

Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

1988

Logic design using programmable logic devices

Loc Bao Nguyen

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Nguyen, Loc Bao, "Logic design using programmable logic devices" (1988). *Dissertations and Theses*. Paper 4103.

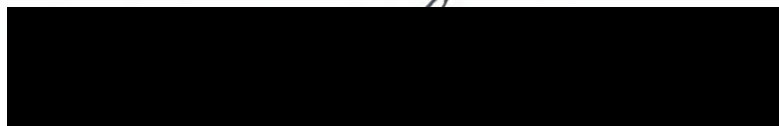
<https://doi.org/10.15760/etd.5987>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

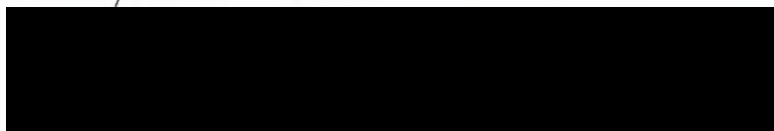
AN ABSTRACT OF THE THESIS OF Loc Bao Nguyen for the Master of Science in Electrical Engineering presented August 18, 1988

Title: Logic Design Using Programmable Logic Devices

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:



Marek Perkowski, Chair



Jack Riley



Mohammed Gafarzade

The Programmable Logic Devices, PLD, have caused a major impact in logic design of digital systems in this decade. For instance, a twenty pin PLD device can replace from three hundreds to six hundreds Transistor Transistor Logic gates, which people have designed with since the 60s. Therefore, by using PLD devices, designers can squeeze more features, reduce chip counts, reduce power consumption, and enhance the reliability of the digital systems.

This thesis covers the most important aspects of logic

design using PLD devices. They are Logic Minimization and State Assignment. In addition, the thesis also covers a seldomly used but very useful design style, Self-Synchronized Circuits.

The thesis introduces a new method to minimize Two-Level Boolean Functions using Graph Coloring Algorithms and the result is very encouraging. The raw speed of the coloring algorithms is as fast as the Espresso, the industry standard minimizer from Berkeley, and the solution is equally good.

The thesis also introduces a rule-based state assignment method which gives equal or better solutions than STASH (an Intel Automatic CAD tool) by as much as twenty percent.

One of the problems with Self-Synchronized Circuits is that it takes many extra components to implement the circuit. The thesis shows how it can be designed using PLD devices and also suggests the idea of a Clock Chip to reduce the chip count to make the design style more attractive.

LOGIC DESIGN USING PROGRAMMABLE LOGIC DEVICES

by
LOC BAO NGUYEN

A thesis submitted in partial fulfillment of the
requirements for the degree of

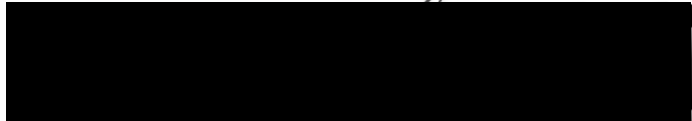
MASTER OF SCIENCE
in
ELECTRICAL ENGINEERING

Portland State University

1988

TO THE OFFICE OF GRADUATE STUDIES:

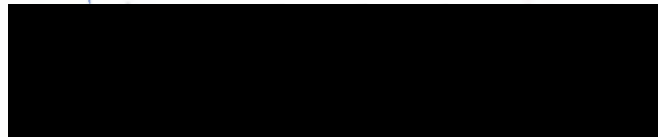
The members of the Committee approve the dissertation
of Loc Bao Nguyen presented August 18, 1988.



Marek Perkowski

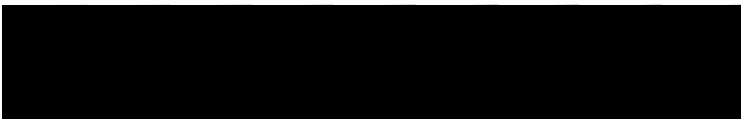


Jack Riley

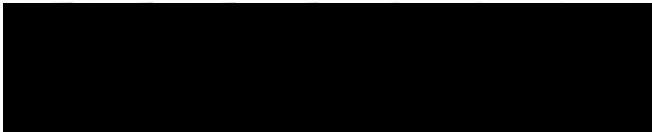


Mohammed Gafarzade

APPROVED:



Lee W. Casperson, Chair, Department of Electrical
Engineering



Bernard Ross, Vice Provost for Graduate Studies

ACKNOWLEDGEMENT

I sincerely thank Dr. Marek Perkowski and my wife, Anhle for their encouragement. Dr. Perkowski has encouraged me to come back to school during 1988 to finish up the thesis after I had discontinued schooling for a year due to the pressure at work. Without the support from those mentioned, I could not have been able to finish this thesis.

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER	
I INTRODUCTION	1
II DESIGNING SELF-SYNCHRONIZED CIRCUIT	
USING PALs OR PLDs	7
Self synchronized circuit structure . . .	11
Clock generator block	14
III INTERNAL STATE ASSIGNMENT FOR FINITE STATE	
MACHINE USING PLDs	37
Heuristic rules for state assignment . .	41
Output consideration	53
IV LOGIC MINIMIZATION OF TWO LEVEL BOOLEAN	
FUNCTION USING GRAPH COLORING	61
Minimal Implicants	64
Compatible implicants and compatible	
sets	74
Minimization of multi-output two-level	
Boolean functions	88

CHAPTER	PAGE
V ZAPAGAL BOARD	117
VI CONCLUSION	155
BIBLIOGRAPHY	158
APPENDIX A	162
APPENDIX B	167
APPENDIX C	217

LIST OF TABLES

TABLE		PAGE
I	Transition Table of D-Flip-Flop	47
II	Matching operator	75
III	Star operation	108
IV	Palmini performance	114

LIST OF FIGURES

FIGURE		PAGE
1.	Huffman Moore machine	8
2.	Rey and Vaucher flow chart	11
3.	Self-synchronized machine	12
4.	MOC machine	13
5.	Clock generator block	14
6.	Change detector	16
7.	TTL implementation of change detector	17
8.	Symmetrical delay	18
9.	Asymmetrical delay	18
10.	Realization of asymmetrical delay	19
11.	UIC machine	26
12.	Crumb road problem	28
13.	Asynchronous circuit for Crumb road problem	29
14.	Synchronous circuit for Crumb road problem	30
15.	Self-synchronized circuit for Crumb road problem	31
16.	UIC case for Crumb road problem	34
17.	Front end chip	36
18.	Combinatorial output	42
19.	Registered output	43

FIGURE		PAGE
20.	Coston and Costoff	45
21.	Coston and Costoff calculation	46
22.	Transition equations	48
23.	Rule 1	50
24.	Reset signal	51
25.	Rule 2	52
26.	Rule 3	52
27.	Grey code assignment	55
28.	Rule based assignment	57
29.	Necessary implicant	69
30.	Necessary implicant 2	70
31.	Minimal implicant	72
32.	Example 4.4	74
33.	Compatible implicants	78
34.	Compatible coloring	84
35.	Multioutput 1	89
36.	Multioutput 2	90
37.	Example 4.8	95
38.	Depth-first strategy with on successor . . .	96
39.	GAL16V8 logic diagram	119
40.	Output macro cell	120
41.	Zapagal board	122
42.	Zapagal block diagram	125
43.	DC-DC converter	129

FIGURE	PAGE
44. Programmable voltage converter	130
45. Edit mode pinout	144
46. Shift register I/O timings	147
47. Array mpas for GAL16V8	148

CHAPTER I

INTRODUCTION

Programmable Logic Devices, PLDs and PALs, were introduced in late 70s; at that time, the state of the art MBI CPU boards from Intel Oregon Division, iSBC 86/12A, iSBC86/30 had only two PALs per board. By the time the iSBC286/20MP and iSBC86C38 boards were designed in 1986 and 1987 respectively, the average number of PALs per board was 20. In 1988, the high performance MBII CPU board, iSBC386/125, has almost 40 PALs per board.

Why are PALs getting so popular? The answer is that we can implement more logic for a given real estate of the printed circuit board with PALs than with discrete logic gates, TTL types. On the average, a PAL16X8 can replace up to 300 logic gates. In addition, a CPU board in the 70s was fairly simple. It contained some ROM, RAM 64K or less, I/O section, and a Microprocessor. However, the CPU board in the 80s is a complete computer system. It may have Cache, Dram up to 64 megabyte on board, DMA capability, I/O and SCSI subsystems, and a lot more. Without using PALs or Custom Gate Array chips, it is impossible to design those features into a board with an area of 9 x 9 inches.

Not only Intel is using PALs; other companies also use PALs extensively. As a consequence, in 1988, there are so many large manufacturers who are producing PLDs like Advance Micro Device, Signetic, Lattice, Altera, Intel, Texas Instrusments, National Semiconductor and many more. In addition, there are a lot of small companies who sell PLD programmers on the market. Some of the big names are Data I/O, Lattice, Altera, and Pcad.

At the time the work on this subject was started in late 1985, there were not many low cost (less than \$5000) tools for PLDs on the market. Actually, there were only two big companies who could support a rather complete CAD tools for PLDs and they were Data I/O with ABEL and Assisted Technology with CUPL. Today there are many vendors who can offer a rather complete system for under \$5000. Some of them are Intel, Altera, Data I/O, Pcad, Signetics etc.

A complete system consists of two parts: software and hardware. The hardware portion is the programmer with firmware on it. The software part consists of the following:

- A high level language (a compiler like ABEL) to translate the state machine description source code to an intermediate level code for processing.
- A State Machine Assignment tool. None of the low

cost tools above can do this. All they can do is State Machine translation which translates the preassigned state assignment to Boolean equations.

- A Boolean Minimization to minimize the logic function so that the function will fit into the target device.
- A JEDEC file generation and programming part.

It is obvious that a complete system will require the support of a company. When the thesis was first started, a complete system was the intention. This thesis has touched on many of the above areas. The details will be covered in the chapters. Following is the summary of the works on this thesis.

- Instead of writing a compiler (high level language), a simple Parser to translate the equations from standard ASCII characters to an intermediate form was provided. From this the Boolean minimization program will read and minimize it. Also Post processing will take this minimized version and retranslate it back to ASCII characters.

- Instead of writing a state machine translation, a set of three rules were offered to do state assignment for PALs. These rules are heuristics but give very good results when compared to those of STASH (a state assignment tool at INTEL). Currently, the author of the thesis does not know

of any CAD tool which is optimized for PALs. There are some tools in the main frame like KISS (DeMicheli, IBM) but KISS will optimize the number of flip-flops rather than the excitation functions. Hence, it does not work well for PAL based designs.

- PALMINI, a Boolean Minimizer using Graph Coloring Algorithms was introduced. At the time it was done in 1986, there were very few Boolean Minimization programs existed in personal computer on the market. They were Espresso from Berkeley, which is considered to be the best, Presto from ABEL, Data I/O corp, A plus from Altera Corp, and CUPL from Assisted Technology. For small examples (PAL based designs), PALMINI is equal or faster than ESPRESSO, much faster than ABEL, and many times faster than ALTERA. As a consequence, two papers were published on two subsequent versions of PALMINI at two conferences: Northcon Conference at Seattle, October 1986 and the other at the Design Automation Conference in Florida, May 1987. In addition, PALMINI offers static hazard elimination for asynchronous machine designs which other Boolean Minimizers do not have.

- A chapter about design Self-synchronized circuits using PALs was introduced. There are very few papers about this topic. However, this design style is very useful. Donald C. Kirkpatrick used it in the state of the art logic analyzer DAS 9200, 1986 at Tektronix. The thesis will show

how we can design self-synchronized circuit using PALs and the idea of building an integrated circuit, an IC chip, to make the design much easier and cheaper is suggested.

- In the last chapter, the thesis shows a complete design of a generic PLD programmer. This low cost programmer is attached to a PC computer and with adequate software, it can perform like a very expensive programmer on the market. It can potentially program all Lattice GAL devices which can emulate many popular PALs, Altera EPLDs 20 and 24 pins, EPROMs from 64K to 1Meg bit, and EEPROMs. With the software already written, it can program EPROMs and GALs, upload and download JEDEC code. Some friends at work have asked me to fabricate this product and market it because it is a very useful tool to have for the lab bench.

Realizing that there is still a lot of work that needs to be done to put together a complete system, however, this thesis has addressed most of the difficult aspects of the system already.

CHAPTER II

DESIGNING SELF-SYNCHRONIZED CIRCUITS USING PALs OR PLDs

INTRODUCTION

Asynchronous Design methods can be used to solve practical problems in the following cases 1) the synchronizing clock in the system is not available, 2) the interface between synchronous circuits, 3) the speed is important and the system can not wait for the next clock pulse to get synchronized.

However, the methods to perform asynchronous designs are much more difficult compared to those of synchronous designs due to stray delays, races, and hazards.

The idea of Self-Synchronized machines originates back to 1971. Bredeson [Bredeson 1971] published the first method to use the input transitions to generate a self-synchronizing clock pulse. He also described how the critical races and logic hazards are avoided by the self-synchronizing clock pulse. However, the design method in his paper is strictly limited to a single-input change mode. Solution to the multiple-input change problem took place in 1973. The machine introduced by Chuang and Das [Chuang, 1973] used the bank of flip-flops for internal registers to utilize the

advantage of arbitrary state assignment of synchronous circuits. The paper published by Rey and Vaucher [Rey, 1974] showed the triggering scheme for multiple input change circuits. The most important paper in the 70s on this subject was probably by Unger [Unger, 1977]. In his paper, Unger discussed the machines of Rey and Vaucher and the machines of Chuang and Das. He also showed how to implement the differentiator circuit using the XOR gates and the latch. In addition, he also discussed the unrestricted input change mode circuits. Between 1976 and 1981, there were some papers by Huertas and Acha [Huertas, 1976], O. Yenersoy [Yenersoy, 1979], El-derini and Hegazy [El-derini, 1981] which did not offer much more information than those previous papers. The latest paper on this subject by Kirkpatrick [Kirkpatrick, 1986] was by far the best paper. He introduced the asymmetrical delay elements which enable machines to operate at a speed limited only by the required function and the choice of circuit technology. His approach is also extended for unrestricted input change mode circuits.

BASIC CONCEPTS AND DEFINITIONS

The general model for a Huffman-Moore machine is shown as follows:

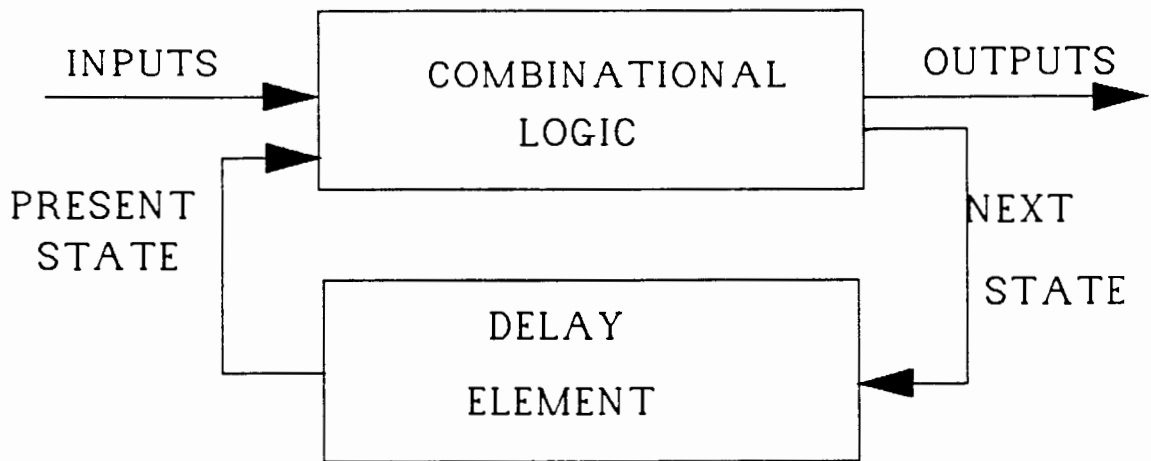


Figure 2.1 Huffman Moore machine

This model applies for both synchronous and asynchronous circuits.

Synchronous Machines:

Synchronous machines are machines which use clocked delay elements or flip-flops. The system clock has a period longer than the sum of the worst-case delay through the combinational logic, plus the worst-case skews of the inputs, and plus the worst-case flip-flop set-up time. The present state value is not allowed to change until the inputs and next states have settled to their proper values. Hence, any arbitrary state-transition function and output function can be easily computed in each clock cycle.

Asynchronous Machines:

Asynchronous machines are machines which do not have a synchronizing clock. The advantages of the asynchronous machine are that no synchronizing clock pulse is needed, and that the state transitions can proceed at a rate limited only by the time delays in the feedback loop. However, they also can suffer many failures which are not encountered in synchronous designs. Some of the failure modes are as follows:

Critical Races:

An asynchronous machine is said to have a critical race if the proper operation of the machine depends upon the relative speed of the state-variable changes.

Essential Hazards:

An asynchronous machine is said to have essential hazards if any state has the following behaviors: Starting in state s , the machine should reach the stable state y with the input change to x . However, due to the improper state assignment and the different delays and races in the circuit, the machine may enter a different stable state under the same input change x at different times.

Static Hazards (Logic Hazards):

Any combinational logic having the potential for spurious outputs is said to have a logic hazard. One way to

avoid this is to introduce redundant prime implicants (consensuses of prime implicants from the selected cover) to subpress the spurious pulses.

Fundamental Mode:

A machine is said to operate in the fundamental mode if the total state (stable state and inputs are stable) is reached between input changes.

Single Input Change (SIC) mode:

A machine can have many inputs but only one input is allowed to change level to cause the machine to enter the next state.

Multiple Input Change (MIC) mode:

More than one input level is allowed to change, and all changes within some small interval are accepted as if they were simultaneous.

Unrestricted Input Change (UIC) mode:

A machine is said to operate in UIC mode if there are no constraints in the possible input sequences.

Single Output Change (SOC) mode:

A machine is said to operate in SOC mode if any input sequence causes only one state transition. All the synchronous circuits operate in SOC mode. We will treat the SOC mode in this chapter.

Multiple Output Change (MOC) mode:

A machine is said to operate in MOC mode if any input sequence causes the machine to perambulate through states before reaching the stable state. Please refer to PH.D Dissertation by Kirpatrick [Kirkpatrick 86] for the detailed discussion of MOC case of Self-synchronized circuits.

SELF SYNCHRONIZED CIRCUIT STRUCTURE

The following diagram by Rey and Vaucher [Rey and Vaucher, 1974] shows how the self synchronized machines would operate.

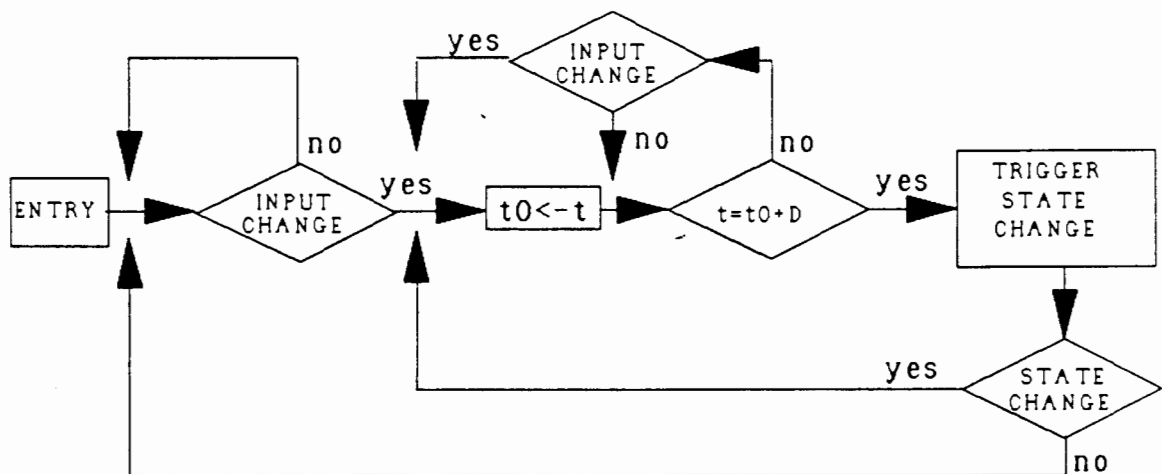


Figure 2.2 Rey and Vaucher flow chart

From the flow chart, the operation can be summarized as follows:

- 1) Detect the input change. (A change detector).

- 2) Let's inputs stable by keep sampling input changes within a window with respect to the last input change.
 - 3) Trigger the state machine by creating a clock pulse.
 - 4) If the state variable are stable then go back to 1).
- (This is for the SOC case).

From the hardware standpoint, the self-synchronized machines can be represented by the following block diagram.

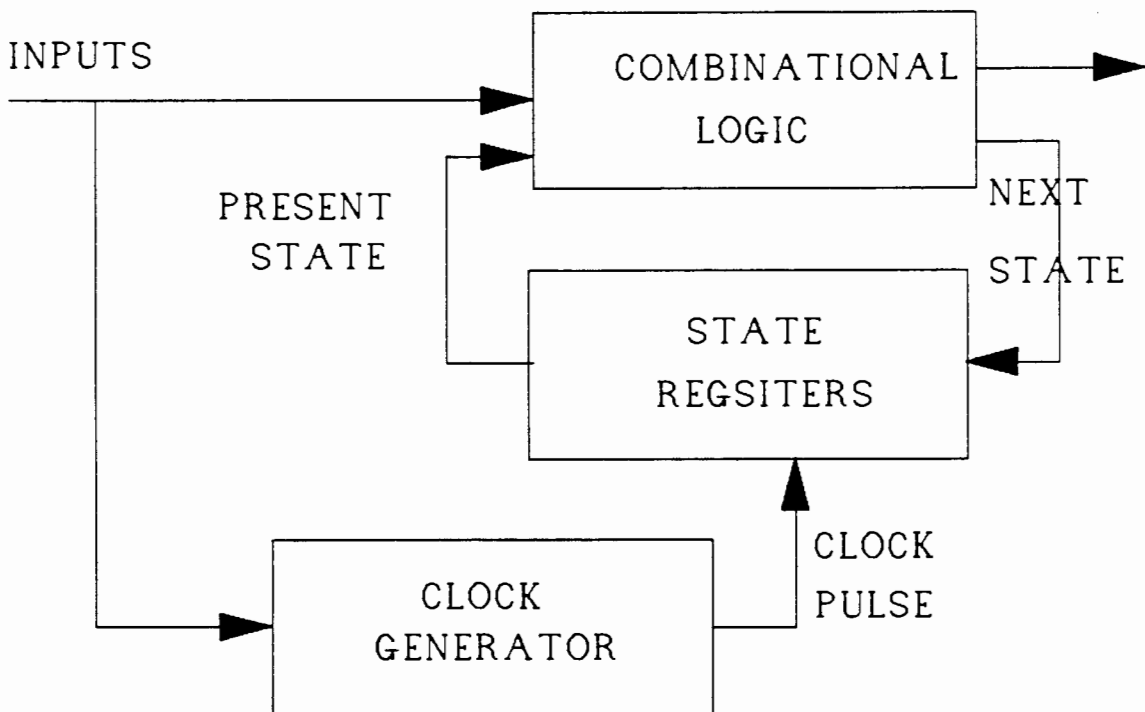


Figure 2.3 Self-synchronized machines

And for the MOC case machine, the following block is used.

