A DIGITAL CIRCUIT DESIGN IMPLEMENTATION USING ABEL-HDL AND PROGRAMMABLE LOGIC DEVICES

by

Richard B. Alcorn

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science in Engineering

in the

Electrical Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

June, 1997

A DIGITAL CIRCUIT DESIGN IMPLEMENTATION USING

ABEL-HDL AND PROGRAMMABLE LOGIC DEVICES

Richard B. Alcorn

I hereby release this thesis to the public. I understand this thesis will be housed at the Circulation Desk of the University library and will be available for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

Richard B. Alcorn 5/26/97

Student Date

Approvals:

Salvatore R. Parsaye 5/26/97

Thesis Advisor Date

Robert S. Tomlinson Jr. 5-28-97

Committee Member Date

Samuel J. Skarola 5/30/97

Committee Member Date

Dean of Graduate Studies 6/2/97

Date

ABSTRACT

The purpose of this thesis is twofold. The first is to use ABEL-HDL design software and Programmable Logic Devices (PLD's) to implement the logic of a digital circuit in a more efficient manner than classic digital design techniques allow. The second is to give enough introductory details about PLD's and the ABEL design process that other students may use this thesis as a guide in learning to use ABEL and PLD's in their digital designs. The thesis explores some ABEL design techniques for programming PLD's and applies the concepts to reducing the total chip count of a digital circuit created by the author using 7400-series logic in a previous graduate course. The results of the experimental design implementation are discussed and ideas given for further study into the topic by future thesis students.

ACKNOWLEDGEMENTS

Very special thanks go to Professor Samuel Skarote for being on my thesis committee, for being my thesis advisor up until his retirement in June 1996, and for all of his help and guidance, both academic and personal, throughout my undergraduate and graduate years at Youngstown State University. I apologize for not finishing this thesis before his retirement. I would also like to express my thanks to Dr. Salvatore Pansino for being my thesis advisor and to Dr. Robert Foulkes for being on my thesis committee. In addition, I wish to thank Makin and Associates, Inc. of Mayfield Village, Ohio for their generosity in giving me free samples of Lattice GAL26CV12 devices to use in my experimental circuit. Finally, credit should be given to William I. Fletcher of Utah State University for the origins of the experimental circuit discussed in this thesis. The concept appeared as a homework problem (# 7-2, pg. 516) in his textbook An Engineering Approach to Digital Design[1] and later appeared in revised form on a test given by Dr. Pansino in the EE825 Sequential Logic Circuit Analysis and Design class. I was inspired by the concept and turned it into a full-fledged design project in Prof. Skarote's EE932 Digital Systems Engineering II class.

TABLE OF CONTENTS

PAGE

ABSTRACT iii

ACKNOWLEDGEMENTS iv

TABLE OF CONTENTS v

LIST OF FIGURES vii

CHAPTER

I. INTRODUCTION 1

II. AN OVERVIEW OF PLD'S 4

2.1 Historical Background 4

2.2 Types of PLD's 6

III. AN OVERVIEW OF ABEL-HDL 16

3.1 Program Features 16

3.2 Command Syntax 17

3.3 Source File Structures 19

3.4 ABEL Design Flow 22

IV. ORIGINAL EXPERIMENTAL DESIGN IMPLEMENTATION 24

4.1 Design Concept 24

4.2 Game Controller 26

4.3 Output Controller 27

4.4 Timing Controller 29

4.5 Input Conditioner 31

PAGE

V. ABEL EXPERIMENTAL DESIGN IMPLEMENTATION 33

5.1 Design Concept 33

5.2 GAMECON 34

5.3 OUTCON 38

5.4 TIMECON 41

5.5 INPUTCON 47

5.6 Additional Notes 48

VI. CONCLUSION 50

6.1 Project Results 50

6.2 Summary 52

6.3 Ideas for Future Research 53

APPENDIX A. ABEL DESIGN PROCESS DOCUMENTATION 54

A.1 GAMECON Documentation 54

A.2 OUTCON Documentation 60

A.3 TIMECON Documentation 65

A.4 INPUTCON Documentation 71

APPENDIX B. MANUFACTURER DATA SHEET EXCERPTS 75

B.1 Lattice GAL16V8 Data Sheet Excerpts 76

B.2 Lattice GAL22V10 Data Sheet Excerpts 85

B.3 Lattice GAL26CV12 Data Sheet Excerpts 90

REFERENCES 95

LIST OF FIGURES

FIGURE		PAGE
1.1	Comparison of a 7400-series circuit and the PLD that replaces it	2
2.1	Example of a PROM device	7
2.2	Example of a PLA device	8
2.3	Example of a PLA device (F105) with registers and feedback	9
2.4	Example of a PAL device	10
2.5	Example of a combinatorial PAL (P16L8)	11
2.6	Example of a registered PAL (P16R4)	12
2.7	Example of a macrocell PAL device (GAL22V10)	13
2.8	GAL22V10 macrocell configurations	14
3.1	Sample ABEL logic design source file	20
3.2	A sample ABEL Design Environment screen	22
4.1	Tug of War game concept	25
4.2	Block diagram of original system design implementation	25
4.3	State diagram of Game Controller circuit	26
4.4	Schematic of original Game Controller design	27
4.5	Schematic for original Output Controller circuit	28
4.6	State diagram for original Timing Controller design	29
4.7	Schematic of original Timing Controller circuit	30
4.8	Timing and state diagrams for original Input Conditioner circuit	31
4.9	Schematic of original Input Conditioner circuit	32

FIGURE **PAGE**

5.1 GAMECON source file 35

5.2 OUTCON source file 39

5.3 TIMECON state diagram 42

5.4 TIMECON source file 43

5.5 INPUTCON source file 47

6.1 Schematic of PLD implementation of Tug of War 51

CHAPTER 1

INTRODUCTION

Major improvements have been made in the field of digital circuit design in the last 15 to 20 years. As many are aware, advancements have been made in chip fabrication techniques allowing greater operating speed, lower power consumption, and improved reliability. However, the most revolutionary improvement in digital design has been the types of devices in which designs may be implemented. One of the most popular of these is a class of devices known as Programmable Logic Devices (PLD's). PLD's and some programming techniques for using them in digital designs will be emphasized in this paper.

Every undergraduate Electrical Engineering student is exposed to digital design techniques during the course of his or her study. These digital design courses generally employ 7400-series TTL logic chips as the devices of choice for implementing designs. From an academic point of view, this makes sense in that these devices typically contain only a few logic gates or flip-flops per chip and are easily understood by newcomers to the subject of digital design. These devices are commonly available, inexpensive, require no special programming, and merely need to be powered and wired properly to work in a circuit. However, breadboarding and debugging anything more than a simple circuit can be extremely time-consuming and frustrating. The number of logic chips needed to implement the design can also become quite large. Power consumption and the physical size of the final circuit are also valid concerns. For these and other reasons industry has tended to shy away in most cases from using 7400-series logic for most digital circuit designs.

On the other hand, Programmable Logic Devices (PLD's) allow digital circuit designers to implement designs with just a few chips, the actual number depending on the complexity of the design and the types of devices used for implementation. Combinatorial or sequential designs may be programmed into a PLD by the user via a device programmer unit. The resultant device is able to perform the same logic functions that would've taken numerous 7400-series chips to accomplish using traditional design methods (Figure 1 shows an example of this). Breadboarding times drop enormously, and most debugging can be done at the design level rather than at the physical device and circuit level. The reduction in chip count also leads to reductions in power consumption, physical circuit size, and cost. For these reasons, programmable logic is now employed in many commercial and industrial digital circuit designs.

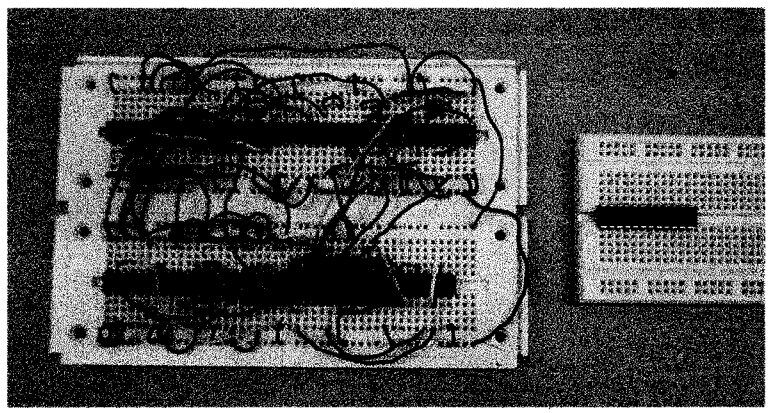


Fig. 1.1 - Comparison of a 7400-series circuit (left) and the PLD that replaces it (right)

The ABEL Hardware Description Language from Data I/O Corporation is used in this project for all PLD programming and in all of the programming examples and descriptions. It's a very powerful software package for programming PLD's, but it has a significant learning curve. To the best of the author's knowledge, the work done in this project goes much further into this subject than what any other YSU Electrical Engineering student has done thus far. Therefore, this project and paper may be useful as

a guide to other students who wish to use ABEL-HDL and PLD's in their designs. The lessons and techniques that were learned by the author and are presented here could save others some struggling and headaches and get them on the road to programming PLD's more quickly. This forms a secondary purpose of the thesis and is responsible for the order of appearance of the subjects presented.

The main purpose of this project is to investigate and test some methods of using ABEL-HDL design techniques and PLD's to reduce the chip count of a digital circuit. After an introduction to PLD types and their features (Chapter II), the ABEL-HDL design software is examined with respect to command types and program file structures (Chapter III). This knowledge is then applied to reduce the chip count in an experimental circuit designed and built by the author in a previous graduate digital design course, employing 7400-series design techniques and devices. Despite efforts to make the original design as efficient as possible, the final chip count totaled 31. In contrast, the final PLD implementation requires only 4 devices in order to duplicate the logic of the original circuit (plus a few chips used to isolate the PLD chips from possible harm). The original experimental circuit design is discussed in Chapter IV. This is then followed by the PLD experimental circuit design in Chapter V.

Finally, the project results are examined and some ideas for future research are given. During the course of this project, various problems or items of interest were encountered that could provide a student an opportunity for further study, but were deemed by the author to be outside the scope of this thesis project. These are presented in Chapter VI.

CHAPTER II

AN OVERVIEW OF PLD'S

2.1 - Historical Background

Until the mid 1970's, digital circuit designers basically had two choices in how to physically implement their designs. The first method was to use off-the-shelf 7400-series devices to build their circuits. This technique had advantages such as low development costs, short design times, and the ability to build and test circuits without the need for specialized programming equipment. However, circuits constructed in this manner tend to be large in size and power consumption. The second method was to use mask-programmed devices. These were custom manufactured (often by photo-etching) devices produced by chip foundries that were designed to satisfy the logical functions specified by the customer's design. Generally, the equivalent logic of hundreds or thousands of gates can be implemented in one device package. Using these devices in designs leads to smaller circuit sizes along with reduced power requirements. However, design times for mask-programmed devices are usually long and development costs can be high. Also, elaborate and complicated testing procedures are often needed to evaluate the programmed devices. For these and other reasons, large production volumes of the end circuit are generally necessary in order to justify using these devices in digital designs. It was apparent that a desirable solution for digital circuit designers would be to have devices with relatively large equivalent gate densities, but also have the ability to be programmed by the designer. This would result in a best of both worlds situation: the

reduced circuit size afforded by mask-programmed devices, and the design flexibility of off-the-shelf devices.

The early 1970's saw the introduction of the first PROM (Programmable Read-Only Memory). It was soon found that these devices could be used to implement some simple logic functions as well as their intended data-storing function. These devices quickly became popular with designers. However, PROM's were limited in what types of logic functions they could implement.

The PLA (Programmable Logic Array) made it's appearance in the mid 1970's. This type of device allowed designers to program logic functions using classic SOP (Sum of Products) format. The PLA was originally just a combinatorial logic device. However, it did not take chip manufacturers long to add feedback paths and flip-flops, thus creating a class of PLA-based devices capable of implementing sequential logic as well.

PLD's finally caught on in popularity with digital designers after the introduction of the PAL (Programmable Array Logic) in the late 1970's. These devices also feature SOP logic implementation and come in combinatorial and registered forms. However, they forfeit some of the user-programmability of the PLA in favor of device speed and simplicity. Regardless, PAL-type devices have grown to be one of the most popular PLD types in use today. They have grown to include many advanced features that can be controlled by the designer. CMOS versions have also been introduced. These devices may be erased electrically and reprogrammed by the designer, making the design and debugging process more efficient and cost-effective.

Universal programming software became readily available in the early 1980's and quickly found a home on personal computers. ABEL-HDL from Data I/O Corporation was one of the first full-featured design software packages and is still an industry standard today. Other software packages are also commonly used in industry, an example of which is CUPL (Common Universal tool for Programmable Logic) from Logical Devices. It was initially developed at about the same time ABEL-HDL was being

developed. Many other software packages exist which tend to offer fewer design features. Device manufacturers also occasionally offer design software that is oriented toward their particular brand of devices.

The last major advancement in programmable logic occurred in the mid-1980's with the introduction of the FPGA (Field Programmable Gate Array). With an equivalent gate density in the thousands and performance rivaling that of mask-programmed devices, this type of device was a radical departure from PLD's. FPGA's are made up of many LCA's (Logic Cell Array) along with signal routing lines. The LCA essentially consists of a small number of gates that can be programmed to implement a simple logic function. These LCA's are then connected by the user with programmable signal routes. This arrangement leads to greater design flexibility than PLD's can offer, but also more complex methods of programming. Except for the fact that they are programmable devices, FPGA's are so different in design from PLD's that they may be considered to be a family of devices outside the realm of PLD's. For this reason, the author chose not to focus attention on them.

2.2 - Types of PLD's

The device structures of PROM's, PLA's, and PAL's each contain programmable fuse arrays that are used by the designer to program the desired logic functions into the device. Array connections that are not needed for the design logic have their respective fuses blown. Fuses are left intact for array connections that are necessary for the programmed logic functions. Each of the three types of PLD have AND and OR arrays that are used to implement logic in SOP form. What distinguishes them from each other is which arrays can be programmed by the user and which ones have permanent connections.

The PROM (see Fig. 2.1) has a fixed AND array that provides all of the possible input product terms. The OR array has fused connection points and can be programmed by the user. Any output in a PROM can be the sum of any or all of the outputs from the AND array. This arrangement is adequate for uses such as address decoders or data storage. SOP logic functions can also be implemented and sequential logic is a possibility, but external storage registers are needed. These uses for PROM's are rather inefficient, however. Every combination of inputs is available in the AND array in a PROM, but very rarely are all of these product terms used in a logic function. Therefore, much of the device resources go to waste. Also, since all product terms are available for a given number of inputs, each additional input doubles the number of terms in the AND array (# of terms = 2^n, where n = # of inputs). This increases the physical size and complexity of the device and tends to put a practical limit on the number of inputs available.

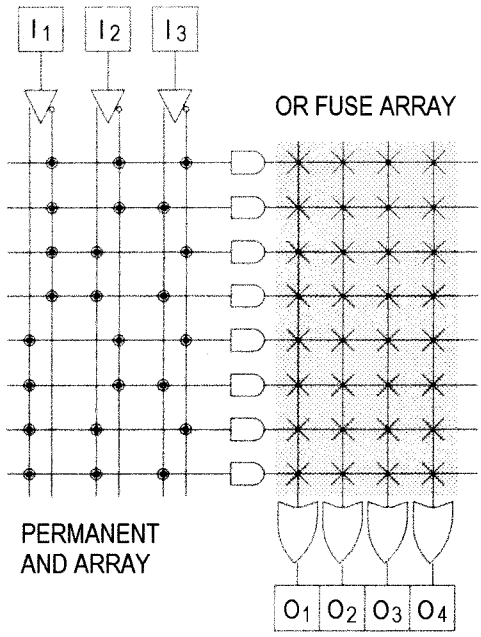


Fig. 2.1 - Example of a PROM device

PLA devices (see Fig. 2.2) are the most flexible type of PLD. They have both AND and OR fuse arrays that may be programmed by the designer. The AND array resources are more limited than in PROM's and it is impossible to have every possible product term represented. This reduces the physical size of the device, but it forces the designer to perform some minimization of the design logic to be implemented. As in PROM's, any output can be the sum of any or all of the product terms in the AND array. This allows for product sharing in the outputs and can lead to more efficient use of device resources. However, while PLA devices are very flexible for design implementations, they are limited in speed due to the need for signals to propagate through two fuse arrays. Figure 2.3 shows an example of a PLA type of device that includes registers and feedbacks.

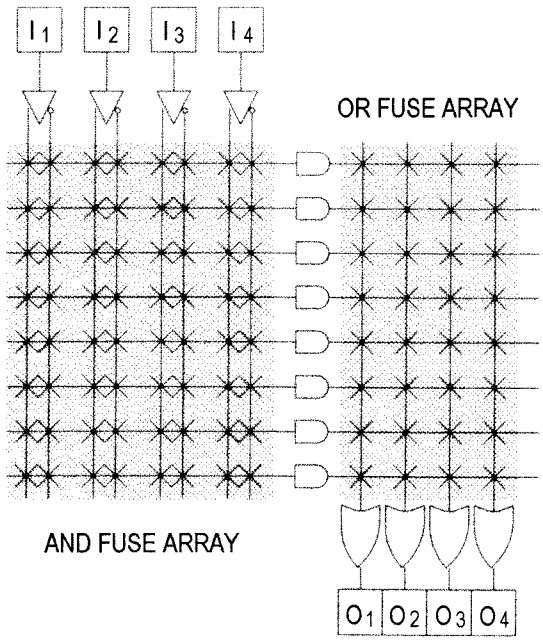


Fig. 2.2 - Example of a PLA device

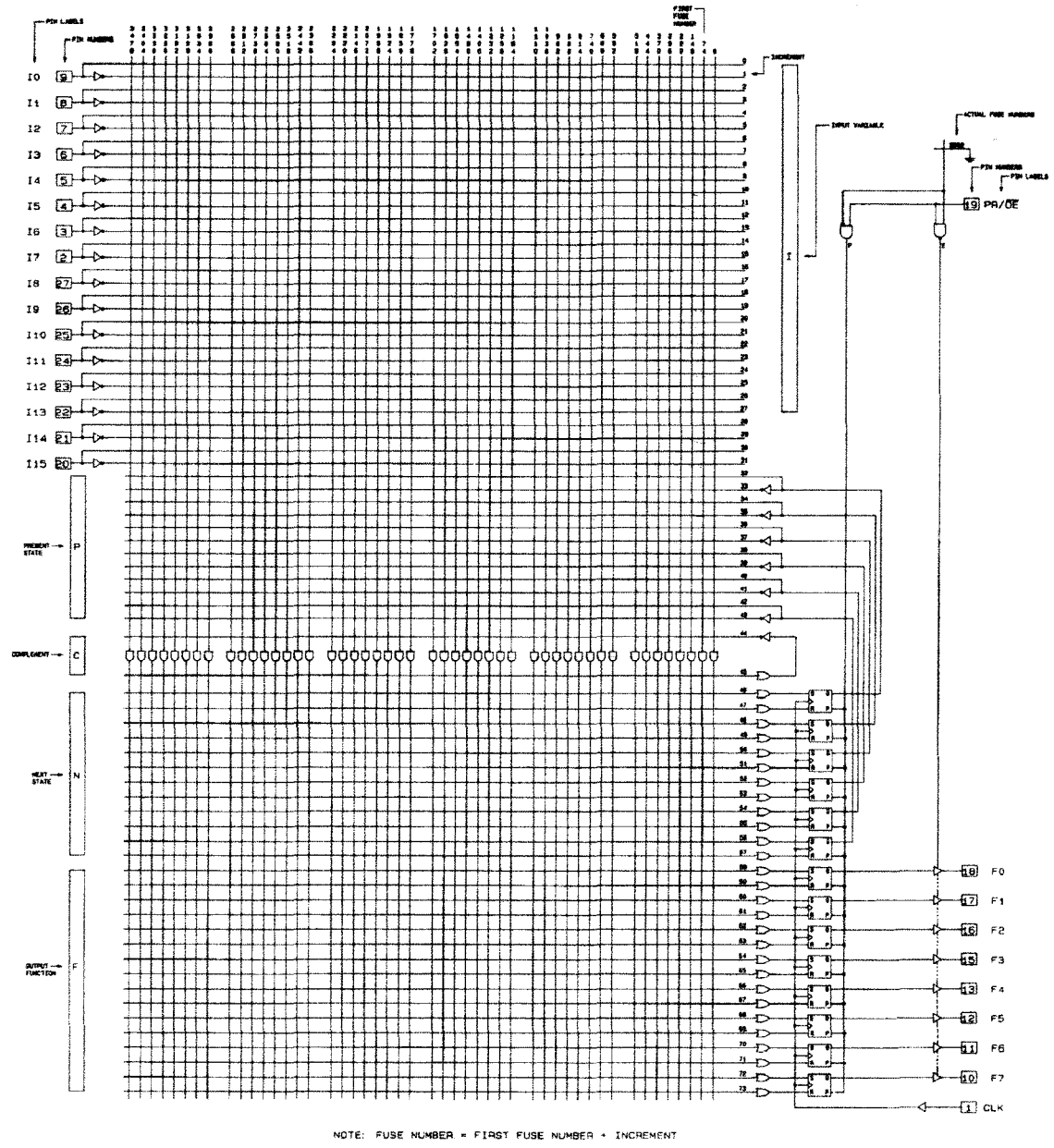


Fig. 2.3 - Example of a PLA device (F105) with registers and feedback

PAL devices (see Fig. 2.4) have fused AND arrays like in the PLA, but have fixed OR arrays. Generally, only one OR gate is assigned to each output pin. The product terms associated with one OR gate are not available to the other OR gates in the array, thus making product sharing all but impossible. In addition, a limited number of product terms are available in the AND array. Therefore, logic minimization is very

important when using PAL devices. These devices have a speed advantage over PLA's, though, since signals only need to propagate through one fuse array. Figure 2.5 shows an example of a combinatorial PAL. A example of a registered PAL is shown in Fig. 2.6. More advanced PAL devices feature output macrocells. These macrocells are configurable by the designer to form combinatorial or registered output configurations with programmable output polarities. These devices with macrocells are extremely flexible and can often directly replace many other types of PAL with just one device. An example of a macrocell-type PAL is shown in Fig. 2.7. The possible macrocell configurations for that device are depicted in Fig. 2.8.

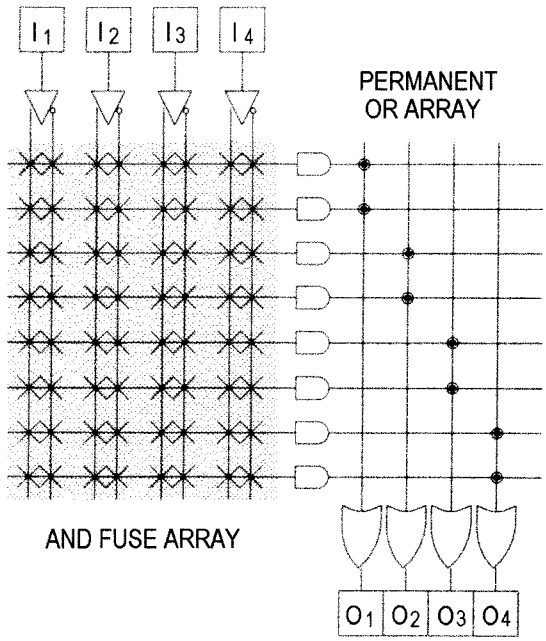


Fig. 2.4 - Example of a PAL device

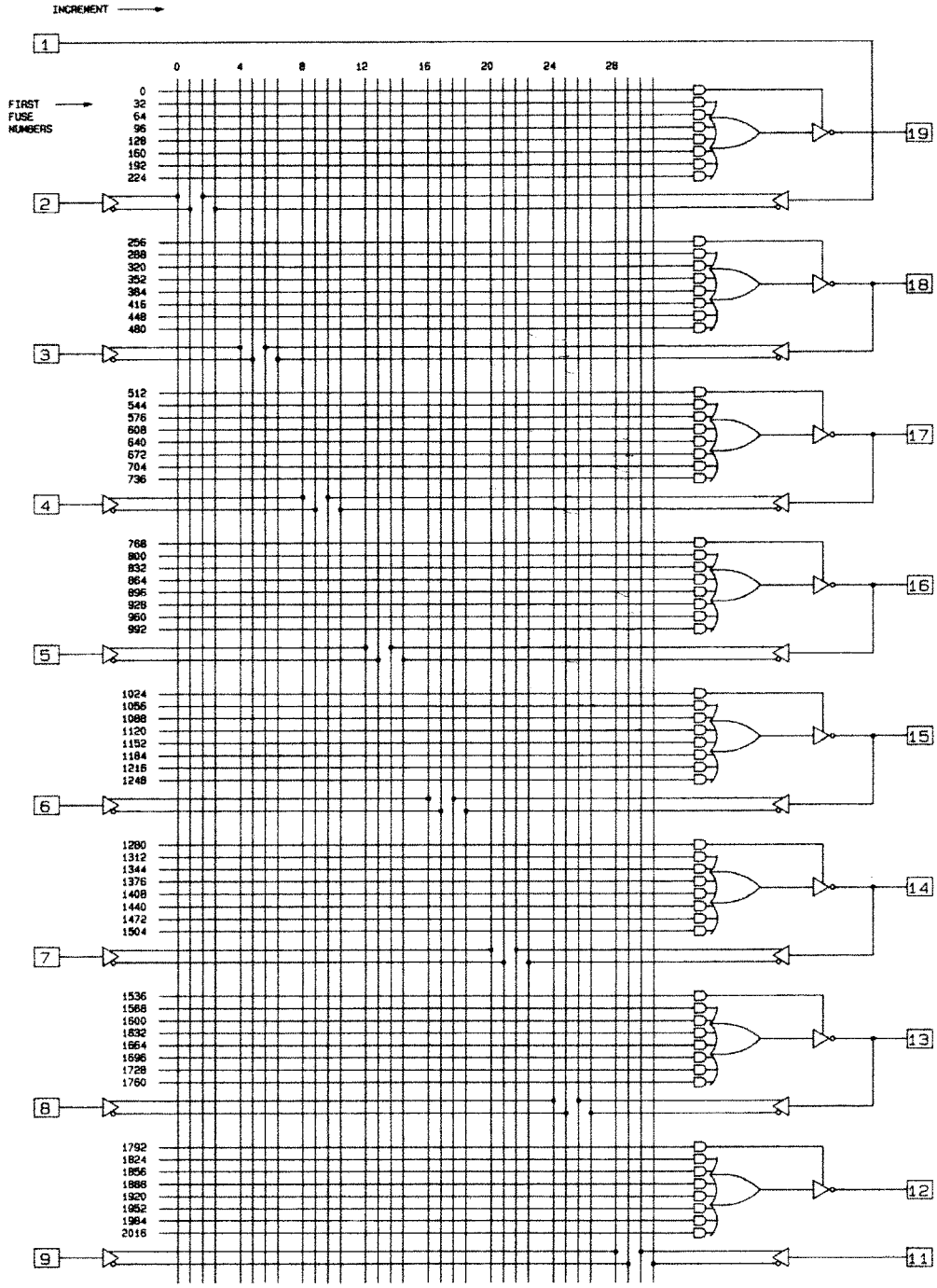


Fig. 2.5 - Example of a combinatorial PAL (P16L8)