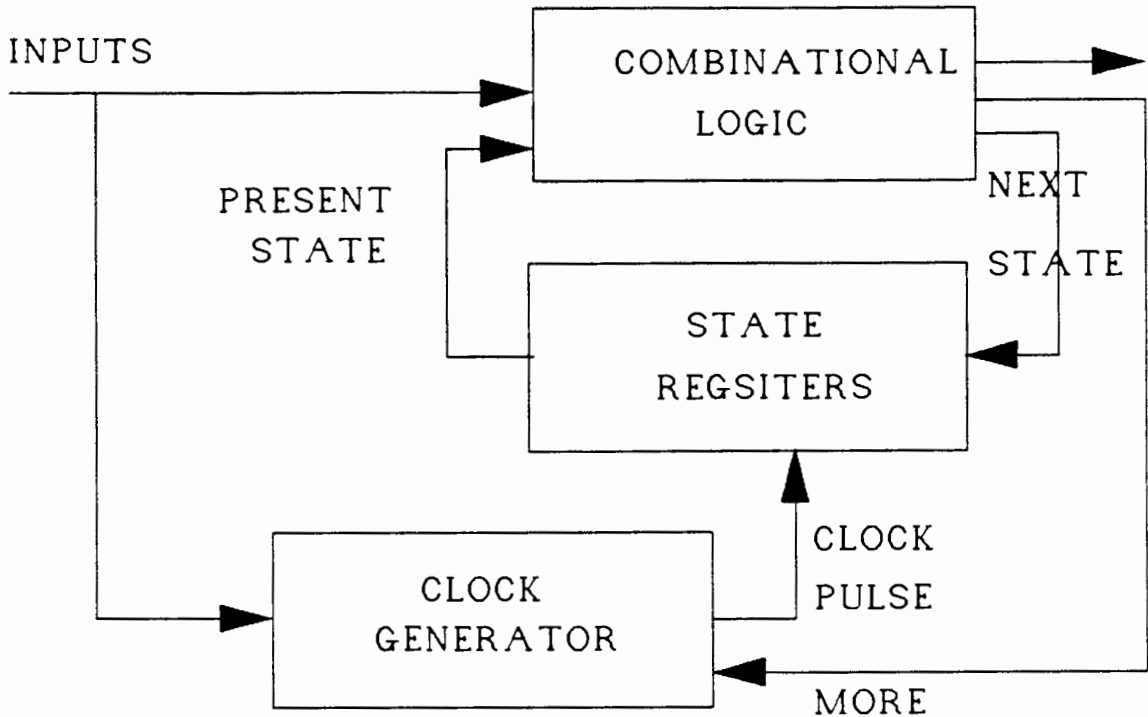


Figure 2.4 MOC machine

Notice that the MORE signal is added to tell the clock generator that more transitions are needed. The clock generator uses the state of MORE each time to generate an additional clock pulse. The signal MORE is produced by a combinational circuit which compares the total state of the machine before the clock with a predetermined final total state. If the states are not equal, MORE will be high. MORE is fed directly to a T flip-flop in the clock generator. So when the clock occurs, the output of the T flip-flop changes. This change will be captured in the change detector to generate another clock pulse. If MORE is low when the

clock occurs, then the sequence ends.

The only block that is different from the synchronous machines is the clock generator.

CLOCK GENERATOR BLOCK

The clock generator scheme presented here is detailed in Kirkpatrick [Kirkpatrick, 1986].

The clock generator consists of two blocks: the Change Detector and the Delay Element.

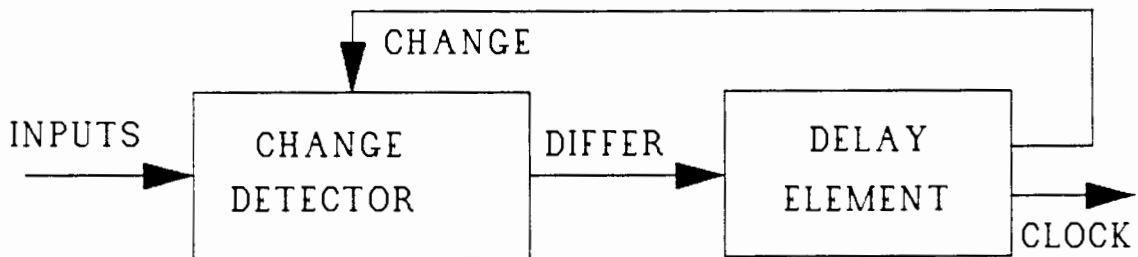


Figure 2.5 CLock generator block

The output of the change detector block is the signal DIFFER.

The outputs of the delay element block are the signals CHANGE and CLOCK.

The behaviour of the circuit is as follows:

- 1) DIFFER, CHANGE, and MORE are low. The change detector and the machine is ready to accept input changes.
- 2) If there is an input change, DIFFER will go high to

indicate a change in inputs has been detected.

3) After a predictable time later through the delay, it emerges as CHANGE. CHANGE is fed back to shut off the change detector. During this time, DIFFER is high and CHANGE is low, more input changes are allowable.

4) Eventually, DIFFER will go low but CHANGE is still high. At this time, changes combined with the present state travel through the combinational logic and setup to the state registers (flip-flops) as the next state condition. MORE is also updated at this time.

5) Lastly, through the delay again, CHANGE goes low (DIFFER = CHANGE = low), and CLOCK goes high to trigger the machine and reenable the machine again. (SOC case).

Note: in the MOC case, the signal MORE will cause more clock pulses so that the machine can perambulate through states until it finds the stable state. During the period of perambulation, the change detector is held off.

6) Now, the machine with DIFFER = CHANGE = MORE = low, it is ready for another input excitation.

CHANGE DETECTOR

The change detector circuit can be realized as shown below:

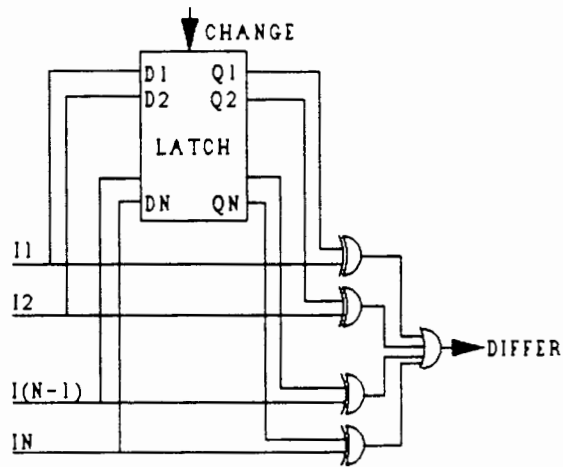


Figure 2.6 Change detector

First, the inputs $I\{1..n\}$ and the output of the latch are the same. Hence, DIFFER is inactive (low). Once, one or more inputs $I\{1..n\}$ change levels, the respective exclusive OR gate will detect the change and go high. DIFFER will follow them. Later, CHANGE is generated to open up the latch. Now, the change from the input propagates through the latch to the exclusive OR gates. Eventually, DIFFER goes low and CHANGE goes low again to shut off the latch. This completes a sequence of input detection.

One can build an eight input change detector with only two commercially available parts: one 74F373 eight-bit latch and one 74F521 eight-bit equality comparator.

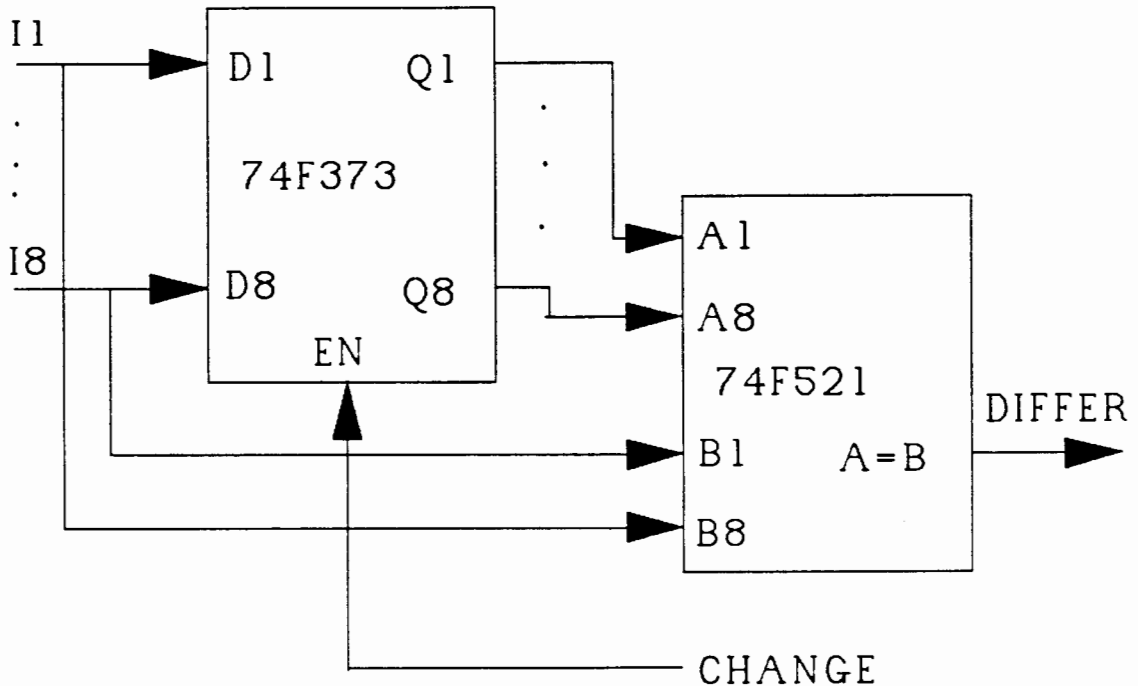


Figure 2.7 TTL implementation of change detector

DELAY BLOCK

SYMMETRICAL DELAY:

A symmetrical delay is a pure delay line where it transforms or shifts the input signal in time by amount D. This delay can be easily realized with gates in series or using available digital delay line.

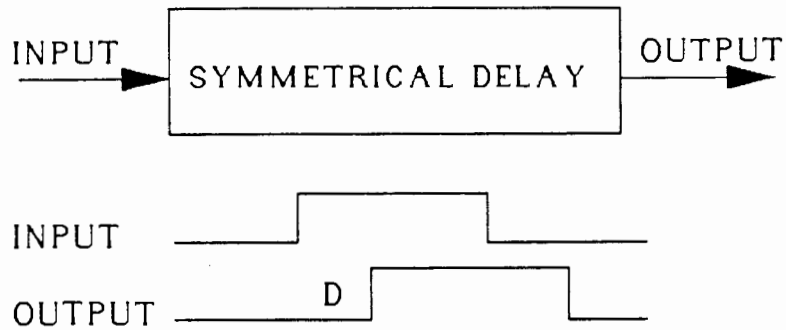


Figure 2.8 Symmetrical delay

ASYMMETRICAL DELAY:

An asymmetrical delay is a delay which the leading edge of the input change is delayed by amount D , but the trailing edge (opposite sense) is propagated without delay.

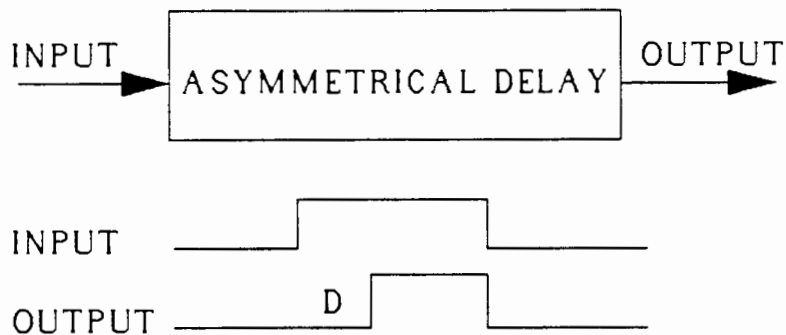


Figure 2.9 Asymmetrical delay

The asymmetrical delay can be realized as follow:

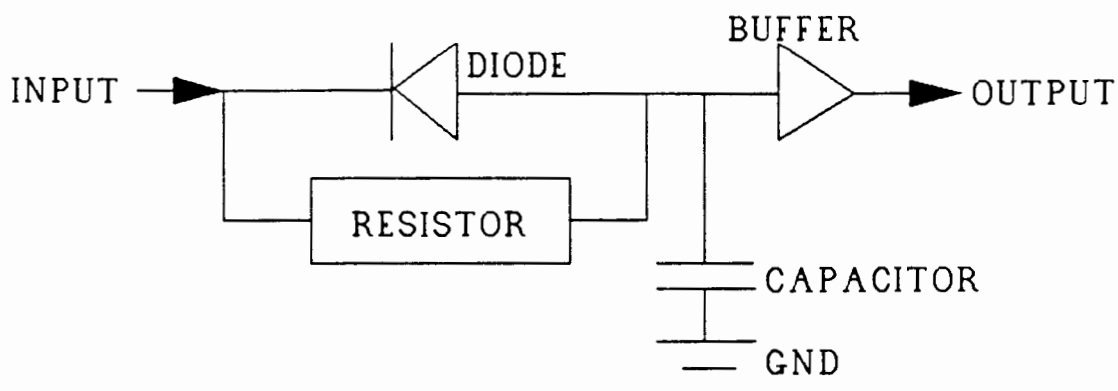


Figure 2.10 Realization of asymmetrical delay

Thus, the trailing edge speed is limited only by the technology. Different implementations are introduced in Kirkpatrick [Kirkpatrick, 1986].

FUNCTIONAL OPERATION

The operation of the self-synchronized circuits can be easily understood by studying the following timing diagram.

Notation:

STATE: <A> means the machine is ready to accept input changes.

 means the inputs have to remain stable for proper operation.

<k1> the time interval for which several input signals may change.

<k2> the time interval for which input signals may not change while the machine perambulates from one state to

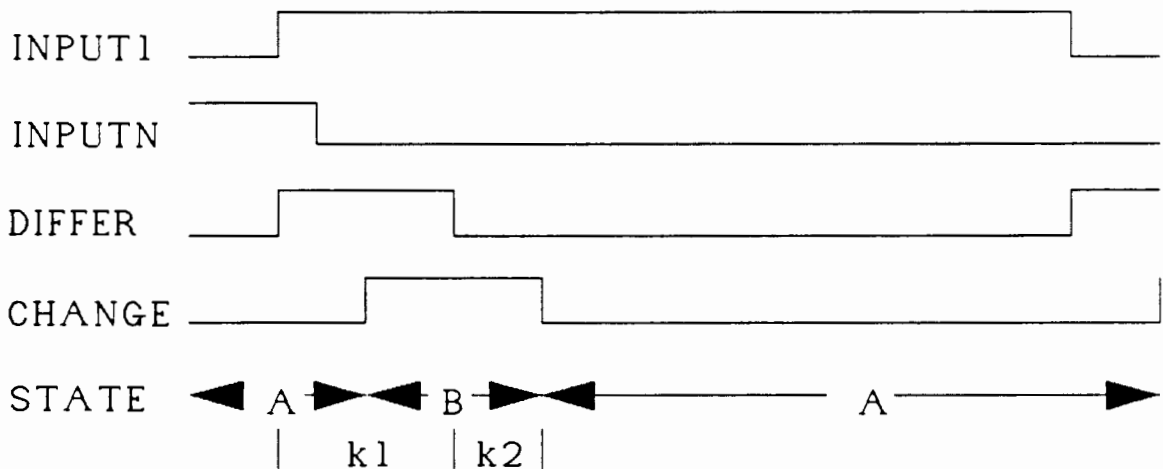
the goal state. If the input signals change during this interval, unpredictable behavior will occur. Hence, the machine may malfunction accordingly.

min = minimum.

max = maximum.

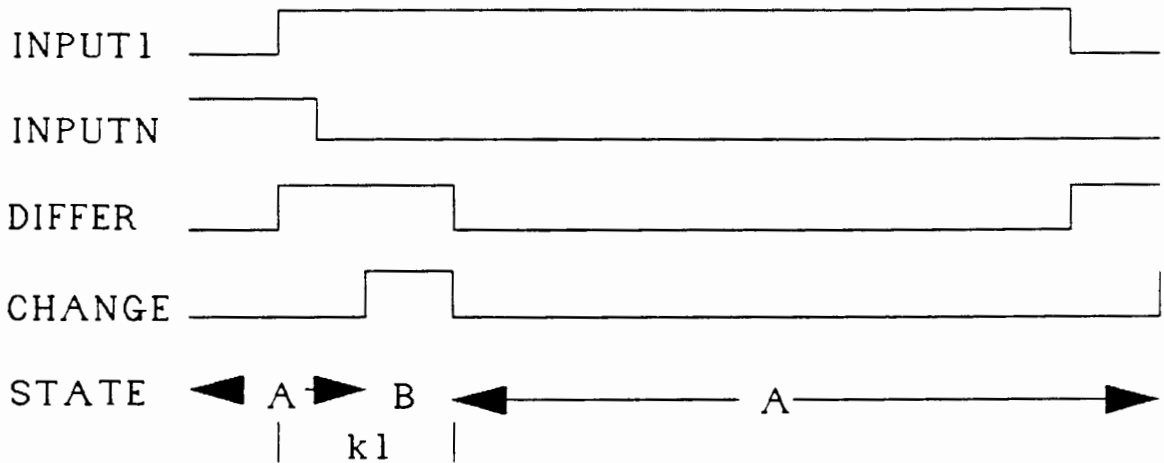
Dm = Delay element.

Case 1) Using symmetrical delays:



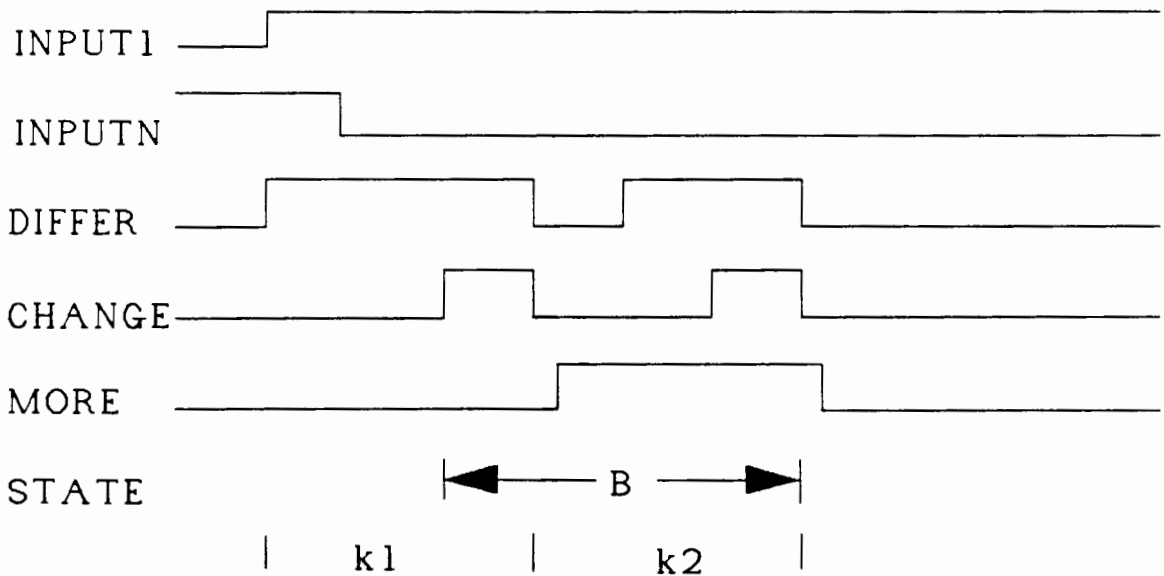
The problem we see with symmetrical delays is that unless we have the control of the inputs, otherwise, the machine may malfunction if the input changes during state . If input changes occur during state , the inputs may change to new state before the clock is generated to clock the flip-flop. Thus, the machine may enter a different state than it should be. In addition, The speed of the machine is also slower due to this type of delay.

Case 2) Using asymmetrical delays:



So we can minimize the problem mentioned above by using the asymmetrical delay elements. The speed of the circuit now is only limited by the chosen technology.

For the MOC case:



The signal MORE is high when the machine has not entered the final stable state.

TIMING ANALYSIS

The following notation will be used from now on to evaluate the speed of the machines.

D : delay through delay elements.

d : Stray delays through combinational logic.

s : set-up time for register elements (flip-flops).

f : propagation delays through register elements
(flip-flops).

k1: the time interval for which several input signals may change.

k2: the time interval for which input signals must remain stable.

min: minimum.

max: maximum.

Asynchronous Huffman-Moore Machines:

A MIC Huffman-Moore machine having a proper critical race-free state assignment will, in general, still require delay elements for proper operation. The earliest that an input change can reach output logic is d_{min} and the latest it can reach the output logic is $k1 + d_{max}$.

Thus the minimum valued for the delay element must be:

$$D_{min} \geq k1 + d_{max} - d_{min}.$$

Or to be safe:

$$D_{min} \geq k1 + d_{max};$$

Hence k_2 is bounded by $D_{\min} + d_{\min}$ and $D_{\max} + d_{\max}$.

For SOC case:

$$k_2 + d_{\min} \geq d_{\max} + (D_{\max} + d_{\max})$$

$$k_2 \geq D_{\max} + 2d_{\max} - d_{\min}.$$

This is the period that inputs have to remain stable after the change.

In the case of MOC, we have another restriction. The time that each state changes is bounded by $D_{\min} + d_{\min}$ and $D_{\max} + d_{\max}$. If n is the longest sequence of state transition in the machine to produce the output then

$$k_2 + d_{\min} \geq d_{\max} + n(D_{\max} + d_{\max})$$

$$\text{or } k_2 \geq nD_{\max} + (n+1)d_{\max} - d_{\min}.$$

and the time between states:

$$k_1 + k_2 \geq k_1 + n(D_{\max} + d_{\max}) + (d_{\max} - d_{\min}) \quad (1)$$

Special case for Huffman-Moore machine:

If the machine is in SOC mode and has no essential hazard, then $D = 0$. Thus,

$$k_2 \geq 2d_{\max} - d_{\min}. \quad (2)$$

Self-Synchronized Machines:

For the machine built using this structure, the clock edge to the register elements (flip-flop2) must not arrive before the input changes have gone through the combinational logic section, reached the state-variable flip-flops, and met the set-up time requirements. Thus,

$$D_{min} \geq k_1 + d_{max} + s$$

and similarly,

$$k_2 + D_{min} \geq D_{max} + f_{max} + d_{max} + s.$$

$$k_2 \geq f_{max} + d_{max} + s + (D_{max} - D_{min}).$$

and input changes are separated by:

$$k_1 + k_2 \geq k_1 + (f_{max} + d_{max} + s) + (D_{max} - D_{min}) \quad (3)$$

for MOC case:

$$k_1 + k_2 \geq k_1 + n(f_{max} + d_{max} + s) + (D_{max} - D_{min}) \quad (4)$$

By comparison between (2) and (3), the Huffman-Moore machine will always be faster if the machine operates in SOC and has no essential hazards. Otherwise, the combination circuit will dictate the speed of the circuit in the Huffman-Moore machines. The more complex the machine, the bigger the combination circuit due to the complicated state assignment to avoid races and hazards. This leads to larger k_1 . On the other hand, the state assignment in Self-Synchronized circuits can be arbitrary. Thus the combinational logic can be made much simpler. Consequently, the speed of the Self-Synchronized machines can be faster than that of Huffman-Moore.

Self synchronized circuit extended to Unrestricted Input Change (UIC) case:

Almost all asynchronous designs assume that the machine will operate in the fundamental mode - once the input-state change is perceived by the machine, the machine

will reach a final stable state before another input-state change is allowed. When the machine operates in UIC mode, the fundamental mode assumption is violated. Since the timing relationships between the inputs are not constrained, ambiguous input-state states will result. This may cause the machine to malfunction. As described in Kirkpatrick [Kirkpatrick, 1986], the extension to the UIC case is straight forward. All we have to do is to add a transparent latch like 74F373 to the input signals. While the machine is in a stable state, the latch is enabled. Thus, the machine is ready to accept input changes. Once, the machine detects new inputs via DIFFER going high, this input latch is disabled and freezing the input state in the latch. Next this input-state is processed and once the machine returns to the stable state, the input latch is again enabled to accept new input changes.

It should be noted that this UIC input latch will exhibit the metastable behavior due to the input changes not meeting the set-up and hold-time requirements for the latch. To compensate for this, an additional delay has to be added to k1 (normally four time the propagation delay of the latch). So the general structure of the UIC self synchronized machine would look like:

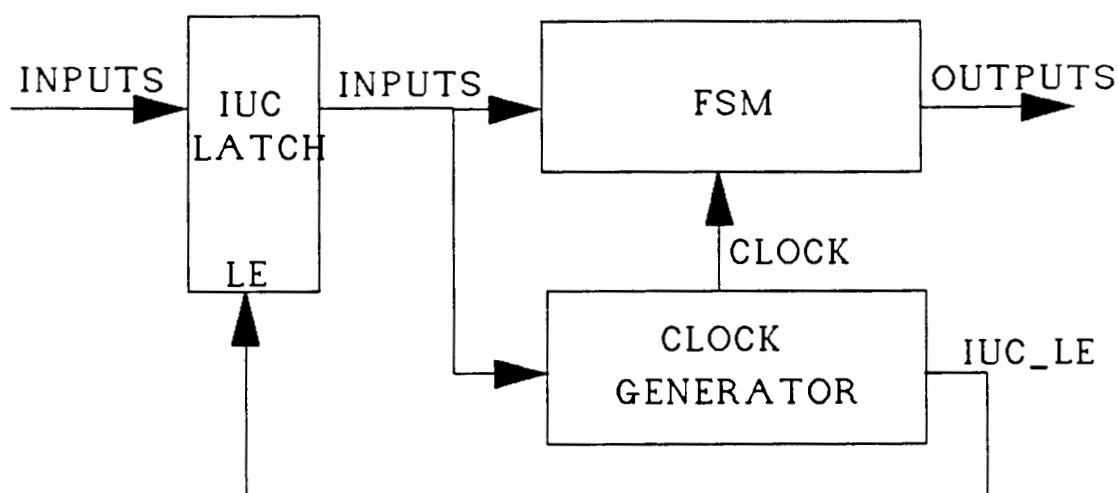


Figure 2.11. UIC machine

And the speed of the circuit is:

For SOC:

$$k_1 + k_2 \geq k_1 + (5f_{\max} + d_{\max} + s) + (D_{\max} - D_{\min}) \quad (5)$$

For MOC:

$$k_1 + k_2 \geq k_1 + n(5f_{\max} + d_{\max} + s) + (D_{\max} - D_{\min}) \quad (6)$$

COMPARISON BETWEEN SYNCHRONOUS, ASYNCHRONOUS, AND SELF-SYNCHRONIZED CIRCUITS

For the following example, assume we use 74FXX technology and also assume each FXX gate delay is 3ns, 10ns for minimum and maximum respectively. For the PAL 16L8B and 16R4B, the set-up time is 15ns, the clock to output time is 12ns, and the propagation delay time is 15ns.

Example 16: The Crumb Road Problem.

This problem is the design of a sequential machine to control the traffic at the intersection of Crumb Road and Route 1. (For a complete description of the problem, see Unger, 1969). Unger derived the following flow matrix.

X1 X2	0 0	0 1	1 1	0 0	y1 y2
1	1,0	2,0	4,0	1,0	0 0
2	2,0	2,0	3,0	3,1	0 1
3	1,0	2,0	3,1	3,1	1 0
4	2,0	2,0	4,0	4,0	1 1

And the circuit is as follows:

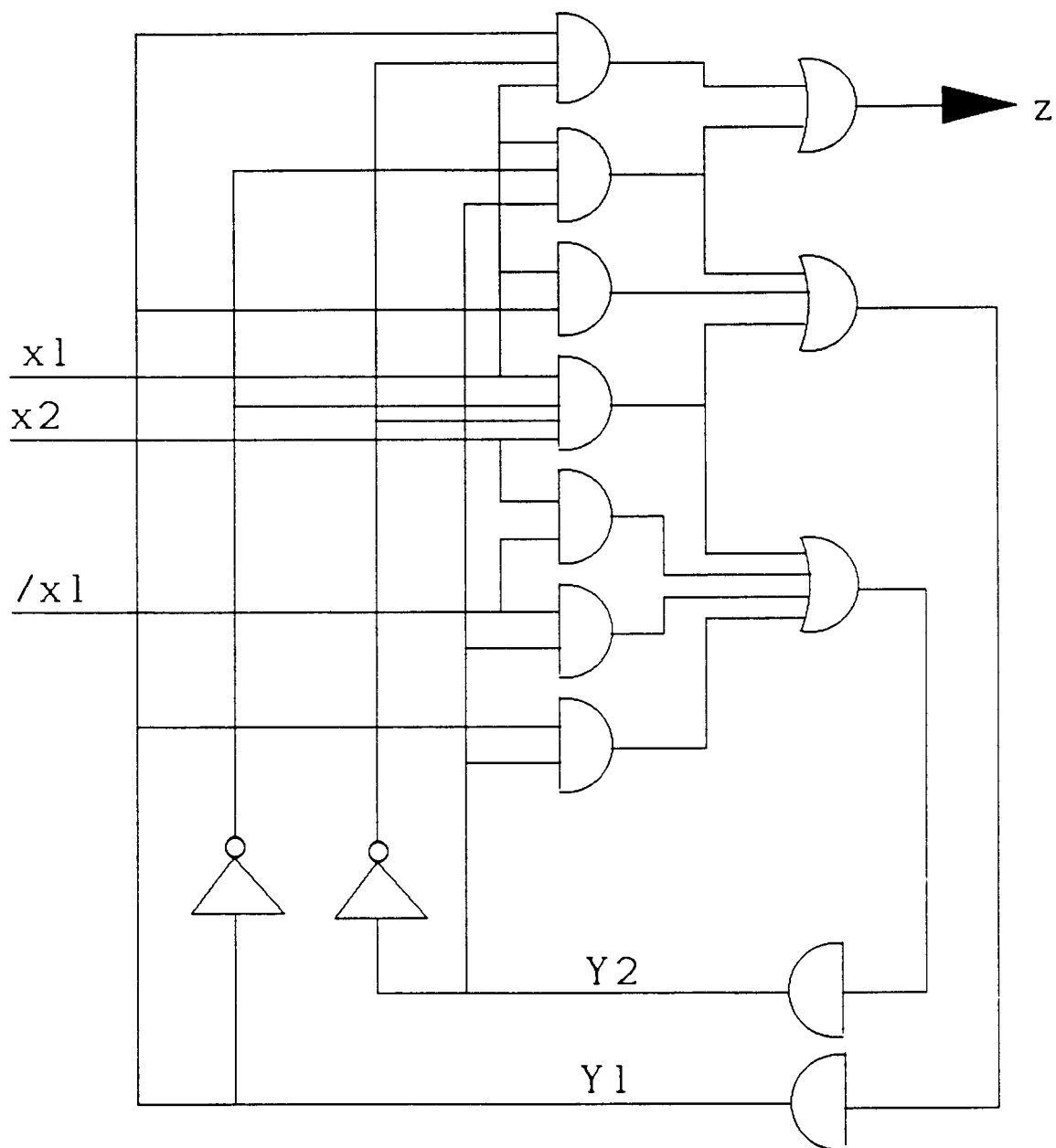


Figure 2.12. Crumb road problem

$$Z = x1./y1.y2 + x1.y1./y2$$

$$Y1 = x1.x2./y1./y2 + x1./y1.y2 + x1.y1$$

$$Y2 = x1./x2 + y1.y2 + /x1.y2 + x1.x2 + /y1./y2$$

Asynchronous machine:

Using PAL 16L8, the Huffman-Moore machine for this example would look like:

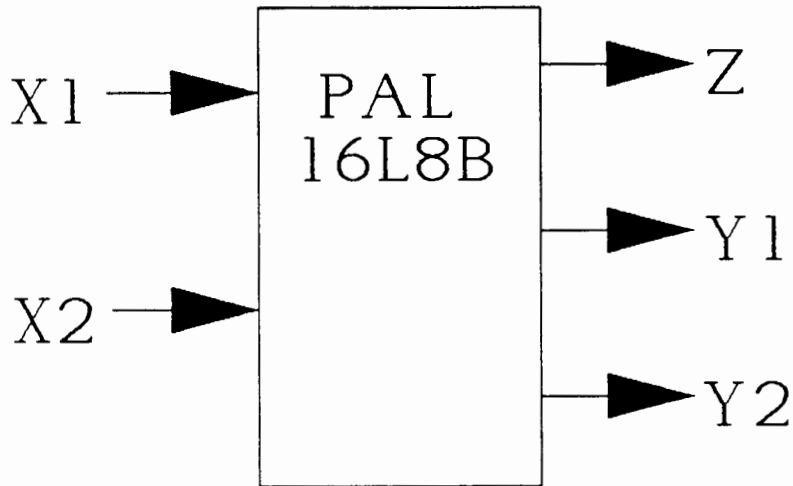


Figure 2.13. Asynchronnous circuit for crumb road problem

The speed of the Asynchronous machine = $TPAL16L8B = 15ns$
or 66.6 mhz.

Synchronous machine:

Using PAL 16R4B, the synchronous machine version of this example would look like:

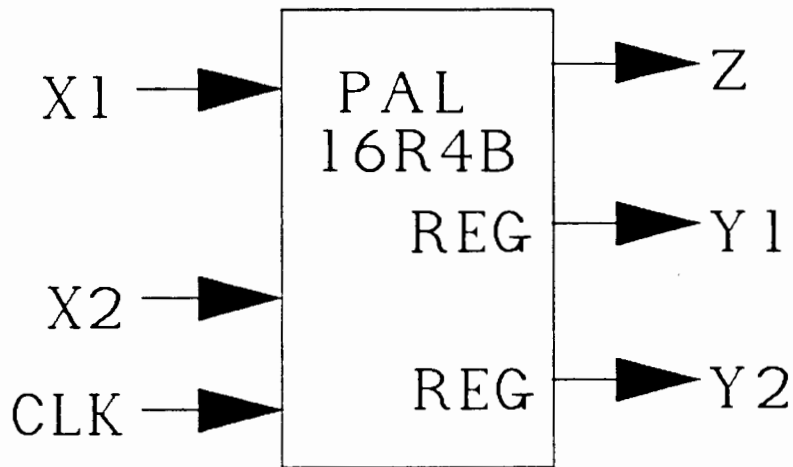


Figure 2.14. Synchronous circuit for crumb road problem

The maximum clock rate = $T_{\text{setup}} + T_{\text{clock-to-output}}$
 $= 15\text{ns} + 12\text{ns} = 27\text{ns}.$

So maximum speed = 27ns or 37 mhz.

Self-synchronized machine:

The circuit realization for the above problem is shown as follows:

delay to turn off CHANGE. Finally, the latch U1 is shut off and CLOCK goes high to clock the PAL 16R4B. Now, the state machine is ready for another input change.

Next we have to determine what is the delay line in the circuit before we can calculate its speed.

The worst case timing analysis is as follows. There are two paths in this circuit. Path 1, P1, is the inputs to the PAL 16R4B. The other path ,P2, is the inputs through the clock generator. The only constraint is that the input change has to arrive the PAL16R4B at least the minimum set-up time, 15ns, before the CLOCK is generated, going from low to high. Hence, the minimum delay through the clock generator block must be equal or greater than the set-up time requirement of the PAL. We have the following inequality.

$$t_{U2min} + t_{Dmin} + t_{U1min} + t_{U2min} + t_{Dmin} + t_{U3min} \geq t_{setup}$$

$$3 + t_{Dmin} + 3 + 3 + t_{Dmin} + 3 \geq 15$$

$$2t_{Dmin} \geq 3 \text{ ns}$$

$$\text{or } t_{Dmin} \geq 1.5\text{ns.}$$

(we can use a non-inverting buffer as the delay in this case).

Suppose, we use a F08 and gate as the delay in this example, then $t_{Dmin} = 3\text{ns}$. Then the speed of the circuit is:

$$\text{Speed} = 2t_{Dmin} + 2t_{U2min} + t_{U1min} + t_{U3min}$$

$$= 2*3 + 2*3 + 3 + 3$$

Speed = 18 ns or 55.5 mhz

So we can see that the self-synchronized circuit under this scheme of implementation is faster than that of the synchronous circuit about 33%.

Asynchronous Huffman-Moore machine = 66.6 mhz.

Self-Synchronized machine = 55.5 mhz.

Synchronous machine = 37 mhz.

For the UIC case:

The UIC latch is added to the self-synchronized circuit and a synchronizer has to be added to the synchronous machine. The speed difference will be less apparent because the self-synchronized circuit will be slower by the extra UIC latch plus the compensation for metastability. On the other hand, the synchronous machine has to wait for an extra clock to synchronize the inputs.

With the above example, the realization for the IUC case is as follows:

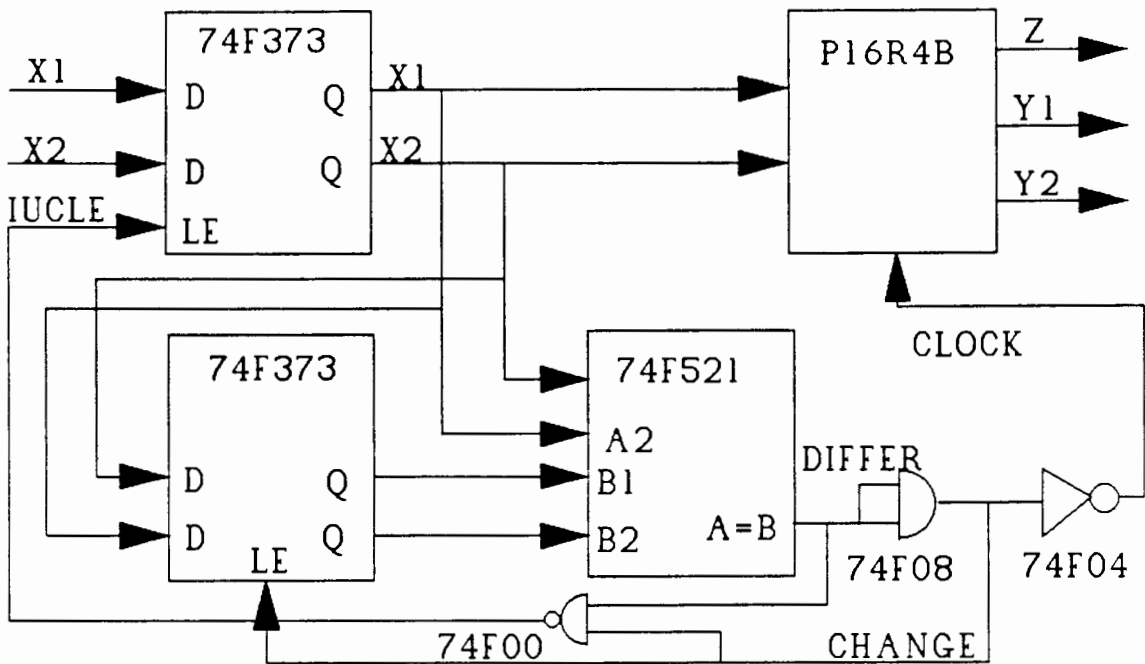


Figure 2.19. UIC case for crumb road problem

As mentioned above, the IUC latch may exhibit the metastable condition, we allow $4 T_{pd}$ to allow the latch to recover. Thus the speed is:

$$\text{Speed} = 2t_{Dmin} + 2t_{U2min} + t_{U1min} + t_{U3min} + T_{UIClatch}$$

$$= 2*3 + 2*3 + 3 + 3 + 40$$

$$\text{Speed} = 58 \text{ ns or } 17.24 \text{ mhz}$$

For the synchronous machine, the metastable problem also has to be taken into account. Hence,

$$\text{Speed} = 27 + 40 = 67 \text{ ns or } 14.9 \text{ mhz.}$$

CONCLUSION

This chapter has shown that the self-synchronized circuits can be designed using commercially available PALs or PLDs and TTL parts. It also shows that the self-synchronized circuits are faster than those of the synchronous circuits when implementing with PALs. The biggest advantage here is that the methods of state assignments and logic reduction of synchronous machines are preserved while the speed can be improved.

The ideas of self-synchronized circuits are not new. However, they were not used very much. Recently, there is a trend for this design style. Kirkpatrick has used this style in the design of Tektronix DAS 9200 Logic Analyzer in 1986 and also in 1987, a Japanese Semiconductor Company introduced Self-timed RAM. I think that this is still a good field to do further research. With respect to PALs or PLDs, there are still a lot of extra components, 5 extra chips, besides the PAL needed to implement a Self-Synchronized circuit. I would like to propose the idea to design a front end chip, CLOCK GENERATOR, so that we can build the Self-Synchronized circuit with only three components: Clock generator, PALs, and a resistor and a capacitor. The pair of resistor and capacitor will set the time delay. The asymmetrical delay element and the UIC mode if selected will

be taken care by this clock generator chip. This chip is fairly small and should be a good project for the VLSI class.