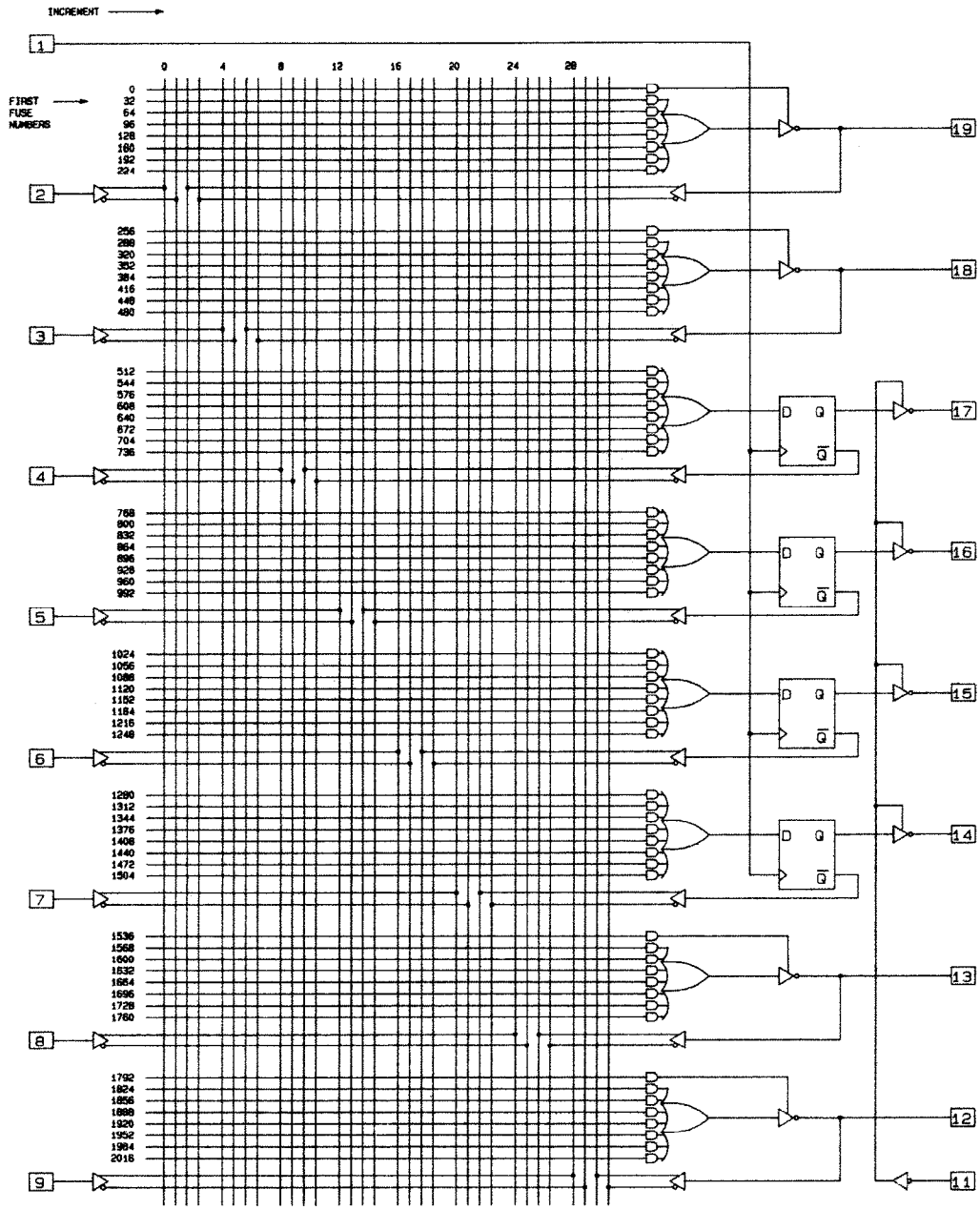


Fig. 2.6 - Example of a registered PAL (P16R4)

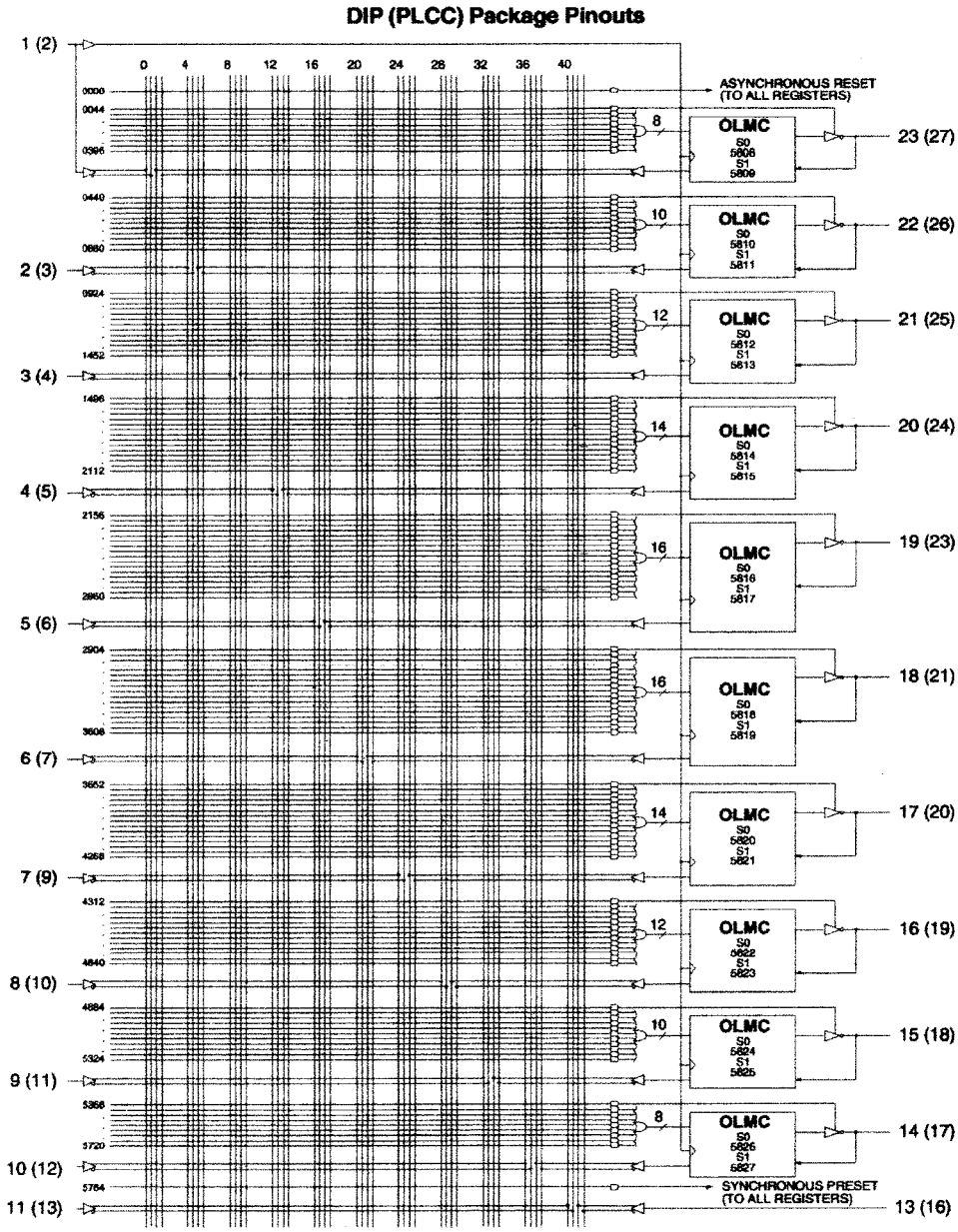
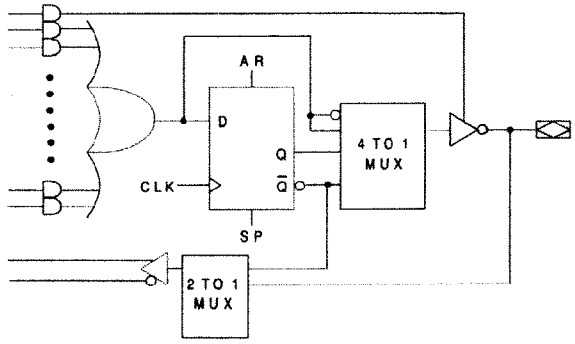
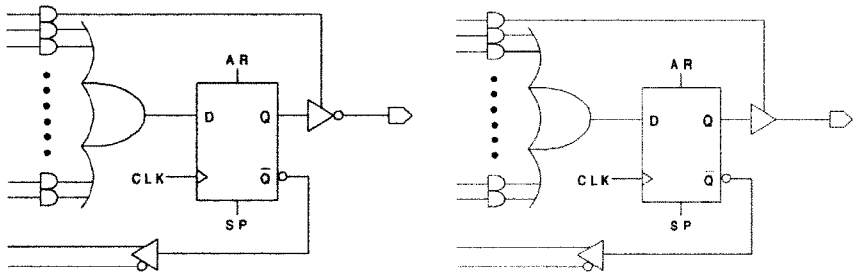


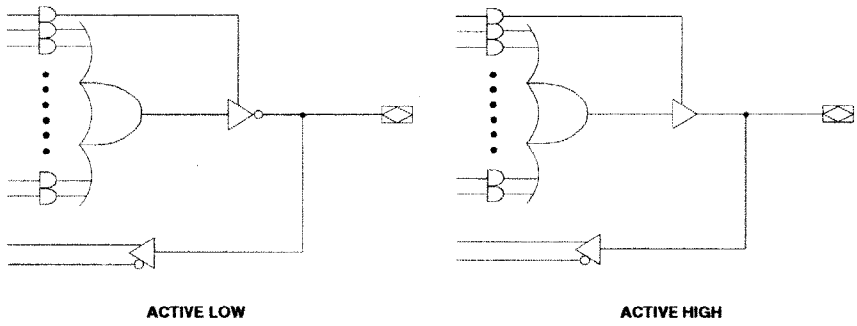
Fig. 2.7 - Example of a macrocell PAL device (GAL22V10)



GAL22V10 OUTPUT LOGIC MACROCELL (OLMC)



REGISTERED MODES



COMBINATORIAL MODES

Fig. 2.8 - GAL22V10 macrocell configurations

This only scratches the surface of the number of PLD types that are available. Those readers who are interested in more information on which types of PLD's exist (including FPGA's) and the history of their development are referred to Practical Design Using Programmable Logic[2] by David Pellerin and Michael Holley. This book proved to be an invaluable resource to the author in preparing this paper. The Data I/O Logic Diagram Package[4] manual (a companion to the ABEL manuals) is also a good reference. It contains detailed schematics of over 200 PLD's.

CHAPTER III

AN OVERVIEW OF ABEL-HDL

3.1 - Program Features

ABEL-HDL is a powerful software package that allows the digital designer to implement designs in PLD's using familiar techniques such as Boolean equations, truth tables, and state diagrams. Using a text editor such as MS-DOS Edit or ABEL's built-in editor, the designer enters the description of the logic to be implemented. The description can be quite specific, even to the point of defining actual fuse assignments, if the end device for implementation is known. However, ABEL also allows the designer to describe the design logic in very generic terms without specifying an end device. ABEL then has the ability to determine (along with an optional set of criteria defined by the user) which devices are capable of implementing the design logic. A variety of logic minimization algorithms are built into the program and can be used to simplify the design logic. Extensive simulation capabilities exist in for determining the validity of the design logic at multiple stages of the ABEL design process. Finally, once the design logic has been "fitted" into a particular device, a fuse data file may be created for downloading to a device programmer unit.

The version of ABEL-HDL used by the author in this project is ABEL 4.10 (Version 4.03 was used initially, but was later upgraded to 4.10). It was released in 1991 or 1992 and runs under MS-DOS. The latest release is ABEL 6. It runs under Microsoft Windows and offers many graphical add-on utilities such as the StateCAD Graphical Design Entry program (allows graphical state diagram design entry) and Waveform

Viewer (allows simulation results to be displayed in familiar timing chart form). However, the same program structures and command syntaxes exist as in ABEL 4 for DOS. Therefore, the design examples depicted in this project are still valid. In fact, the DOS version of ABEL still exists under a new title: ABEL-PLD. Data I/O offers it as an entry-level means of programming PLD's. However, it is not capable of using the graphics utilities and other advanced features of ABEL 6.

3.2 - Command Syntax

ABEL uses a set of symbols for Boolean operators that is different from those introduced in classical digital design courses. These are used throughout this paper and in the design examples and are summarized in Table 3.1.

TABLE 3.1

ABEL BOOLEAN OPERATORS

Symbol	Description
!	NOT
&	AND
#	OR
$	Exclusive OR
!$	Exclusive NOR

Assignment operators are used to assign values to an output. These are used when writing equations. Separate symbols are used for combinatorial or registered outputs. Using a registered assignment operator in describing the logic for an output indicates that the output will take on the evaluated value of that logic at the next clock cycle. Assignment operators are shown in Table 3.2.

TABLE 3.2

ABEL ASSIGNMENT OPERATORS

Symbol	Description
=	combinatorial output assignment
:=	registered output assignment

Relational operators are used for comparing two items in an expression. They are primarily employed for conditional decision-making in state diagrams, but can also be used in equations. These are listed in Table 3.3. They are straight-forward except for the symbol for the 'equal' condition. The usage of two '=' symbols was necessary in order to differentiate the condition of 'equal' from the combinatorial assignment operator.

TABLE 3.3

ABEL RELATIONAL OPERATORS

Symbol	Description
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

ABEL allows signals, either inputs or outputs, to be grouped into sets. This allows some simplification for the designer in writing the design logic. An example would be in the case of state variables. Say there are four state variables in the design, y1 through y4. If the design logic requires the comparing of state values, it would simplify things if the state variable were grouped into a set. For example:

State_Value = [y1,y2,y3,y4]

Then, instead of writing logic that compares the value of each state variable individually, the designer may compare the value of the set as a whole. For example:

IF (State_Value == 5) THEN...

ABEL also allows numeric values to be expressed in bases other than ten. Base ten is assumed whenever a number is used. If the designer wishes to express a numeric value in a different base, a base operator prefix must be used. These are shown in Table 3.4.

TABLE 3.4

ABEL BASE OPERATOR PREFIXES

Base System	Prefix
binary	^b
octal	^o
decimal	^d
hexadecimal	^h

Using these base prefixes, the state value in the above example could also be expressed in another base, say binary:

IF (State_Value == ^b0101) THEN...

These descriptions of command syntax that have been given are by no means exhaustive, but are sufficient for understanding the design examples to follow in this paper. If further syntax-related information is desired, the reader is referred to the ABEL User Manual.[3]

3.3 - Source File Structures

A digital logic design that is implemented in ABEL must conform to a standard file format. There are a number of required structures in an ABEL source file. However, there are also a number of optional items that the designer may include to simplify or clarify the design. A sample ABEL source file is shown in Fig. 3.1. It's part of the experimental circuit that will be discussed later. At the moment, it will serve to demonstrate the structures, both required and optional, of an ABEL file.

```
module     inputcon
title      'Input Conditioner, Tug-o-war Game'

declarations

   inputcon device 'p16v8as';

   CLK      pin 1;
   P1_IN    pin 2;
   P2_IN    pin 3;
   Y1       pin 17;
   Y2       pin 16;
   Y3       pin 15;
   Y4       pin 14;
   P1_OUT   pin 19;
   P2_OUT   pin 18;

equations

   Y1 = (P1_IN & Y1.pin) # (CLK & Y2.pin);
   Y2 = (!CLK & P1_IN) # (P1_IN & Y2.pin) # (CLK & Y2.pin);
   P1_OUT = (P1_IN & !Y2.pin) # (!CLK & P1_IN & !Y1.pin);
   Y3 = (P2_IN & Y3.pin) # (CLK & Y4.pin);
   Y4 = (!CLK & P2_IN) # (P2_IN & Y4.pin) # (CLK & Y4.pin);
   P2_OUT = (P2_IN & !Y4.pin) # (!CLK & P2_IN & !Y3.pin);

test_vectors    'Pulse catcher for P1'

  ( [ CLK,P1_IN ] -> [ P1_OUT ] )
    [  0,   0   ] -> [ 0 ];
    [  0,   1   ] -> [ 1 ];
    [  1,   1   ] -> [ 0 ];
    [  1,   1   ] -> [ 0 ];
    [  0,   1   ] -> [ 0 ];
    [  0,   1   ] -> [ 0 ];
    [  1,   1   ] -> [ 0 ];
    [  1,   0   ] -> [ 0 ];
    [  0,   0   ] -> [ 0 ];
    [  0,   0   ] -> [ 0 ];
    [  1,   0   ] -> [ 0 ];
    [  1,   1   ] -> [ 1 ];
    [  0,   1   ] -> [ 1 ];
    [  0,   1   ] -> [ 1 ];
    [  1,   1   ] -> [ 0 ];
    [  1,   1   ] -> [ 0 ];
    [  0,   1   ] -> [ 0 ];
    [  0,   0   ] -> [ 0 ];
    [  1,   0   ] -> [ 0 ];

end
```

Fig. 3.1 - Sample ABEL logic design source file

The first few lines at the top of the file form a Header. The Header is a required structure, but includes optional elements as well. The 'module' statement is required. It defines the beginning of the source file and gives the module a name. In this example, the module name is 'inputcon'. An optional title may be given to the module. It

is indicated with the 'title' statement. This statement is not acted on by the ABEL compiler, but it does allow the title to show up in the documentation produced by ABEL.

Next comes the Declarations structure. This area of the file is reserved for indicating details such as device type, signal-to-pin assignments, signal attributes, and user-defined constants and sets. The 'declarations' statement is not a required element. ABEL assumes that any statements following the Header and preceding the Logic Structure(s) (equations, truth_table, and state_diagram) are indeed declarations. In this example, the 'device' statement is used to declare the PLD intended to be used and the 'pin' statement is used to declare signal-to-pin assignments. If a generic approach is being taken in the design process, the device line and the pin numbers may be eliminated. ABEL can be made to find devices that the module will 'fit' into and automatically assign pin numbers to the defined input/output signals.

Following the Declarations are the Logic Structures. These portions of the source file define the design logic. The descriptions may be equations, truth tables, or state diagrams. At least one of these Logic Structures are required in a module, but it's also valid in many situations to have more than one type in the same source file. A Logic Structure begins with a required keyword. These keywords are 'equations', 'truth_table', and 'state_diagram'. The Logic Structure used in the example source file is equations.

After the Logic Structures, the designer may include a set of Test Vectors. This structure is optional, but it is highly advisable that it be used. It allows the designer to verify that the design logic functions as intended. This structure begins with the 'test_vectors' keyword. The designer then writes a set of vectors that tell ABEL how particular outputs behave when subjected to certain input stimuli. Using these test vectors, it is possible to describe all of the functional behavior of the design logic. Once the ABEL source file has been compiled, the resulting compiled logic may be simulated through the use of the test vectors. The input portion of the vectors are used to stimulate the compiled logic and the results are compared to the output portion of the vectors.

ABEL alerts the designer to any discrepancies between the expected output values and the values determined by the simulation. This is a great aid to the designer in debugging a logic design.

Finally, the module is terminated with the 'end' keyword. This is a required structure. It's optional for the module name to follow the 'end' keyword, but it's rarely used in practice.

3.4 - ABEL Design Flow

All processing of an ABEL source file takes place in the ABEL Design Environment. Upon typing 'ABEL4' at the DOS prompt, the ABEL Design Environment screen appears. A sample screen is shown in Fig. 3.2.

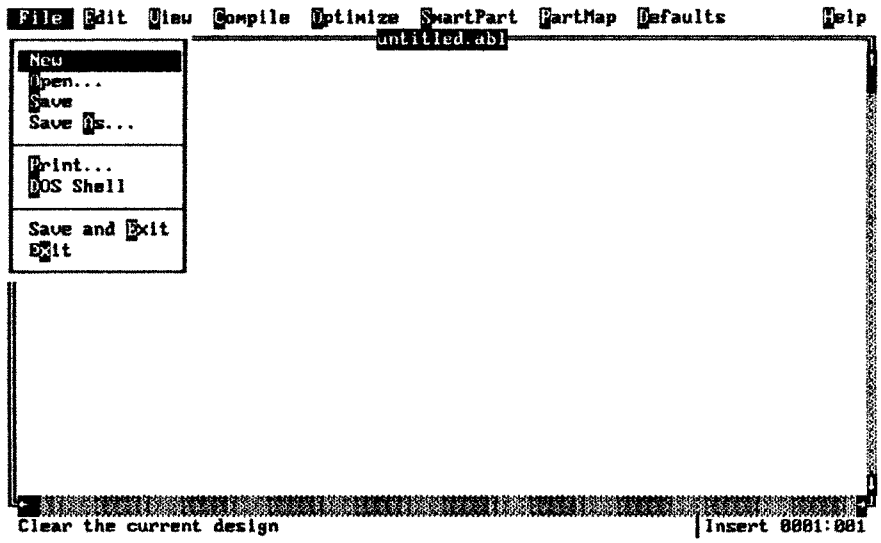


Fig. 3.2 - A sample ABEL Design Environment screen

The ABEL source file may be created in the built-in screen editor or created externally. Either way, once it is loaded into the ABEL Design Environment, it then may be compiled via commands in the 'Compile' pull-down menu. If the source file includes test vectors, they then may be simulated (also from the 'Compile' menu). If errors occur

in the compilation or simulation processes, they may be seen using commands from the 'View' menu. The 'View' menu also allows the designer to view documentation produced by other ABEL processes.

Having been compiled, the source file then is optimized. There are a number of logic minimization options available in the 'Optimize' menu. The optimized source file may then be simulated (also in the 'Optimize' menu).

The optimized file is then 'fitted' into a device. This is accomplished from the 'SmartPart' menu. If the design is generic in nature, the Device Database may be searched for PLD's in which the design is capable of being implemented. If a device-specific design is being used, the device is already known. Either way, the source file may be fitted into the device and the results simulated. The signal-to-pin assignments may also be calculated by ABEL at this time, if they have not already been assigned in the source file.

Once the source file is fitted into a device, a JEDEC fuse file is created. The options for doing this are found in the 'PartMap' menu. This fuse file may then also be simulated. The resulting <filename>.jed file should then be copied onto a 3-1/2" disk to be read by the Unisite programmer. This JEDEC file contains the actual fuse data for the physical process of programming the PLD with the design logic.

The above directions are meant to be only an introduction into the ABEL design process. For further information, the reader is again referred to the ABEL User Manual.[3]

CHAPTER IV

ORIGINAL EXPERIMENTAL DESIGN IMPLEMENTATION

4.1 - Design Concept

The ABEL design examples that will be discussed are derived from a digital circuit designed and built by the author in a graduate digital logic design course during the summer of 1992. 7400-series devices were used in the design implementation. Every effort was made to limit the total chip count, but in spite of these efforts, the final tally came to 31 chips.

The circuit was designed to be a 'Tug of War' game. The concept is depicted in Fig. 4.1. The 'rope' consisted of nine red LED's arranged in a row. A lit LED indicated the 'knot' position on the rope. A green LED on either end of the rope indicated the respective winner of the game. A game was started by pressing the RESET button. A time delay of about 1.5 seconds occurred and then a piezo buzzer sounded to signal both players to begin pushing their respective buttons. They would continue to push their buttons until the knot was 'pulled' to their respective winner LED. The piezo buzzer would then sound three times to indicate that a win condition occurred. The game then waited in this state until the RESET button was pressed again and another game would start. In addition, there were two more features. First, all LED's would be lit at power up to test that they were functioning. Also, if a player would push his button prior to the 'GO' signal from the buzzer, it was considered cheating and the other player would automatically win. The final circuit was tested and functioned as intended.

A block diagram of the system design is shown in Fig. 4.2. The system consisted of four main circuits: the Game Controller, Output Controller, Timing Controller, and Input Conditioner.

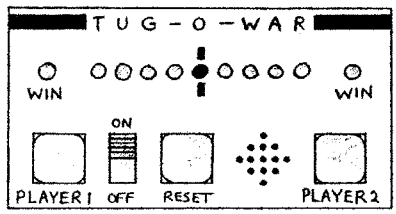


Fig. 4.1 - Tug of War game concept

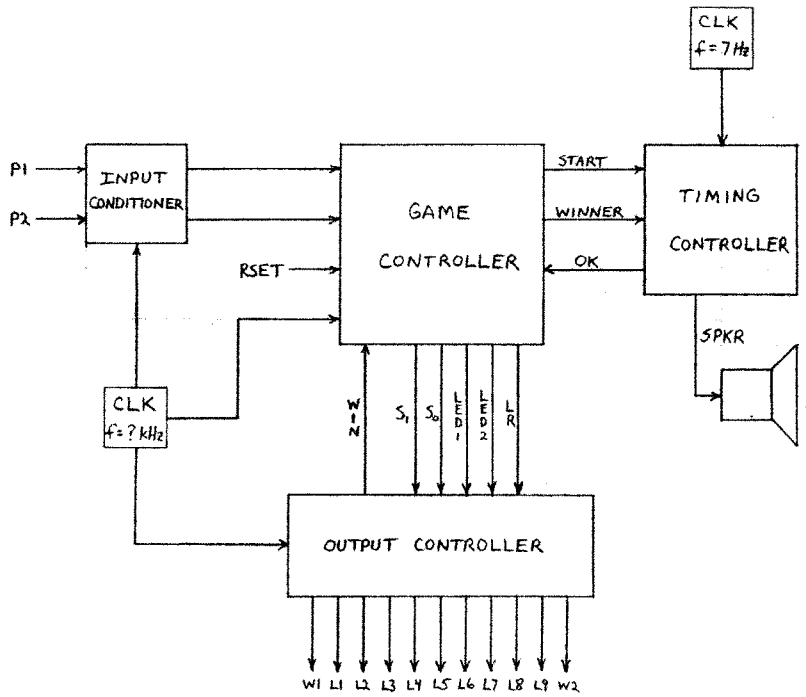


Fig. 4.2 - Block diagram of original system design implementation

4.2 - Game Controller

The Game Controller managed all of the game rules and course of play. It would issue signals to the Output Controller shift registers to dictate LED 'movement' and to the Timing Controller to indicate that a time delay or a winner signal should be started. Communications with the Output Controller were handled synchronously. However, because the Timing Controller operated at a much slower clock frequency than the Game Controller, communications with it were accomplished through asynchronous handshaking. The Game Controller design itself was implemented as a Moore-type sequential circuit. The state diagram is shown in Fig. 4.3 and the circuit schematic is depicted in Fig. 4.4. It may also be interesting to note that the 7400-series circuit seen in Fig. 1.1 is the actual original Game Controller circuit.