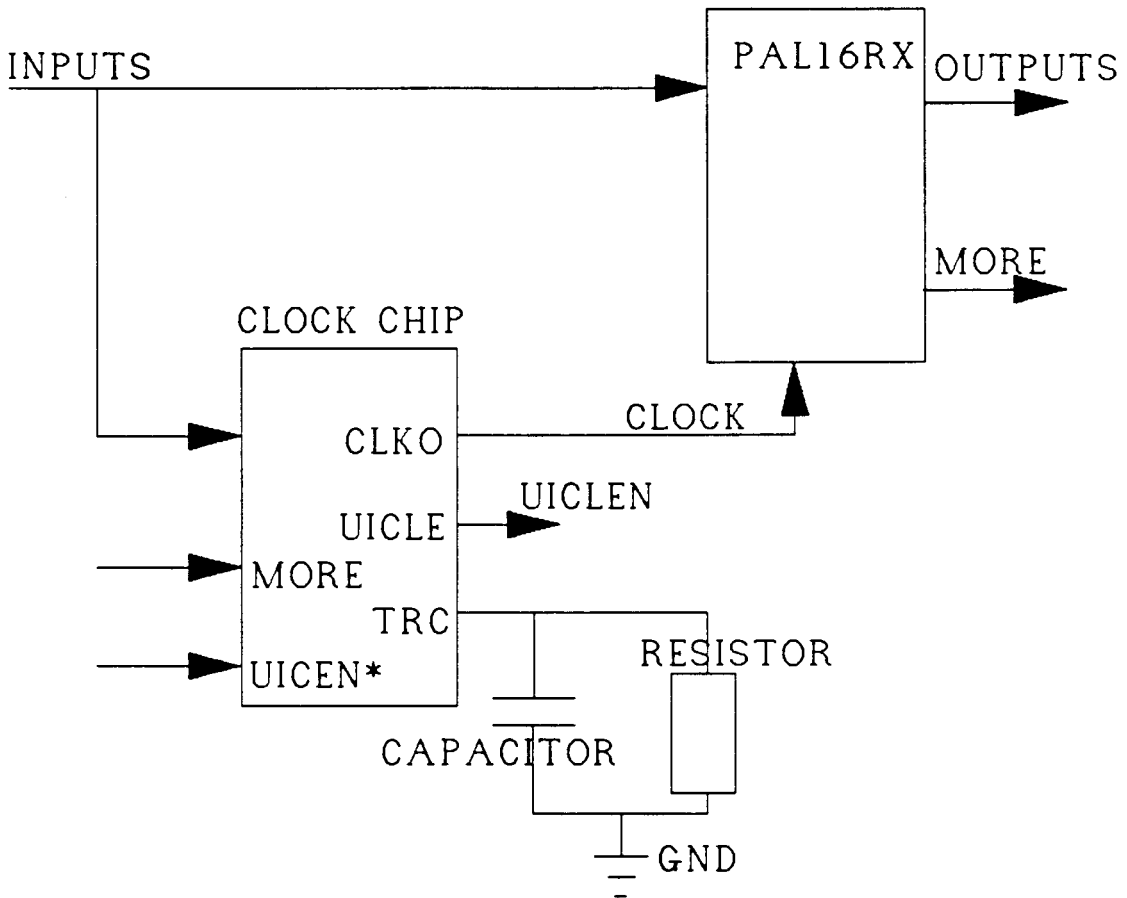


Figure 2.17. Front end chip

CHAPTER III

INTERNAL STATE ASSIGNMENT FOR FINITE STATE MACHINES USING PLDs

INTRODUCTION

The following constraints must be taken into account when designing state machines using PALs or PLDs. (From now on, the term PLDs will be used for both PALs and PLDs)

1) Most of the commercial registered PLDs implement only D-type flip-flop. This type is still the most popular among high speed PLDs.

2) For the 20 and 24 pin PLDs, there are at most 8 registered outputs. Hence, this will limit how big the finite state machine can be.

3) Each D-input of the above eight registered outputs has at most eight products in the sum term. This condition will severely limit the design.

4) The number of inputs is limited to 21 and it is found adequate.

From these restrictions, only small and medium state machines can be designed using PLDs. From my personal experience, state machines of less than 15 inputs and 8 states are frequently encountered. In addition, each machine

normally has more than one output. It is then obvious to see that the output pins are scarce resources in a PLD. As a consequence, the outputs of the machines are normally encoded in the state variables to save I/O pins for extra functions (either for output or input). With this design style, the designer often knows the minimum number of flip-flops that are to be used in the design in advance. All that he needs is a method to assign the binary code to state variables such that the excitation functions described by the Boolean equations will fit into the device. At the moment, there are some CAD tools to do the automatic state-assignment. However, these tools try to minimize the number of flip-flops in the design rather than the excitation functions [KISS by Michelli] and [STASH in Logmin]. The author has not seen and does not know of any CAD tool which minimizes the excitation functions for PALs or PLDs yet on the market. Therefore, he would like to show some set of heuristic rules which are based on his personal experience with a hope that some future student who will be designing such a system may take them into account.

BASIC DEFINITION

FSM : Finite State Machine.

ASM chart: a flow chart method to represent the state transition of a FSM.

Bubble Diagram: A method to represent the state transition of a FSM. States are represented in a circle and the transitions are represented by arrows going out or going in to the state.

X and $\neg X$: variable X and the complement of X respectively.

STATE ASSIGNMENT

The procedure for designing a two level AND-OR Finite State Machine can be summarized as follows:

- 1) Formulate the problem using:
 - Bubble Diagram
 - ASM chart
 - Karnaugh Maps
- 2) State Reduction: find minimum number of flip-flop needed. This step is not needed in many cases for PAL based designs.
- 3) State Assignment: assign binary code to the state variables. This step is very important. A bad state assignment will cause a more complex excitation function, more expensive to build and less reliable due to more power consumption.
- 4) Minimization of excitation functions: using PALMINI, Espresso, or others.

For a PAL based design, the method can be summarized as follows:

Begin

Step1: Formulate the problem.

Step2: State reduction.

While (the excitation functions do not fit the device and the possibility of state assignment has not been exhausted)
do

begin

Step3: State Assignment.

Step4: Minimization of excitation functions.

end while;

End.

step5: if the design does not fit the device, then show the best solution. At this point, the designer has the following options:

1) Combine output pin of PALs together to increase the product of sum terms for the excitation function.

2) Partition the design into smaller machines.

3) Go to a bigger device like Gate Array for example.

The rest of this chapter will only address step3 and step5 described above.

STEP 3: HEURISTIC RULES FOR STATE ASSIGNMENT.

As mentioned earlier, the number of product terms for a registered output PAL is very limited (only 8 terms). Hence, the excitation input equations frequently exceed the limit imposed by PAL architecture. So, the method to assign binary codes to states is very important because the complexity of excitation equations and the number of product terms in particular are the direct result of the state assignment. So, we would like to have a method that will always produce an optimum solution.

Basically, there are two classes of designs.

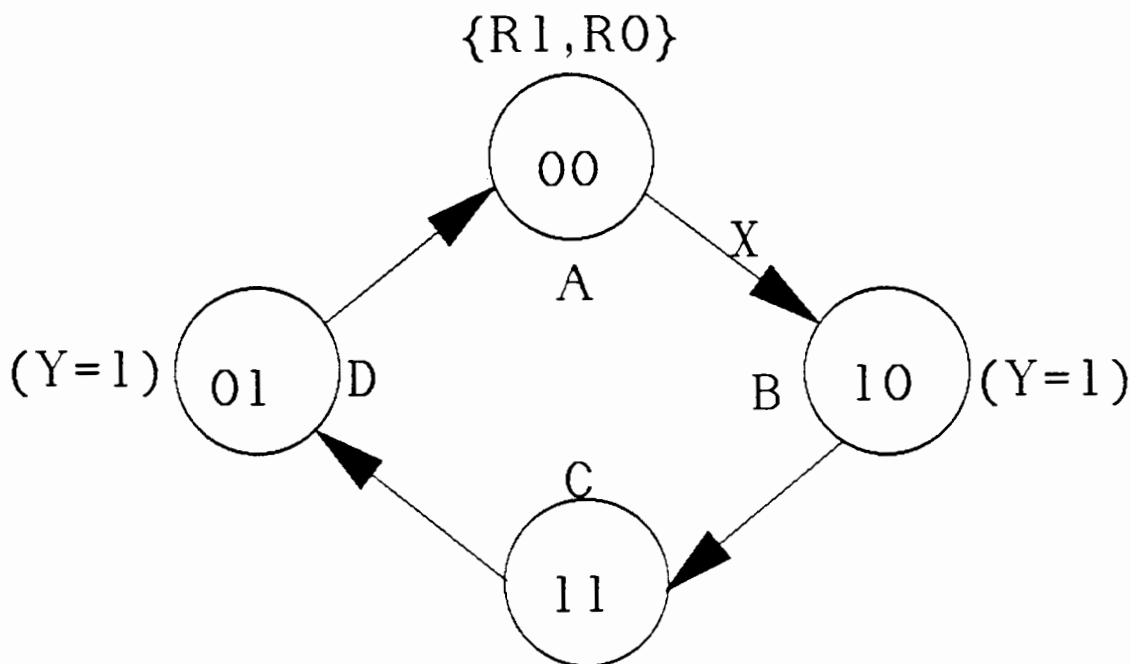
A) The outputs are separate from the state variables.

- Outputs are functions of inputs and state variables. (Mealy machines).
- Outputs are functions of only state variables. (Moore machines).

B) The outputs are encoded as state variables.

(Moore machines).

In class B), the designer has less freedom to perform the state assignment than in class A) because the output signals' polarity dictates the state assignment.

Example 3.1:Figure 3.1. Combinatorial output

State Variable = $V = \{R1, R0\}$

Output $y = R1./R0 + /R1.R0$

This design takes 3 output pins.

Whereas

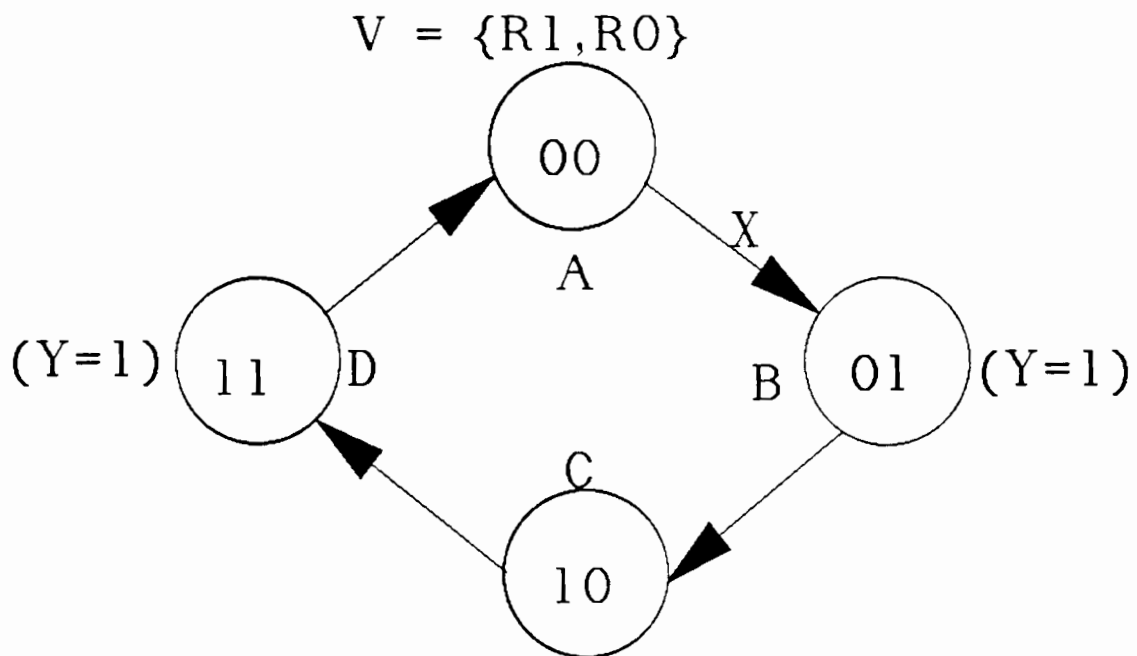


Figure 3.2. Registered output

Output $y = R0$.

This design takes only 2 output pins. However, in this scheme, the state variable $R0$ in state B and state D is dictated by the polarity of the output y .

The following is a set of heuristic rules which will attempt to minimize the excitation function for the state assignment.

Definition 3.1: Definition of COSTON, COSTOFF.

Let set V is the set which contains the state variable assignment and

$$V_i = (0,1) \text{ for all } V_i \in V$$

Where subscript i = state variable i

subscript n = current state.

X = set of branching conditions. ie $A = (X, XY, Z)$.

Thus $|A| = 3$.

COSTON:

If $V_{in} = 0$, then COSTON = 0

If $V_{in} = 1$, then COSTON = number of product terms going into the state plus the number of product terms looping in that state. The set D in figure 3.3 is considered to be the set of looping product terms for that state.

COSTOFF:

If $V_{in} = 0$, then COSTOFF = 0.

If $V_{in} = 1$, then COSTOFF = number of product terms going out of the state.

Example 3.2:

$$\text{COSTON} = |A + B + C + D| = 4$$

$$\text{COSTOFF} = E = 1$$

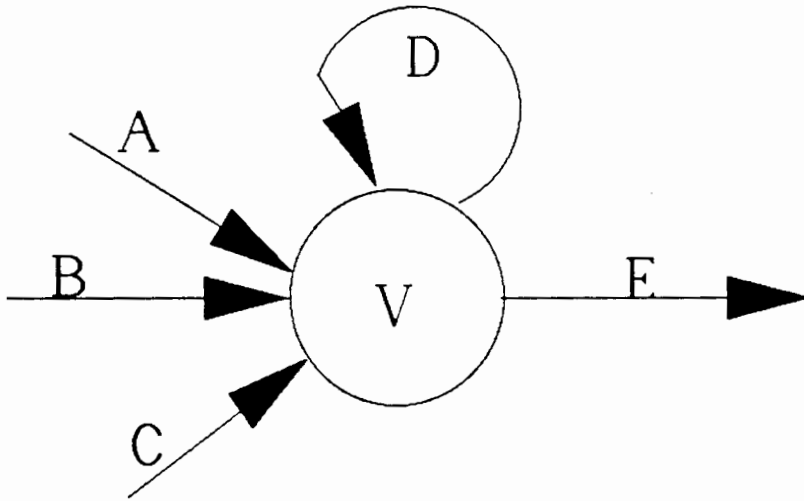


Figure 3.3. Coston and costoff

Note that: $E = \text{complement of } D$. Otherwise, the transition from state n to next state $n+1$ would be not deterministic.

Example 3.3:

The transition function for state B is shown below:

State Variables: $V = \{V_2, V_1, V_0\}$

The variables: X, Y, K , and Z are input variables and they constitute the branching conditions.

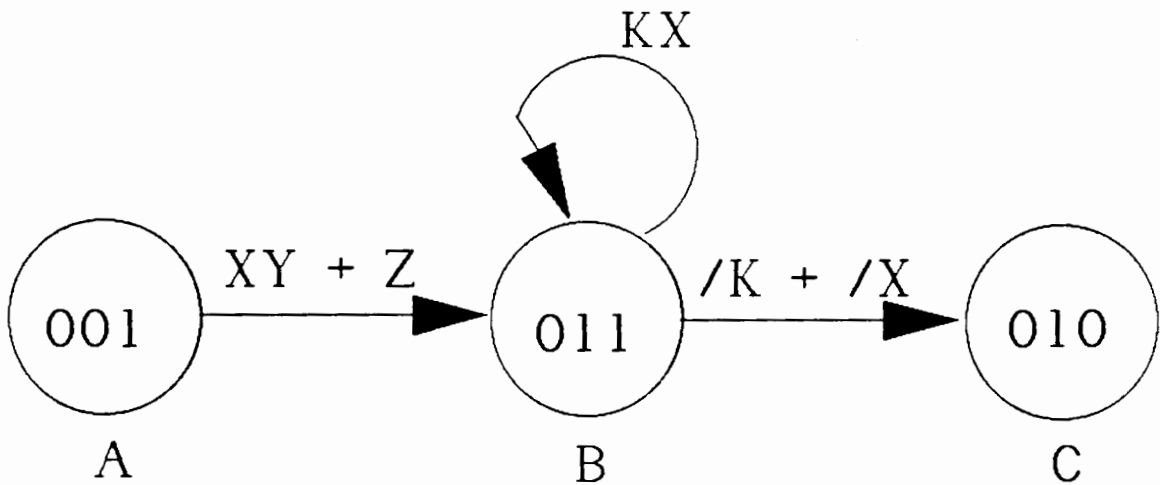


Figure 3.4. Coston and costoff calculation

State Vars	COSTON	COSTOFF
v_2	0	0
v_1	3	2
v_0	3	2

Implication of COSTON and COSTOFF:

The COSTON and COSTOFF together determine the number of product terms that we have to write for the state variable under consideration when the state machine transits from the current state to the next state.

Method for writing equations directly from the flow chart.

For the D-type flip-flop, the transition table is as follows:

Table I
TRANSITION TABLE OF D-FLIP-FLOP

D\Q	0	1
0	0	0
1	1	1

The following rules apply:

- 1) If $V_{in} = 0$ and $V_{in+1} = 0$, then no equation is needed. It is a free transition.
- 2) If $V_{in} = 1$ and $V_{in+1} = 0$, and there is no looping back at V_{in} , then no equation is needed. It is a free transition.
- 3) If $V_{in} = 1$ or 0 and $V_{in+1} = 1$, then equation is needed. The number of product terms depends on the input set.

Example 3.4:

Write the transition equation for state J:

$$V = \{V_3, V_2, V_1, V_0\}$$

State I = 0101.

State J = 0011 = next state.

Branching condition = $\{xy + \bar{z}\}$

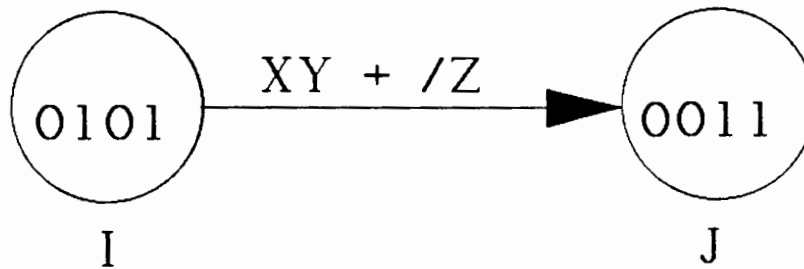


Figure 3.5. Transistion equations

Equation for state J:

For V_3 = none, cost = 0

For V_2 = none, cost = 0

For V_1 = $(0101) * (XY + /Z)$ = two terms, cost = 2.

For V_0 = $(0101) * (XY + /Z)$ = two terms, cost = 2.

RULE 1:

Find the state which has the greatest COSTON, then assign as many zero bits as possible to the state variables. This is called the Hot Code Assignment.

Note: for any FSM, the reset signal is needed to reset the FSM to a known state on the power up or during the reset condition. Thus, the reset state normally has the highest

COSTON and is assigned binary code 0.

There is a method which can bring the FSM to a known state without using the reset signal. This is achieved by assigning all of the unused states to branch to a selected state in the state diagram.

Example 3.5:

Consider the following two bit up counter. When the input x is high, the counter will count up. To be able to control the counter, we introduce the signal reset to bring it to the known state A during reset. Thus, at every state, the counter will enter state A and stay there until the reset signal is removed. The cost of state A in this example is thus 5 and is the highest cost. So, to optimize the excitation function for this example, we assign state A to be 00.

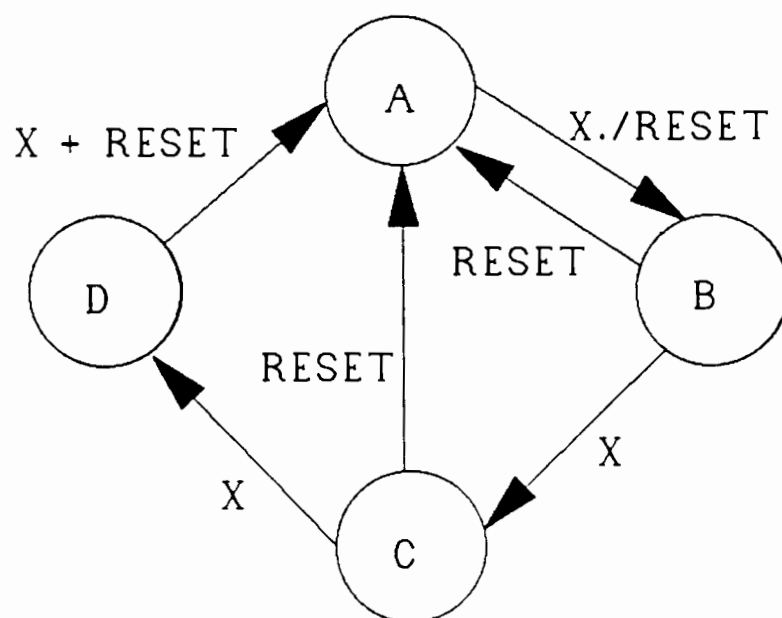


Figure 3.6. Rule 1.

Normally the reset is shown as follows:

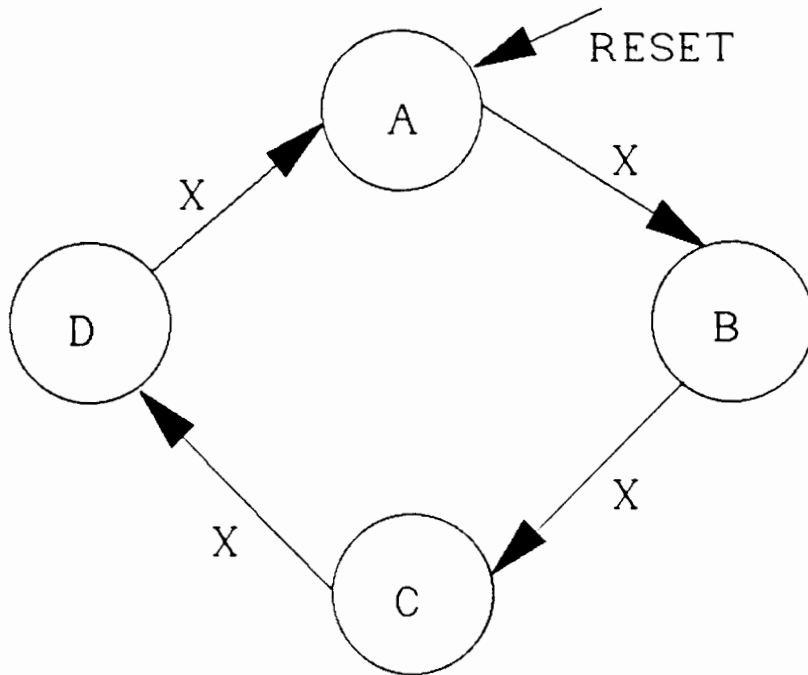


Figure 3.7. Reset signal

Rule 2:

If there is a transition from state SA to state SB , and the state variable V_i in state SA is already assigned to be 1, and there is looping condition in state SA , then assign V_i in state SB to be 1 if possible.

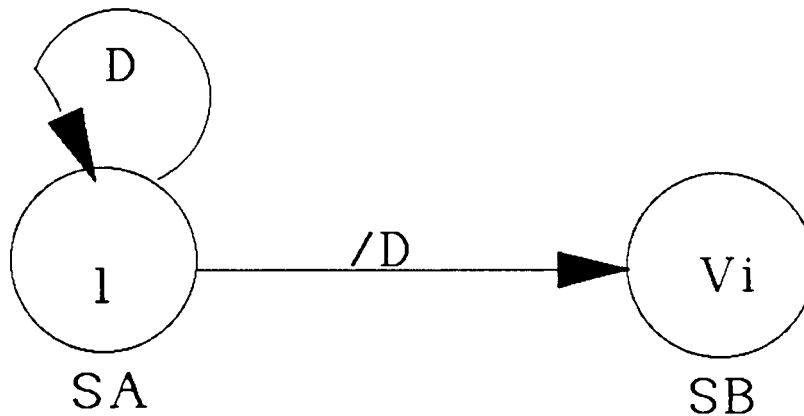


Figure 3.8. Rule 2.

If in state SA, $V_i = 1$, then assign 1 to V_i in state SB. Hence, COSTON of V_i in state SB = 1 since, COSTON of V_i in state SB = SA * (|D + /D|) = SA * (1) = SA = one term.

Rule 3:

When there is a transition from state SA to state SB and there is no looping condition in SA, assign 0 to V_i in state SB to achieve a free transition.

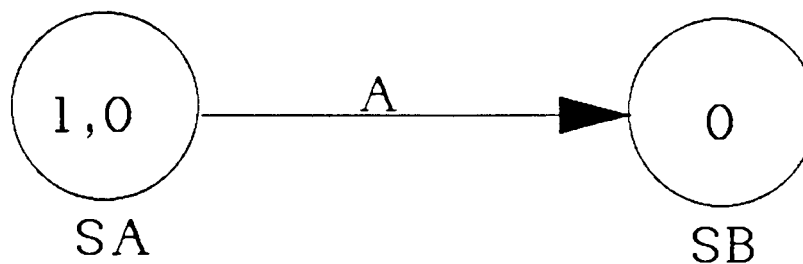


Figure 3.9. Rule 3.

Note: Rule 3 will give better result than that of Grey Code assignment. However, one has to pay attention to the combinatorial outputs of the state machine because since the state assignment are not Grey coded, the output may glitch due to more than one variables are changing and their delays are not equal.

We have introduced 3 rules which should be used in doing the state assignment. Note that the number of times that symbol 0 or 1 that one can assign to any variable is limited by the number of flip-flops used in the design. So for some machines, in order to fit the device when using the above rules, more state variables need to be introduced.

OUTPUT CONSIDERATION

The outputs of FSM can be registered outputs or combinational outputs. In the latter case, it can be in the Moore or Mealy machine form. This type of outputs required the Grey Code assignment (only one variable changes per any state transition) or the consensus must be added to avoid glitches (static hazards). In the first case, the outputs are clocked. Therefore, glitches will not occur. In addition, registered outputs are faster than that of combinational outputs by a tpd (15 ns if B-PAL type is used); and 15 ns is a lot of time in a high speed design.

Observation:

- The two schemes occupy the same number of pinouts.
- Registered output is more reliable due to no glitching.
- Registered output is faster.

The following is a complete example of a DRAM BUS INTERFACE design. The first part will illustrate the result of the Grey Code assignment. The second part will show the result of using the above rules.

Example 3.6:

The state diagram shown in Figure 3.10 is encoded using Grey Code. The Boolean equation version (the output from LOGMIN) is given in the next page. We observe that:

Variable R2 has two terms.

Variable R1 has four terms.

Variable R0 has six terms.

The state diagram shown in Figure 3.11 is encoded using the above rules. The Boolean equation version is given in the following page. We also observe the following:

Variable R2 has four terms.

Variable R1 has four terms.

Variable R0 has two terms.

The result has shown that by using above rules we have achieved a better solution compared to that of Grey Code assignment method for this example. In fact, after years of experience, my colleagues and I have used the above rules

almost every cases and every time the result is either equal or better when compared to results from STASH (a CAD tool of INTEL which does heuristic state assignment).

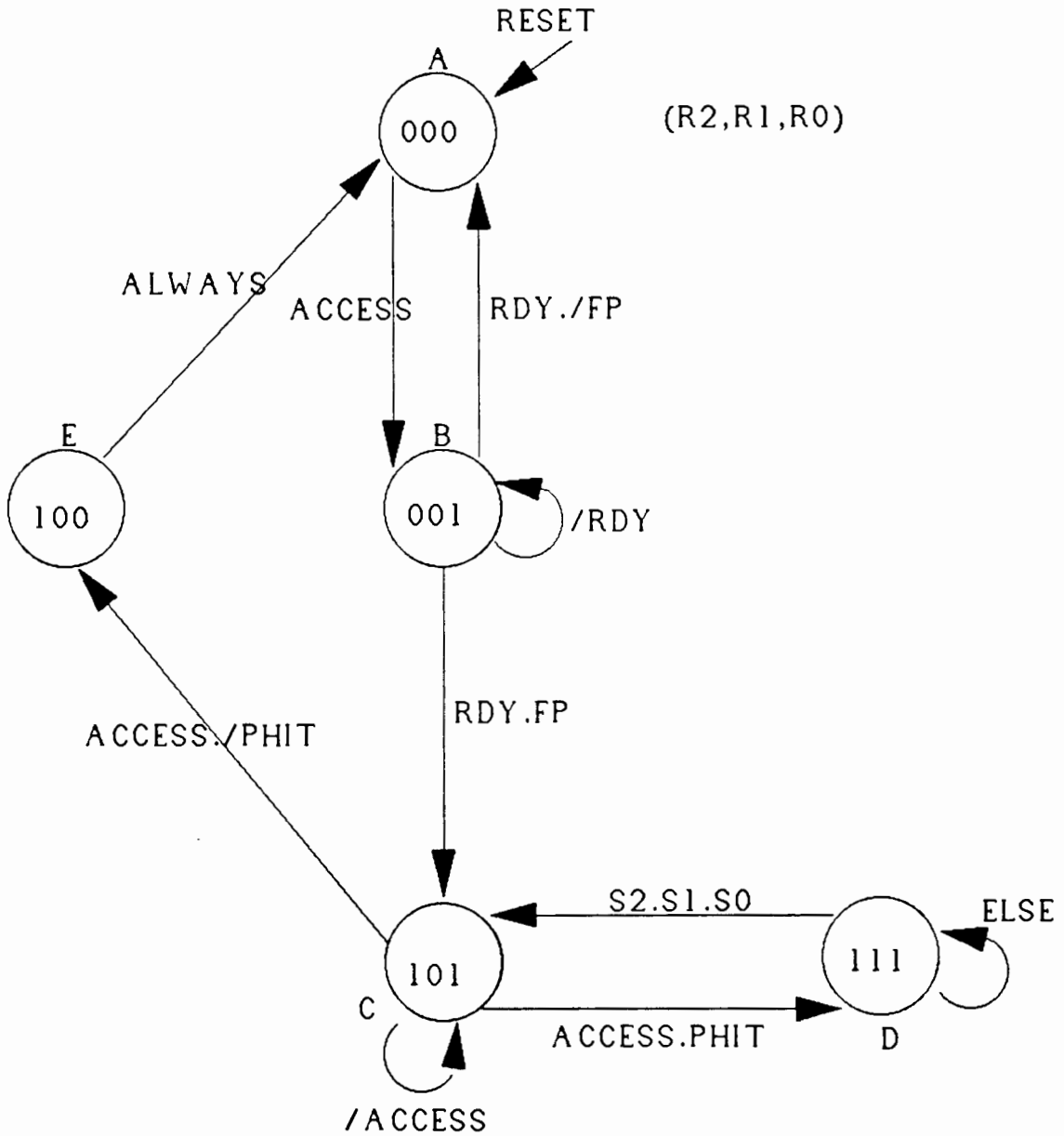


Figure 3.10. Grey code assignment

PAL: ESPTEST

Intel Corporation

RESET ACCESS PHIT FP RDY S2 S1 S0

/R2 /R1 /R0

$$R2 := \overline{\text{RESET}} * \text{FP} * \text{RDY} * \overline{R2} * \overline{R1} * R0 \\ + \overline{\text{RESET}} * R2 * R0$$

$$R1 := \overline{\text{RESET}} * \text{ACCESS} * \text{PHIT} * R2 * \overline{R1} * R0 \\ + \overline{\text{RESET}} * \overline{S0} * R2 * R1 * R0 \\ + \overline{\text{RESET}} * \overline{S1} * R2 * R1 * R0 \\ + \overline{\text{RESET}} * \overline{S2} * R2 * R1 * R0$$

$$R0 := \overline{\text{RESET}} * \text{ACCESS} * \overline{R2} * \overline{R1} * \overline{R0} \\ + \overline{\text{RESET}} * \text{FP} * \text{RDY} * \overline{R2} * \overline{R1} * R0 \\ + \overline{\text{RESET}} * \overline{\text{RDY}} * \overline{R2} * \overline{R1} * R0 \\ + \overline{\text{RESET}} * \text{ACCESS} * \text{PHIT} * R2 * \overline{R1} * R0 \\ + \overline{\text{RESET}} * \overline{\text{ACCESS}} * R2 * \overline{R1} * R0 \\ + \overline{\text{RESET}} * R2 * R1 * R0$$

DESCRIPTION:

PAL ESPTEST = [PLA ESPRESSO REDUCED FROM @ TEST]

Number of Inputs: 8

Number of Outputs: 4

Largest Number of Inputs for a Minterm: 6

Largest Number of Minterms for an Outputs:5

Line Count:

27

STATE ASSIGNMENT METHOD: GREY CODE

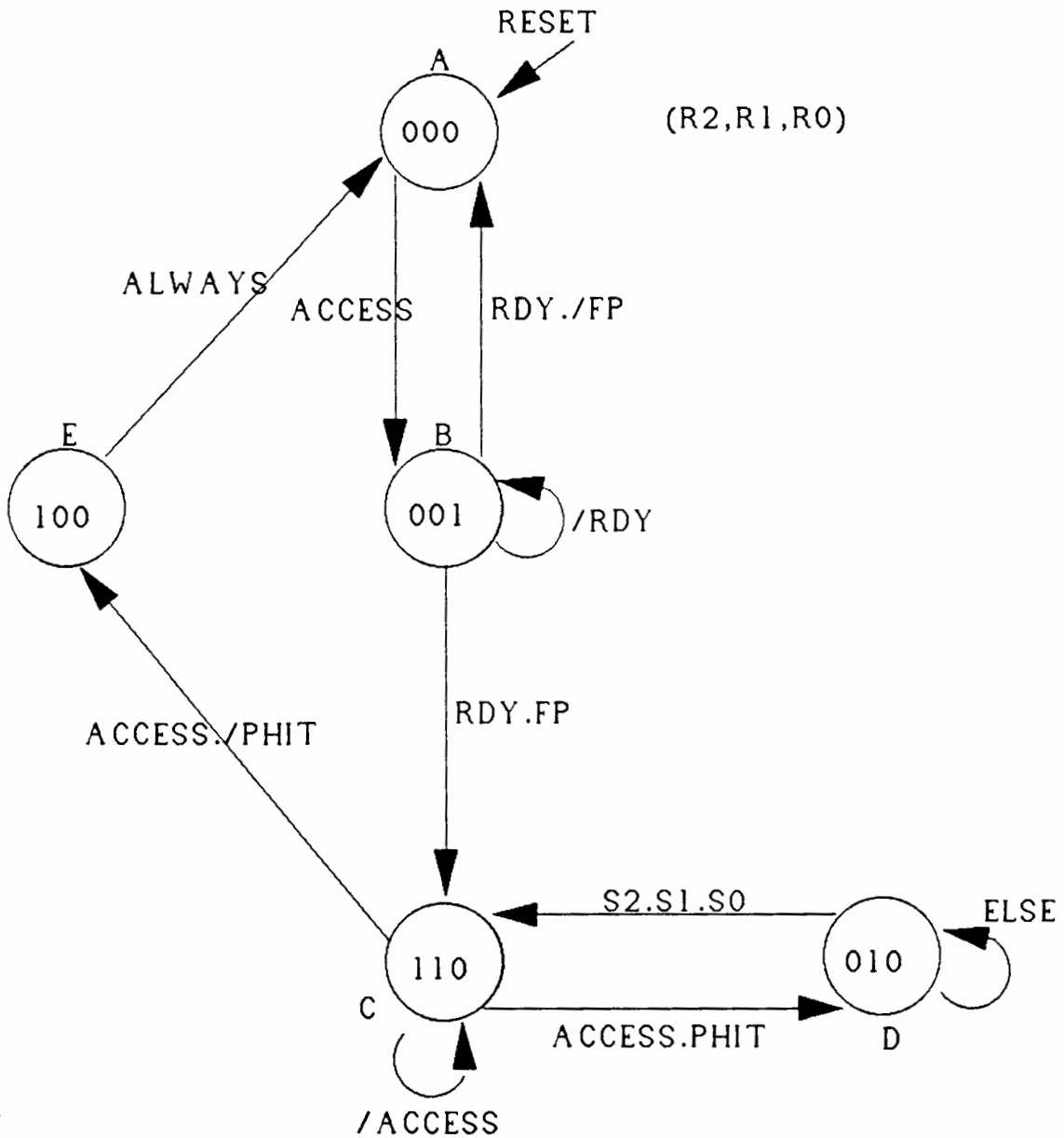


Figure 3.11. Rule based assignment

The COSTON and COSTOFF are found as belows:

State	COSTON	COSTOFF	TOTAL	CODE
A	5	1	6	000
B	2	1	3	001
C	2	1	3	110
D	4	1	5	010
E	1	1	2	100

COSTON of state A = 5 due to the RESET signal.

COSTON of state D = 4 due to the inversion of S2.S1.S0 and the transition of ACC.PHIT into the node.

COSTOFF of state B = 1 because $RDY.(FP + /FP) = RDY$.

- So by RULE 1, 000 is assigned to state A because it has the highest cost.

- Next node D is considered. RULE 1 is applied again and the code 010 is arbitrarily chosen.

- Next node C is considered. RULE 2 is applied on variable R1. Thus the code 110 is chosen.

- Next node B is considered. RULE 1 is applied and the code 001 is chosen.

- Lastly, node E is considered. RULE 3 is applied and the code 100 is chosen. The equations are listed below and

it can be seen that the maximum number of sum terms for each variable is four compared to six of the Grey Code assignment above. This will have a better chance of fitting the device. Following is the listing of the equations after using these rules.

PAL: ESPTEST

Intel Corporation

RESET ACCESS PHIT FP RDY S2 S1 S0

/R2 /R1 /R0

R2 := /RESET * /R2 * /R1 * R0 * RDY * FP
 + /RESET * R2 * R1 * /R0 * /ACCESS
 + /RESET * R2 * R1 * /R0 * ACC * /PHIT
 + /RESET * /R2 * R1 * /R0 * S2 * S1 * S0

R1 := /RESET * /R2 * /R1 * R0 * RDY * FP
 + /RESET * R2 * R1 * /R0 * /ACCESS
 + /RESET * R2 * R1 * /R0 * ACCESS * PHIT
 + /RESET * /R2 * R1 * /R0

R0 := /RESET * /R2 * /R1 * /R0 * ACCESS
 + /RESET * /R2 * /R1 * R0 * /RDY

DESCRIPTION:

PAL ESPTEST = [PLA ESPRESSO REDUCED FROM @ TEST]

Number of Inputs:	8
Number of Outputs:	4
Largest Number of Inputs for a Minterm:	6
Largest Number of Minterms for an Outputs:	5
Line Count:	27

STATE ASSIGNMENT METHOD: USING 3 RULES.

CONCLUSION

This chapter has introduced three new rules regarding internal state assignment for finite state machines using PLDs.

The result has shown that by using above rules we have achieved a better solution compared to that of Grey Code assignment method for this example. In fact, after years of experience, my colleagues and I have used the above rules almost every cases and every time the result is either equal or better when compared to results from STASH (a CAD tool of INTEL to do heuristic state assignment). Actually, these three rules are best when used after the initial state assignment is done (can be via other methods). If the initial assignment does not give a good result, then one can try applying the above rules to reduce the number of product terms of selected variables.

CHAPTER IV

LOGIC MINIMIZATION OF TWO LEVEL BOOLEAN FUNCTION USING GRAPH
COLORING

INTRODUCTION

There has been recently an interest in programs for optimization of Programmable Logic Array (PLA) and Programmable Array Logic (PAL) such as Presto (Brown 1981), Espresso, Espresso-mv, Espresso-exact (Rudell 1985), Prestol-II (Bartholomeus 1985), Mini (Hong 1974). Two approaches are currently known: algorithms that look for the minimum solution and approximate algorithms. The most advanced programs for minimum solutions are Espresso-exact (Rudell 1985), and McBoole (Dagenais 1986). All algorithms which search for the minimal solutions include two stages:

- generation of prime implicants
- minimum covering of minterms with prime implicants.

The number of prime implicants increases rapidly with the number of minterms, especially for functions with many don't cares. The set of prime implicants can become too large to enumerate even if it is possible to represent the function in two-level form. This result limits the application of algorithms based on generating all prime

implicants. The covering problem is NP-hard. Some functions that lead to extremely hard to solve covering problems have been constructed. It results then that there are two reasons why the current approaches to exact minimization will meet limited success.

In this chapter, we will introduce a new method to solve the covering problems without generating prime implicants. We reduce the covering problems to the coloring problems. Instead of solving the covering problem with prime implicants, we solve the coloring problem for a graph whose nodes correspond to minterms or some implicants of a new type. Therefore, we solve one NP-hard problem (graph coloring) instead of two NP-hard problems (the generation of prime implicants and the covering).

Graph Coloring can be solved approximately or exactly. We have written different algorithms for both solution method. In this chapter, we will show one for each type. The graph for coloring is created with any on-cubes of the function as nodes. These can be minterms, arbitrary cubes (product implicants), minimal product implicants of the function or disjoint minimum implicants. Minimal implicant for a minterm M is a product of all prime implicants covering M . The number of such implicants never exceeds the number of minterms or the number of prime implicants.

SOME BASIC DEFINITIONS AND NOTATIONS

ON[f] = set of ON-cubes of function f.

OFF[f] = set of OFF-cubes of function f.

DC[f] = set of Don't care cubes of function f.

Minterm = a cube which is contained in ON(f) set.

OFF-cell = a cell which is contained in OFF(f).

Set of cubes = array of cubes.

Cube C_i = a string of 0's, 1's, and X's; it represents a product of literals of function f.

An implicant of a function = an arbitrary subset of its minterms.

A product implicant = an implicant being a cube.

A prime implicant = a product implicant which is not covered by any other product implicant of that function.

\in = belongs to a set.

\supseteq = inclusion of sets.

\cap = intersection of arrays of cubes.

\prod = product

Example:

$$\begin{aligned} \{01X, 0X1\} \cap \{X10, 0X1\} &= (01X \cap X10) \cup (01X \cap 0X1) \\ &\quad \cup (0X1 \cap X10) \cup (0X1 \cap 0X1) \\ &= 010 \cup 011 \cup 011 = 010 \cup 011 = 01X. \end{aligned}$$

= sharp operator. It is equivalent to subtraction of arrays of cubes.

Example: $0XX \# 01X = 00X$.

$$\{XX0X, 1X1X\} - \{010X, X11X\} = \{X0X1, 1X01, 101X\}$$

MINIMAL IMPLICANTS

The set of minimal implicants constitutes the initial data to the optimum graph coloring. If this set is too large, we can use the set of disjoint cubes. Below, we will describe the generation of these minimal implicants.

Definition 4.1

A product implicant of a function f is any cube which is an implicant of that function.

Definition 4.2

The minimal implicant, MI, for minterm m_i , denoted by $MI(m_i)$, is the product of all prime implicants which cover minterm m_i .

Definition 4.3

Redundant minimal implicants are those which are properly included in other minimal implicants.

The following properties hold.

Theorem 1

Each essential implicant of the function is a minimal implicant, but a minimal implicant is not necessarily an essential implicant.

Proof: Recall the definition for essential implicant: an essential implicant is one which includes a singly covered minterm. Therefore, if a minterm can be covered by one and only one implicant, it will be by definition the minimal implicant for that minterm.

Theorem 2

There exists exactly one set of nonredundant minimal implicants for a Boolean function.

Proof: Follows from the fact that there exists exactly one minimal implicant for each minterm.

Theorem 3

Let $CUBES[j]m_i$ be the set of all j -cubes that cover minterm m_i and do not cover any OFF-cell. Let $CUBS[j]m_i$ be the set such that

$$CUBS[j]m_i = CUBES[0]m_i \cup CUBES[1]m_i \cup \dots \cup CUBES[j]m_i$$

If $CUBS[j]m_i = CUBS[j+1]m_i$ then $MI(m_i) = \cap CUBS[j]m_i$

Proof: the above algorithm generates the prime implicants that cover minterm m_i . Since $CUBS[j]m_i$ will be all the j -cubes that cover m_i , when we have completed adding all $CUBS[j]m_i$ for $j = 0$ to n , all cubes included in a larger cube will have been absorbed, and the terms that are left will be the prime implicants that cover minterm m_i . Then, from the definition of minimal implicant, Theorem 4 follows.

The input data to the algorithm at this point for generation of minimal implicants is the array DIC of disjoint ON-cubes, ON(f), and the array OFF(f) not necessarily disjoint cubes. Hence, the algorithms 1a and 1b below will create the array CC of minimal disjoint cubes.

The algorithm 1.b is the enhanced version from the algorithm 1.a. It was invented by Ciesielski and was used in PALMINI-MV:Multivalued Logic Minimizer by Ciesielski (1988).

Algorithm 1.a

Begin

1. Find set CONS of all consensuses of cubes from DIC(f).
 2. Find all products of pairs of cubes from DIC(f) and CONS.
- $$PROD = \{C_i \cap C_j \mid C_i \in DIC(f) \wedge C_j \in CONS\}$$
3. Find set CC = (DIC(f) # CONS) \cup PROD.
 4. Order the set CC according to the decreasing valued of INDEX.

The value of INDEX is found using Algorithm 2.