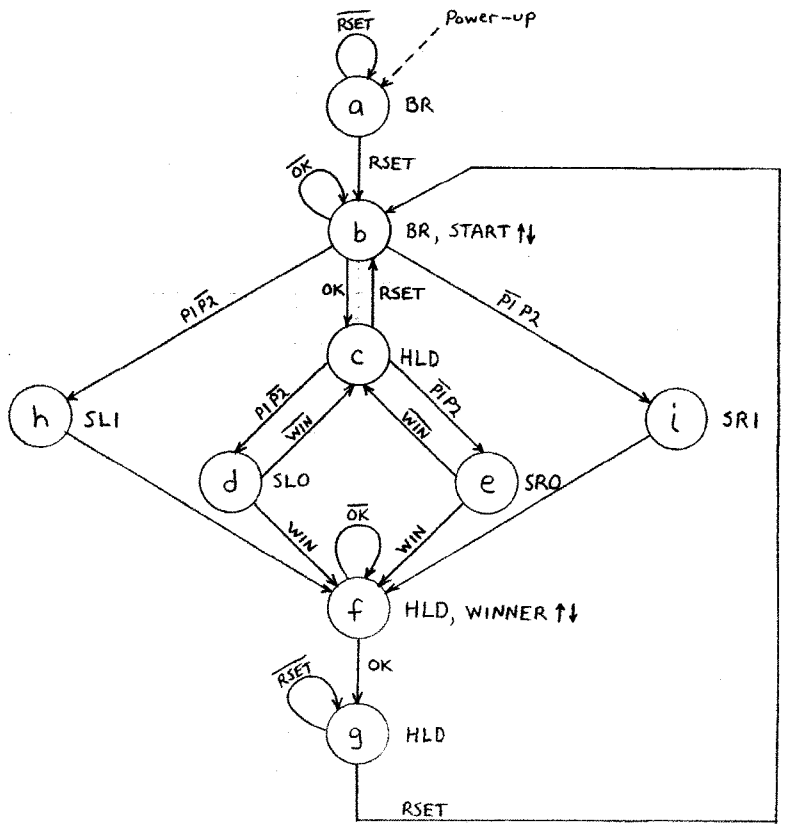


Fig. 4.3 - State diagram of Game Controller circuit

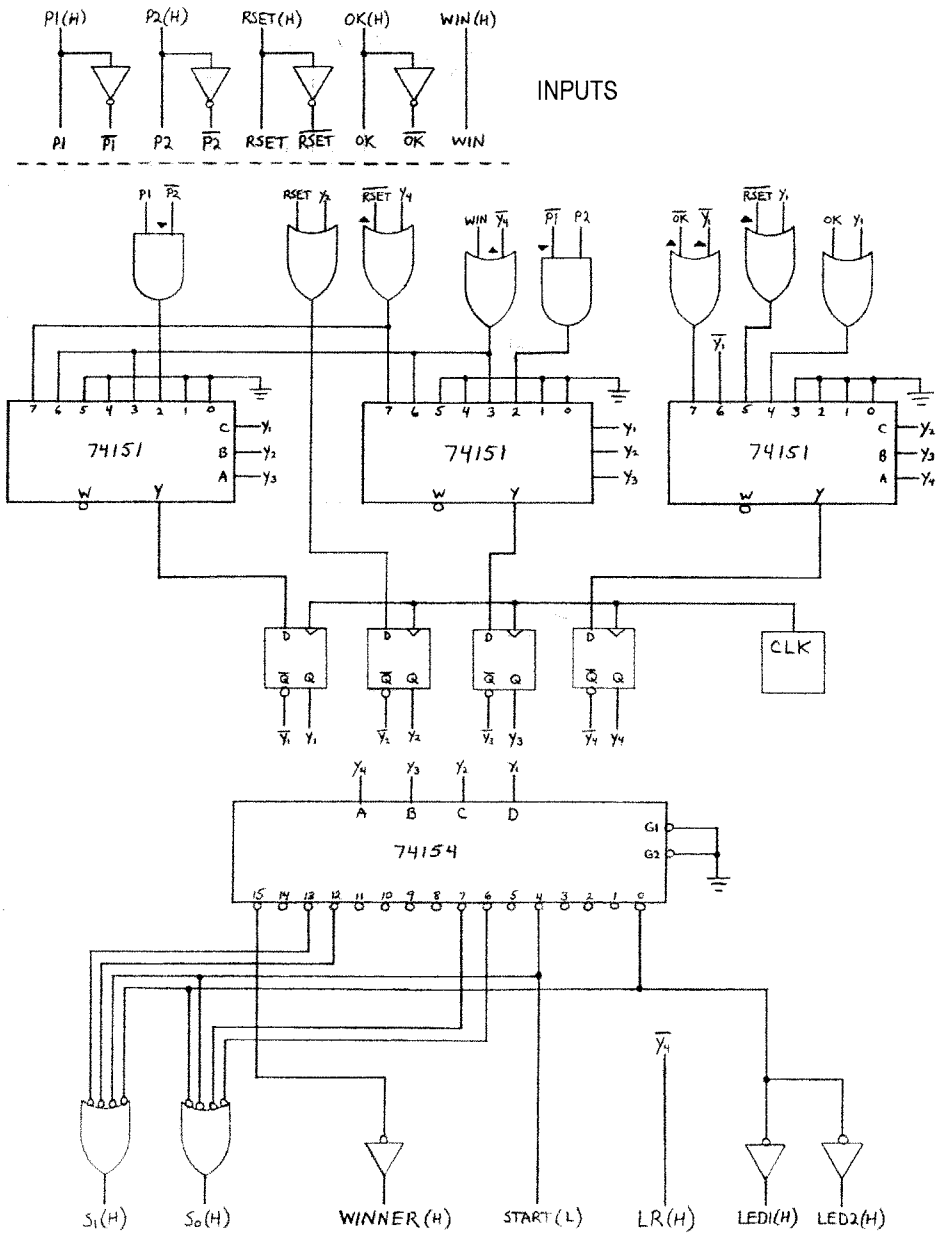


Fig. 4.4 - Schematic of original Game Controller design

4.3 - Output Controller

The Output Controller was in charge of controlling the status of the LED's and alerting the Game Controller when a win condition had occurred. The circuit (see Fig. 4.5) was composed primarily of three 4-bit universal shift registers connected in series. The LED signals came directly from the shift register outputs. The shift and broadside

load signals and data bits came from the Game Controller. In order to get win status information back to the Game Controller in a synchronous manner, the shift registers were made to clock on the falling edge of the system clock pulse (note inverter on clock signal). This ensured that no signal lag occurred.

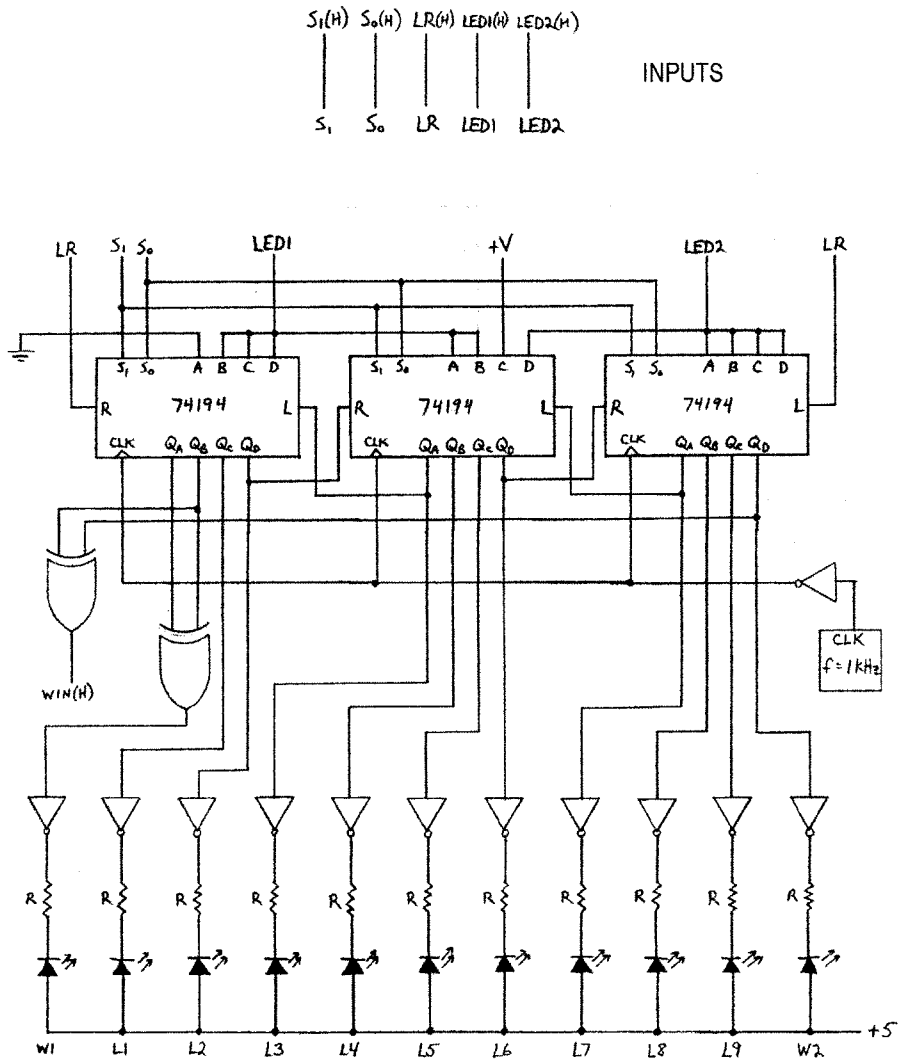


Fig. 4.5 - Schematic for original Output Controller circuit

4.4 - Timing Controller

The Timing Controller took care of all timing routine and sound effect duties. The circuit was built around a 4-bit binary counter clocked at a very slow pace (approximately 5-10 Hz). This difference in clock speeds (the system clock frequency was set to about 1-5 kHz) mandated that the timing functions be relegated to a separate circuit and that communications between it and the Game Controller be asynchronous in nature. The Timing Controller design was implemented as a Mealy-type synchronous circuit, using the counter outputs as state registers. The state diagram is shown in Fig. 4.6 and the circuit schematic is depicted in Fig. 4.7.

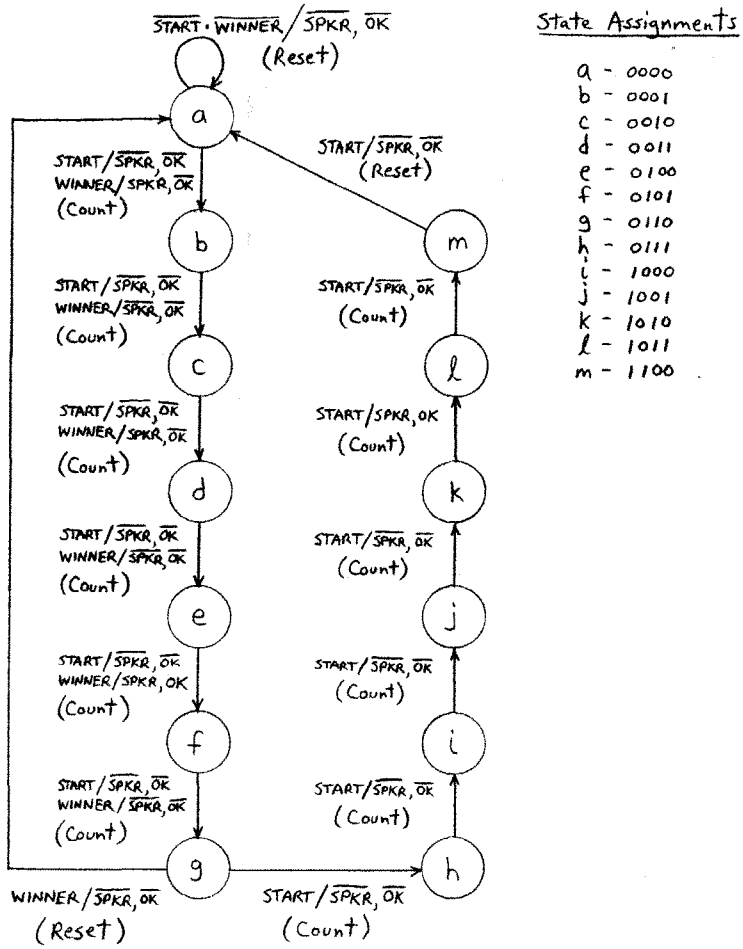


Fig. 4.6 - State diagram for original Timing Controller design

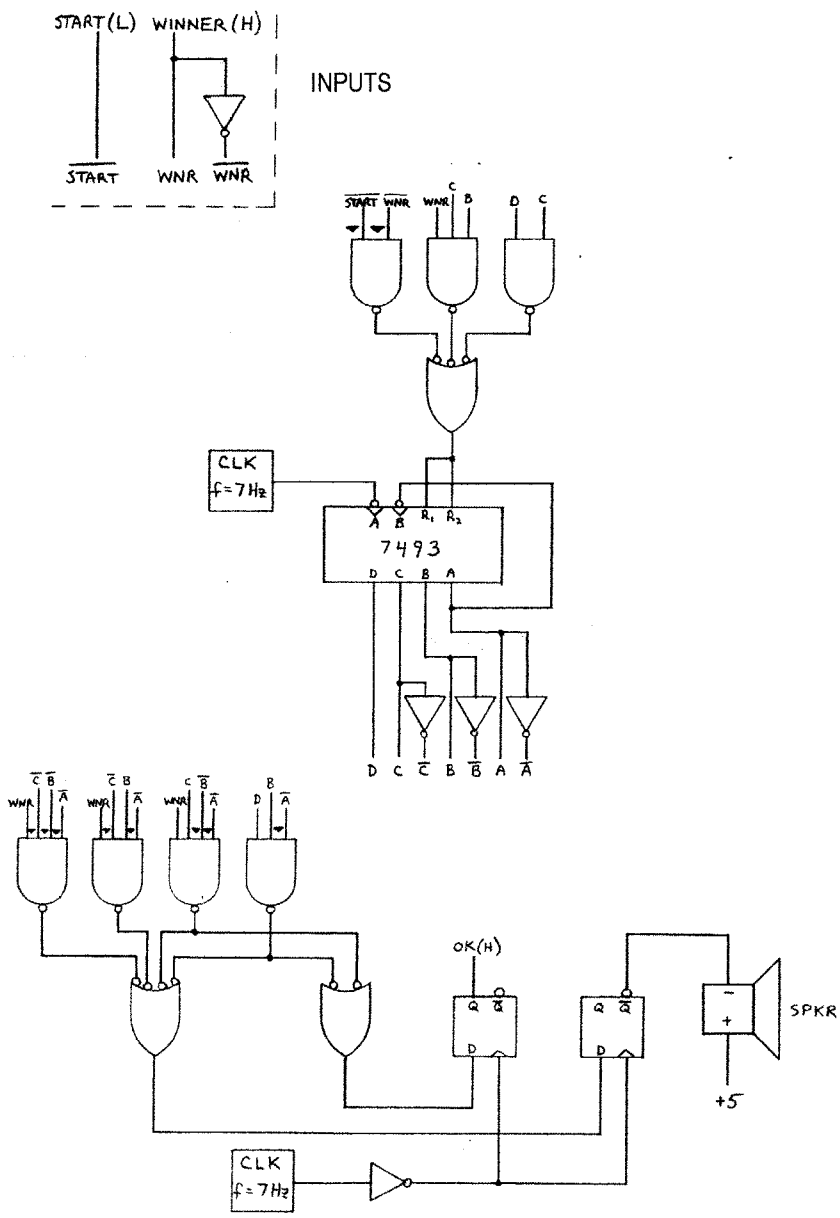
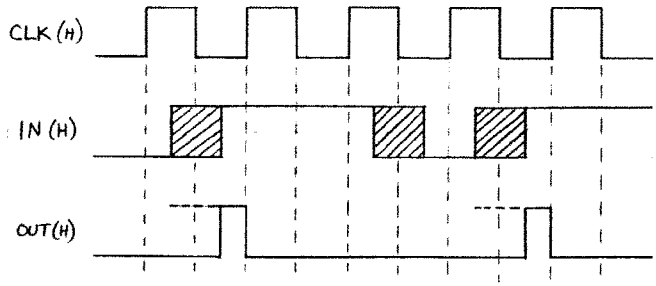


Fig. 4.7 - Schematic of original Timing Controller circuit

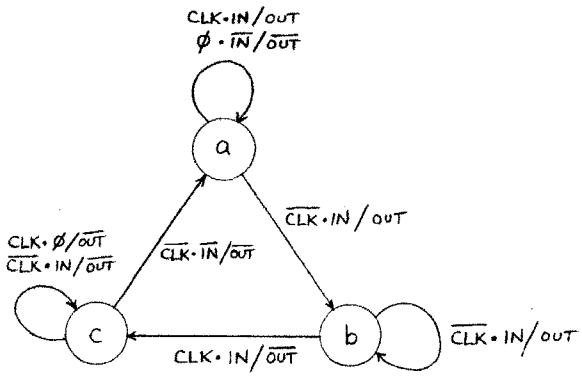
The 7493 4-bit counter counts when its control inputs (R1 and R2) are held low and asynchronously resets the count outputs to zero when the control inputs are high. This feature was used in the design to enable the START time delay to be interrupted upon a player cheating. The counter would then immediately (i.e. asynchronously) reset to state 'a' and the WINNER routine could be performed.

4.5 - Input Conditioner

The Input Conditioner was needed due to the asynchronous nature of the signals from the player pushbuttons. The pulses from the pushbuttons needed to be captured and synchronized with the system clock so that they could be serviced by the Game Controller. It was also desired that the pulses be turned into one-shots in order to prevent a player from being lazy and simply hold his button down. These conditions made it a logical choice to implement the design as an asynchronous sequential circuit that had the system clock signal as one of its inputs. Some cross-coupled NAND's and pull-up resistors were also employed to debounce the pushbuttons. The timing and state diagrams are shown in Fig. 4.8 and the circuit schematic in Fig. 4.9.



TIMING DIAGRAM



STATE DIAGRAM

Fig. 4.8 - Timing and state diagrams for original Input Conditioner circuit

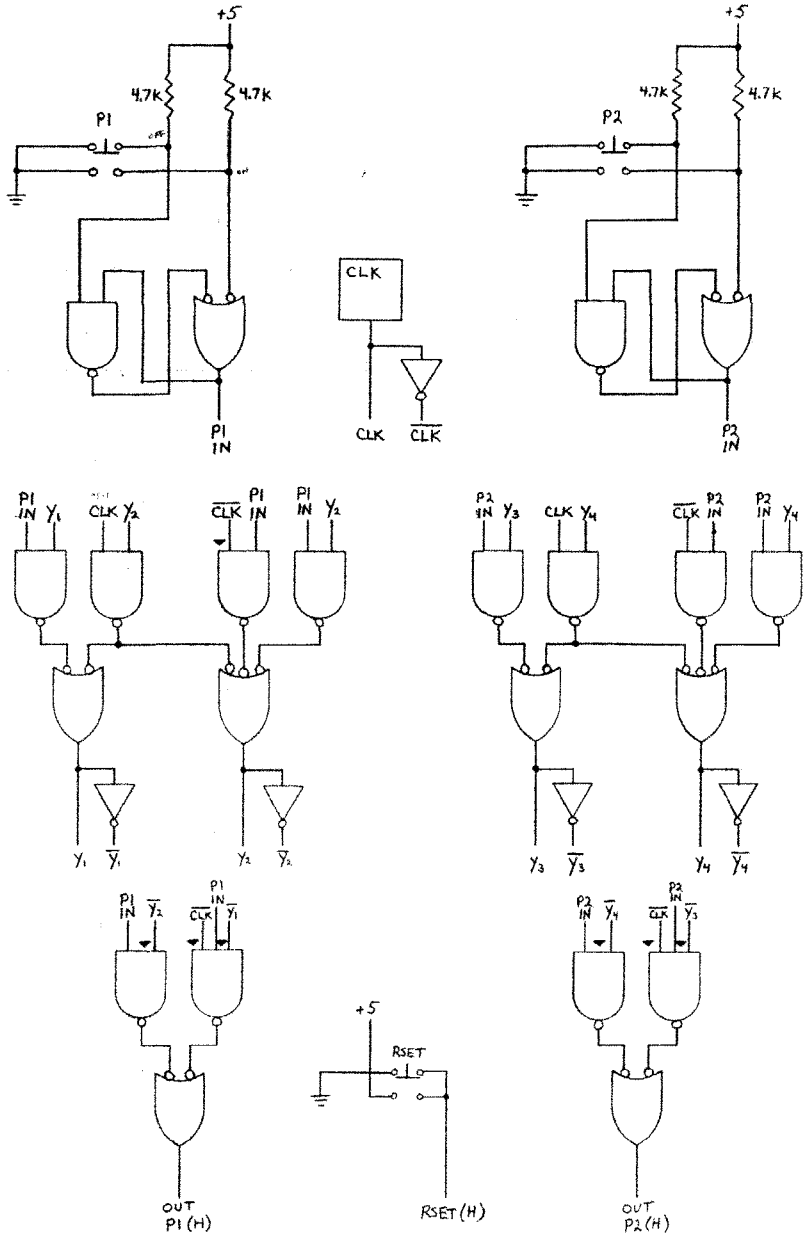


Fig. 4.9 - Schematic of original Input Conditioner circuit

CHAPTER V

ABEL EXPERIMENTAL DESIGN IMPLEMENTATION

5.1 - Design Concept

The main objective for the ABEL implementation of the Tug of War game circuit was to maintain the functionality of the original circuit while only using one PLD for each of the four system components (i.e. Game Controller, Output Controller, Timing Controller, and Input Conditioner). Two deviations from this plan were made, however. First, it was necessary to make some changes in the signals between the Game Controller and the Output Controller. This was due to device limitations and will be discussed in more detail. Secondly, the decision was made to separate the debouncing circuitry from the Input Conditioner and the LED output inverters from the Output Controller. The reasoning for this was to isolate the logic portions (i.e. the PLD's) from the input/output circuits and protect them from any possible harm. Quad NAND and hex inverter chips are cheap and plentiful and can be replaced easily if damaged. PLD's are more expensive and generally are not stocked at consumer electronics stores, necessitating ordering from catalogs.

ABEL 4, being DOS-based, only supports up to eight character length filenames. Therefore, the filenames (and module names) for the Game Controller, Output Controller, Timing Controller, and Input Conditioner became GAMECON, OUTCON, TIMECON, and INPUTCON, respectively.

Macrocell-type PAL devices were used for all four of the system components. This resulted in great flexibility in terms of the availability of equivalent device architectures because of the configurable nature of the macrocells. In particular GAL

devices from Lattice Semiconductor Corporation were used. These are CMOS devices and are electrically erasable and reprogrammable. This feature reduced the number of chips needed for experimentation since they could be reused if the logic programmed into them did not function as intended. Manufacturer data sheets for these devices are included in Appendix B.

5.2 - GAMECON

The GAMECON module (see Fig. 5.1) was programmed as a Moore-type sequential circuit just as the original Game Controller had been designed. The state assignments and state transitions remain true to the original design specifications. Three separate modifications were made to the input/output signals, though. The LED1 and LED2 outputs from the original circuit were combined into one named LED. LED1 and LED2 are identical logically and were only separated in the first place because of fanout concerns. These concerns don't exist inside the PLD. Secondly, the WIN input to the original circuit was eliminated and both W1 and W2 from the Output Controller were brought in instead. Therefore, the win status determination would be made in the logic of the GAMECON module rather than in the OUTCON module. This was done because of resource restrictions in the device used for the OUTCON design that will be explained later. Finally, an output named CLOCK was created. This signal consists only of an inversion of the CLK input. It was also necessary for use in the OUTCON design for reasons that will be explained later.

There are a number of items in the GAMECON file that need to be introduced. First, in the Declarations section, the reader will notice the use of the 'istype' statement. This statement is used to tell ABEL what type of attribute should be assigned to a given output. For example, S1 is defined to have the output attribute of a 'buffer' (i.e. positive logic). Register output attributes were assigned to y1 through y4. If this had not been

```
Module       gamecon
Title        'Game Controller, Tug-o-war Game'

Declarations

        gamecon device 'p26cv12';

        CLK                 pin 1;
        P1                  pin 2;
        P2                  pin 3;
        RSET                pin 4;
        OK                  pin 8;
        W1                  pin 9;
        W2                  pin 10;
        S1                  pin 27 istype 'buffer';
        S0                  pin 26 istype 'buffer';
        WINNER              pin 17 istype 'buffer';
        START               pin 18 istype 'buffer';
        LR                  pin 25 istype 'buffer';
        LED                 pin 24 istype 'buffer';
        y1,y2,y3,y4         pin 23,22,20,19 istype 'reg';
        CLOCK               pin 16 istype 'buffer';

        Stval = [y1,y2,y3,y4];
        a = [0,0,0,0];
        b = [0,1,0,0];
        c = [0,1,0,1];
        d = [1,1,0,1];
        e = [0,1,1,1];
        f = [1,1,1,1];
        g = [1,1,1,0];
        h = [1,1,0,0];
        i = [0,1,1,0];

Equations

        Stval.clk = CLK;
        CLOCK = !CLK;

State_Diagram  Stval

    state a:  S1 = 1;
              S0 = 1;
              LED = 1;
              If RSET then b else a;

    state b:  S1 = 1;
              S0 = 1;
              LED = 0;
              START = 1;
              If (P1 & !P2) then h
              Else
              If (!P1 & P2) then i
              Else
              If OK then c
              Else b;

    state c:  S1 = 0;
              S0 = 0;
              If (P1 & !P2) then d
              Else
              If (!P1 & P2) then e
              Else
              If RSET then b
              Else c;
```

Fig. 5.1 - GAMECON source file

```
    state d:  S1 = 1;
              S0 = 0;
              LR = 0;
              If W1 then f else c;

    state e:  S1 = 0;
              S0 = 1;
              LR = 0;
              If W2 then f else c;

    state f:  S1 = 0;
              S0 = 0;
              WINNER = 1;
              If OK then g else f;

    state g:  S1 = 0;
              S0 = 0;
              If RSET then b else g;

    state h:  S1 = 1;
              S0 = 0;
              LR = 1;
              Goto f;

    state i:  S1 = 0;
              S0 = 1;
              LR = 1;
              Goto f;

Test_vectors  'Test !CLOCK'

      ( [ CLK ] -> [ CLOCK ] )
        [  0  ] -> [   1   ];
        [  1  ] -> [   0   ];

Test_vectors  'power-up and reset/start routine'

      ( [CLK, P1,P2,RSET,OK ] -> [ Stval,S1,S0,LED,START ] )
        [.c.,.x.,.x., 0 ,.x.] -> [    a , 1, 1,  1 ,  0  ];
        [.c.,.x.,.x., 0 ,.x.] -> [    a , 1, 1,  1 ,  0  ];
        [.c.,.x.,.x., 1 ,.x.] -> [    b , 1, 1,  0 ,  1  ];
        [.c., 0 , 0 ,.x., 0 ] -> [    b , 1, 1,  0 ,  1  ];
        [.c., 0 , 0 ,.x., 1 ] -> [    c , 0, 0, .x.,  0  ];

Test_vectors  'Player 1 cheat routine'

      ( [ CLK, P1,P2,RSET,OK ] -> [ Stval,S1,S0,LR,WINNER ] )
        [ .c., 0 , 0,  1 ,.x.] -> [    b , 1, 1,.x.,  0   ];
        [ .c., 1 , 0, .x.,.x.] -> [    h , 1, 0, 1 ,  0   ];
        [ .c.,.x.,.x.,.x.,.x.] -> [    f , 0, 0,.x.,  1   ];
        [ .c.,.x.,.x.,.x., 0 ] -> [    f , 0, 0,.x.,  1   ];
        [ .c.,.x.,.x.,.x., 1 ] -> [    g , 0, 0,.x.,  0   ];
        [ .c.,.x.,.x., 0 ,.x.] -> [    g,  0, 0,.x.,  0   ];
        [ .c.,.x.,.x., 1 ,.x.] -> [    b,  1, 1,.x.,  0   ];

Test_vectors  'Player 2 cheat routine'

      ( [ CLK, P1,P2,RSET,OK ] -> [ Stval,S1,S0,LR,WINNER ] )
        [ .c., 0 , 0, .x., 0 ] -> [    b , 1, 1,.x.,  0   ];
        [ .c., 0 , 1, .x.,.x.] -> [    i , 0, 1, 1 ,  0   ];
        [ .c.,.x.,.x.,.x.,.x.] -> [    f , 0, 0,.x.,  1   ];
        [ .c.,.x.,.x.,.x., 1 ] -> [    g , 0, 0,.x.,  0   ];
        [ .c.,.x.,.x., 1 ,.x.] -> [    b , 1, 1,.x.,  0   ];
```

Fig. 5.1 (cont'd.) - GAMECON source file

```
Test_vectors  'Normal competition'

      ( [ CLK, P1,P2,RSET, OK, W1,W2 ] -> [ Stval,S1,S0,LR ] )
        [ .c., 0 , 0, .x., 1 ,.x.,.x.] -> [   c  , 0, 0,.x.];
        [ .c., 0 , 0,  0 ,.x.,.x.,.x.] -> [   c  , 0, 0,.x.];
        [ .c., 0 , 0,  1 ,.x.,.x.,.x.] -> [   b  , 1, 1,.x.];
        [ .c., 0 , 0, .x., 1 ,.x.,.x.] -> [   c  , 0, 0,.x.];
        [ .c., 1 , 1,  0 ,.x.,.x.,.x.] -> [   c  , 0, 0,.x.];
        [ .c., 1 , 0, .x.,.x.,.x.,.x.] -> [   d  , 1, 0, 0 ];
        [ .c.,.x.,.x.,.x.,.x., 0 , 0 ] -> [   c  , 0, 0,.x.];
        [ .c., 0 , 1, .x.,.x.,.x.,.x.] -> [   e  , 0, 1, 0 ];
        [ .c.,.x.,.x.,.x.,.x., 0 , 0 ] -> [   c  , 0, 0,.x.];
        [ .c., 1 , 0 ,.x.,.x.,.x.,.x.] -> [   d  , 1, 0, 0 ];
        [ .c.,.x.,.x.,.x.,.x., 1 , 0 ] -> [   f  , 0, 0,.x.];
        [ .c.,.x.,.x.,.x., 1 ,.x.,.x.] -> [   g  , 0, 0,.x.];
        [ .c.,.x.,.x., 1 ,.x.,.x.,.x.] -> [   b  , 1, 1,.x.];
        [ .c., 0 , 0 ,.x., 1 ,.x.,.x.] -> [   c  , 0, 0,.x.];
        [ .c., 0 , 1 ,.x.,.x.,.x.,.x.] -> [   e  , 0, 1, 0 ];
        [ .c.,.x.,.x.,.x.,.x., 0 , 1 ] -> [   f  , 0, 0,.x.];

end
```

Fig. 5.1 (cont'd.) - GAMECON source file

done, ABEL may have assumed y1 through y4 to be combinatorial outputs. Also, in the Declarations section, notice the use of a set named 'Stval' that includes all four of the state variables. Below that is a list of defined set constants 'a' through 'i' that cover all of the state values in the circuit. The set and set constants were useful in simplifying the writing of the state diagram section of the source file.