End

End Algorithm 1.a

Algorithm 1.b

Begin

1. Find all consensuses of cubes from $DIC(f)$.

2. Expand consensuses to prime implicants.

 $DIC(f) \leftarrow \text{Consensus}(DIC(f)) \cup DIC(f)$.

3. Obtain products of all pairs of cubes.

 $DIC(f) \leftarrow \text{Product}(DIC(f)) \cup DIC(f)$.

4. Delete cubes which are unions of other cubes.

5. Delete cubes contained in single cubes.

6. Make the resulting cubes disjoint:

 $\forall \{X, Y\} : P = X \cap Y \neq 0 \quad \text{split}\{X, Y\} \rightarrow \{P, X \# P, Y \# P\}$.

End

End Algorithm 1.b

Algorithm 2 generates an index for every minterm, corresponding to the number of OFF-cells, adjacent to that minterm in the function.

Algorithm 2

Begin

For each cube $C_i = x_1.x_2...x_j...x_k \in CC$

(where the x_i are variables in their true or complemented form)

do begin

INDEX [C_i] = 0;MINTERMS = [C_i]*;

```

    (create set MINTERMS = set of minterms included in
    Ci )
  for each minterm ∈ MINTERMS
  do begin
    j = 1;
    while j < k
    do begin
      change xj to /xj in minterm;
      if x1x2.../xj...xk is the OFF-cell then
        INDEX [Ci] = INDEX [Ci] + 1
      end
    end
  end
end
end
end algorithm 2

```

Let CUBS[j] be a set of all prime implicants covering a minimal implicant of cube C_i, MI(C_i). We introduce the relation of domination of prime implicants

$$p_1 \geq p_2 \leftrightarrow [p_1 \prod ON(f)]^* \subseteq [p_2 \prod ON(f)]^*$$

Definition 4

Let CUBS[j] be a set of all prime implicant p₁ in CUBS[j] such that $(\forall p_r \in \text{CUBS}[j]) [p_1 \geq p_r]$

then p_1 is called a necessary implicant for the minimal implicant

$$MI = \prod CUBS[i].$$

Necessary implicants are added to the minimal solution and all cubes covered by it are deleted from CC.

Example 4.1

For K-map of Figure 4.1: $[p_1 \prod ON]' - [p_2 \prod ON]' - [MI(0X0X)]'$

then $p_1 \geq p_2 \wedge p_2 \geq p_1$,

so either of them can be selected as necessary implicant.

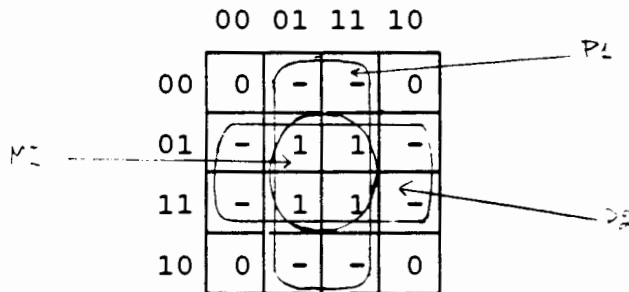


Figure 4.1. Necessary implicant

Example 4.2

For K-map of Figure 4.2:

$[p_2 \prod ON]' > [p_1 \prod ON]'$, then $p_2 \geq p_1$.

Hence P_2 is selected as the necessary implicant.

	00	01	11	10
00	-	-	-	0
01	1	1	1	-
11	-	1	1	-
10	0	-	-	0

Figure 4.2. Necessary implicant 2

Algorithm 3 generates the minimal and the necessary implicant for the cube C_i of CC. We denote the set of all necessary implicants of function f by NEI.

Algorithm 3: Procedure MINIMPL (C_i)

Begin

$j = 0$;

CUBS[0] = C_i ;

repeat

$j = j + 1$;

create set CUBES[j] of j -cubes covering C_i ;

delete from CUBES[j] the j -cubes that are not
implicants;

$CUBS[j] = CUBES[j] \cup CUBES[j-1]$;

delete from CUBS[j] the products covered by other
products

until CUBS[j] = \emptyset ;

MINIMPL = \prod CUBS[j];

(product of all cubes in array CUBS[j])

if there exists a necessary implicant $p_r \in CUBS[j]$ then

```

begin
    NEI = NEI  $\cup$  { $p_r$ };
    CC = CC #  $p_r$  ;
end

```

End Algorithm 3

Example 3

Given the function f such that $ON(f) = \{0111, 1111\}$,
 $OFF(f) = \{XXX0, X00X\}$, and the rest is $DC[f]$.

Find the minimal implicant $MI(0111)$ and the necessary
 implicant.

Solution:

The K-map and stages for generating $MI(0111)$ are shown
 in Figure 4.3.

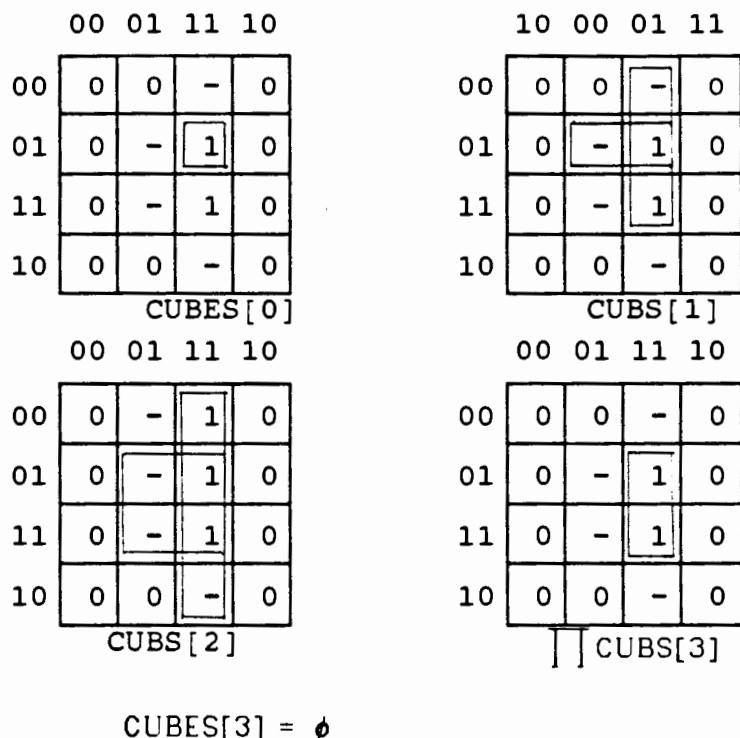


Figure 4.3. Minimal implicants

We will denote $\text{SMI}(f)$ is the set of minimal implicants of function f .

Algorithm 4 will generate set $\text{SMI}(f)$ from the disjoint set $\text{CC}(f)$.

Algorithm 4

begin

$\text{SMI} = \emptyset; \text{NEI} = \emptyset;$

while $\text{CC} \neq \emptyset$ do

begin

a) $C_i = \text{first cube from CC};$

b) $MI(C_i) = MINIMPL(C_i);$

c) if $MI(C_i) \supseteq M_r$, where M_r is some minimal implicant from SMI.
then

delete M_r from SMI;

d) $SMI = SMI \cup \{MI(C_i)\};$

e) $CC = CC - \{C, \in CC \mid MI(C_i) \supseteq C, \}$

end;

end algorithm 4

Example 4.4

$OFF(f) = \{XX10, X001, 011X\}$, $ON(f) = \{0X00, 11X1, X011\}$,

and the K-map is shown in figure 4.4

Consensus is computed: $CONS = \{1X11\}$

Product implicant from CONSENSUS is computed: $PROD = \{1111, 1011\}$

Disjoint set CC is then computed: $CC = \{0X00, 0011, 1101, 1111, 1011\}$

Now, the algorithm 4 is invoked to compute set SMI.

$MI(0X00) = XX00$, $M(0011) = X011$ (1011 deleted),

$MI(1111) = 1111$, $MI(1101) = 1101$,

$SMI(f) = \{XX00, X011, 1111, 1101\}$,

$NEI(f) = \{XX00, X011\}$.

	00	01	11	10
00	1	0	1	0
01	1	-	0	0
11	-	1	1	0
10	-	0	1	0

Figure 4.4. Example 4.4

It is important to realize that with this approach we do not have to store minterms, nor need we store at the same time all the prime implicants of the function. The sets of disjoint cubes or minimal cubes are almost always smaller than the respective sets of minterms or prime implicants. In the worst case, the set of minimal implicant is equal to the set of minterms. However, this is rarely the case.

COMPATIBLE MINIMAL IMPLICANTS AND COMPATIBLE SETS

The goal of this section is to discuss some properties of minimal implicants, which are essential to the method of reduction which we shall present in section 4.

First, we introduce the MATCHING operator, which is a main logic operation in our system.

Definition 4.5

$C_1 = (C_1^1, \dots, C_1^n) \wedge C_2 = (C_2^1, \dots, C_2^n)$ be cubes.

The matching operator $\$$ is defined as follows

$$C_{12} = (C_{12}^1, C_{12}^2, \dots, C_{12}^n) = C_1 \$ C_2 = (C_1^1 \$ C_2^1, C_1^2 \$ C_2^2, \dots, C_1^n \$ C_2^n)$$

where the operation $\$$ is defined in Table 1

TABLE I
MATCHING OPERATOR

$\$$	0	1	X
0	0	X	X
1	X	1	X
X	X	X	X

The operator $\$$ is commutative and associative and the result of its operation is always a cube.

Theorem 4.4

Let PI be a prime implicant of a completely or a partially specified Boolean function f . Then, for each set of minterms SM of function f which are covered by PI

$$SM = \{m_1, m_2, \dots, m_r\} \subseteq [PI]^* \subseteq ON(f) \cup DC(f)$$

The following relation holds

$$\$m_i \subseteq PI \quad (1)$$

$$m_i \in SM \subseteq [PI]^*$$

i.e., a cube resulting from matching minterms included in any subset of minterms of a prime implicant of a Boolean

function is an implicant (not necessarily prime) of this function.

Proof:

a) If $m \in [PI]^* \rightarrow m \subseteq PI$

b) From the definition of

$$C \subseteq PI \leftrightarrow (\forall i = 1, \dots, n) [C^i = PI^i \vee C^i \subseteq PI^i = X] \quad (2)$$

Let $C_{12} = C_1 \$ C_2$, where $C_1 \subseteq PI, C_2 \subseteq PI$.

Then from the definition of the matching operator

$$\begin{aligned} (\forall i) [C_{12}^i = C_1^i \text{ when } C_1^i = C_2^i \\ = X \text{ in any other case}] \quad (3) \end{aligned}$$

Using (2) for C_1 and C_2 we get

$$\begin{aligned} C_1 \subseteq PI \wedge C_2 \subseteq PI \rightarrow (\forall i) [(C_1^i = PI^i \vee C_1^i \subseteq PI^i = X) \subseteq (C_2^i = PI^i \vee C_2^i \subseteq PI^i = X)] \\ \leftrightarrow (\forall i) [C_1^i = PI^i = C_2^i \vee C_2^i \subseteq C_1^i = PI^i = X \vee C_1^i \subseteq C_2^i = PI^i = X \\ \vee C_1^i = C_2^i \subseteq PI^i = X \vee (C_1^i \neq C_2^i) \subseteq PI^i = X] \quad (4) \end{aligned}$$

If $C_1^i = PI^i = C_2^i$ then taking (3) into account we get $C_{12}^i = C_1^i = PI^i$.

In the next two cases of (4) we get $C_{12}^i = X$ from the definition of the matching operator. From (3) we then have $C_{12} = X = PI^i$. In the last case we may have $C_1^i = C_2^i$, therefore $C_2^i = C_1^i \subseteq PI^i = X \vee C_1^i \neq C_2^i$,

then $C_2^i = X = PI^i$, which by (2) gives $C_2 \subseteq PI$.

c) Using (a) and (b) we conclude that

$$\$m_i \subseteq PI$$

$$m_i \in SM \subseteq [PI]^*$$

Definition 4.6

Minimal implicants MI_i and MI_j are called compatible implicants when $MI_i \$ MI_j$ is an implicant of f , i.e. when there exists OFF-cell, $Z \subseteq OFF(f)$ such that $MI_i \$ MI_j \subseteq Z$.

Minimal implicants MI_i and MI_j which are not compatible will be called incompatible. A set of minimal implicants CM will be called compatible set when

$$\$MI_i \cap OFF(f) = \emptyset \text{ where } MI_i \in CM \quad (7)$$

A set of minimal implicants CP will be called set of compatible pairs when

$$(\forall \{MI_i, MI_j\} \subseteq CP) [(MI_i \$ MI_j) \cap OFF(f) = \emptyset] \quad (8)$$

Any subset of the set of minimal implicants included in a prime implicant is then compatible, and the matching of any compatible set of minimal implicants is a product implicant of the function, while the matching of any pair of compatible minimal implicants is a product implicant.

Theorem 4.5A

For each set of minimal implicants of the function f which are covered by PI

$$SMI = \{MI_i \mid MI_i \subseteq [PI]^* \subseteq ON(f) \cup DC(f)\}$$

The following relation holds

$$\$MI_i \subseteq PI \text{ where } MI_i \in SMI \subseteq [PI]^*$$

i.e., a cube resulting from matching minimal implicants included in any subset of minimal implicants of a prime implicants of a Boolean function is an implicant (not

necessarily prime) of this function.

Any compatible set is also a set of compatible pairs. The opposite statement is however not true, as shown in the following example.

Example 4.5

The Karnaugh map for function f is given in Figure 4.5 where $m_1 = 000$, $m_2 = 110$, $m_3 = 101$.

	0	1
00	1	-
01	-	0
11	1	-
10	-	1

Figure 4.5. Compatible implicants

The minimal implicants are:

$$MI_1 = X00$$

$$MI_2 = 1X0$$

$$MI_3 = 10X$$

We have that $MI_1, MI_2 = XX0 \notin Z = 011$

$$MI_1, MI_3 = X0X \notin Z = 011$$

$$MI_2, MI_3 = 1XX \notin Z = 011$$

$$\text{but } MI_1, MI_2, MI_3 = XXX \geq Z = 011$$

Hence, set of compatible pairs $CP = \{MI_1, MI_2, MI_3\}$ is then not a compatible set.

Lemma 1

If $A \subseteq C_1 \wedge B \subseteq C_2$ then

$$A \$ B \subseteq C_1 \$ C_2$$

Proof

For some indices i :

$$C_1^i \$ C_2^i \neq X,$$

which means that $C_1^i = C_2^i \neq X$.

If $C_1^i = C_2^i = 0$ then $A^i = B^i = 0$.

If $C_1^i = C_2^i = 1$ then $A^i = B^i = 1$.

Therefore, $C_1^i = C_2^i = A^i = B^i$ and for these indices i

$$A^i \$ B^i \subseteq C_1^i \$ C_2^i$$

for other indices j : $C_1^j \$ C_2^j = X$.

Then $A^j \$ B^j \subseteq C_1^j \$ C_2^j$ for those indices j .

It thus holds for all indices that $A^i \$ B^i \subseteq C_1^i \$ C_2^i$ and we have $A \$ B \subseteq C_1 \$ C_2$

Theorem 4.5B

Let CPR be any set of cubes covering all minterms and don't cares, (i.e., the cells of Karnaugh map) included in product implicant PR.

Then $\$C_i = PR$ where $C_i \in CPR$

Theorem 4.5C

Let C be the set of cubes covering cells (0-cubes) with minterms and don't cares.

If $(\forall C_i, C_j \in C)[(C_i \$ C_j) \cap OFF(f) = \emptyset]$

Then

- 1) $PR = \sum C_i$, is a product implicant where $C_i \in C$ and
- 2) $(\forall C_i \in C)[PR \supseteq C_i]$

REDUCTION OF TWO-LEVEL SINGLE-OUTPUT BOOLEAN FUNCTION MINIMIZATION PROBLEM TO THE MINIMAL GRAPH-COLORING PROBLEM.

The purpose of this section is to discuss how the minimization of a single-output Boolean function can be reformulated as a Graph-Coloring Problem.

Let us create the non-ordered graph $GIM = (SMI, RS)$, where SMI is equal to the set of minimal implicants of f and RS is the set of edges where

$$e = (MI_1, MI_2) \in SMI \times SMI \text{ such that } MI_1 \text{ is incompatible with } MI_2.$$

This graph will be called graph of incompatibility of minimal implicants.

Digression

The nodes of the graph correspond to minimal implicants. However, it must be kept in mind that only for moderately sized functions we can actually create graph GIM with the minimal implicants to provide the minimum solutions. For difficult functions of many variables, the number of minimal implicants can be equal to the number of minterms, which in turn can be equal to 2^n , where n is the number of variables. For more than $n = 14$ input variables, there exists functions (they are rare for examples taken from practice) for which product implicants can not be

generated. However, the method is still applicable, if we use the disjoint cubes of the initial specification instead of the minimal implicants or minimum disjoint cubes. This can lead to nonminimal solutions. Nodes of the graph can also correspond to arbitrary nondisjoint cubes; but this would degrade the result even further.

In a normal sum of products form, each minimal implicant MI from $SMI(f)$ must be covered by some set $\{PI_1, PI_2, \dots, PI_m\}$ of prime implicants of this function. This denotes the monomorphism $SMI(f) \rightarrow 2^{PI(f)}$, where $PI(f)$ is the set of prime implicants for function f .

Then, for each prime implicant cover of the function, we can assign to each minimal implicant a set of numbers of the prime implicants that cover this minimal implicant. We will call these numbers the colors of the minimal implicant. To each cover there corresponds then a certain coloring function: $COLF: SMI(f) \rightarrow 2^N$ where N is the set of natural numbers.

This function has the property that any two incompatible minimal implicants are colored by different colors. We will call this the property of "proper coloring".

$$MI_1 \in SMI(f) \wedge MI_2 \in SMI(f) \wedge (MI_1, MI_2) \in RS \rightarrow COLF(MI_1) \cap COLF(MI_2) = \emptyset.$$

Let us now consider the inverse mode. We will find the coloring satisfying this property. If each set of minimal implicants with the same color denotes some prime

implicant then a prime implicant cover of the function corresponds to this coloring. To the coloring with the minimal number of colors, there corresponds a cover with the minimum number of implicants. Because nodes which are linked with an edge must belong to different implicants, local fulfillment of the condition of proper coloring for each node implies that the set of colors of any node is disjoint with the set of colors of any of its adjacent (linked) nodes. Let us now assume that each node has only one color:

$$\text{COLF} : \text{ON}(f) \rightarrow N$$

A proper coloring will be defined as one in which different values of the function COLF are assigned to any pair of nodes which are connected by an edge $(MI_1, MI_2) \in RS$.

Definition 4.7: Compatible Coloring.

A Compatible coloring is a proper coloring in which each set of nodes of the graph having the same color is a compatible set of minimal implicants of the function.

By finding the compatible coloring of the graph with minimum number of colors, we minimize the number of compatible sets of minimal implicants, and then the number of product implicants in the cover, and as a consequence the number of prime implicants in the cover. This result is stated in the following theorem

Theorem 4.6

The minimal number of compatible sets of minimal implicants is the same as the number of prime implicants in the minimal cover of the function.

Proof: Let PI_i be any prime implicant of function f , then there exists for it exactly one matching cube

$$C = \prod MI_i,$$

$$MI_i \in SMI(f) \wedge MI_i \subseteq PI_i$$

which is a product implicant. Let us assume then that MCP is a minimal cover of the function f with prime implicants, and MMC is a minimal cover of this function with matchings of compatible sets of minimal implicants and $CARD(MCP) < CARD(MMC)$. This is inconsistent with the fact that MMC is a minimal cover, because if we find the corresponding matching group for each prime implicant in MCP, we will obtain the cover MMC' such that $CARD(MCP) = CARD(MMC')$, and then MMC is not the minimal cover.

There are different optimal and quasioptimal proper graph-coloring algorithms, both for sequential and parallel computers (Gare 73), (John 84), (Kauf 68), (Perk 83), (McDia 79), (Vizi 64), (Perk 84), (Perk 84b). The compatible coloring algorithms are presented in (Perk 83).

After completing the compatible coloring of graph GIM, the algorithm returns a set of cubes that are matchings of compatible sets of minimal implicants. Depending on the

coloring algorithm that is used, this set of product implicants has a minimal or quasi-minimal number of implicants. Where our intention is to find only the minimal number of implicants (minimization of cost function CF_1), then the minimization process is finished. However, if we intend to find the minimal number of inputs to gates under the assumption that it is the number of gates that is to be minimized first, then we will attempt to delete all possible subsets of the set of literals from each product implicant independently.

Example 4.6

Consider the following incompletely specified function:

ON($f(X_1, X_2, X_3, X_4)$) = {0000, 0100, 0011, 1101, 1111, 1011}

OFF($f(X_1, X_2, X_3, X_4)$) =

{0010, 0101, 0111, 1110, 1001, , 1010}

	00	01	11	10
00	1	-	1	0
01	1	0	0	-
11	-	1	1	0
10	-	0	1	0

Figure 4.6. Compatible coloring

Method 1: Necessary implicant is taken into account.

First $SMI(f) = ON(f) = \{0000, 0100, 0011, 1101, 1111, 1011\}$

The necessary implicants are : XX00 and X011

Hence, $SMI(f) = \{1101, 1111\}$ (others are absorbed in $NEI(f)$)

The graph GIM is as follows:



By matching operator: $1101 \oplus 1111 = 11X1$, where

$$11X1 \cap OFF(f) = \emptyset$$

Thus, we can color this graph with one color. In other word, we can combine the two cubes into one: $11X1$.

Hence, the solution is $f = NEI(f) + 11X1 = \{XX00, X011, 11X1\}$

Method 2: Necessary implicant is not taken into account.

$$SMI(f) = ON(f) = \{0000, 0100, 0011, 1101, 1111, 1011\}$$

$$\text{Node 1} = 0000$$

$$\text{Node 2} = 0100$$

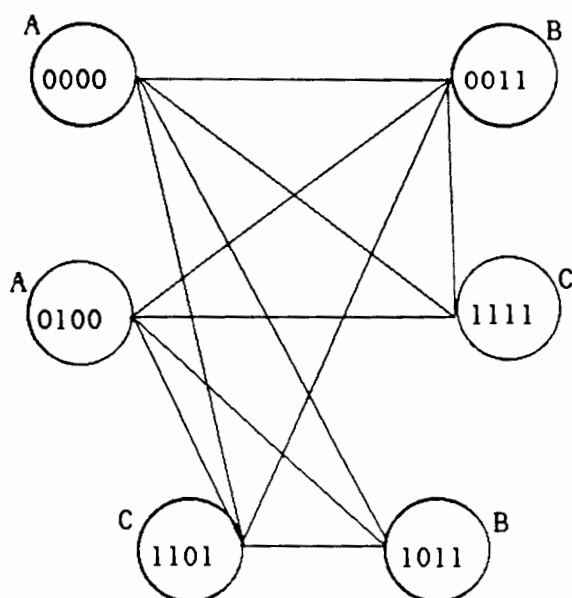
$$\text{Node 3} = 0011$$

$$\text{Node 4} = 1101$$

$$\text{Node 5} = 1111$$

$$\text{Node 6} = 1011$$

By matching each pairs of node, we create graph GIM as follows



The graph can be represented as an Incompatibility Matrix as follows:

	1	2	3	4	5	6
Node						
1	0	0	1	1	1	1
2	0	0	1	1	1	1
3	1	1	0	1	1	0
4	1	1	1	0	0	1
5	1	1	1	0	0	0
6	1	1	0	1	0	0

1 = an edge between two nodes

0 = there is no edge between two nodes.

Now we can start coloring the nodes. Remember that if there is an edge between two nodes, then the two nodes must

have different colors. The minimum number of colors needed for this graph is three. The coloring with colors A, B, and C is shown on the graph. This means that we can realize the minimal solution for this function with three product implicants. By matching minterms with colors A, we get $0000 \oplus 0100 = 0X00$. Similarly, by matching minterms with color B, we get $0011 \oplus 1011 = X011$. Finally, by matching minterms with color C, we get $1101 \oplus 1111 = 11X1$.

$$\text{So, } f(X_1, X_2, X_3, X_4) = \{0X00, X011, 11X1\}$$

$$\text{Or } f = \bar{X}_1 \cdot \bar{X}_3 \cdot \bar{X}_4 + \bar{X}_2 \cdot X_3 \cdot X_4 + X_1 \cdot X_2 \cdot X_4$$

$$I_1 = 0X00, I_2 = X011, I_3 = 11X1$$

If our goal is to minimize the cost function CF2, then we want to minimize the number of literals. So we will try to delete literals from the product implicants. For I_1 and I_3 this is not possible

$$\bar{X}_2 \cdot X_3 \supseteq 0010, \bar{X}_2 \cdot X_4 \supseteq 0001, X_3 \cdot X_4 \supseteq 0111$$

and

$$X_2 \cdot X_4 \supseteq 0111, X_1 \cdot X_4 \supseteq 1001, X_1 \cdot X_2 \supseteq 1110.$$

However, deleting X_1 from I_1 gives us the prime implicant $I_{11} = \bar{X}_3 \cdot \bar{X}_4$.

Other deletions do not lead to new implicants. We have then obtained

$$f = \bar{X}_3 \cdot \bar{X}_4 + \bar{X}_2 \cdot X_3 \cdot X_4 + X_1 \cdot X_2 \cdot X_4.$$

MINIMIZATION OF MULTI-OUTPUT TWO-LEVEL BOOLEAN FUNCTIONS

Full minimization of multi-output two-level functions consists of: reducing such a function to a single output function using the method presented by [Mill 65], then minimizing this function using the method of section 4, and, finally, finding multioutput implicants from the implicants of the single-output function.

From function f we define an $n+m$ -input, 1-output function f_f as follows:

$$ON(f_f) = \{C_f = C \circ Z_r \mid (\exists r \in \{1, \dots, m\}) [C \in ON(f^r)]\}, \text{ and}$$

$$OFF(f_f) = \{C_f = C \circ Z_r \mid (\exists r \in \{1, \dots, m\}) [C \in OFF(f^r)]\}.$$

where $Z^m_r = (Z_1, Z_2, \dots, Z_i, \dots, Z_m)$ is the m -tuple defined for each component function f_r , in which $Z_i = 1$

For $i \neq r \wedge Z_i = 0$ for $i = r$.

Symbol \circ means concatenation.

We minimize this new function f_f using the method described in the previous section. Then, from the implicants of f_f , we find the implicants of the initial multi-output function f . Each of the generated implicants of f_f can be presented in the form $I = IC_i \circ Z^m$ where the m -tuple Z^m has one of the following forms:

1. $Z_k = 1$ or $Z_k = X$ - then IC_i is an implicant of f^k iff

$$Z_k = X, \quad k = 1, \dots, m,$$

2. $Z_k = 0$ or $Z_k = X$ - then IC_i is an implicant of f^k

iff

$$Z_k = 0, k = 1, \dots, m,$$

3. $Z_k = X$ - then IC_i is an implicant of f^k , $k = 1, \dots, m$.

Example 4.7

The goal of this example is to minimize the two function f_1 and f_2 at the same time. The K-map of the functions are shown in Figure 4.7a.

Number of inputs = 3.

Number of outputs (functions) = 2.

Hence, we will create a function f_3 which has 5 input variables as shown in Figure 4.7b. With this method, as the number of outputs increases, we can quickly see that the function f_3 is strongly incompletely specified.

	0	1		0	1
00	1	0		1	0
01	1	1		0	0
11	1	0		1	0
10	0	0		1	1
	f1			f2	

Figure 4.7a. Multioutput 1

	00	01	11	10
000	-	1	-	1
001	-	0	-	0
011	-	1	-	0
010	-	1	-	0
110	-	1	-	1
111	-	0	-	0
101	-	0	-	1
100	-	0	-	1
		f1		f2

Figure 4.7.b Multioutput 2

Now $\text{SMI}(f_3) = \{0001, 01X01, 11001, 00010, 11010, 10X10\}$

Using the previous method,

The necessary implicants are:

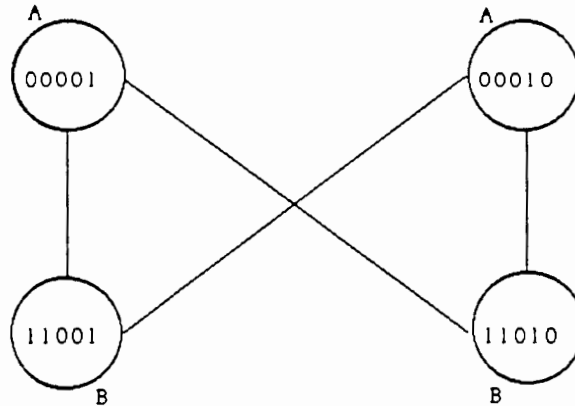
For $10X10$ - $10X1X$ or $10XX0$ - we select the first one:

$10X1X$.

For $01X01$ - $01X0X$ or $01XX1$ - we select the first one:

$01X0X$.

The graph of incompatibility for the remaining $\text{SMI}(f_3)$
 $= \{0001, 11001, 00010, 11010\}$ is then computed.



As a result of coloring of the graph, we get

$$I^1 = 00001 \ \$ \ 00010 = 000XX$$

$$I^2 = 11001 \ \$ \ 11010 = 110XX$$

$$\text{Then } f_3 = \{10X1X, 01X0X, 000XX, 110XX\}$$

After the separation into component functions according to the above method, we obtain:

000 belongs to both f_1 and f_2

110 belongs to both f_1 and f_2

10X belongs to f_2

01X belongs to f_1

$$\text{Then } f_1 = \{000, 110, 01X\}$$

$$f_2 = \{000, 110, 10X\}$$

EXTENSION OF PRODUCT IMPLICANTS

After using the graph coloring to minimize the function f . The implicants can be further extended by

deleting redundant literals. The result can in some case lead to less input pins to the PLA. We will show two algorithms for extension: approximate and optimum.

Algorithm 5

An approximate method for extending product implicants

Given: the set II_1 of product implicants for function f

K is the number of variables in cubes.

Begin

$II_2 = \emptyset$;

for each product implicant $I \in II_1$ do

begin

$N = 1$;

while $N \leq K$ do

begin

$I_1 = I$ with the N_{th} literal from the left deleted;

if $(\exists Z \in [OFF(f)])(I_1 \supseteq Z)$;

then

$N = N + 1$;

end

else

begin

$I = I_1$;

$N = N + 1$;


```

        end
    end
     $II_2 = II_2 \cup I$ 
end
End algorithm 5;
```

This algorithm is very fast and is sufficient for most problems. It is implemented in PALMINI.

Algorithm 6

Exact method for extending product implicants

Given: set II_1 of product implicants of function f

Begin

E1. $II_2 = \emptyset$;

E2. For each product implicant $I \in II_1$ do

Begin

a.- SOLUTION = I, $CF_{\min} = CF_3(I)$

(Cost function CF_3 calculates number of literals in implicant I);

b.- place initial state of the tree ($N=0$): $[QS(N), GS(N), CF_3(N)] = [I, \text{set of indices "in" of cube I for which } I_{in} \neq X, CF_3(I)]$, on the list BT (BT stands for Branch of Tree). At this point BT has only one element (the triple $(QS(0), GS(0), CF_3(0))$);

```

c.- FE = (QS(N), GS(N), CF3(N)) = first element from
    list BT;
if GS(N) =  $\emptyset$ 
begin
    delete FE from BT;
    go to d;
end
INDEX = first element from GS(N),
QS(N+1) = cube QS(N) with symbol X inserted in the
position INDEX;
GS(N+1) = (GS(N) with INDEX deleted),
if ( $\exists Z \in [OFF(f)]$ ) [ $QS(N+1) > Z$ ] then
    "cut-off and backtrack in tree" go to d;
CF3(N+1) = CF3(N) - 1;
if CF3(N+1) < CFmin
begin
    CFmin = CF3(N+1);
    SOLUTION = QS(N+1);
end;
if GS(N+1) =  $\emptyset$ 
    go to d;
else
    add new state (QS(N+1), GS(N+1), CF3(N+1)) to the
    top of list BT;
d.- if BT =  $\emptyset$ 

```

```

    add prime implicant SOLUTION to the set  $II_2$ 
  else go to c;
end;
end algorithm 6;

```

The following example will illustrate the operation of this algorithm.

Example 4.8

The Karnaugh map for function f is given in Figure 4.8

$ON(f) = \{0001, X100, 10X1\}$

$OFF(f) = \{0000, 0010, 0111, 1010\}$

	00	01	11	10
00	0	1	-	0
01	1	-	0	-
11	1	-	-	-
10	-	1	1	0

Figure 4.8. Example 4.8

From coloring the graph GIM, the product implicant 0001 was found. This is the case where the necessary implicants have not been taken into account. Figure 4.8b shows the tree for deleting literals. Deleting literal INDEX corresponds to replacing the corresponding index with the symbol X. The tree is created as a tree of subsets of the given set. When a newly created cube is found not to be

an implicant, the cut-off in the tree is executed. The enumeration of nodes in the figure corresponds to the Depth-first strategy with one successor (Perk 80b) applied in this algorithm 6. As a solution, cubes XX01, X0X1 were found.

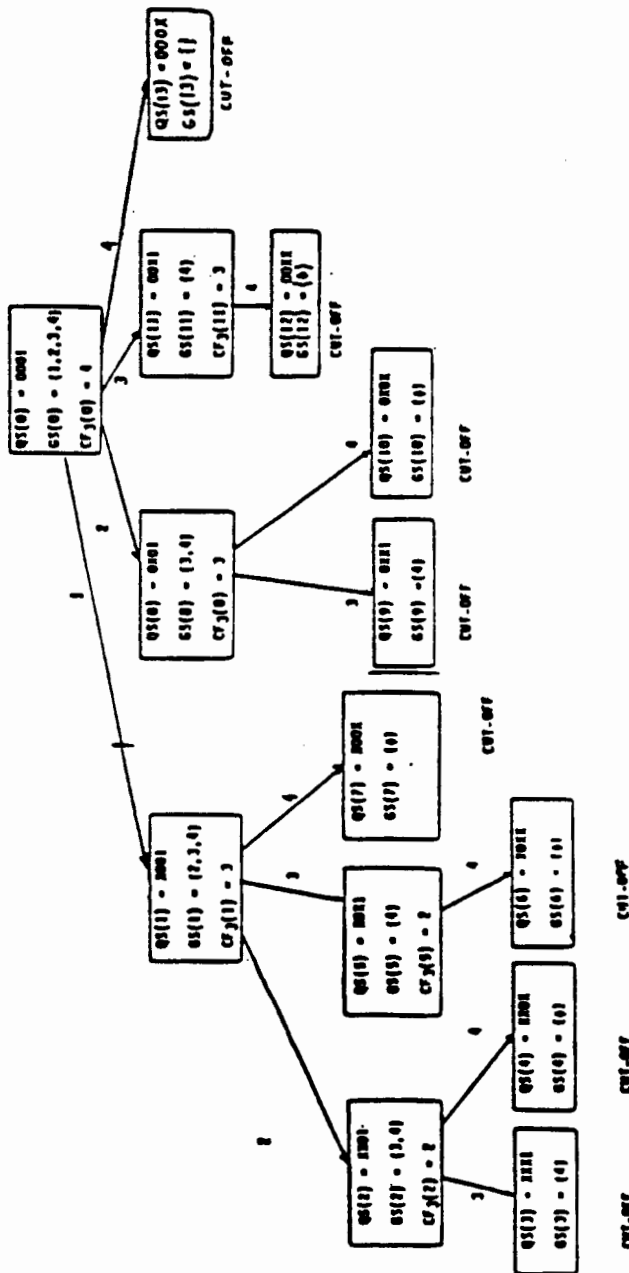


Figure 4.8.b Depth-first strategy with one successor

ALGORITHMS FOR GRAPH COLORING

In this section, we will introduce two algorithms of proper coloring which can be used for Boolean Minimization.

1) The first algorithm colors node after node with one of the colors admissible for this node. The remaining colors are stored for later possible use after backtracking. We initially assume that the number of colors is equal to the number of nodes in the graph. The tree is searched with a Depth First Strategy With One Successor. After finding each solution, the algorithm calculates its cost $CF(N)$. The solution with lower cost is printed and stored. This cost $CF(N)$ is now used as a new upper estimate of the chromatic number of the graph. From the sets of the possible colors for used in the nodes (sets $GS(N)$), all those colors not included in the last solution are deleted. The process of tree search is executed applying the cut-off principle based on the cost function, $CF(N)$.

This algorithm will give us the optimum solution. The complete listing and example of this algorithm is given in the Appendix C.

2) The second algorithm is based on a heuristic approach. This is a non backtracking and approximate algorithm. However, it is very fast and gives good results. This procedure is currently implemented in PALMINI.

Algorithm 9:

Approximate Coloring of the Graph

```

Color(Node1) = 1; {first color}
for Nodei = Node2 to Noden do
begin
    Color(Nodei) = 1;
    for Nodej = Node1 to Nodei - 1 do
    begin
        if {Nodei,Nodej} ∈ RS and Color(Nodei) = Color(Nodej) then
            Color(Nodei) = Color(Nodei) + 1;
    end;
end;
End algorithm 10;

```

PERFORMANCE EVALUATION

The above algorithms were implemented in two versions. The first one was written in PASCAL and called PLAMCO and the other was written in C and called PALMINI. The major difference between PLAMCO and PALMINI is the data structure being used to represent the cubes. In PLAMCO, the bits of the cubes are realized as elements of two dimensional arrays. Hence, all the operations operate on arrays. Whereas in PALMINI, the bits of the cubes are represented as pairs of bits in registers. Hence, all the operators operate on

registers which is much faster and occupy much less memory.

PLAMCO:

- PLAMCO did not have the complementation part. We originally assumed that when we designed a Boolean function, we will also know the $OFF(f)$ along with $ON(f)$ and we treated the rest as $DC(f)$. However, this is not true at all. The reality is that most of the time, we only know the set $ON(f)$.

- PLAMCO used Boolean arrays to represent cubes. Hence, each bit takes a lot of memory (on the average, two integers, it varies from compiler to compiler).

- PLAMCO did not have the Static Hazardless feature. This feature will be described in detail in PALMINI section.

- PLAMCO used back-tracking Graph Coloring Algorithm to color graph GIM. The result turned out to be very dissapointing. A function with 19 terms/6 inputs could take more than half an hour. With PLAMCO, we observed the following things:

- 30 % of the time was spent in coloring graph GIM. So, the back-tracking Graph Coloring Algorithm was somewhat slow.

- 70 % of the time was spent in generating Minimal Implicants.

- The time spent in other procedures is too small to bring it into the picture. Hence, they are not accounted

for here.

It was 1985 and PALs and PLDs began to gain popularity in industry. However, the software support was still weak. The only CAD tools available for PC at the time was from CUPL and DATA I/O. Therefore, our goal was to focus on a Boolean Minimizer for PAL-Based circuits. The main goal was to provide a reasonably good solution (does not have to be optimal) within a reasonable amount of time. And, the next product was PALMINI.

It is worth while to insert a reminder here that most commercially available minimizers are only approximate, including PRESTO, ESPRESSO, etc. For exact minimization procedure, only McBOOLE (Degais 85) and ESPRESSO-EXACT (Rudell 85) have been designed.

PALMINI.

- PALMINI has a complementation part. We decided to use the Disjoint Sharp method because it was easy to implement. This method is the worst one compared to those used in ESPRESSO or MINI. For PALs and PLDs, where the number of products of sums are not large (normally less than 20) and the number of input variable are not large (normally less than 24), the Disjoint Sharp is manageable. For better algorithm, we should have used the one described by (Brayton 84) or (Sasao 83).

- PALMINI uses bits inside a register to represent

Boolean bits.

A Boolean bit of a cube is represented by two binary bits. Hence, a short integer or a byte (8 bits) can store 4 Boolean bits. Thus it offers a lot memory saving compared to the case in PLAMCO. In addition, the operations on cubes can now be done with operations on registers which include: AND, OR, and XOR, and they are many orders of magnitude faster than in the case of PLAMCO.

- Instead of generating the Minimal Implicants, we chose to generate minimum disjoint cubes from set $SMI(f)$. The method used is the Disjoint Sharp method.

- Instead of using the back-tracking Algorithm to color graph GIM, we invented a heuristic non-backtracking Algorithm. This method is very fast and gives good solutions. However, it is only approximate.

The result is very encouraging. For small single-output functions, the speed is far better than APLUS 1.0 from ALTERA CORP, many times faster than ABEL 1.1 (Presto) from DATA I/O CORP, and comparable and even faster than ESPRESSO.

With the current version of the program, we observe the following:

- 60 % of the time is to compute the complementation.
- 20 % of the time is to compute disjoint cubes.
- 10 % of the time is to compute graph GIM.

- 5 % of the time is to color the graph.
- 5 % of the time is to delete the literal.

The fact that 60 % of the time is to compute the complementation suggests that by having a better algorithm such as the one used in Espresso, the speed of the program can be improved even further.

PALMINI

Description of PALMINI

- input: cubes (product implicants) of completely specified functions in terms of sum of products. The input cubes can be overlapping.
- output: a minimized version of the function.
- features as options:
 - 1- Form of input cubes for Graph Coloring.
 - 2- Optimal and quasi-optimal Graph-coloring algorithms.
 - 3- Invert the polarity of the output.
 - 4- Check for Static Hazards for combinatorial outputs.
 5. Minimize the number of literals in each term of the function.

Main Procedures of PALMINI.

procedure COMPL(SMI);

This procedure returns the complementation of the

input function contained in SMI. The Disjoint Sharp method (Ulug 1974) is currently employed. At the end of each loop, the list OFF which contains new ON-cubes that were created in the previous pass, is passed to procedure ABSORBE to delete redundant terms.

procedure CREATEDISJOINT(SMI);

This procedure receives data from the input set SMI. It then returns a set of disjoint cubes back into set SMI. The algorithm is as follows:

```

for i = 1 to (last cube in SMI -1)
begin
  for j = i + 1 to last cube in SMI
  begin
    if cubei intersects cubej then
    begin
      list D = cubei # cubej;
      cubej is deleted from SMI;
      list D is added to SMI;
    end;
  end;
end;

```

procedure CREATEMINIMAL(SMI);

This procedure is used to create disjoint minimum product implicants. In general, only implicants of this

type (or ones included in them, like minterms) assure the minimum solution if the solution to graph coloring problem is also optimal.

At the moment the Disjoint Sharp method is used. This will give a worse result than the algorithm below.

The following algorithm will be implemented later.

1. Find all consensuses of cubes from SMI and add them to the set SMI.
2. Find all products of pairs, pairs of pairs, pairs of pairs of pairs, ... etc. of cubes from SMI; remembering for each new product cube the product cubes that it originates from. This is done in the form of the (directed, acyclic) graph. An arrow points from cube1 to cube2 if cube2 originates from cube1.
3. Remove from the tree all cubes, that are cube unions of other cubes from the graph. This is done from top to bottom of the graph (starting from the largest cubes).
4. Remove from the tree all the cubes that are included into a single cube only.

The remaining cubes in the tree are the disjoint minimum implicants. Return them as the value of CREATEMINIMAL.

Example 4.10. For function $f(a,b,c,d) = \{0X01, X1X1, 011X, 1100, 1011\}$. The consensus are $\{110X, 1X11, 01X1\}$. The products of cubes are $\{0101, 0111, 1101, 1111\}$. After removal of products being unions of other products the set SMI is $\{1100, 1011, 011X, 0X01, 0101, 0111, 1111, 1101\}$. After removing of cubes that are included into only one cube, the set SMI = $\{1100, 1101, 1111, 1011, 011X, 0X01\}$. This set is used to create graph GIM.

procedure GRAPH(SMI, OFF, GIM);

This procedure will construct graph GIM from disjoint set SMI and set OFF which contains the complementation of the input function.