Secondly, in the Equations section, the reader will notice the use of a 'dot extension' in describing the clock source for the 'Stval' registers. Dot extensions are used to accurately describe register and feedback signals and clear up any potential ambiguities. For example, the registers used for 'Stval' have attributes such as register input (.D), register output (.Q), register clock input (.clk), asynchronous reset (.AR), synchronous preset (.SP), etc. Without specifying a dot extension for 'Stval' in an equation involving it, ABEL would be clueless in many cases as to which part of the registers it should be addressing with that equation.

Next, the reader is for the first time introduced to an example of a State_Diagram structure. Immediately following the State_Diagram keyword are the state variables for the state diagram. In this case, the set 'Stval' is used as a shortcut. The set constants that were defined previously are also used in the individual state definitions in

the state diagram. It's also interesting to note that the syntax for the branching logic is strikingly similar to BASIC or FORTRAN.

Lastly, the Test_Vectors section deserves some attention. When writing test vectors to simulate the actions of a state machine, it is necessary to write them such that the simulation begins in the power-up or initial state. The following vectors should then step the state machine through its states just as the normal operational flow will occur. It is not possible to skip states while doing this. At this time, also note the use of the special constants '.c.' (clocked input; low-high-low) and '.x.' (don't care) in the vectors.

ABEL generates reports at every step of the design process. These include simulation results, compiled equations, device resource allocations, etc. These are included for GAMECON and the other system component designs in Appendix A.

5.3 - OUTCON

In programming the OUTCON module (see Fig. 5.2), a different approach was taken. Since this was not a state machine, the state diagram method of description was of no use. In addition, a truth table description would be highly impractical. The only recourse was to use equations to describe the design logic.

A couple of set definitions were made in the Declarations section that are worth noting prior to discussing the equations. First, the two shift register control signals were put in a set named 'Select'. Second, all of the LED outputs were grouped into a set named 'OutLED'. The 'OutLED' set was created to simplify the equations that had to do with broadside load conditions since all of the LED outputs could be set properly in the same equation.

Also of note in the Declarations section is the CLOCK input signal. This signal is the inverted system clock pulse output from the GAMECON module. Referring back to the original Output Controller design, the shift registers needed to be clocked on the

```
Module       Outcon
Title        'Output Controller, Tug-o-war Game'

Declarations

        Outcon device 'P26CV12';

        CLOCK                                   pin 1;
        S1,S0                                   pin 2,3;
        LR                                      pin 4;
        LED                                     pin 5;
        W1,L1,L2,L3,L4,L5,L6,L7,L8,L9,W2    pin 27,26,25,24,23,22,20,19,18,17,16 istype
'buffer';

        Select = [S1,S0];
        OutLED = [W1,L1..L9,W2];

Equations

        OutLED.CLK = CLOCK;
        When (Select == 0) then OutLED := OutLED.fb;
        When (Select == 1) & (LR == 0) then W2 := L9.fb;
        When (Select == 1) & (LR == 0) then L9 := L8.fb;
        When (Select == 1) & (LR == 0) then L8 := L7.fb;
        When (Select == 1) & (LR == 0) then L7 := L6.fb;
        When (Select == 1) & (LR == 0) then L6 := L5.fb;
        When (Select == 1) & (LR == 0) then L5 := L4.fb;
        When (Select == 1) & (LR == 0) then L4 := L3.fb;
        When (Select == 1) & (LR == 0) then L3 := L2.fb;
        When (Select == 1) & (LR == 0) then L2 := L1.fb;
        When (Select == 1) & (LR == 0) then L1 := W1.fb;
        When (Select == 1) & (LR == 0) then W1 := 0;
        When (Select == 2) & (LR == 0) then W1 := L1.fb;
        When (Select == 2) & (LR == 0) then L1 := L2.fb;
        When (Select == 2) & (LR == 0) then L2 := L3.fb;
        When (Select == 2) & (LR == 0) then L3 := L4.fb;
        When (Select == 2) & (LR == 0) then L4 := L5.fb;
        When (Select == 2) & (LR == 0) then L5 := L6.fb;
        When (Select == 2) & (LR == 0) then L6 := L7.fb;
        When (Select == 2) & (LR == 0) then L7 := L8.fb;
        When (Select == 2) & (LR == 0) then L8 := L9.fb;
        When (Select == 2) & (LR == 0) then L9 := W2.fb;
        When (Select == 2) & (LR == 0) then W2 := 0;
        When (Select == 3) then OutLED := [LED,LED,LED,LED,LED,1,LED,LED,LED,LED,LED];
        When (Select == 1) & (LR == 1) then OutLED := [1,0,0,0,0,0,1,0,0,0,0];
        When (Select == 2) & (LR == 1) then OutLED := [0,0,0,0,1,0,0,0,0,0,1];

Test_vectors        'Broadside load and hold operation'

     ( [CLOCK,S1,S0, LR,LED] -> [ W1,L1,L2,L3,L4,L5,L6,L7,L8,L9,W2 ] )
       [ .C. , 1, 1,.X., 1 ] -> [  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ];
       [ .C. , 0, 0,.X., 1 ] -> [  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ];
       [ .C. , 1, 1,.X., 0 ] -> [  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ];
```

Fig. 5.2 - OUTCON source file

falling-edge of the system clock pulse in order to return the win status results to the Game Controller by the next rising-edge of the clock. Unfortunately, the GAL26CV12 (and most PLD's for that matter) does not allow signal inversion on the signal lines leading to the clock inputs on the registers. This necessitated feeding the clock input pin on the

```
Test_vectors        'Shift zeros right and then left'

     ( [CLOCK,S1,S0, LR,LED] -> [ W1,L1,L2,L3,L4,L5,L6,L7,L8,L9,W2 ] )
       [ .C. , 1, 1,.X., 0 ] -> [  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ];
       [ .C. , 0, 1, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 ];
       [ .C. , 0, 1, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 ];
       [ .C. , 0, 1, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 ];
       [ .C. , 0, 1, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 ];
       [ .C. , 0, 1, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ];
       [ .C. , 1, 0, 0 ,.X.] -> [  1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ];

Test_vectors          'Shift 1 left'

     ( [CLOCK,S1,S0, LR,LED] -> [ W1,L1,L2,L3,L4,L5,L6,L7,L8,L9,W2 ] )
       [ .C. , 1, 1,.X., 0 ] -> [  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ];
       [ .C. , 1, 0, 1 ,.X.] -> [  0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1 ];

Test_vectors          'Shift 1 right'

     ( [CLOCK,S1,S0, LR,LED] -> [ W1,L1,L2,L3,L4,L5,L6,L7,L8,L9,W2 ] )
       [ .C. , 1, 1,.X., 0 ] -> [  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ];
       [ .C. , 0, 1, 1 ,.X.] -> [  1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 ];

End
```

Fig. 5.2 (cont'd.) - OUTCON source file

26CV12 with the defined CLOCK output from the GAMECON device in order to simulate a falling-edge clock trigger.

The Equations section includes a rather large number of equations. Since ABEL doesn't have a mathematical or logical operator for shifting bits in a defined set such as 'OutLED', it was necessary to write an equation for each LED output in the set for both the left and right shift operations. This was a rather inelegant, brute-force method of accomplishing shift operations, but it worked. On a side note, the '.fb' dot extension probably was not necessary in the LED outputs. The use of the registered assignment operator in the equations probably should have indicated to ABEL that the LED signals involved in the equation logic were registered feedbacks (it never hurts to be prepared, though).

With regards to the previously mentioned elimination of the WIN output and subsequent usage of both W1 and W2 as signals back to GAMECON, this was not the intent in the first attempt at the OUTCON module design. WIN had been defined as an output of OUTCON and used the same logic as in the Output Controller design:

$$WIN = W1 \$ W2$$

However, while attempting to 'fit' the OUTCON module into the 26CV12 device, ABEL indicated that this logic could not be assigned due to the fact that the 26CV12 has no resources available for exclusive-OR functions. The logical alternative would have been to use the SOP equivalent of the equation:

$$WIN = (W1 \& !W2) \# (!W1 \& W2)$$

Unfortunately, adding this equation to the design logic resulted in a "too many feedbacks used" error message for the W1 and W2 outputs. This was due to feedback resource limitations in the 26CV12 device. In the end, the W1 and W2 outputs were each routed to the GAMECON module as inputs and both designs interacted fine.

5.4 - TIMECON

The design for the TIMECON module differs from the original Timing Controller design, yet the functionality remains the same. TIMECON is implemented as a Moore-type sequential design (see Fig. 5.3), whereas the Timing Controller was a Mealy-type design. The decision to use a Mealy implementation in the Timing Controller was forced by the fact that the 4-bit binary counter used as state variables could not be branched. States could only be sequenced in numerical order. This necessitated the

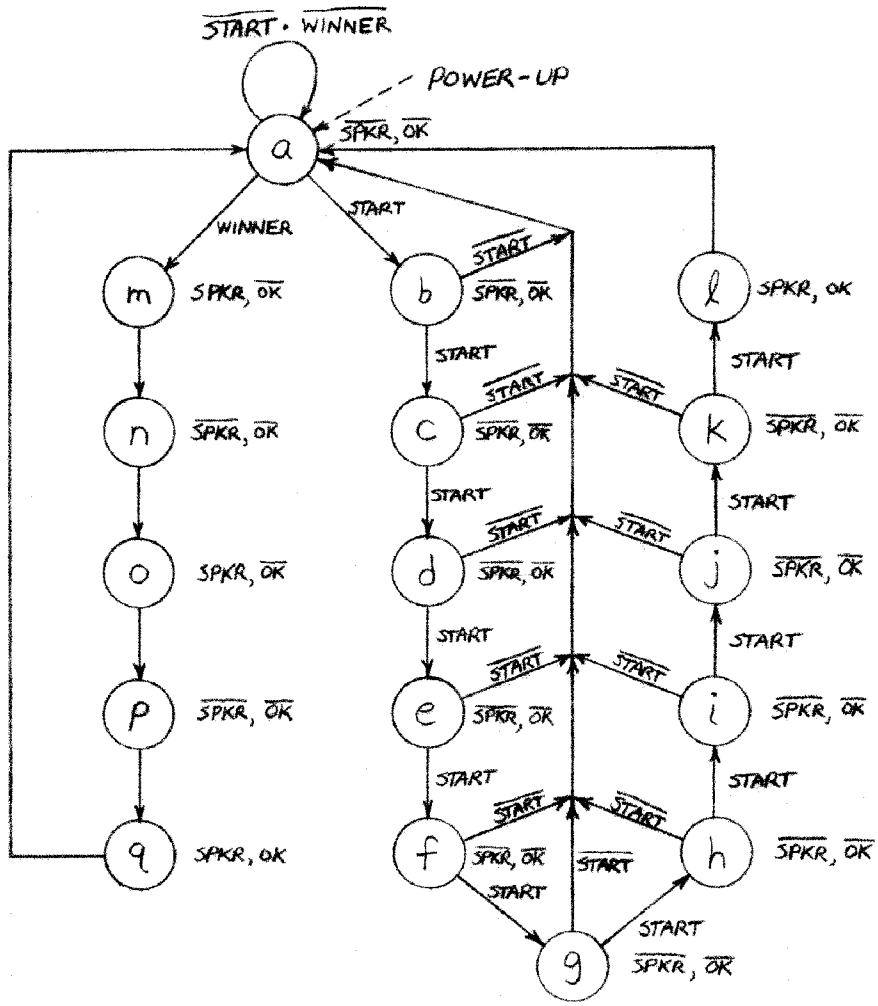


Fig. 5.3 - TIMECON state diagram

sharing of states for differing output conditions, thus a Mealy design was needed. This constraint does not exist in PLD's, so more flexibility was available in the design for TIMECON. Using a Moore design enabled the START and WINNER routines to be separated, thus eliminating the need for an asynchronous reset of the registers upon an interruption of the START time delay routine (i.e. when a player cheats) as was used in the Timing Controller design. In the TIMECON design, when an START routine interruption occurred, the next state transition was a synchronous one to state 'a'. This situation made it much easier to write test vectors that accurately described the behavior of the design. The ABEL source file for TIMECON is shown in Fig. 5.4.

```
Module     timecon
Title      'Timing Controller - Tug-o-War Game'

Declarations

        timecon device 'p22v10';

        CLK              pin 1;
        START            pin 2;
        WINNER           pin 3;
        y1,y2,y3,y4,y5   pin 21,20,19,18,17 istype 'reg';
        OK               pin 22 istype 'buffer';
        SPKR             pin 16 istype 'buffer';

        Stval = [y1,y2,y3,y4,y5];

        a = [0,0,0,0,0];
        b = [0,0,0,0,1];
        c = [0,0,0,1,1];
        d = [0,0,0,1,0];
        e = [0,0,1,1,0];
        f = [0,1,1,1,0];
        g = [1,1,1,1,0];
        h = [1,1,0,1,0];
        i = [0,1,0,1,0];
        j = [0,1,0,1,1];
        k = [0,1,0,0,1];
        l = [0,1,0,0,0];
        m = [0,0,1,0,0];
        n = [0,1,1,0,0];
        o = [1,1,1,0,0];
        p = [1,0,1,0,0];
        q = [1,0,0,0,0];

Equations

        Stval.clk = CLK;

State_Diagram      Stval

        state a:  SPKR = 0;
                  OK = 0;
                  If START then b
                  Else
                  If WINNER then m
                  Else a;

        state b:  SPKR = 0;
                  OK = 0;
                  If START then c
                  Else a;

        state c:  SPKR = 0;
                  OK = 0;
                  If START then d
                  Else a;

        state d:  SPKR = 0;
                  OK = 0;
                  If START then e
                  Else a;

        state e:  SPKR = 0;
                  OK = 0;
                  If START then f
                  Else a;
```

Fig. 5.4 - TIMECON source file

```
 state f:  SPKR = 0;
           OK = 0;
           If START then g
           Else a;

state g:   SPKR = 0;
           OK = 0;
           If START then h
           Else a;

state h:   SPKR = 0;
           OK = 0;
           If START then i
           Else a;

state i:   SPKR = 0;
           OK = 0;
           If START then j
           Else a;

state j:   SPKR = 0;
           OK = 0;
           If START then k
           Else a;

state k:   SPKR = 0;
           OK = 0;
           If START then l
           Else a;

state l:   SPKR = 1;
           OK = 1;
           Goto a;

state m:   SPKR = 1;
           OK = 0;
           Goto n;

state n:   SPKR = 0;
           OK = 0;
           Goto o;

state o:   SPKR = 1;
           OK = 0;
           Goto p;

state p:   SPKR = 0;
           OK = 0;
           Goto q;

state q:   SPKR = 1;
           OK = 1;
           Goto a;
```

Fig. 5.4 (cont'd.) - TIMECON source file

The source file is rather straightforward and follows the state diagram in Fig. 5.3. As in GAMECON, the state variables are grouped into a set named 'Stval' and set constants are defined for the individual state values. The test vectors are perhaps the most interesting feature of the TIMECON source file. The vectors for the 'Interruptions of start loop' are quite large in number. This was necessary due to the fact that the vectors

```
Test_Vectors   'power-up, normal start loop'

     ( [CLK,START,WINNER] -> [Stval,SPKR, OK] )
       [.c.,  0  ,   0  ] -> [  a  ,  0 , 0 ];
       [.c.,  0  ,   0  ] -> [  a  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  b  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  c  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  d  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  e  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  f  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  g  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  h  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  i  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  j  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  k  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  l  ,  1 , 1 ];
       [.c., .x. ,  .x. ] -> [  a  ,  0 , 0 ];

Test_Vectors   'Normal winner loop'

     ( [CLK,START,WINNER] -> [Stval,SPKR, OK] )
       [.c.,  0  ,   1  ] -> [  m  ,  1 , 0 ];
       [.c., .x. ,  .x. ] -> [  n  ,  0 , 0 ];
       [.c., .x. ,  .x. ] -> [  o  ,  1 , 0 ];
       [.c., .x. ,  .x. ] -> [  p  ,  0 , 0 ];
       [.c., .x. ,  .x. ] -> [  q  ,  1 , 1 ];
       [.c., .x. ,  .x. ] -> [  a  ,  0 , 0 ];

Test_Vectors   'Interruptions of start loop'

     ( [CLK,START,WINNER] -> [Stval,SPKR, OK] )
       [.c.,  1  ,   0  ] -> [  b  ,  0 , 0 ];
       [.c.,  0  ,  .x. ] -> [  a  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  b  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  c  ,  0 , 0 ];
       [.c.,  0  ,  .x. ] -> [  a  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  b  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  c  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  d  ,  0 , 0 ];
       [.c.,  0  ,  .x. ] -> [  a  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  b  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  c  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  d  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  e  ,  0 , 0 ];
       [.c.,  0  ,  .x. ] -> [  a  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  b  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  c  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  d  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  e  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  f  ,  0 , 0 ];
       [.c.,  0  ,  .x. ] -> [  a  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  b  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  c  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  d  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  e  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  f  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  g  ,  0 , 0 ];
       [.c.,  0  ,  .x. ] -> [  a  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  b  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  c  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  d  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  e  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  f  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  g  ,  0 , 0 ];
       [.c.,  1  ,   0  ] -> [  h  ,  0 , 0 ];
       [.c.,  0  ,  .x. ] -> [  a  ,  0 , 0 ];
```

Fig. 5.4 (cont'd.) - TIMECON source file

```
[.c., 1 , 0 ] -> [ b , 0 , 0 ];
[.c., 1 , 0 ] -> [ c , 0 , 0 ];
[.c., 1 , 0 ] -> [ d , 0 , 0 ];
[.c., 1 , 0 ] -> [ e , 0 , 0 ];
[.c., 1 , 0 ] -> [ f , 0 , 0 ];
[.c., 1 , 0 ] -> [ g , 0 , 0 ];
[.c., 1 , 0 ] -> [ h , 0 , 0 ];
[.c., 1 , 0 ] -> [ i , 0 , 0 ];
[.c., 0 , .x. ] -> [ a , 0 , 0 ];
[.c., 1 , 0 ] -> [ b , 0 , 0 ];
[.c., 1 , 0 ] -> [ c , 0 , 0 ];
[.c., 1 , 0 ] -> [ d , 0 , 0 ];
[.c., 1 , 0 ] -> [ e , 0 , 0 ];
[.c., 1 , 0 ] -> [ f , 0 , 0 ];
[.c., 1 , 0 ] -> [ g , 0 , 0 ];
[.c., 1 , 0 ] -> [ h , 0 , 0 ];
[.c., 1 , 0 ] -> [ i , 0 , 0 ];
[.c., 1 , 0 ] -> [ j , 0 , 0 ];
[.c., 0 , .x. ] -> [ a , 0 , 0 ];
[.c., 1 , 0 ] -> [ b , 0 , 0 ];
[.c., 1 , 0 ] -> [ c , 0 , 0 ];
[.c., 1 , 0 ] -> [ d , 0 , 0 ];
[.c., 1 , 0 ] -> [ e , 0 , 0 ];
[.c., 1 , 0 ] -> [ f , 0 , 0 ];
[.c., 1 , 0 ] -> [ g , 0 , 0 ];
[.c., 1 , 0 ] -> [ h , 0 , 0 ];
[.c., 1 , 0 ] -> [ i , 0 , 0 ];
[.c., 1 , 0 ] -> [ j , 0 , 0 ];
[.c., 1 , 0 ] -> [ k , 0 , 0 ];
[.c., 0 , .x. ] -> [ a , 0 , 0 ];
[.c., 1 , 0 ] -> [ b , 0 , 0 ];
[.c., 1 , 0 ] -> [ c , 0 , 0 ];
[.c., 1 , 0 ] -> [ d , 0 , 0 ];
[.c., 1 , 0 ] -> [ e , 0 , 0 ];
[.c., 1 , 0 ] -> [ f , 0 , 0 ];
[.c., 1 , 0 ] -> [ g , 0 , 0 ];
[.c., 1 , 0 ] -> [ h , 0 , 0 ];
[.c., 1 , 0 ] -> [ i , 0 , 0 ];
[.c., 1 , 0 ] -> [ j , 0 , 0 ];
[.c., 1 , 0 ] -> [ k , 0 , 0 ];
[.c., 1 , 0 ] -> [ l , 1 , 1 ];
[.c., .x. , .x. ] -> [ a , 0 , 0 ];

end
```

Fig. 5.4 (cont'd.) - TIMECON source file

must step through all of the states leading up to the target state. Since the START routine can possibly be interrupted in any state from 'b' to 'k', it was necessary to simulate a reset from each of these states. Every time an interruption occurred and the circuit reset to state 'a', it was necessary to begin stepping through all of the preceding states in order to reach the next state to be simulated as being interrupted. This is a prime example of how tedious writing test vectors can sometimes be. This set of vectors (nearly 100 in number) probably took longer to write than the logic portion of the file.

5.5 - INPUTCON

The design for the INPUTCON module followed exactly the design and functionality of the original Input Conditioner circuit. Not only were the inputs and outputs identical to the original circuit, but the equations used to describe the logic in the ABEL source file (see Fig. 5.5) were exactly the same as those that made up the asynchronous sequential logic in the original circuit.

```
module     inputcon
title      'Input Conditioner, Tug-o-war Game'

declarations

   inputcon device 'p16v8as';

   CLK      pin 1;
   P1_IN    pin 2;
   P2_IN    pin 3;
   Y1       pin 17;
   Y2       pin 16;
   Y3       pin 15;
   Y4       pin 14;
   P1_OUT   pin 19;
   P2_OUT   pin 18;

equations

   Y1 = (P1_IN & Y1.pin) # (CLK & Y2.pin);
   Y2 = (!CLK & P1_IN) # (P1_IN & Y2.pin) # (CLK & Y2.pin);
   P1_OUT = (P1_IN & !Y2.pin) # (!CLK & P1_IN & !Y1.pin);
   Y3 = (P2_IN & Y3.pin) # (CLK & Y4.pin);
   Y4 = (!CLK & P2_IN) # (P2_IN & Y4.pin) # (CLK & Y4.pin);
   P2_OUT = (P2_IN & !Y4.pin) # (!CLK & P2_IN & !Y3.pin);

test_vectors    'Pulse catcher for P1'

  ( [ CLK,P1_IN ] -> [ P1_OUT ] )
    [  0,   0   ] -> [ 0 ];
    [  0,   1   ] -> [ 1 ];
    [  1,   1   ] -> [ 0 ];
    [  1,   1   ] -> [ 0 ];
    [  0,   1   ] -> [ 0 ];
    [  0,   1   ] -> [ 0 ];
    [  1,   1   ] -> [ 0 ];
    [  1,   0   ] -> [ 0 ];
    [  0,   0   ] -> [ 0 ];
    [  0,   0   ] -> [ 0 ];
    [  1,   0   ] -> [ 0 ];
    [  1,   1   ] -> [ 1 ];
    [  0,   1   ] -> [ 1 ];
    [  0,   1   ] -> [ 1 ];
    [  1,   1   ] -> [ 0 ];
    [  1,   1   ] -> [ 0 ];
    [  0,   1   ] -> [ 0 ];
    [  0,   0   ] -> [ 0 ];
    [  1,   0   ] -> [ 0 ];

end
```

Fig. 5.5 - INPUTCON source file

Equations were used to describe the logic instead of a state diagram structure due to the asynchronous sequential behavior of the design. During the course of the author's research, no evidence was found that ABEL will allow a state diagram structure to describe an asynchronous design. It apparently assumes a registered design when a state diagram logic structure is used in a source file. In any event, combinatorial feedback equations were used to describe the sequential logic of the design, as is evident from the usage of the combinatorial output assignment operator '=' and the '.pin' feedback dot extension.

The reader will also notice that test vectors only exist for the player 1 (P1_OUT) equations. The equations that determine P2_OUT are identical to those for P1_OUT except for the substitution of P2_IN for P1_IN. Therefore, it was easier to simply change the test vector heading signals for P1 to P2 and re-simulate the vectors rather than typing an additional set of them.

5.6 - Additional Notes

Truth tables were considered for describing design logic at various points in the design process of each of the four modules. However, in each case, it was eventually decided that this was too inefficient a method. In the GAMECON and TIMECON modules synchronous sequential designs were used. Although these may be implemented in truth table form, the state diagram method of description is more intuitive since the sequential design process basically begins by drawing a state diagram. In the case of OUTCON, there were a large number of inputs involved (including feedback signals) and the truth table would soon have become unwieldy. Finally, in the case of INPUTCON, the equation method of logic description made so much sense in that the equations already existed from the original 7400-series design. It would have required more effort to investigate using a truth table implementation.

In addition, Mealy-type synchronous sequential design methods were not used in any of the four module designs. However, the state diagram structure in ABEL does support the Mealy form of state machine logic. The overall format of a Mealy state diagram in ABEL is similar to that of a Moore design with one exception. The fact that outputs are assigned during the state transition rather than upon arriving in a state (as in a Moore design) is accommodated with the 'with...endwith' statements. For example:

If START then c with [SPKR,OK] := [0,0]; endwith;

The 'with...endwith' statements ensure that the outputs are decoded at the proper time and do not lag by a clock cycle.

If the reader is interested in using truth tables or Mealy-type sequential logic in his design, he is referred again to the ABEL User Manual[3] and Practical Design Using Programmable Logic.[2] These references have numerous design examples of both techniques.

CHAPTER VI

CONCLUSION

6.1 - Project Results

The circuit schematic for the PLD implementation of the Tug of War game is shown in Fig. 6.1. Now, go back and compare this to Fig. 4.2. The reader will notice a striking similarity between the block diagram of the original 7400-series system design and the schematic of the PLD system design. Each system component (block) corresponds to a single PLD, as was the project intent. This graphically demonstrates one of the strongest advantages of using PLD's in a digital circuit design: the conceptual design in many cases may be directly translated into a hardware design.

Once assembled and tested, it was determined by the author that the entire system, with all of its components, was functioning properly. The author even went as far as to drag a couple of students from an adjacent laboratory to witness this fact. They played a few rounds of the game and were satisfied with its operations.

The final device count, including the Quad-NAND and Hex-Inverter chips, was 7. This is versus a total chip count of 31 in the original 7400-series design. That equates to a reduction factor of over 4 to 1. If the input debouncing and output buffering logic had been implemented in their respective PLD's, the total device count would stand at 4, for a reduction of nearly 8 to 1. This would probably be the effective limit for least chip count while using these types of devices (PAL's) in implementing this design. While the individual module designs could have been more complex and still fit into their respective PLD's (many product terms went unused on all 4 devices), only one module may be programmed into a particular device. Also, the small number of outputs on a given PLD

effectively limit the logic that can be assigned to a module and still fit into the device. Therefore, a 1 to 1 correspondence between each of the system components and PLD would appear to be the practical limit for device count limitation in this design.

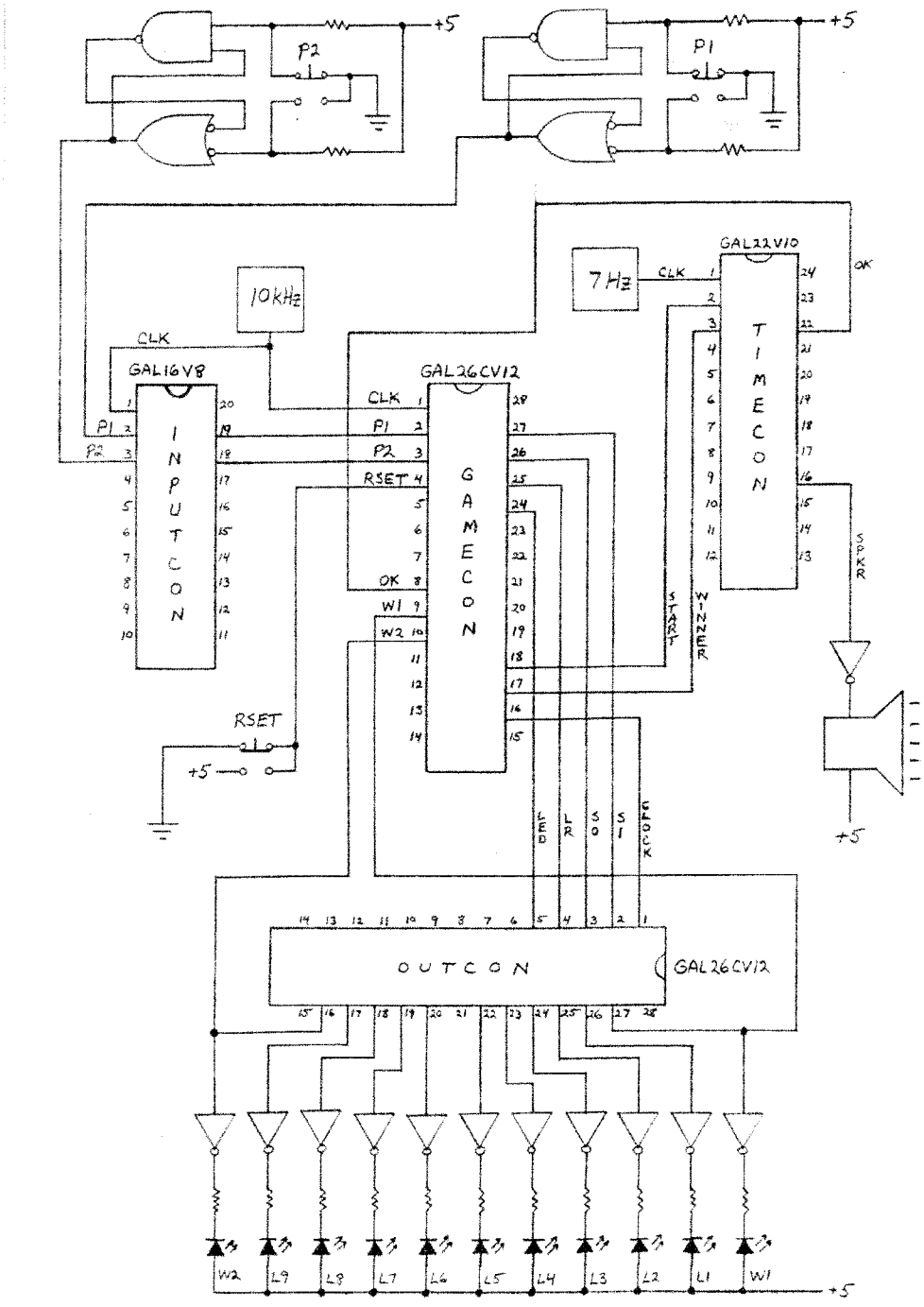


Fig. 6.1 - Schematic of PLD implementation of Tug of War

6.2 - Summary

Programmable Logic Devices are a valuable resource to the digital system designer. When they made their appearance in the 1970's, they finally enabled the designer to implement efficient and practical digital designs without the need to rely on the response time of a chip manufacturer in designing a mask-programmed device. Design and debug times, as well as physical circuit dimensions, were drastically reduced.

With the advent of the PC in the early to mid 1980's, universal PLD programming software became widely available. Sophisticated programming languages such as ABEL-HDL, along with commercially available universal programmer units, allowed the designer to design, program, and test PLD-based digital designs at his desk, enjoying previously unknown levels of flexibility in the design process.

Using ABEL-HDL, the digital designer is capable of turning design concepts such as Boolean equations, truth tables, and state diagrams directly into hardware implementations. A specific target device does not even need to be identified when the design process is begun. ABEL has the ability to take a generic design and determine which device architectures it will fit into. However, the designer may also use hardware-specific terms in the logic descriptions to tailor a design to a particular device, if so desired.

In order to demonstrate the concepts above, an example of the process of designing a digital system with PLD's was conducted by the author. Using ABEL design techniques, the logic of a previously constructed 7400-series digital circuit was implemented more efficiently using a smaller number of chips.

6.3 - Ideas for Future Research

There are a number of related topics that the author came upon during the course of this project that may be worthy of further study. First is the writing of test vectors. A properly written set of test vectors is crucial to the success of a PLD program. They must be written to accurately simulate all logical behavior in a design, including such things as asynchronous inputs. Often, this is a difficult task. Perhaps a more systematic approach to writing test vectors could be developed.

Also, the proper use of tools such as dot extensions and signal attributes could be investigated. To this day, despite all the work performed on this project, the author is still often confused as to when it is necessary to use these tools and when it is not. Most of the time, ABEL is capable of determining the proper signal attributes and device resource usage from the logic descriptions in the source file. However, this is not always the case. Learning when to use dot extensions and signal attributes and when not to could be an interesting area of study.

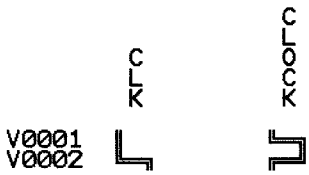
Finally, ABEL has FPGA programming capabilities (ABEL-FPGA). Also, the Data I/O Unisite programmer can program such devices with the proper adapter modules. Both of these currently exist at the YSU Electrical Engineering department. While the author chose not to research FPGA's, the reader is encouraged to look into them as possible target devices for digital designs.

APPENDIX A

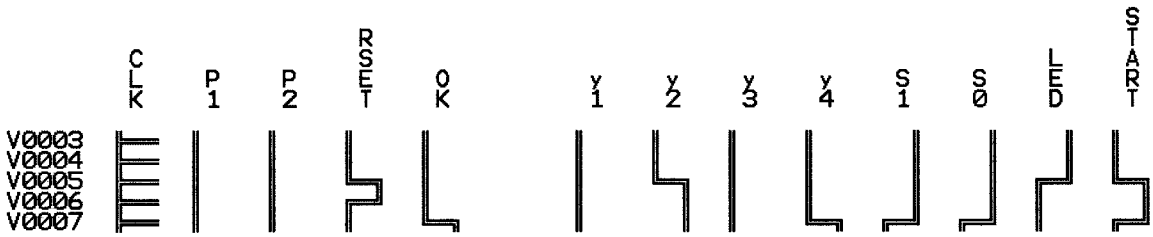
ABEL DESIGN PROCESS DOCUMENTATION

A.1 - GAMECON Documentation

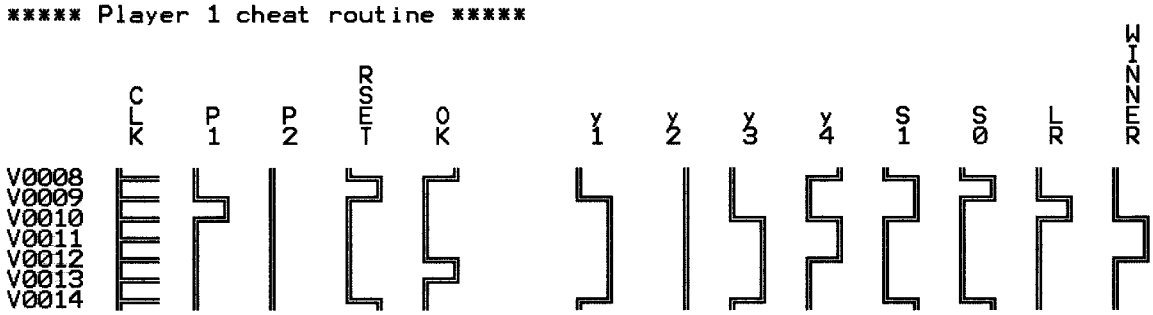
```
Simulate ABEL 4.03  Date Thu Jun  1 15:26:41 1995

Fuse file: 'gamecon.jed'  Vector file: 'gamecon.tmv'  Part: 'P26CV12'

ABEL 4.03 Data I/O Corp. JEDEC file for: P26CV12 V9.0
Created on: Thu Jun  1 15:26:35 1995

Game Controller, Tug-o-war Game


***** Test !CLOCK *****
```



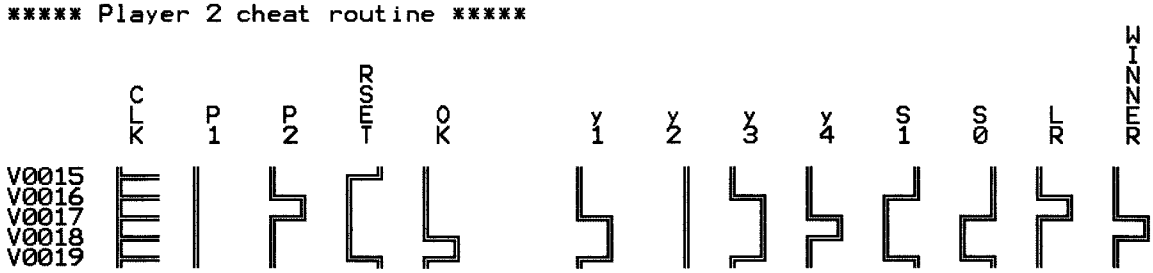

```
***** power-up and reset/start routine *****
```



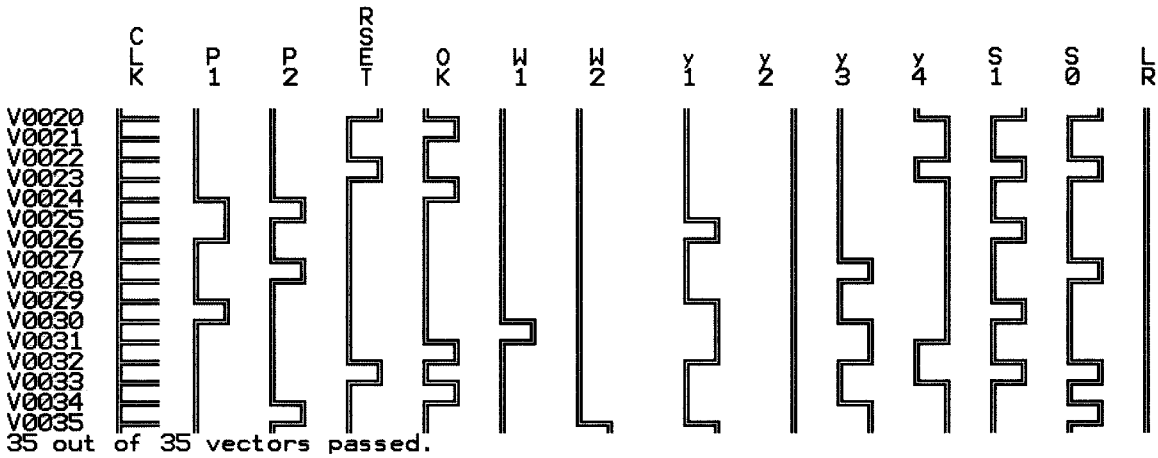

```
***** Player 1 cheat routine *****
```



***** Player 2 cheat routine *****



***** Normal competition *****

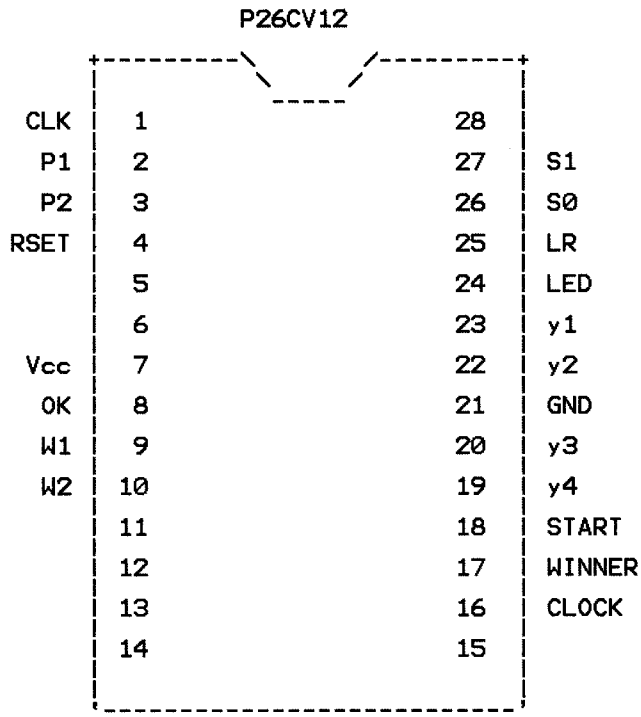


35 out of 35 vectors passed.

```
ABEL 4.03  -  Device Utilization Chart          Thu Jun  1 15:26:35 1995

Game Controller, Tug-o-war Game

    ==== P26CV12 Programmed Logic ====

y1.D    = (  !y1.FB & y2.FB & y3.FB & !y4.FB
          #    y1.FB & y2.FB & !y3.FB & !y4.FB
          #    !RSET & y1.FB & y2.FB & y3.FB
          #    y1.FB & y2.FB & y3.FB & y4.FB
          #    W2 & y2.FB & y3.FB & y4.FB
          #    W1 & y1.FB & y2.FB & y4.FB
          #    P1 & !P2 & !y1.FB & y2.FB & !y3.FB );  " ISTYPE 'BUFFER'
y1.C    = (  CLK );

y2.D    = (  y2.FB
          #    RSET & !y1.FB & !y3.FB & !y4.FB );  " ISTYPE 'BUFFER'
y2.C    = (  CLK );

y3.D    = (  !y1.FB & y2.FB & y3.FB & !y4.FB
          #    y1.FB & y2.FB & !y3.FB & !y4.FB
          #    !RSET & y1.FB & y2.FB & y3.FB
          #    y1.FB & y2.FB & y3.FB & y4.FB
          #    W2 & y2.FB & y3.FB & y4.FB
          #    W1 & y1.FB & y2.FB & y4.FB
          #    !P1 & P2 & !y1.FB & y2.FB & !y3.FB );  " ISTYPE 'BUFFER'
y3.C    = (  CLK );

y4.D    = (  y1.FB & y2.FB & !y3.FB
          #    !y1.FB & y2.FB & y3.FB
          #    !OK & y2.FB & y3.FB & y4.FB
          #    P2 & !RSET & y2.FB & !y3.FB & y4.FB
          #    !P1 & !RSET & y2.FB & !y3.FB & y4.FB
          #    !P1 & P2 & y2.FB & !y3.FB & y4.FB
          #    P1 & !P2 & y2.FB & !y3.FB & y4.FB
          #    P1 & P2 & OK & y2.FB & !y3.FB & !y4.FB
          #    !P1 & !P2 & OK & y2.FB & !y3.FB & !y4.FB );  " ISTYPE 'BUFFER'
y4.C    = (  CLK );

CLOCK     = (  !CLK );

S1      = (  y1.FB & y2.FB & !y3.FB
          #    !y1.FB & !y3.FB & !y4.FB );

S0      = (  !y1.FB & !y3.FB & !y4.FB
          #    !y1.FB & y2.FB & y3.FB );

LED     = (  !y1.FB & !y2.FB & !y3.FB & !y4.FB );

START     = (  !y1.FB & y2.FB & !y3.FB & !y4.FB );

LR      = (  !y1.FB & y2.FB & y3.FB & !y4.FB
          #    y1.FB & y2.FB & !y3.FB & !y4.FB );

WINNER    = (  y1.FB & y2.FB & y3.FB & y4.FB );
```

Game Controller, Tug-o-war Game

==== P26CV12 Chip Diagram ====



SIGNATURE: N/A

ABEL 4.03 - Device Utilization Chart Thu Jun 1 15:26:36 1995

Game Controller, Tug-o-war Game

==== P26CV12 Resource Allocations ====

Device Resources	Resource Available	Design Requirement	Part Utilization	Unused
Dedicated input pins	14	7	7	7 (50 %)
Combinatorial inputs	14	7	7	7 (50 %)
Registered inputs	-	0	-	-
Dedicated output pins	-	11	-	-
Bidirectional pins	12	0	11	1 (8 %)
Combinatorial outputs	-	7	-	-
Registered outputs	-	4	-	-
Reg/Com outputs	12	-	11	1 (8 %)
Two-input XOR	-	0	-	-
Buried nodes	-	0	-	-
Buried registers	-	0	-	-
Buried combinatorials	-	0	-	-

ABEL 4.03 - Device Utilization Chart Thu Jun 1 15:26:36 1995

Game Controller, Tug-o-war Game

==== P26CV12 Product Terms Distribution ====

Signal Name	Pin Assigned	Terms Used	Terms Max	Terms Unused
y1.REG	23	7	10	3
y2.REG	22	2	12	10
y3.REG	20	7	12	5
y4.REG	19	9	10	1
CLOCK	16	1	8	7
S1	27	2	8	6
S0	26	2	8	6
LED	24	1	8	7
START	18	1	8	7
LR	25	2	8	6
WINNER	17	1	8	7

==== List of Inputs/Feedbacks ====

Signal Name	Pin	Pin Type
CLK	1	CLK/IN
P1	2	INPUT
P2	3	INPUT
RSET	4	INPUT
OK	8	INPUT
W1	9	INPUT
W2	10	INPUT

ABEL 4.03 - Device Utilization Chart Thu Jun 1 15:26:36 1995

Game Controller, Tug-o-war Game

==== P26CV12 Unused Resources ====

Pin Number	Pin Type	Product Terms	Flip-flop Type
5	INPUT	-	-
6	INPUT	-	-
11	INPUT	-	-
12	INPUT	-	-
13	INPUT	-	-
14	INPUT	-	-
15	BIDIR	NORMAL 8	D
28	INPUT	-	-

ABEL 4.03 - Device Utilization Chart Thu Jun 1 15:26:36 1995

Game Controller, Tug-o-war Game

==== I/O Files ====

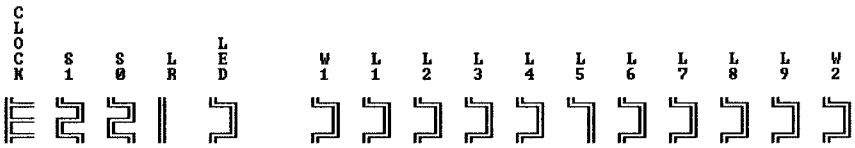
Module: 'gamecon'

Input files
===========
ABEL PLA file: gamecon.tt3
Vector file: gamecon.tmv
Device library: P26CV12.dev

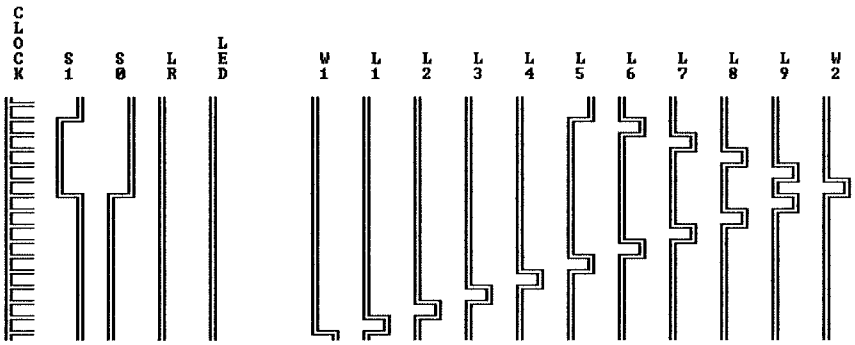
Output files
============
Report file: gamecon.doc
Programmer load file: gamecon.jed

A.2 - OUTCON Documentation

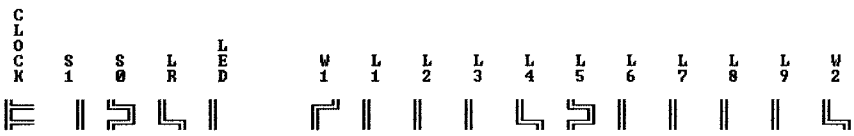
```
Simulate ABEL 4.03  Date Wed May 31 18:03:35 1995

Fuse file: 'outcon.jed'  Vector file: 'outcon.tmv'  Part: 'P26CV12'

ABEL 4.03 Data I/O Corp. JEDEC file for: P26CV12 V9.0
Created on: Wed May 31 18:03:30 1995

Output Controller, Tug-o-war Game


***** Broadside load and hold operation *****

```



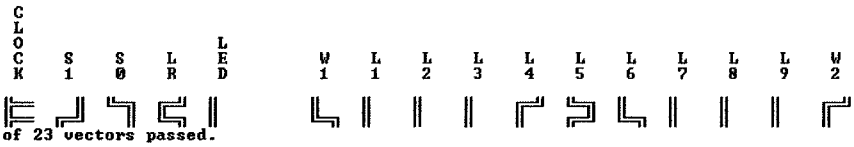

```
***** Shift zeros right and then left *****

```




```
***** Shift 1 left *****

```




```
***** Shift 1 right *****

```




```
23 out of 23 vectors passed.
```

ABEL 4.03 - Device Utilization Chart Wed May 31 18:03:30 1995

Output Controller, Tug-o-war Game

==== P26CV12 Programmed Logic ====

```
W1.D   = (  !S1 & S0 & LR
         #    S1 & S0 & LED
         #    S1 & !S0 & !LR & L1.FB
         #    !S1 & !S0 & W1.FB );  " ISTYPE 'BUFFER'
W1.C   = (  CLOCK );

L1.D   = (  S1 & S0 & LED
         #    S1 & !S0 & !LR & L2.FB
         #    !S1 & S0 & !LR & W1.FB
         #    !S1 & !S0 & L1.FB );  " ISTYPE 'BUFFER'
L1.C   = (  CLOCK );

L2.D   = (  S1 & S0 & LED
         #    S1 & !S0 & !LR & L3.FB
         #    !S1 & S0 & !LR & L1.FB
         #    !S1 & !S0 & L2.FB );  " ISTYPE 'BUFFER'
L2.C   = (  CLOCK );

L3.D   = (  S1 & S0 & LED
         #    S1 & !S0 & !LR & L4.FB
         #    !S1 & S0 & !LR & L2.FB
         #    !S1 & !S0 & L3.FB );  " ISTYPE 'BUFFER'
L3.C   = (  CLOCK );

L4.D   = (  S1 & S0 & LED
         #    S1 & !S0 & LR
         #    S1 & !S0 & L5.FB
         #    !S1 & S0 & !LR & L3.FB
         #    !S1 & !S0 & L4.FB );  " ISTYPE 'BUFFER'
L4.C   = (  CLOCK );

L5.D   = (  S1 & S0
         #    S1 & !LR & L6.FB
         #    S0 & !LR & L4.FB
         #    !S1 & !S0 & L5.FB );  " ISTYPE 'BUFFER'
L5.C   = (  CLOCK );

L6.D   = (  !S1 & S0 & LR
         #    S1 & S0 & LED
         #    S1 & !S0 & !LR & L7.FB
         #    !S1 & S0 & L5.FB
         #    !S1 & !S0 & L6.FB );  " ISTYPE 'BUFFER'
L6.C   = (  CLOCK );

L7.D   = (  S1 & S0 & LED
         #    S1 & !S0 & !LR & L8.FB
         #    !S1 & S0 & !LR & L6.FB
         #    !S1 & !S0 & L7.FB );  " ISTYPE 'BUFFER'
L7.C   = (  CLOCK );
```

ABEL 4.03 - Device Utilization Chart Wed May 31 18:03:30 1995

Output Controller, Tug-o-war Game

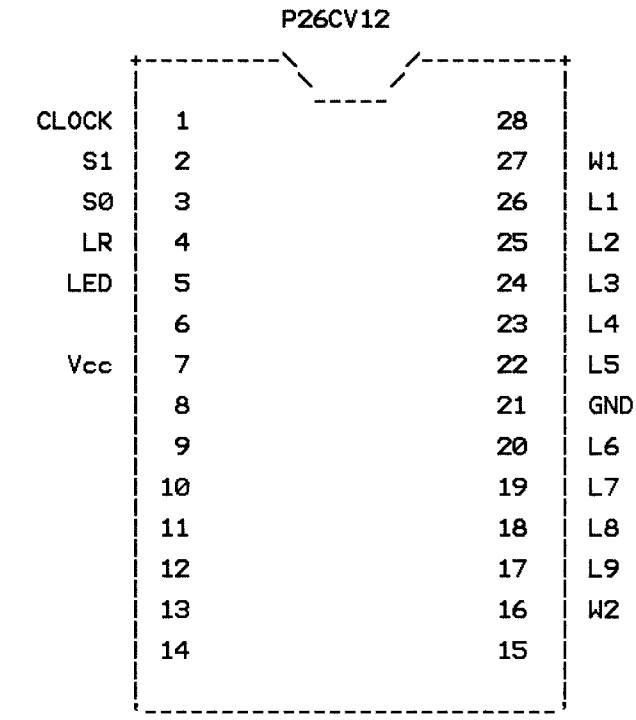
==== P26CV12 Programmed Logic ====

```
L8.D    = (  S1 & S0 & LED
         #   S1 & !S0 & !LR & L9.FB
         #   !S1 & S0 & !LR & L7.FB
         #   !S1 & !S0 & L8.FB );  " ISTYPE 'BUFFER'
L8.C    = (  CLOCK );

L9.D    = (  S1 & S0 & LED
         #   S1 & !S0 & !LR & W2.FB
         #   !S1 & S0 & !LR & L8.FB
         #   !S1 & !S0 & L9.FB );  " ISTYPE 'BUFFER'
L9.C    = (  CLOCK );

W2.D    = (  S1 & S0 & LED
         #   S1 & !S0 & LR
         #   !S1 & S0 & !LR & L9.FB
         #   !S1 & !S0 & W2.FB );  " ISTYPE 'BUFFER'
W2.C    = (  CLOCK );
```


ABEL 4.03 - Device Utilization Chart Wed May 31 18:03:30 1995

Output Controller, Tug-o-war Game

==== P26CV12 Chip Diagram ====



SIGNATURE: N/A

ABEL 4.03 - Device Utilization Chart Wed May 31 18:03:30 1995

Output Controller, Tug-o-war Game

==== P26CV12 Resource Allocations ====

Device Resources	Resource Available	Design Requirement	Part Utilization	Unused
Dedicated input pins	14	5	5	9 (64 %)
Combinatorial inputs	14	5	5	9 (64 %)
Registered inputs	-	0	-	-
Dedicated output pins	-	11	-	-
Bidirectional pins	12	0	11	1 (8 %)
Combinatorial outputs	-	0	-	-
Registered outputs	-	11	-	-
Reg/Com outputs	12	-	11	1 (8 %)
Two-input XOR	-	0	-	-
Buried nodes	-	0	-	-
Buried registers	-	0	-	-
Buried combinatorials	-	0	-	-

ABEL 4.03 - Device Utilization Chart Wed May 31 18:03:30 1995

Output Controller, Tug-o-war Game

==== P26CV12 Product Terms Distribution ====

Signal Name	Pin Assigned	Terms Used	Terms Max	Terms Unused
W1.REG	27	4	8	4
L1.REG	26	4	8	4
L2.REG	25	4	8	4
L3.REG	24	4	8	4
L4.REG	23	5	10	5
L5.REG	22	4	12	8
L6.REG	20	5	12	7
L7.REG	19	4	10	6
L8.REG	18	4	8	4
L9.REG	17	4	8	4
W2.REG	16	4	8	4

==== List of Inputs/Feedbacks ====

Signal Name	Pin	Pin Type
CLOCK	1	CLK/IN
S1	2	INPUT
S0	3	INPUT
LR	4	INPUT
LED	5	INPUT

ABEL 4.03 - Device Utilization Chart Wed May 31 18:03:30 1995

Output Controller, Tug-o-war Game

==== P26CV12 Unused Resources ====

Pin Number	Pin Type	Product Terms	Flip-flop Type
6	INPUT	-	-
8	INPUT	-	-
9	INPUT	-	-
10	INPUT	-	-
11	INPUT	-	-
12	INPUT	-	-
13	INPUT	-	-
14	INPUT	-	-
15	BIDIR	NORMAL 8	D
28	INPUT	-	-

ABEL 4.03 - Device Utilization Chart Wed May 31 18:03:30 1995

Output Controller, Tug-o-war Game

==== I/O Files ====

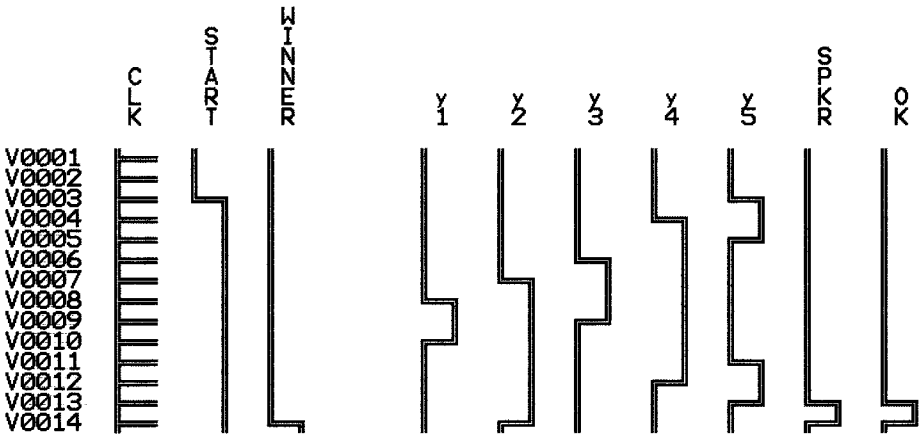
Module: 'outcon'

Input files
===========
ABEL PLA file: outcon.tt3
Vector file: outcon.tmv
Device library: P26CV12.dev

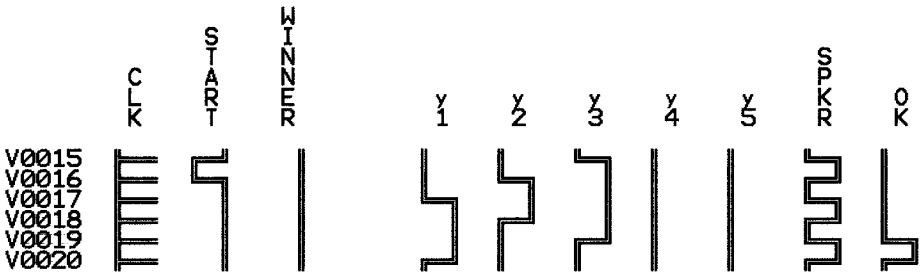
Output files
============
Report file: outcon.doc
Programmer load file: outcon.jed

A.3 - TIMECON Documentation

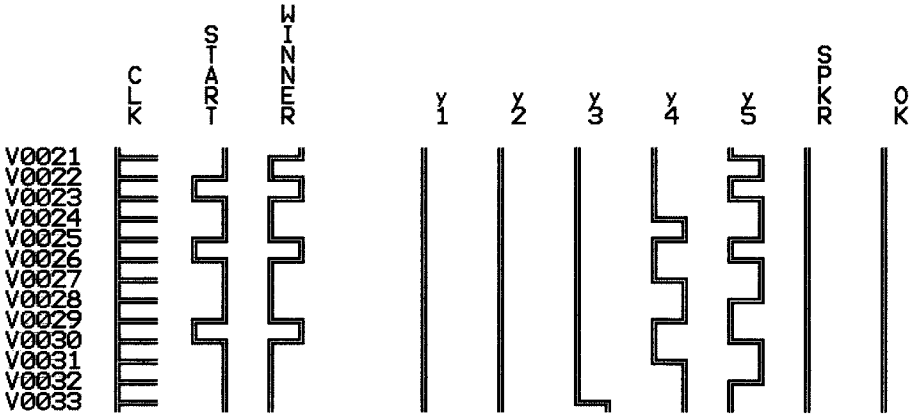
```
Simulate ABEL 4.10  Date Wed May  7 23:56:38 1997

Fuse file: 'timecon.jed'  Vector file: 'timecon.tmv'  Part: 'P22V10'

ABEL 4.10 Data I/O Corp. JEDEC file for: P22V10 V9.0
Created on: Wed May  7 23:56:25 1997

Timing Controller - Tug-o-War Game


***** power-up, normal start loop *****
```

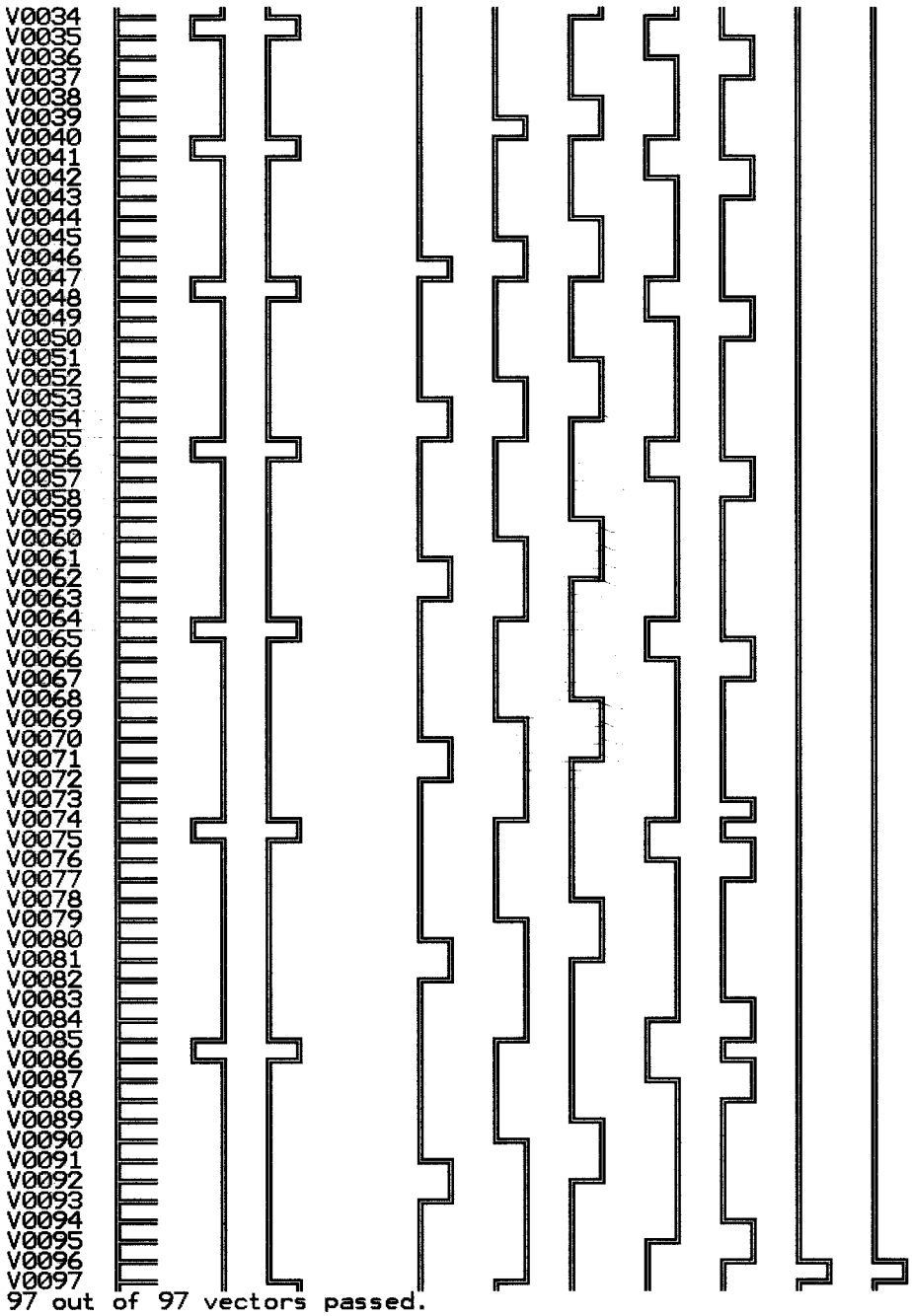



```
***** Normal winner loop *****
```




```
***** Interruptions of start loop *****
```

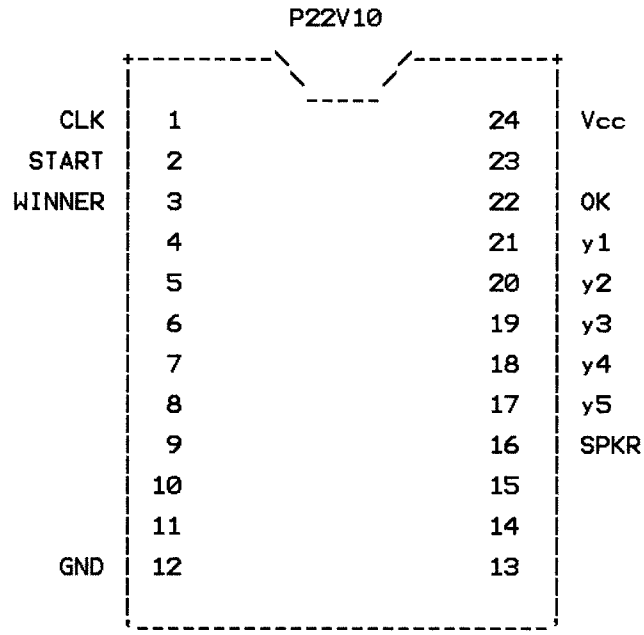



V0034
V0035
V0036
V0037
V0038
V0039
V0040
V0041
V0042
V0043
V0044
V0045
V0046
V0047
V0048
V0049
V0050
V0051
V0052
V0053
V0054
V0055
V0056
V0057
V0058
V0059
V0060
V0061
V0062
V0063
V0064
V0065
V0066
V0067
V0068
V0069
V0070
V0071
V0072
V0073
V0074
V0075
V0076
V0077
V0078
V0079
V0080
V0081
V0082
V0083
V0084
V0085
V0086
V0087
V0088
V0089
V0090
V0091
V0092
V0093
V0094
V0095
V0096
V0097
97 out of 97 vectors passed.

```
ABEL 4.10  -  Device Utilization Chart          Wed May  7 23:56:25 1997
Timing Controller - Tug-o-War Game
    ==== P22V10 Programmed Logic ====

y1.D    = (  y2.FB & y3.FB & !y4.FB & !y5.FB
          #    y1.FB & y3.FB & !y4.FB & !y5.FB
          #    START & y2.FB & y3.FB & !y5.FB ); " ISTYPE 'BUFFER'
y1.C    = (  CLK );

y2.D    = (  START & y2.FB & y4.FB & !y5.FB
          #    START & !y1.FB & y3.FB & !y5.FB
          #    !y1.FB & y3.FB & !y4.FB & !y5.FB
          #    START & !y1.FB & y2.FB & !y3.FB & y5.FB ); " ISTYPE 'BUFFER'
y2.C    = (  CLK );

y3.D    = (  !START & WINNER & !y1.FB & !y2.FB & !y4.FB & !y5.FB
          #    START & !y1.FB & !y2.FB & y4.FB & !y5.FB
          #    y2.FB & y3.FB & !y4.FB & !y5.FB
          #    START & !y1.FB & y3.FB & !y5.FB
          #    !y1.FB & y3.FB & !y4.FB & !y5.FB ); " ISTYPE 'BUFFER'
y3.C    = (  CLK );

y4.D    = (  START & !y1.FB & !y2.FB & !y3.FB & y5.FB
          #    START & y2.FB & y4.FB & !y5.FB
          #    START & !y1.FB & y4.FB & !y5.FB ); " ISTYPE 'BUFFER'
y4.C    = (  CLK );

y5.D    = (  START & !y1.FB & !y2.FB & !y3.FB & !y4.FB
          #    START & !y1.FB & y2.FB & !y3.FB & y4.FB ); " ISTYPE 'BUFFER'
y5.C    = (  CLK );

SPKR      = (  y1.FB & !y2.FB & !y3.FB & !y4.FB & !y5.FB
           #    !y1.FB & y2.FB & !y3.FB & !y4.FB & !y5.FB
           #    !y1.FB & !y2.FB & y3.FB & !y4.FB & !y5.FB
           #    y1.FB & y2.FB & y3.FB & !y4.FB & !y5.FB );

OK      = (  y1.FB & !y2.FB & !y3.FB & !y4.FB & !y5.FB
          #    !y1.FB & y2.FB & !y3.FB & !y4.FB & !y5.FB );
```

Timing Controller - Tug-o-War Game

==== P22V10 Chip Diagram ====



SIGNATURE: N/A

ABEL 4.10 - Device Utilization Chart Wed May 7 23:56:25 1997

Timing Controller - Tug-o-War Game

==== P22V10 Resource Allocations ====

Device Resources	Resource Available	Design Requirement	Part Utilization	Unused
Dedicated input pins	12	3	3	9 (75 %)
Combinatorial inputs	12	3	3	9 (75 %)
Registered inputs	-	0	-	-
Dedicated output pins	-	7	-	-
Bidirectional pins	10	0	7	3 (30 %)
Combinatorial outputs	-	2	-	-
Registered outputs	-	5	-	-
Reg/Com outputs	10	-	7	3 (30 %)
Two-input XOR	-	0	-	-
Buried nodes	-	0	-	-
Buried registers	-	0	-	-
Buried combinatorials	-	0	-	-

ABEL 4.10 - Device Utilization Chart Wed May 7 23:56:25 1997

Timing Controller - Tug-o-War Game

==== P22V10 Product Terms Distribution ====

Signal Name	Pin Assigned	Terms Used	Terms Max	Terms Unused
y1.REG	21	3	12	9
y2.REG	20	4	14	10
y3.REG	19	5	16	11
y4.REG	18	3	16	13
y5.REG	17	2	14	12
SPKR	16	4	12	8
OK	22	2	10	8

==== List of Inputs/Feedbacks ====

Signal Name	Pin	Pin Type
CLK	1	CLK/IN
START	2	INPUT
WINNER	3	INPUT

ABEL 4.10 - Device Utilization Chart Wed May 7 23:56:26 1997

Timing Controller - Tug-o-War Game

==== P22V10 Unused Resources ====

Pin Number	Pin Type	Product Terms	Flip-flop Type
4	INPUT	-	-
5	INPUT	-	-
6	INPUT	-	-
7	INPUT	-	-
8	INPUT	-	-
9	INPUT	-	-
10	INPUT	-	-
11	INPUT	-	-
13	INPUT	-	-
14	BIDIR	NORMAL 8	D
15	BIDIR	NORMAL 10	D
23	BIDIR	NORMAL 8	D

ABEL 4.10 - Device Utilization Chart Wed May 7 23:56:26 1997

Timing Controller - Tug-o-War Game

==== I/O Files ====

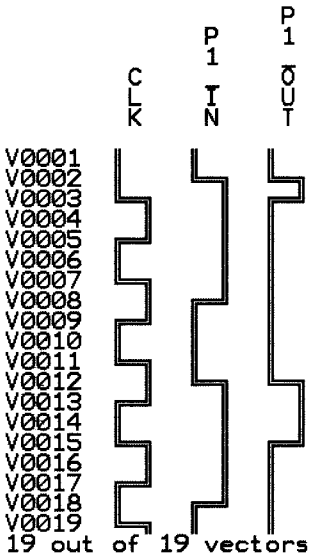
Module: 'timecon'

Input files
===========
ABEL PLA file: timecon.tt3
Vector file: timecon.tmv
Device library: P22V10.dev

Output files
============
Report file: timecon.doc
Programmer load file: timecon.jed

A.4 - INPUTCON Documentation

```
Simulate ABEL 4.03  Date Thu Jun  1 16:38:35 1995

Fuse file: 'inputcon.jed'  Vector file: 'inputcon.jed'  Part: 'P16V8C'

ABEL 4.03 Data I/O Corp. JEDEC file for: P16V8C V9.0
Created on: Thu Jun  1 16:38:15 1995

Input Conditioner, Tug-o-war Game
```




```
19 out of 19 vectors passed.
```

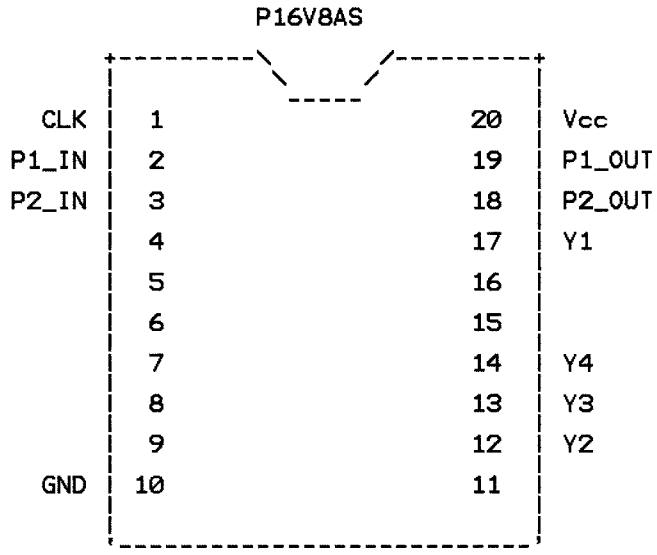


```
ABEL 4.10  -  Device Utilization Chart         Tue Mar 18 18:39:26 1997

Input Conditioner, Tug-o-war Game

    ==== P16V8AS Programmed Logic ====

Y1      = (  CLK & Y2.PIN
         #   P1_IN & Y1.PIN );

Y2      = !(  !CLK & !P1_IN
         #   CLK & !Y2.PIN );

P1_OUT      = (  !CLK & P1_IN & !Y1.PIN
             #   P1_IN & !Y2.PIN );

Y3      = (  CLK & Y4.PIN
         #   P2_IN & Y3.PIN );

Y4      = !(  !CLK & !P2_IN
         #   CLK & !Y4.PIN );

P2_OUT      = (  !CLK & P2_IN & !Y3.PIN
             #   P2_IN & !Y4.PIN );
```

ABEL 4.10 - Device Utilization Chart Tue Mar 18 18:39:27 1997

Input Conditioner, Tug-o-war Game

==== P16V8AS Chip Diagram ====



SIGNATURE: N/A

ABEL 4.10 - Device Utilization Chart Tue Mar 18 18:39:27 1997

Input Conditioner, Tug-o-war Game

==== P16V8AS Resource Allocations ====

Device Resources	Resource Available	Design Requirement	Part Utilization	Unused
Dedicated input pins	10	3	3	7 (70 %)
Combinatorial inputs	10	3	3	7 (70 %)
Registered inputs	-	0	-	-
Dedicated output pins	2	2	0	2 (100 %)
Bidirectional pins	6	4	6	0 (0 %)
Combinatorial outputs	8	6	6	2 (25 %)
Registered outputs	-	0	-	-
Two-input XOR	-	0	-	-
Buried nodes	-	0	-	-
Buried registers	-	0	-	-
Buried combinatorials	-	0	-	-

ABEL 4.10 - Device Utilization Chart Tue Mar 18 18:39:27 1997

Input Conditioner, Tug-o-war Game

==== P16V8AS Product Terms Distribution ====

Signal Name	Pin Assigned	Terms Used	Terms Max	Terms Unused
Y1	17	2	8	6
Y2	12	2	8	6
P1_OUT	19	2	8	6
Y3	13	2	8	6
Y4	14	2	8	6
P2_OUT	18	2	8	6

==== List of Inputs/Feedbacks ====

Signal Name	Pin	Pin Type
CLK	1	INPUT
P1_IN	2	INPUT
P2_IN	3	INPUT
Y1.PIN	17	COMB FB
Y2.PIN	12	COMB FB
Y3.PIN	13	COMB FB
Y4.PIN	14	COMB FB

ABEL 4.10 - Device Utilization Chart Tue Mar 18 18:39:27 1997

Input Conditioner, Tug-o-war Game

==== P16V8AS Unused Resources ====

Pin Number	Pin Type	Product Terms	Flip-flop Type
4	INPUT	-	-
5	INPUT	-	-
6	INPUT	-	-
7	INPUT	-	-
8	INPUT	-	-
9	INPUT	-	-
11	INPUT	-	-
15	OUTPUT	NORMAL 8	-
16	OUTPUT	NORMAL 8	-

ABEL 4.10 - Device Utilization Chart Tue Mar 18 18:39:27 1997

Input Conditioner, Tug-o-war Game

==== I/O Files ====

Module: 'inputcon'

Input files
===========
ABEL PLA file: inputcon.tt3
Vector file: inputcon.tmv
Device library: P16V8AS.dev

Output files
============
Report file: inputcon.doc
Programmer load file: inputcon.jed

APPENDIX B

MANUFACTURER DATA SHEET EXCERPTS

(Used with permission from Lattice Semiconductor Corp. - see below)

Date: Tue, 3 Jun 97 09:34:37 PDT
To: "Rick Alcorn" <ralcorn@cisnet.com> (by way of Travis Illig <tillig@latticesemi.com>)
From: steve stark <steve_stark@latticesemi.com>
Subject: Re:

You have our permission to use these pages. Would you be willing to send me
a copy of your thesis when completed?

Regards,

Steve Stark

At 08:20 AM 6/3/97 -0600, you wrote:
>Dear Sir or Madam:
>
>I downloaded Data Sheets (1996 Data Book) for the GAL16V8, GAL22V10, and
>GAL26CV12 from your web site during the course of my Master's Thesis work
>at Youngstown State University. I would like to include some of these Data
>Sheet pages in my Thesis paper. The pages that I wish to include are:
>
>GAL16V8: 1,3,4,5,6,7,8,9,19
>GAL22V10: 1,3,4,5,13
>GAL26CV12: 1,3,4,5,13
>
>May I have permission to include these in my paper?
>
>Rick Alcorn
>ralcorn@cisnet.com
>

****
* Steve Stark *
* Lattice Semiconductor Corporation *
* 5555 N.E. Moore Ct. *
* Hillsboro, OR 97124-6421 *
* 503-693-0279 (Ph.) *
* 503-681-3037 (Fax) *
* steve_stark@latticesemi.com *
****

B.1 - Lattice GAL16V8 Data Sheet Excerpts

Lattice Semiconductor Corporation

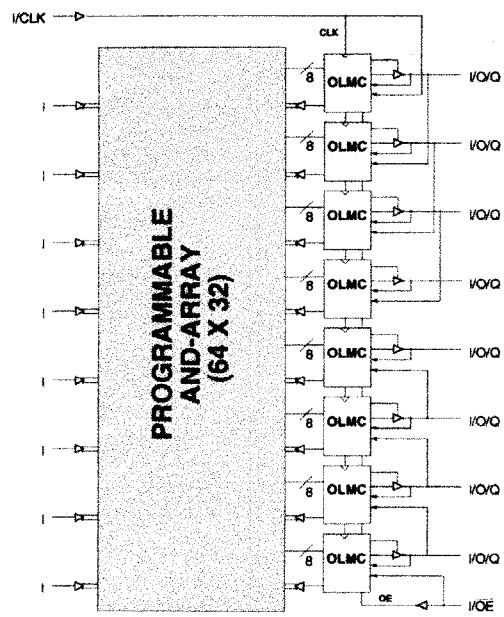
GAL16V8

High Performance E²CMOS PLD
Generic Array Logic™

FEATURES

- **HIGH PERFORMANCE E²CMOS® TECHNOLOGY**
  - **3.5 ns Maximum Propagation Delay**
  - **Fmax = 250 MHz**
  - **3.0 ns Maximum from Clock Input to Data Output**
  - **UltraMOS® Advanced CMOS Technology**
- **50% to 75% REDUCTION IN POWER FROM BIPOLAR**
  - **75mA Typ Icc on Low Power Device**
  - **45mA Typ Icc on Quarter Power Device**
- **ACTIVE PULL-UPS ON ALL PINS**
- **E² CELL TECHNOLOGY**
  - **Reconfigurable Logic**
  - **Reprogrammable Cells**
  - **100% Tested/Guaranteed 100% Yields**
  - **High Speed Electrical Erasure (<100ms)**
  - **20 Year Data Retention**
- **EIGHT OUTPUT LOGIC MACROCELLS**
  - **Maximum Flexibility for Complex Logic Designs**
  - **Programmable Output Polarity**
  - **Also Emulates 20-pin PAL® Devices with Full Function/Fuse Map/Parametric Compatibility**
- **PRELOAD AND POWER-ON RESET OF ALL REGISTERS**
  - **100% Functional Testability**
- **APPLICATIONS INCLUDE:**
  - **DMA Control**
  - **State Machine Control**
  - **High Speed Graphics Processing**
  - **Standard Logic Speed Upgrade**
- **ELECTRONIC SIGNATURE FOR IDENTIFICATION**

FUNCTIONAL BLOCK DIAGRAM



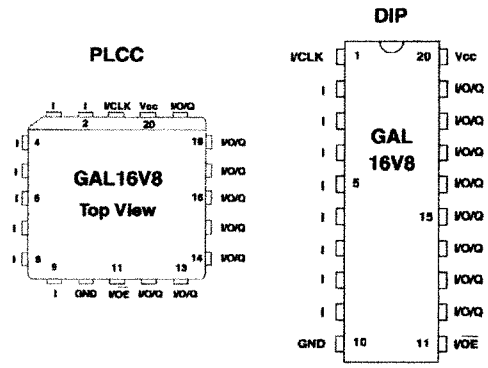
DESCRIPTION

The GAL16V8D, at 3.5 ns maximum propagation delay time, combines a high performance CMOS process with Electrically Erasable (E²) floating gate technology to provide the highest speed performance available in the PLD market. High speed erase times (<100ms) allow the devices to be reprogrammed quickly and efficiently.

The generic architecture provides maximum design flexibility by allowing the Output Logic Macrocell (OLMC) to be configured by the user. An important subset of the many architecture configurations possible with the GAL16V8 are the PAL architectures listed in the table of the macrocell description section. GAL16V8 devices are capable of emulating any of these PAL architectures with full function/fuse map/parametric compatibility.

Unique test circuitry and reprogrammable cells allow complete AC, DC, and functional testing during manufacture. As a result, Lattice Semiconductor guarantees 100% field programmability and functionality of all GAL products. In addition, 100 erase/write cycles and data retention in excess of 20 years are guaranteed.

PIN CONFIGURATION



LATTICE SEMICONDUCTOR CORP., 5555 Northeast Moore Ct., Hillsboro, Oregon 97124, U.S.A.
Tel. (503) 681-0118; 1-888-ISP-PLDS; FAX (503) 681-3037; http://www.latticesemi.com

OUTPUT LOGIC MACROCELL (OLMC)

The following discussion pertains to configuring the output logic macrocell. It should be noted that actual implementation is accomplished by development software/hardware and is completely transparent to the user.

There are three global OLMC configuration modes possible: **simple**, **complex**, and **registered**. Details of each of these modes are illustrated in the following pages. Two global bits, SYN and AC0, control the mode configuration for all macrocells. The XOR bit of each macrocell controls the polarity of the output in any of the three modes, while the AC1 bit of each of the macrocells controls the input/output configuration. These two global and 16 individual architecture bits define all possible configurations in a GAL16V8 . The information given on these architecture bits is only to give a better understanding of the device. Compiler software will transparently set these architecture bits from the pin definitions, so the user should not need to directly manipulate these architecture bits.

The following is a list of the PAL architectures that the GAL16V8 can emulate. It also shows the OLMC mode under which the GAL16V8 emulates the PAL architecture.

PAL Architectures Emulated by GAL16V8	GAL16V8 Global OLMC Mode
16R8	Registered
16R6	Registered
16R4	Registered
16RP8	Registered
16RP6	Registered
16RP4	Registered
16L8	Complex
16H8	Complex
16P8	Complex
10L8	Simple
12L6	Simple
14L4	Simple
16L2	Simple
10H8	Simple
12H6	Simple
14H4	Simple
16H2	Simple
10P8	Simple
12P6	Simple
14P4	Simple
16P2	Simple

COMPILER SUPPORT FOR OLMC

Software compilers support the three different global OLMC modes as different device types. These device types are listed in the table below. Most compilers have the ability to automatically select the device type, generally based on the register usage and output enable (OE) usage. Register usage on the device forces the software to choose the registered mode. All combinatorial outputs with OE controlled by the product term will force the software to choose the complex mode. The software will choose the simple mode only when all outputs are dedicated combinatorial without OE control. The different device types listed in the table can be used to override the automatic device selection by the software. For further details, refer to the compiler software manuals.

When using compiler software to configure the device, the user must pay special attention to the following restrictions in each mode.

In **registered mode** pin 1 and pin 11 are permanently configured as clock and output enable, respectively. These pins cannot be configured as dedicated inputs in the registered mode.

In **complex mode** pin 1 and pin 11 become dedicated inputs and use the feedback paths of pin 19 and pin 12 respectively. Because of this feedback path usage, pin 19 and pin 12 do not have the feedback option in this mode.

In **simple mode** all feedback paths of the output pins are routed via the adjacent pins. In doing so, the two inner most pins (pins 15 and 16) will not have the feedback option as these pins are always configured as dedicated combinatorial output.

	Registered	Complex	Simple	Auto Mode Select
ABEL	P16V8R	P16V8C	P16V8AS	P16V8
CUPL	G16V8MS	G16V8MA	G16V8AS	G16V8
LOG/IC	GAL16V8_R	GAL16V8_C7	GAL16V8_C8	GAL16V8
OrCAD-PLD	"Registered"[1]	"Complex"[1]	"Simple"[1]	GAL16V8A
PLDesigner	P16V8R[2]	P16V8C[2]	P16V8C[2]	P16V8A
TANGO-PLD	G16V8R	G16V8C	G16V8AS[3]	G16V8

1) Used with **Configuration** keyword.
2) Prior to Version 2.0 support.
3) Supported on Version 1.20 or later.

REGISTERED MODE

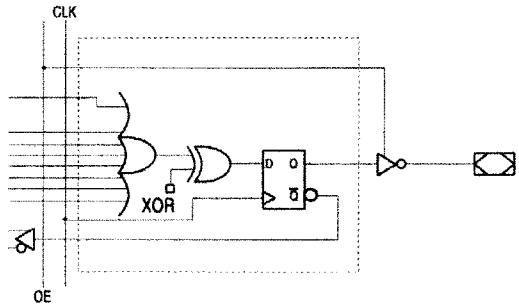
In the Registered mode, macrocells are configured as dedicated registered outputs or as I/O functions.

Architecture configurations available in this mode are similar to the common 16R8 and 16RP4 devices with various permutations of polarity, I/O and register placement.

All registered macrocells share common clock and output enable control pins. Any macrocell can be configured as registered or I/O. Up to eight registers or up to eight I/O's are possible in this mode. Dedicated input or output functions can be implemented as subsets of the I/O function.

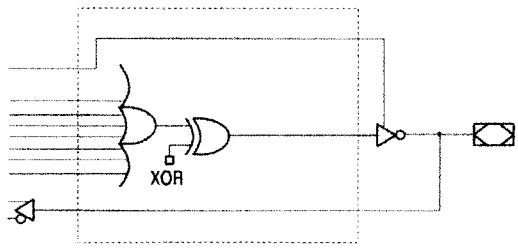
Registered outputs have eight product terms per output. I/O's have seven product terms per output.

The JEDEC fuse numbers, including the User Electronic Signature (UES) fuses and the Product Term Disable (PTD) fuses, are shown on the logic diagram on the following page.



Registered Configuration for Registered Mode

- SYN=0.
- AC0=1.
- XOR=0 defines Active Low Output.
- XOR=1 defines Active High Output.
- AC1=0 defines this output configuration.
- Pin 1 controls common CLK for the registered outputs.
- Pin 11 controls common $\overline{OE}$ for the registered outputs.
- Pin 1 & Pin 11 are permanently configured as CLK & $\overline{OE}$.



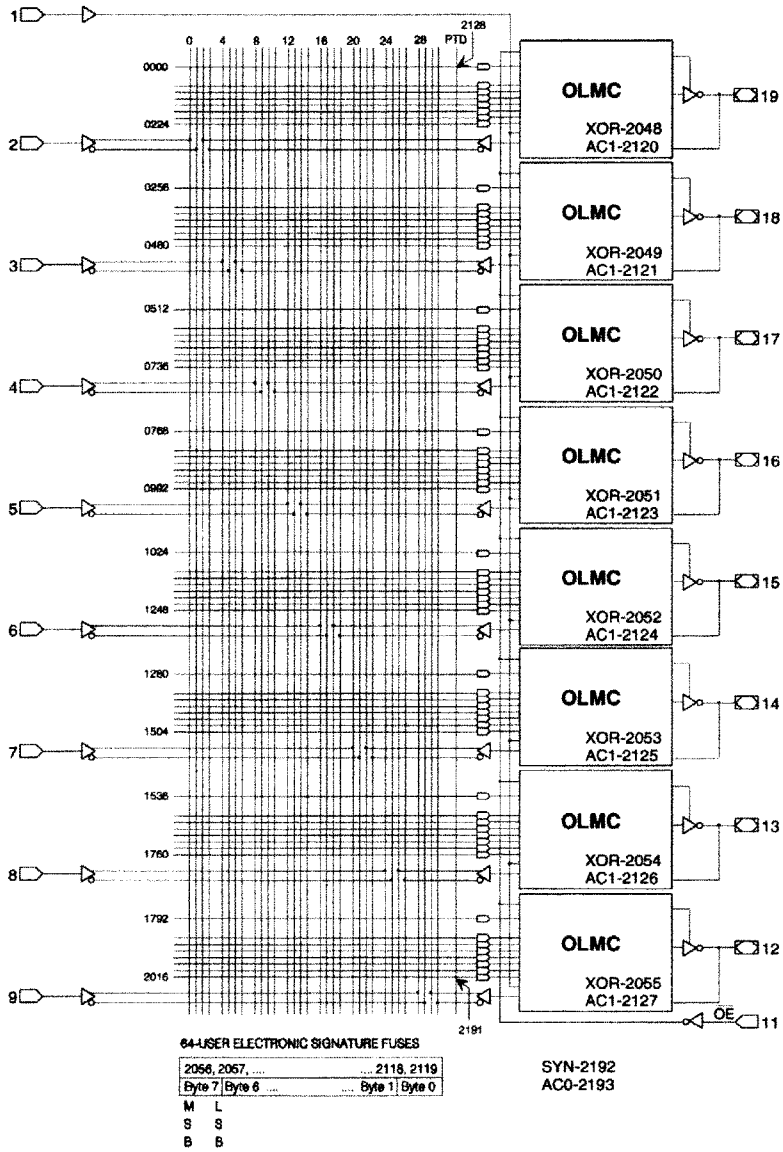
Combinatorial Configuration for Registered Mode

- SYN=0.
- AC0=1.
- XOR=0 defines Active Low Output.
- XOR=1 defines Active High Output.
- AC1=1 defines this output configuration.
- Pin 1 & Pin 11 are permanently configured as CLK & $\overline{OE}$.

Note: The development software configures all of the architecture control bits and checks for proper pin usage automatically.

REGISTERED MODE LOGIC DIAGRAM

DIP & PLCC Package Pinouts



COMPLEX MODE

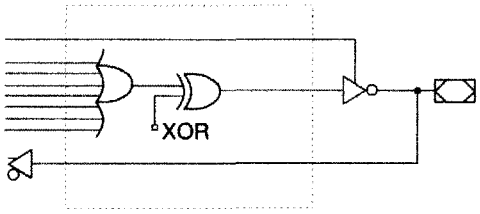
In the Complex mode, macrocells are configured as output only or I/O functions.

Architecture configurations available in this mode are similar to the common 16L8 and 16P8 devices with programmable polarity in each macrocell.

Up to six I/O's are possible in this mode. Dedicated inputs or outputs can be implemented as subsets of the I/O function. The two outer most macrocells (pins 12 & 19) do not have input capability. Designs requiring eight I/O's can be implemented in the Registered mode.

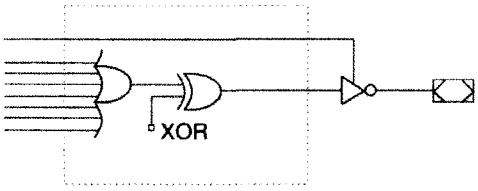
All macrocells have seven product terms per output. One product term is used for programmable output enable control. Pins 1 and 11 are always available as data inputs into the AND array.

The JEDEC fuse numbers including the UES fuses and PTD fuses are shown on the logic diagram on the following page.



Combinatorial I/O Configuration for Complex Mode

- SYN=1.
- AC0=1.
- XOR=0 defines Active Low Output.
- XOR=1 defines Active High Output.
- AC1=1.
- Pin 13 through Pin 18 are configured to this function.



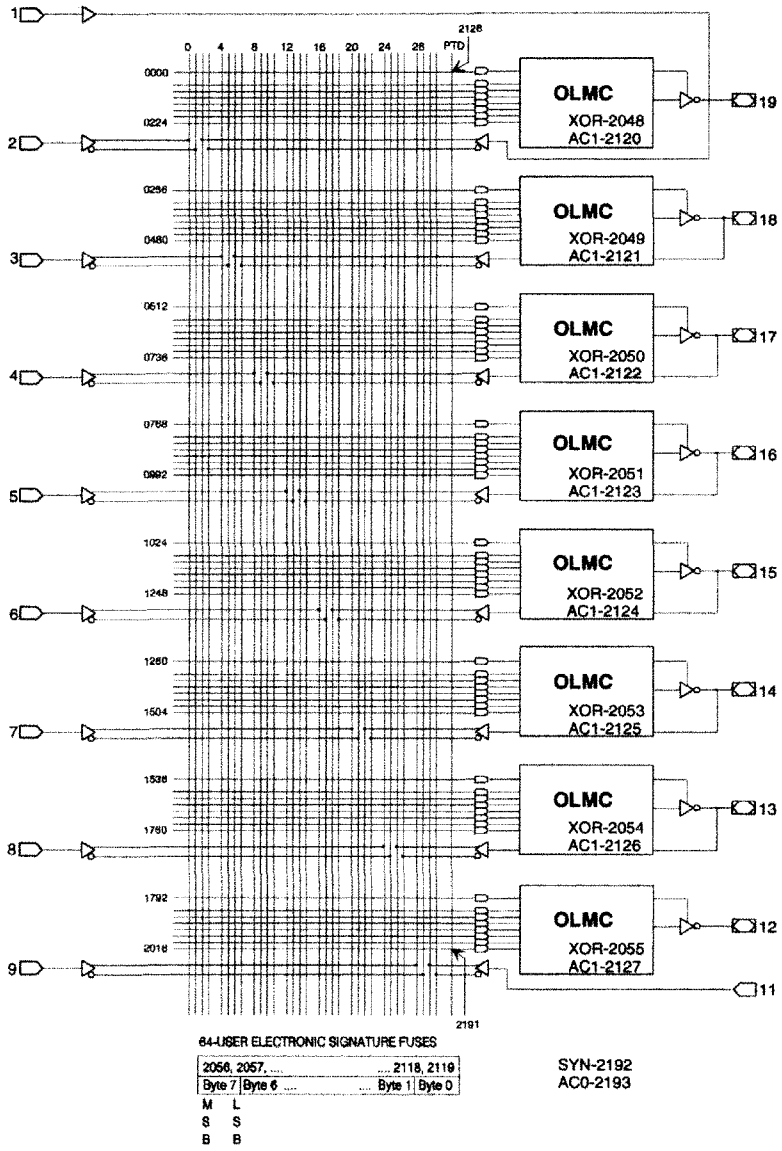
Combinatorial Output Configuration for Complex Mode

- SYN=1.
- AC0=1.
- XOR=0 defines Active Low Output.
- XOR=1 defines Active High Output.
- AC1=1.
- Pin 12 and Pin 19 are configured to this function.

Note: The development software configures all of the architecture control bits and checks for proper pin usage automatically.

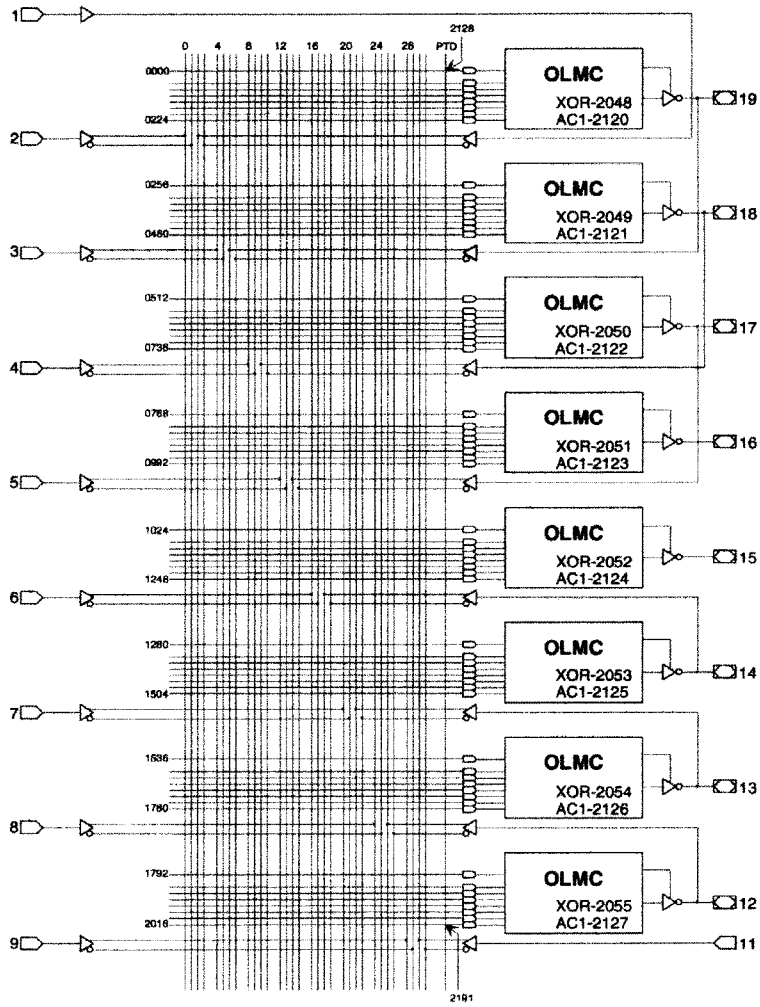
COMPLEX MODE LOGIC DIAGRAM

DIP & PLCC Package Pinouts



SIMPLE MODE LOGIC DIAGRAM

DIP & PLCC Package Pinouts



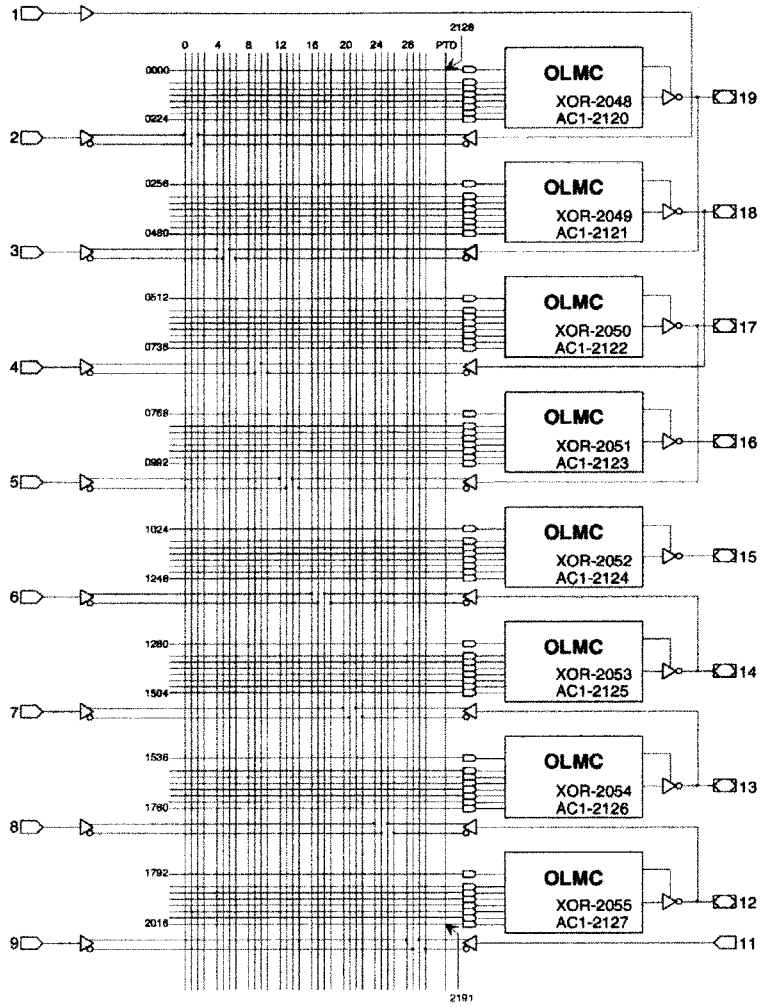
64-USER ELECTRONIC SIGNATURE FUSES

2056, 2057, ...	.... 2118, 2119
Byte 7 Byte 6	.... Byte 1 Byte 0
MSB	LSB

SYN-2192
AC0-2193

SIMPLE MODE LOGIC DIAGRAM

DIP & PLCC Package Pinouts



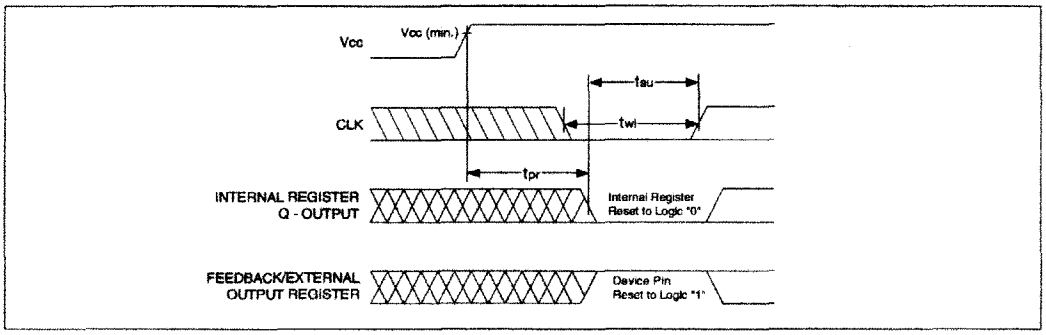
64-USER ELECTRONIC SIGNATURE FUSES

2056, 2057, ...	.... 2118, 2119
Byte 7 | Byte 6	.... Byte 1 | Byte 0

MSB LSB

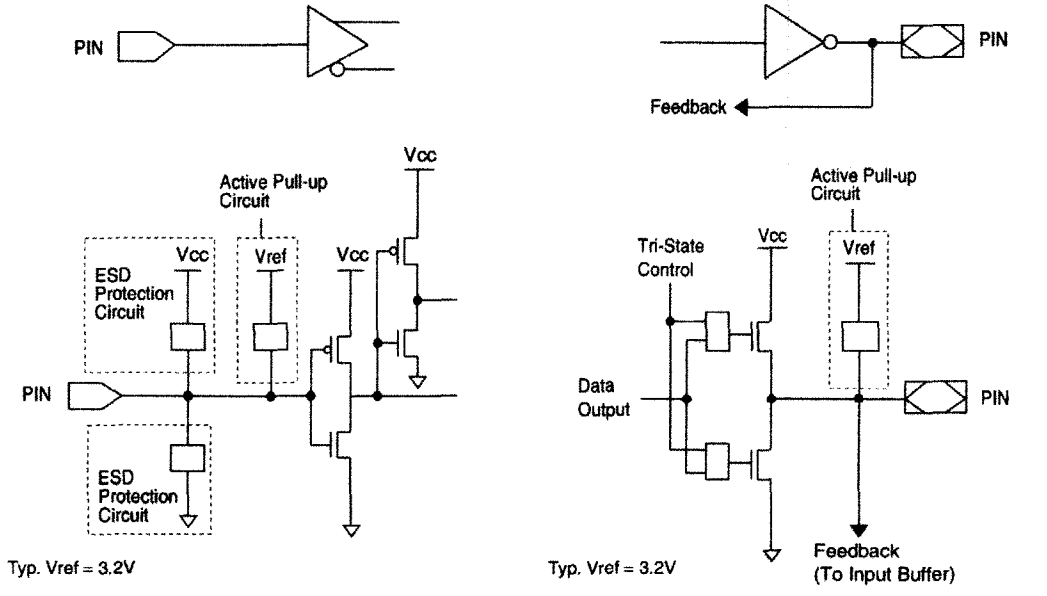
SYN-2192
AC0-2193

POWER-UP RESET



Circuitry within the GAL16V8 provides a reset signal to all registers during power-up. All internal registers will have their Q outputs set low after a specified time (tpr, 1µs MAX). As a result, the state on the registered output pins (if they are enabled) will always be high on power-up, regardless of the programmed polarity of the output pins. This feature can greatly simplify state machine design by providing a known state on power-up. Because of the asynchronous nature of system power-up, some conditions must be met to guarantee a valid power-up reset of the device. First, the Vcc rise must be monotonic. Second, the clock input must be at static TTL level as shown in the diagram during power up. The registers will reset within a maximum of tpr time. As in normal system operation, avoid clocking the device until all input and feedback path setup times have been met. The clock must also meet the minimum pulse width requirements.

INPUT/OUTPUT EQUIVALENT SCHEMATICS



B.2 - Lattice GAL22V10 Data Sheet Excerpts

Lattice™ Semiconductor Corporation

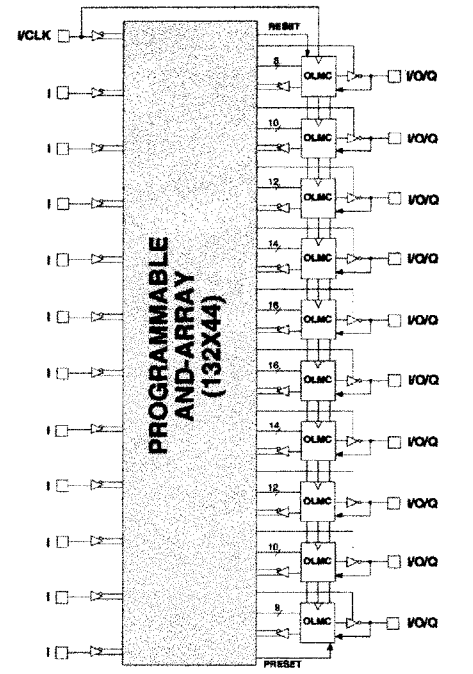
GAL22V10

High Performance E²CMOS PLD
Generic Array Logic™

FEATURES

- **HIGH PERFORMANCE E²CMOS® TECHNOLOGY**
  - **5 ns Maximum Propagation Delay**
  - **Fmax = 200 MHz**
  - **4 ns Maximum from Clock Input to Data Output**
  - **UltraMOS® Advanced CMOS Technology**
- **ACTIVE PULL-UPS ON ALL PINS**
- **COMPATIBLE WITH STANDARD 22V10 DEVICES**
  - **Fully Function/Fuse-Map/Parametric Compatible with Bipolar and UVCMOS 22V10 Devices**
- **50% to 75% REDUCTION IN POWER VERSUS BIPOLAR**
  - **90mA Typical Icc on Low Power Device**
  - **45mA Typical Icc on Quarter Power Device**
- **E² CELL TECHNOLOGY**
  - **Reconfigurable Logic**
  - **Reprogrammable Cells**
  - **100% Tested/Guaranteed 100% Yields**
  - **High Speed Electrical Erasure (<100ms)**
  - **20 Year Data Retention**
- **TEN OUTPUT LOGIC MACROCELLS**
  - **Maximum Flexibility for Complex Logic Designs**
- **PRELOAD AND POWER-ON RESET OF REGISTERS**
  - **100% Functional Testability**
- **APPLICATIONS INCLUDE:**
  - **DMA Control**
  - **State Machine Control**
  - **High Speed Graphics Processing**
  - **Standard Logic Speed Upgrade**
- **ELECTRONIC SIGNATURE FOR IDENTIFICATION**

FUNCTIONAL BLOCK DIAGRAM



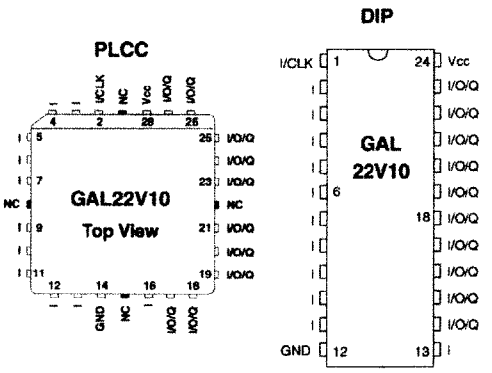
DESCRIPTION

The GAL22V10C, at 5ns maximum propagation delay time, combines a high performance CMOS process with Electrically Erasable (E²) floating gate technology to provide the highest performance available of any 22V10 device on the market. CMOS circuitry allows the GAL22V10 to consume much less power when compared to bipolar 22V10 devices. E² technology offers high speed (<100ms) erase times, providing the ability to reprogram or reconfigure the device quickly and efficiently.

The generic architecture provides maximum design flexibility by allowing the Output Logic Macrocell (OLMC) to be configured by the user. The GAL22V10 is fully function/fuse map/parametric compatible with standard bipolar and CMOS 22V10 devices.

Unique test circuitry and reprogrammable cells allow complete AC, DC, and functional testing during manufacture. As a result, Lattice Semiconductor guarantees 100% field programmability and functionality of all GAL products. In addition, 100 erase/write cycles and data retention in excess of 20 years are guaranteed.

PIN CONFIGURATION



LATTICE SEMICONDUCTOR CORP., 5555 Northeast Moore Ct., Hillsboro, Oregon 97124, U.S.A.
Tel. (503) 681-0118; 1-888-ISP-PLDS; FAX (503) 681-3037; http://www.latticesemi.com

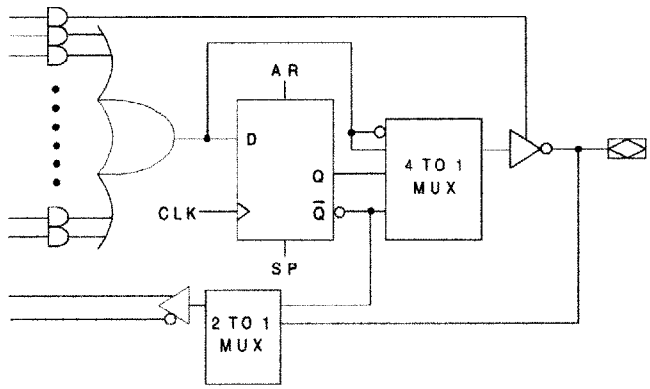
OUTPUT LOGIC MACROCELL (OLMC)

The GAL22V10 has a variable number of product terms per OLMC. Of the ten available OLMCs, two OLMCs have access to eight product terms (pins 14 and 23, DIP pinout), two have ten product terms (pins 15 and 22), two have twelve product terms (pins 16 and 21), two have fourteen product terms (pins 17 and 20), and two OLMCs have sixteen product terms (pins 18 and 19). In addition to the product terms available for logic, each OLMC has an additional product-term dedicated to output enable control.

The output polarity of each OLMC can be individually programmed to be true or inverting, in either combinatorial or registered mode. This allows each output to be individually configured as either active high or active low.

The GAL22V10 has a product term for Asynchronous Reset (AR) and a product term for Synchronous Preset (SP). These two product terms are common to all registered OLMCs. The Asynchronous Reset sets all registers to zero any time this dedicated product term is asserted. The Synchronous Preset sets all registers to a logic one on the rising edge of the next clock pulse after this product term is asserted.

NOTE: The AR and SP product terms will force the Q output of the flip-flop into the same state regardless of the polarity of the output. Therefore, a reset operation, which sets the register output to a zero, may result in either a high or low at the output pin, depending on the pin polarity chosen.



GAL22V10 OUTPUT LOGIC MACROCELL (OLMC)

OUTPUT LOGIC MACROCELL CONFIGURATIONS

Each of the Macrocells of the GAL22V10 has two primary functional modes: registered, and combinatorial I/O. The modes and the output polarity are set by two bits (SO and S1), which are normally controlled by the logic compiler. Each of these two primary modes, and the bit settings required to enable them, are described below and on the following page.

REGISTERED

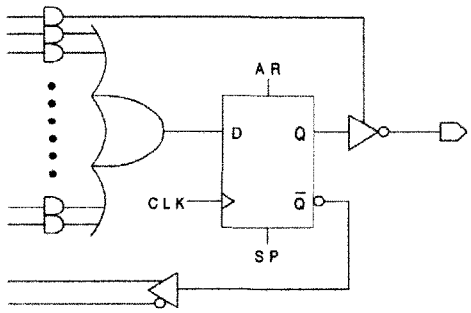
In registered mode the output pin associated with an individual OLMC is driven by the Q output of that OLMC's D-type flip-flop. Logic polarity of the output signal at the pin may be selected by specifying that the output buffer drive either true (active high) or inverted (active low). Output tri-state control is available as an individual product-term for each OLMC, and can therefore be defined by a logic equation. The D flip-flop's /Q output is fed back into the AND array, with both the true and complement of the feedback available as inputs to the AND array.

NOTE: In registered mode, the feedback is from the /Q output of the register, and not from the pin; therefore, a pin defined as registered is an output only, and cannot be used for dynamic I/O, as can the combinatorial pins.

COMBINATORIAL I/O

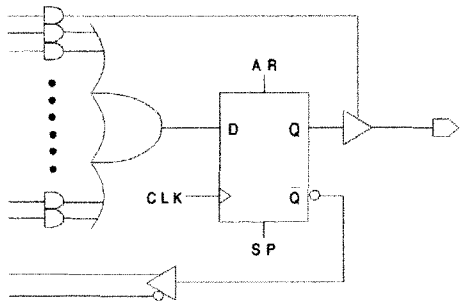
In combinatorial mode the pin associated with an individual OLMC is driven by the output of the sum term gate. Logic polarity of the output signal at the pin may be selected by specifying that the output buffer drive either true (active high) or inverted (active low). Output tri-state control is available as an individual product-term for each output, and may be individually set by the compiler as either "on" (dedicated output), "off" (dedicated input), or "product-term driven" (dynamic I/O). Feedback into the AND array is from the pin side of the output enable buffer. Both polarities (true and inverted) of the pin are fed back into the AND array.

REGISTERED MODE



ACTIVE LOW

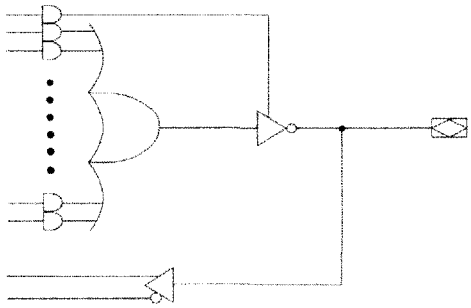
$S_0 = 0$
$S_1 = 0$



ACTIVE HIGH

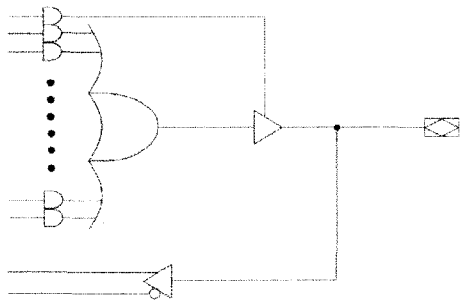
$S_0 = 1$
$S_1 = 0$

COMBINATORIAL MODE



ACTIVE LOW

$S_0 = 0$
$S_1 = 1$

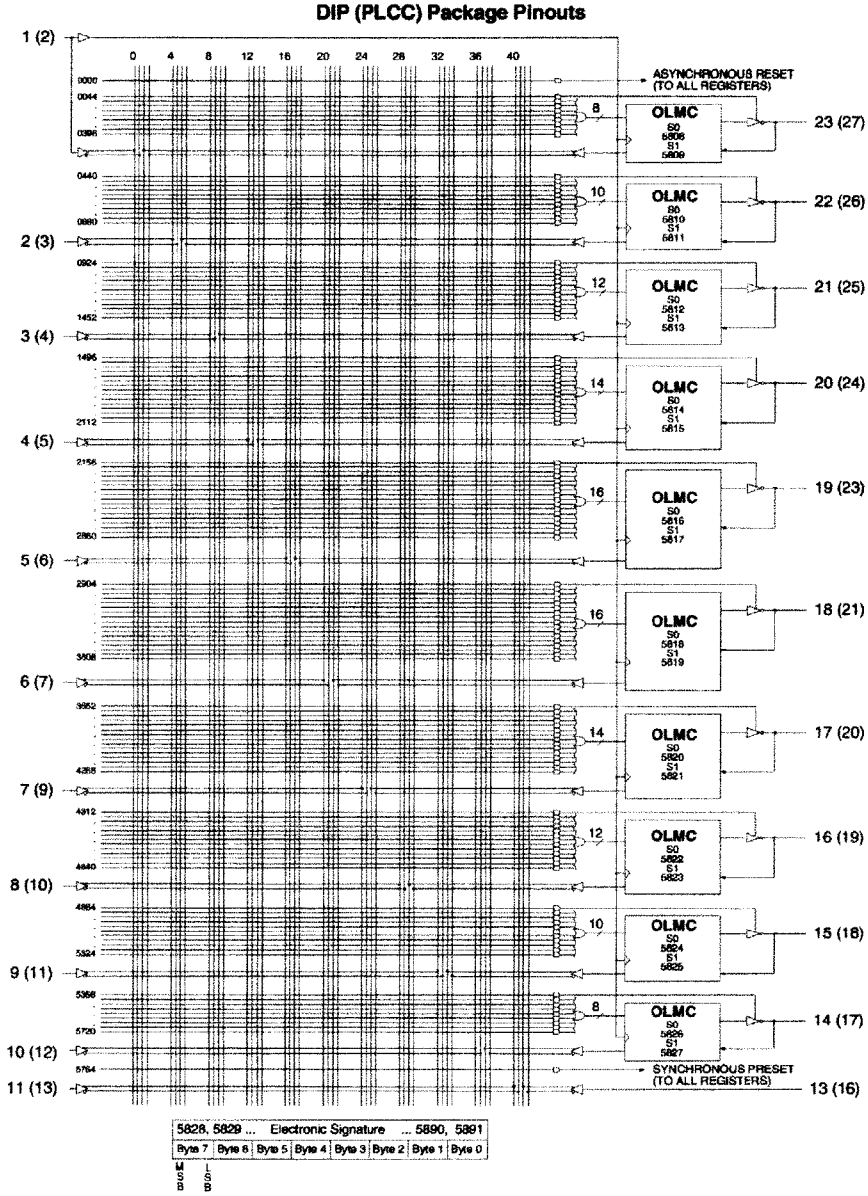


ACTIVE HIGH

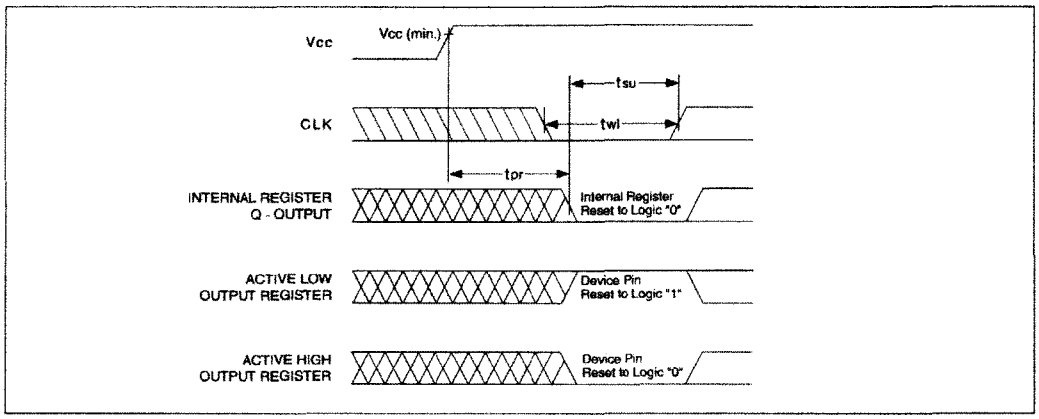
$S_0 = 1$
$S_1 = 1$

GAL22V10 LOGIC DIAGRAM / JEDEC FUSE MAP

DIP (PLCC) Package Pinouts

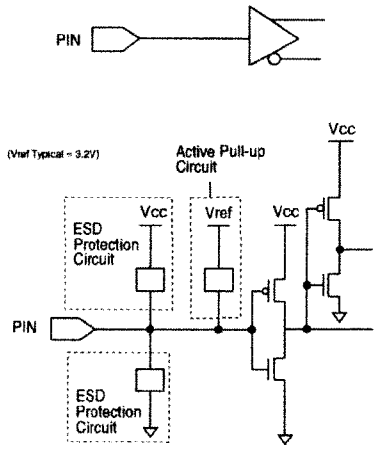


POWER-UP RESET

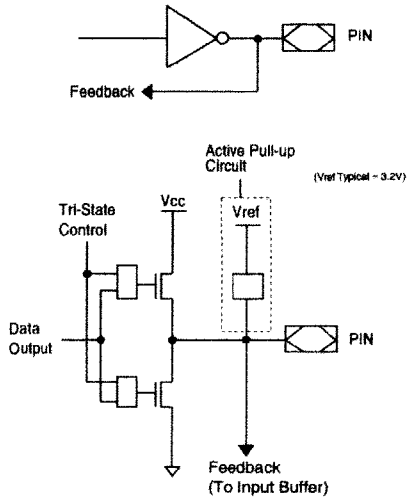


Circuitry within the GAL22V10 provides a reset signal to all registers during power-up. All internal registers will have their Q outputs set low after a specified time (tpr, 1µs MAX). As a result, the state on the registered output pins (if they are enabled) will be either high or low on power-up, depending on the programmed polarity of the output pins. This feature can greatly simplify state machine design by providing a known state on power-up. The timing diagram for power-up is shown below. Because of the asynchronous nature of system power-up, some conditions must be met to guarantee a valid power-up reset of the GAL22V10. First, the Vcc rise must be monotonic. Second, the clock input must be at static TTL level as shown in the diagram during power up. The registers will reset within a maximum of tpr time. As in normal system operation, avoid clocking the device until all input and feedback path setup times have been met. The clock must also meet the minimum pulse width requirements.

INPUT/OUTPUT EQUIVALENT SCHEMATICS



Typical Input



Typical Output

B.3 - Lattice GAL26CV12 Data Sheet Excerpts

Lattice™ Semiconductor Corporation

GAL26CV12

High Performance E²CMOS PLD
Generic Array Logic™

FEATURES

- **HIGH PERFORMANCE E²CMOS® TECHNOLOGY**
  - **7.5 ns Maximum Propagation Delay**
  - **Fmax = 142.8 MHz**
  - **4.5ns Maximum from Clock Input to Data Output**
  - **TTL Compatible 16 mA Outputs**
  - **UltraMOS® Advanced CMOS Technology**
- **ACTIVE PULL-UPS ON ALL PINS**
- **LOW POWER CMOS**
  - **90 mA Typical Icc**
- **E² CELL TECHNOLOGY**
  - **Reconfigurable Logic**
  - **Reprogrammable Cells**
  - **100% Tested/Guaranteed 100% Yields**
  - **High Speed Electrical Erasure (<100ms)**
  - **20 Year Data Retention**
- **TWELVE OUTPUT LOGIC MACROCELLS**
  - **Uses Standard 22V10 Macrocells**
  - **Maximum Flexibility for Complex Logic Designs**
- **PRELOAD AND POWER-ON RESET OF REGISTERS**
  - **100% Functional Testability**
- **APPLICATIONS INCLUDE:**
  - **DMA Control**
  - **State Machine Control**
  - **High Speed Graphics Processing**
  - **Standard Logic Speed Upgrade**
- **ELECTRONIC SIGNATURE FOR IDENTIFICATION**

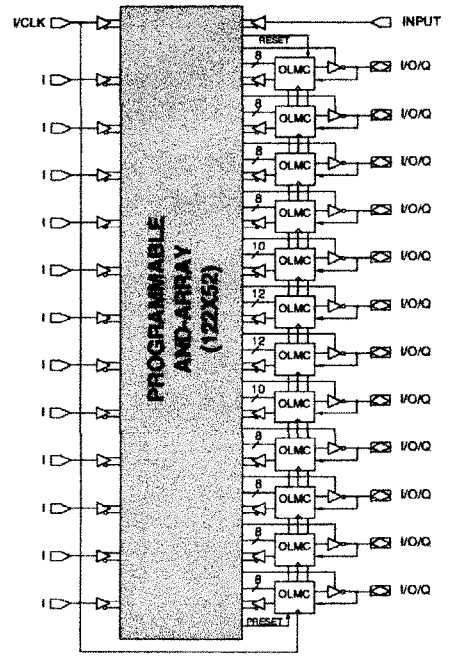
DESCRIPTION

The GAL26CV12, at 7.5 ns maximum propagation delay time, combines a high performance CMOS process with Electrically Erasable (E^2) floating gate technology to provide the highest performance 28-pin PLD available on the market. E^2 technology offers high speed (<100ms) erase times, providing the ability to reprogram or reconfigure the device quickly and efficiently.

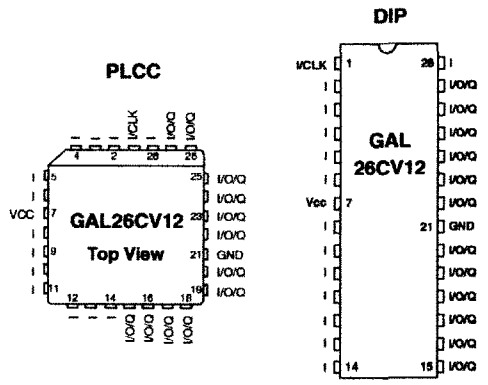
Expanding upon the industry standard 22V10 architecture, the GAL26CV12 eliminates the learning curve typically associated with using a new device architecture. The generic architecture provides maximum design flexibility by allowing the Output Logic Macrocell (OLMC) to be configured by the user. The GAL26CV12 OLMC is fully compatible with the OLMC in standard bipolar and CMOS 22V10 devices.

Unique test circuitry and reprogrammable cells allow complete AC, DC, and functional testing during manufacture. As a result, Lattice Semiconductor guarantees 100% field programmability and functionality of all GAL products. In addition, 100 erase/write cycles and data retention in excess of 20 years are guaranteed.

FUNCTIONAL BLOCK DIAGRAM



PIN CONFIGURATION



LATTICE SEMICONDUCTOR CORP., 5555 Northeast Moore Ct., Hillsboro, Oregon 97124, U.S.A.
Tel. (503) 681-0118; 1-888-ISP-PLDS; FAX (503) 681-3037; http://www.latticesemi.com

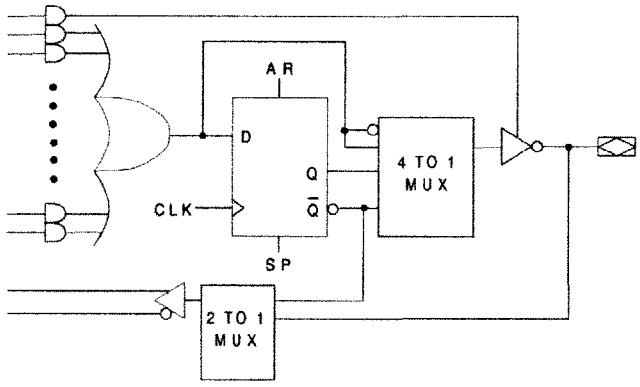
OUTPUT LOGIC MACROCELL (OLMC)

The GAL26CV12 has a variable number of product terms per OLMC. Of the twelve available OLMCs, two OLMCs have access to twelve product terms (pins 20 and 22), two have access to ten product terms (pins 19 and 23), and the other eight OLMCs have eight product terms each. In addition to the product terms available for logic, each OLMC has an additional product term dedicated to output enable control.

The output polarity of each OLMC can be individually programmed to be true or inverting, in either combinatorial or registered mode. This allows each output to be individually configured as either active high or active low.

The GAL26CV12 has a product term for Asynchronous Reset (AR) and a product term for Synchronous Preset (SP). These two product terms are common to all registered OLMCs. The Asynchronous Reset sets all registered outputs to zero any time this dedicated product term is asserted. The Synchronous Preset sets all registers to a logic one on the rising edge of the next clock pulse after this product term is asserted.

NOTE: The AR and SP product terms will force the Q output of the flip-flop into the same state regardless of the polarity of the output. Therefore, a reset operation, which sets the register output to a zero, may result in either a high or low at the output pin, depending on the pin polarity chosen.



GAL26CV12 OUTPUT LOGIC MACROCELL (OLMC)

OUTPUT LOGIC MACROCELL CONFIGURATIONS

Each of the Macrocells of the GAL26CV12 has two primary functional modes: registered, and combinatorial I/O. The modes and the output polarity are set by two bits (SO and S1), which are normally controlled by the logic compiler. Each of these two primary modes, and the bit settings required to enable them, are described below and on the the following page.

REGISTERED

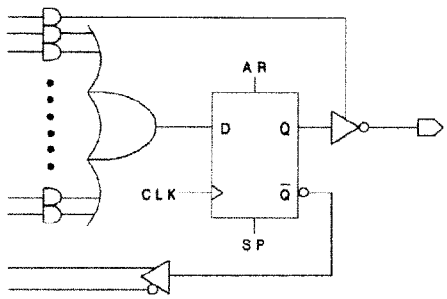
In registered mode the output pin associated with an individual OLMC is driven by the Q output of that OLMC's D-type flip-flop. Logic polarity of the output signal at the pin may be selected by specifying that the output buffer drive either true (active high) or inverted (active low). Output tri-state control is available as an individual product term for each OLMC, and can therefore be defined by a logic equation. The D flip-flop's /Q output is fed back into the AND array, with both the true and complement of the feedback available as inputs to the AND array.

NOTE: In registered mode, the feedback is from the /Q output of the register, and not from the pin; therefore, a pin defined as registered is an output only, and cannot be used for dynamic I/O, as can the combinatorial pins.

COMBINATORIAL I/O

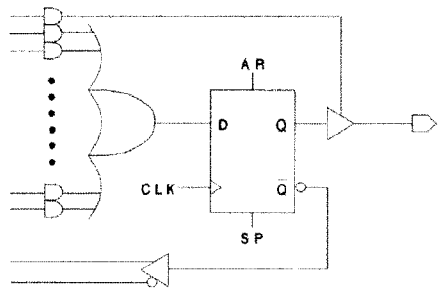
In combinatorial mode the pin associated with an individual OLMC is driven by the output of the sum term gate. Logic polarity of the output signal at the pin may be selected by specifying that the output buffer drive either true (active high) or inverted (active low). Output tri-state control is available as an individual product term for each output, and may be individually set by the compiler as either "on" (dedicated output), "off" (dedicated input), or "product term driven" (dynamic I/O). Feedback into the AND array is from the pin side of the output enable buffer. Both polarities (true and inverted) of the pin are fed back into the AND array.

REGISTERED MODE



ACTIVE LOW

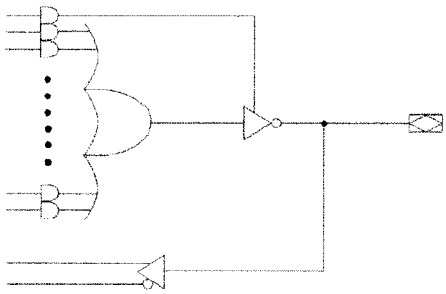
$S_0 = 0$
$S_1 = 0$



ACTIVE HIGH

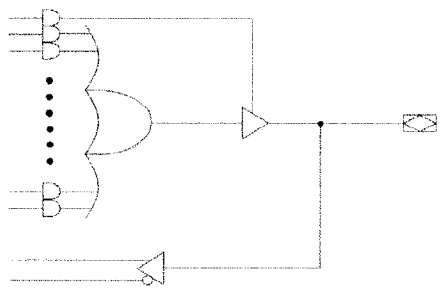
$S_0 = 1$
$S_1 = 0$

COMBINATORIAL MODE



ACTIVE LOW

$S_0 = 0$
$S_1 = 1$

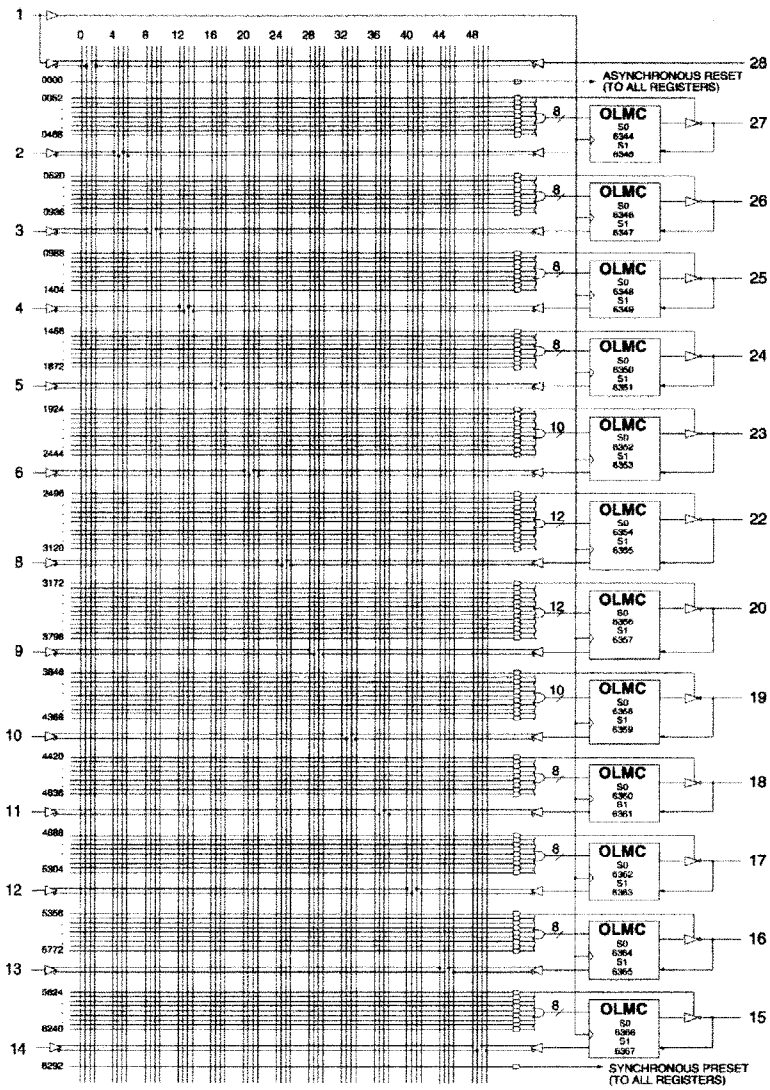


ACTIVE HIGH

$S_0 = 1$
$S_1 = 1$

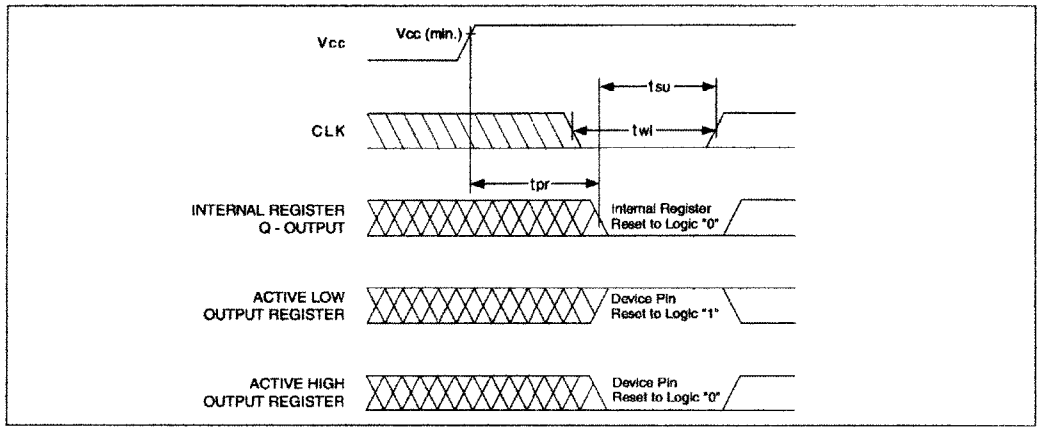
GAL26CV12 LOGIC DIAGRAM / JEDEC FUSE MAP

DIP & PLCC Package Pinouts



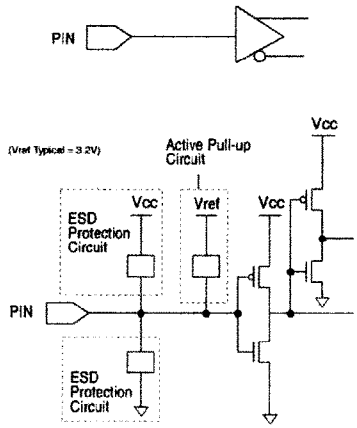
6368, 6369 ... Electronic Signature ... 6430, 6431							
Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
MSB							LSB

POWER-UP RESET

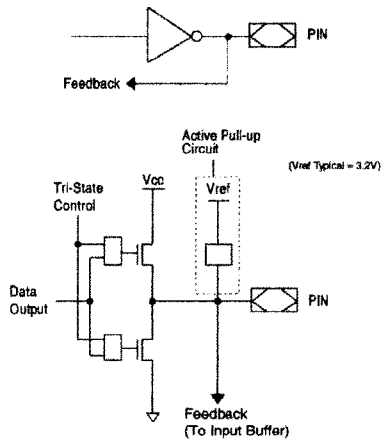


Circuitry within the GAL26CV12 provides a reset signal to all registers during power-up. All internal registers will have their Q outputs set low after a specified time (tpr, 1µs MAX). As a result, the state on the registered output pins (if they are enabled) will be either high or low on power-up, depending on the programmed polarity of the output pins. This feature can greatly simplify state machine design by providing a known state on power-up. Because of the asynchronous nature of system power-up, some conditions must be met to guarantee a valid power-up reset of the device. First, the Vcc rise must be monotonic. Second, the clock input must be at static TTL level as shown in the diagram during power up. The registers will reset within a maximum of tpr time. As in normal system operation, avoid clocking the device until all input and feedback path setup times have been met. The clock must also meet the minimum pulse width requirements.

INPUT/OUTPUT EQUIVALENT SCHEMATICS



Typical Input



Typical Output

REFERENCES

[1] Fletcher, William I. An Engineering Approach to Digital Design. New Jersey: Prentice-Hall, Inc., 1980.

[2] Pellerin, David and Holley, Michael. Practical Design Using Programmable Logic. New Jersey: Prentice-Hall, Inc., 1991.

[3] ABEL User Manual. Data I/O Corporation, 1990.

[4] Logic Diagram Package. Data I/O Corporation, 1990.

[5] Data I/O Corporation Web Site: http://www.data-io.com

[6] Lattice Semiconductor Corporation Web Site: http://www.latticesemi.com