The algorithm is as follows:

```

for i = 1 to (last cube in SMI - 1)
begin
  for j = 1 to last cube in SMI
  begin
    if (cubei $ cubej)  $\cap$  OFF  $\neq \emptyset$  then
      GIM(i,j) = GIM(j,i) = 1;
      {an edge exists between node i and node j}
    else GIM(i,j) = GIM(j,i) = 0;
      {no edge exists between node i and node j}
    end;
  end;
end;
```

procedure COLOR(GIM,cost1);

The algorithm in PALMINI is a non-backtracking, approximate algorithm. The optimal algorithm is implemented in PASCAL version called PLAMCO.

This procedure uses GIM as its input and returns cost1 as the number of colors needed to color this graph. The Algorithm 10 was implemented in this procedure.

procedure DELETELITERAL(SOL,OFF);

This procedure takes each term in SOL and tries to remove as many redundant variables as possible according to the following algorithm:

```

for i = 1 to last cube in SOL
begin
  for j = 1 to max number of input variables
  begin
    temp = cubei[j];
    cubei[j] = X;
    if cubei  $\cap$  OFF  $\neq \emptyset$  then
      cubei[j] = temp;
  end;
end;

```

Hazardless minimization

Product implicants PI1 and PI2 are adjacent when they include two minterms, $m_1 \in PI1 \wedge m_2 \in PI2$

such that m_1 and m_2 differ in a single bit only (are adjacent in a sense of a Gray Code). The static hazard in ones occurs in a two-level circuit when there are two ANDs realizing adjacent product implicants but lacking a third product to cover the adjacent minterms of the two products. The result of such hazards is a glitch (short pulse zero) in the output before it reaches the stable state 1.

Example 4.11. Let us assume a two-level realization of an expression

$$f = \bar{a}.\bar{c}.d + a.b.\bar{c} + \bar{a}.b.c + a.c.d$$

Assume that all the gates have the same delay "tpd". The pair of cells 0101 and 0111 is a pair of adjacent minterms not covered by a single implicant. So are also the pairs: 0111 and 1111, 1111 and 1101, 1101 and 0101. This is then a circuit with four static hazards. Depending on the later stages of the circuitry, these glitches may cause catastrophic failures to the rest of the operation of the circuitry (for instance if hazard occurs in a feedback loop of an asynchronous circuit or if a counter is driven from a circuit with hazard). By introducing a fifth cube to cover the adjacent 1's between the original product implicants, we effectively eliminate all four hazards.

Solution: $f = \bar{a}.\bar{c}.d + a.b.\bar{c} + \bar{a}.b.c + a.c.d + bd$ is then hazardless.

One of the features of PALMINI is the ability to correct all the static hazards that exist in the solution. After the solution is obtained from the Graph Coloring Algorithm and if the hazardless option is selected, PALMINI will compute all the consensuses which exist among the cubes in SOL. Next it will find all mergings (distant-one merge groups $A.B + A./B = A$) of consensuses and of consensuses and product implicants. This operation is repeated until no more groups are created. It will then remove the consensuses that are properly included into some mergings. The consensuses and the mergings are attached to SOL as a part of the final solution.

Below we will present the algorithmic way to find all the hazard eliminating cubes. The consensus of two cubes A and B is created as follows. First, we calculate the bit-by-bit operation star (*) on cubes A and B. The STAR operation per bit is defined as follows:

TABLE II
STAR OPERATION: *

*	0	1	X
0	0	e	0
1	e	1	1
X	0	1	X

Next if the resultant cube includes exactly one e, it is changed to X.

Otherwise the cube is not a consensus of the function.

Example 4.12: from the example 4.11 above, we have

$$\text{cube}_1 * \text{cube}_2 = 0X01 * 011X = 01e1 = 01X1.$$

Note: 01e1 contains only one "e". Therefore, it can be changed to "X". There are four consensuses in this example: 01X1, 11X1, X101, and X111. Merging of 01X1 and 11X1 produces cube X1X1. All consensuses are now removed since they are covered by this cube. This leads to a hazardless solution for example 4.11.

procedure HAZARDLESS(SOL);

1. {Find the set of all consensuses cube_c of cubes from solution SOL}

for i = 1 to (last cube in SOL - 1)

begin

for j = (i + 1) to last cube in SOL

begin

$\text{cube}_c = \text{cube}_i * \text{cube}_j;$

{if there is no result of consensus operation cube_c is an empty set}

if cube_c is not empty and $\text{cube}_c \notin \text{SOL}$

```

        then add cubec to SOL;
    end;
end;

2. {Find the set NEW_CUBES of all cubes cubem being
    results of merging operations (cubem = cubei m cubej)
    off all cubei and cubej in SOL}
    for i = 1 to (last cube in SOL - 1)
    begin
        for j = (i + 1) to last cube in SOL
        begin
            cubem = cubei m cubej;
            { m is a merging operator, if cubes do not
              merge, the result cubem is an empty set}
            if cubem is not empty and cubem ∉ SOL then
            begin
                add cubem to SOL;
                add cubem to NEW_CUBES;
            end;
        end;
    end;

3. NEW_CUBES = MERGING(NEW_CUBES, SOL);
    if NEW_CUBES = ∅ then
        return SOL = SOL with removed cubes included in
        other cubes of SOL;
    else begin

```

```

    SOL = SOL  $\cup$  NEW_CUBES;
    goto 3;
end;

```

function MERGING(NEW_CUBES, ALL_CUBES);

```

NEW_CUBES =  $\emptyset$ ;
for i = 1 to last cube in ALL_CUBES
begin
    for j = 1 to last cube in ALL_CUBES
    begin
        cubem = cubei m cubej;
        if cubem  $\neq \emptyset$  and cubem  $\notin$  SOL then
            add cubem to NEW_CUBES;
        end;
    end;
end;
return NEW_CUBES;

```

Note: m = merging operation. If two cubes are different by only one variable in their literal, they will be merged.

Flow chart of PALMINI

Get input: set SMI \leftarrow sum of products

1. Find the complementation from this set:

OFF \leftarrow COMPL(SMI).

2. If invert polarity is selected then

begin

```
SMI <- OFF.
```

```
OFF <- SMI.
```

```
end.
```

3. If createdisjoint variant is selected then create disjoint set from SMI:

```
SMI <- CREATEDISJOINT(SMI).
```

```
else create minimal set from SMI:
```

```
SMI <- CREATEMINIMAL(SMI).
```

4. Create graph GIM: GIM <- GRAPH(SMI, OFF, GIM).

5. Color graph GIM to find cost:

```
cost <- COLOR(GIM, cost1).
```

6. Find solution and store in array SOL.

7. If the Static Hazardless option is selected then

```
SOL <- HAZARDLESS(SOL).
```

8. If the literal delete option is selected then delete redundant literals in each term of solution SOL.

```
SOL <- DELETEDLITERAL(SOL, OFF).
```

```
Solution is now contained in SOL.
```

Performance Evaluation of PALMINI

Palmini is written in C using computer words (registers) to represent cubes. We have tried about thirty examples from work (at INTEL) ranging from 4 terms/4 inputs to 20 terms/18 inputs. The solutions were then compared to those of LOGMIN and were the same. LOGMIN is an INTEL's

proprietary CAD tool which consists of many different CAD programs and one of them is Espresso which is used to minimize PLA's. The table below used a set of nine selected examples to compare PALMINI with LOGMIN, APLUS Ver1.0 (tool from ALTERA Corp for EPLD), and ABEL Ver1.1 (tool from DATA I/O Corp). All tests were done on a PC XT compatible machine with 8 MHz clock. The minimizers from ALTERA, DATA I/O, and LOGMIN run on the same machine. The algorithm used in ALTERA software is an order of magnitude slower than PALMINI and is not shown here. On the other hand, Presto from ABEL is very reasonable. The version used is 1.1 which is much better than version 1.04. PALMINI is found to be equal or better than ABEL. Depending on the types of functions, sometimes, PALMINI is faster than both Espresso and ABEL and sometimes it is not.

In the following table, the numbers of terms and input variables are given for each example. Next, the times (in seconds) and numbers of terms in solution are given for PALMINI, ABEL, and ESPRESSO.

TABLE III
PALMINI PERFORMANCE

EX# Function PALMINI ABEL ESPRESSO

EX#	Term	Input	Time	Term	Time	Term	Time	Term
1	19	6	2	13	12	13	4.5	14
2	8	9	2	4	9	4	2	4
3	15	9	1	4	9	4	2	4
4	10	10	1	10	12	10	3	10
5	10	11	2	9	8	9	3	9
6	12	12	2	10	27	10	5	10
7	13	13	4	10	26	10	4	10
8	11	17	9	10	7	10	4	10
9	20	18	6	17	--	--	8	17

Note: -- means no answer in 20 minutes and the test is aborted.

As we can see that PALMINI on the average is much faster than ABEL 1.1 and gives good results as compared to ESPRESSO for small examples. We can easily see that PALMINI is adequate for PALs or PLDs based designs.

CONCLUSION AND FUTURE WORK

The examples discussed above were taken from examples at work. PALMINI has shown us that it indeed gives good solutions within an acceptable time frame. Besides the fact that its speed on functions of small size is comparable or

better than ESPRESSO and many times faster than ABEL (Presto), it has an useful feature which other low cost minimizers do not have. That is Static Hazard correction.

PALMINI is easily recompiled to run on various personal and home computers which support standard C like IBM PC, APPLE, Commodore, and etc. The compiled code is small. It can easily fit into 64K of memory. This includes all the code and data areas, which permits the use of this program together with other memory-resident programs. Executable code of PALMINI is only 30K, versus 177K of Espresso.

The limitation of the current version is as follows:

- up to 64 input variables. (PAL or PLD only allow up to 23 inputs)
- up to 60 product terms. (PAL or PLD only allow 8 product terms)

The current version also supports multi-output function.

With respect to the Graph Coloring Algorithm, we can summarize the limitations as follows:

- The reduction and coloring algorithms are fast.
- The weakest part is the complementation.

Two improvements are possible:

- 1) Better complementation algorithm.
- 2) Avoid complementation and check inclusion of

matchings of ON-cubes instead of checking intersection of matchings with OFF-cubes while creating the graph GIM.

The algorithm to find Minimal Implicant is slow and needs to be reinvestigated. This must be done to insure a good (optimal) solutions.

The limitations of the program result could be due to the way of the implementation itself rather than the method.

In short, the result of the study of Graph Coloring, PLAMCO and PALMINI, gives us a good foundation for further investigation of other variants. With little effort, next students can easily extend the algorithm to support

- Multivalued Logic Functions.
- Multilevel Logic Functions.

CHAPTER V

ZAP A GAL BOARD

INTRODUCTION

The purpose of this chapter is to show how to design a GAL programmer. Actually, the design is capable of programming EPROMs, EEPROMs, EPLDs, and GALs. For the scope of the thesis, only the GAL section is mentioned in detail. The author chooses the Lattice GAL for the following reasons:

- GAL can emulate many different types of PAL.
- GAL is reprogrammable while PAL is not. This makes GAL ideal for prototypes.
- Building a GAL programmer is much easier and cheaper.
- GAL is designed with new technology, EECMOS technology, with very low power consumption.

The design of the ZAP A GAL board consists of two parts. One is HARDWARE and the other is SOFTWARE. The host of the ZAP A GAL board is a PC XT or AT personal computer.

INTRODUCTION TO GENERIC ARRAY LOGIC (GAL)

The Lattice E²CMOS GAL device combines a high performance CMOS process with electrically erasable floating gate technology. This programmable memory technology applied to array logic provides designers with reconfigurable and bipolar performance at significant reduced power levels when compared with bipolar PALs. Lattice also guarantees that a GAL device can be programmed and erased at least 100 times and data retention will be at least twenty years.

The 20-pin GAL16V8, which will be described in this chapter, features 8 programmable Output Logic Macrocells (OLMCs) allowing each output to be configured by the user. Each output can be configured as a dedicated input, dedicated asynchronous output, bidirectional output, and bidirectional synchronous output. With these OLMCs, the GAL16V8 is capable of emulating, in a functional/fuse map/parametric compatible device, all common 20-pin PAL device architectures. The output of each OLMC can be program as active high or low. If it is programmed as dedicated output pin, that particular OLMC can have eight product terms instead of seven for PALs. In addition, Lattice GAL offers a very useful feature. That is the security protection via the Security Cell. After programming the GAL, one can prevent others from observing or

copying the content of the design by programming the Security bit. Following is the picture of a GAL16V8 logic diagram.

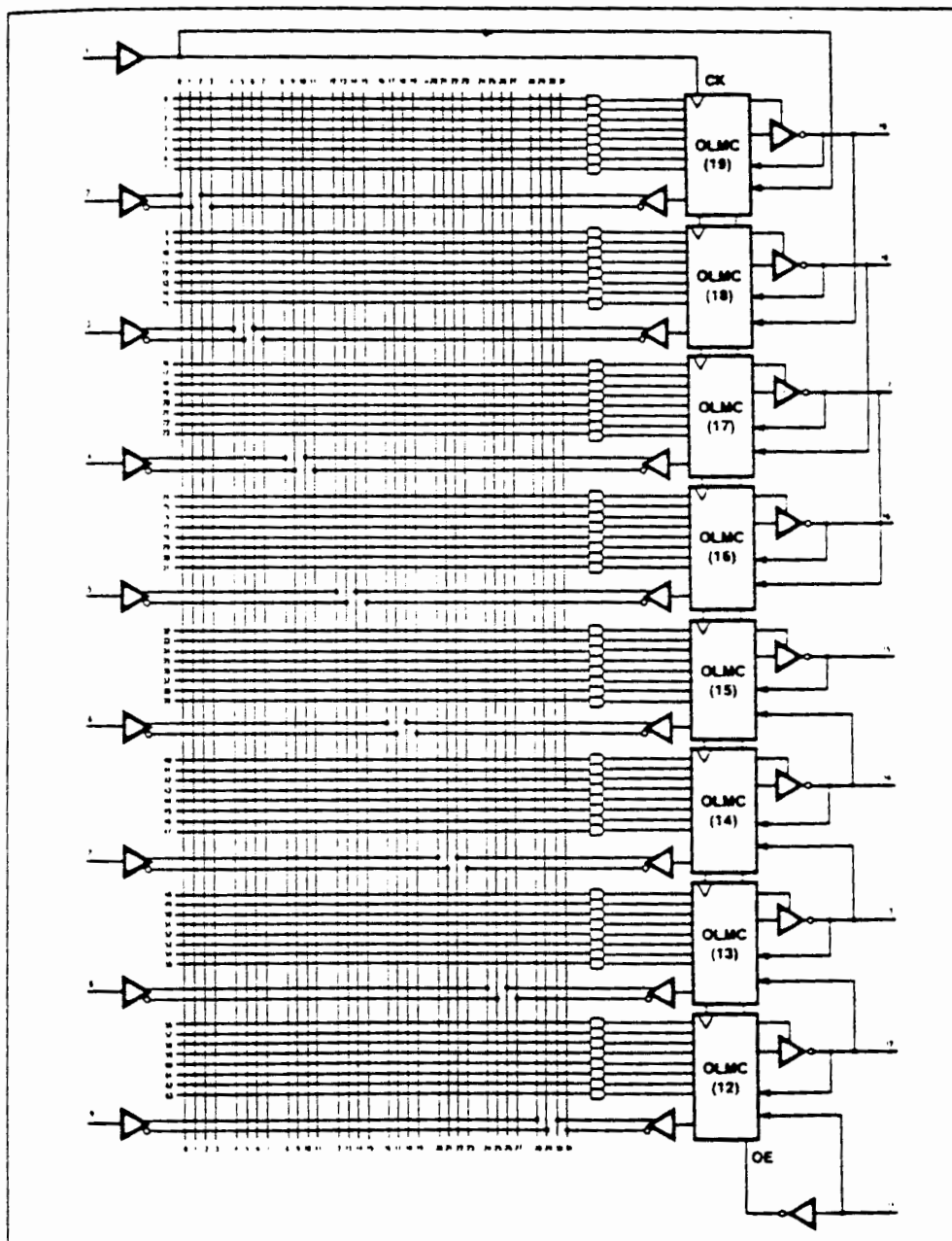


Figure 5.1. GAL16V8 logic diagram

From the logic diagram, Pin 1 can be used either as input pin or clock pin as in registered PALs. Pin 11 can be used as input or as Output Enable Control pin as in registered PALs. Lastly, OLMC12 through OLMC19 can be user prorammmable. The Figure 5.2 shows the logic diagram of one of the OLMC cells.

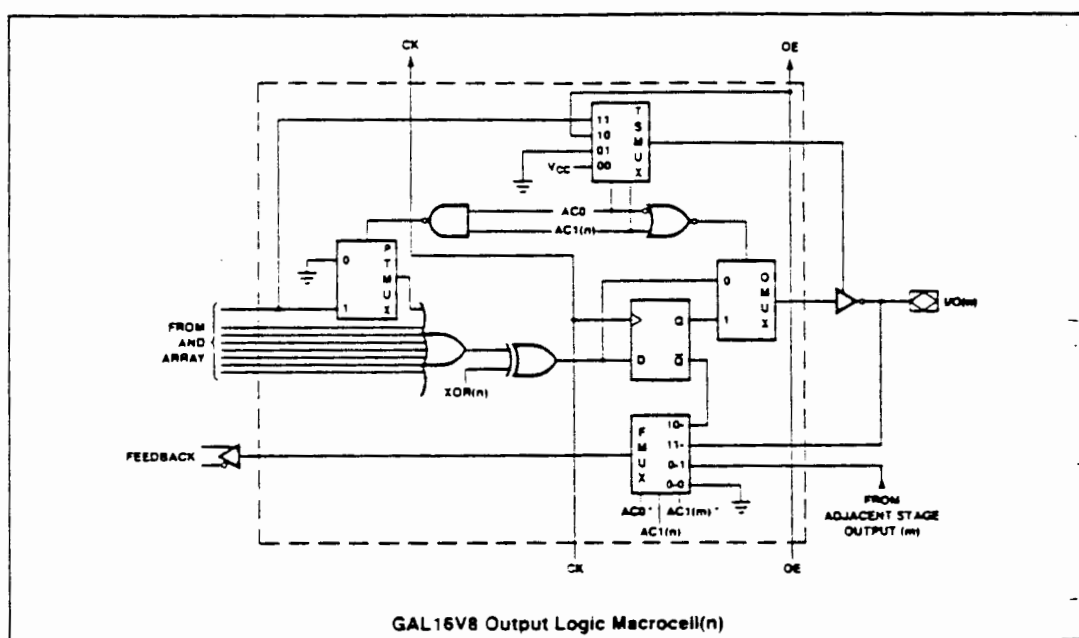


Figure 5.2. Output macro cell

The designer can configure the OLMC to one of the options described above by programming the bits AC0, AC1, and SYN for each OLMC. These bits are located in the Architecture Array which will be described in detail later in the chapter.

OVERVIEW OF THE ZAPAGAL BOARD

The Zapagal board consists of two pieces. The first piece is the adapter board which can be plugged in any eight bit slot of any PC XT or AT computers. At the end of the board is a 50-pin locking edge connector. A 50-pin ribbon cable connects the adapter board to a socket board which is a small printed circuit board which contains a 20-pin dip socket. The length of the cable can be as much as three feet long. The reason to have a separate socket adapter is as follows. To support many different devices with different pinouts and possible future devices, all we have to change are the socket adapter board and software. The Figure 5.3 below shows the block diagram of the Zapagal board.

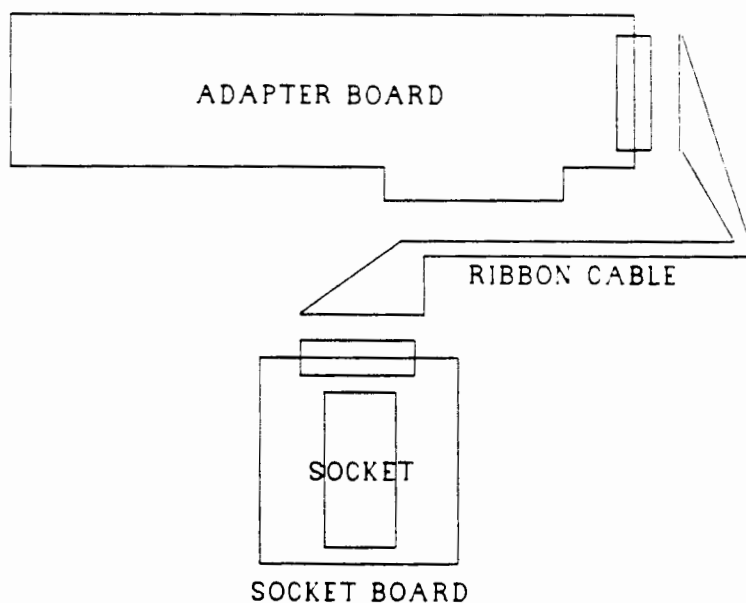


Figure 5.3. Zapagal block diagram

ZAPAGAL HARDWARE

The Zapagal board is a PC XT add in card. It fits and meets all the electrical interface for both PC XT and AT (8mhz) computers. It functions as an I/O board. It does not have any firmware on board nor any microprocessor. Hence, all the control software is coming from the host PC. Thus, it is very convenient to develop software for it because we can use all the features of PC DOS.

This Zapagal board can potentially perform as a very expensive programmer in the commercial market. It costs less than \$100 to build and it can do the task of programmers in \$1000 range.

The board is designed to program the following devices:

- Lattice GAL 20 and 24 pin devices.
- Altera EPLD 20 and 24 pin devices.
- Erasic EPLD 20 and 24 pin devices.
- EPROM and EEPROM from 2764 upto 27010.
- Any CMOS PLDs in the future.

One of the features of this board is that it is device programmable selectable. The Zapagal board behaves like a permanent adapter. All we need to do is to change the socket board which contains device sockets and software to accommodate new devices. All the address and data pin to the socket board are tristatable and bidirectional; this

allows the board to accommodate any CMOS PLDs in the future.

The picture 5.4 shows the block level schematic of the board. It has six main blocks.

1) DECODE1: this block contains the circuit for PC interface.

2) DECODE2: this block contains circuits for the timer and I/O pins.

3) DECODE3: this block contains circuits for more of I/O pins.

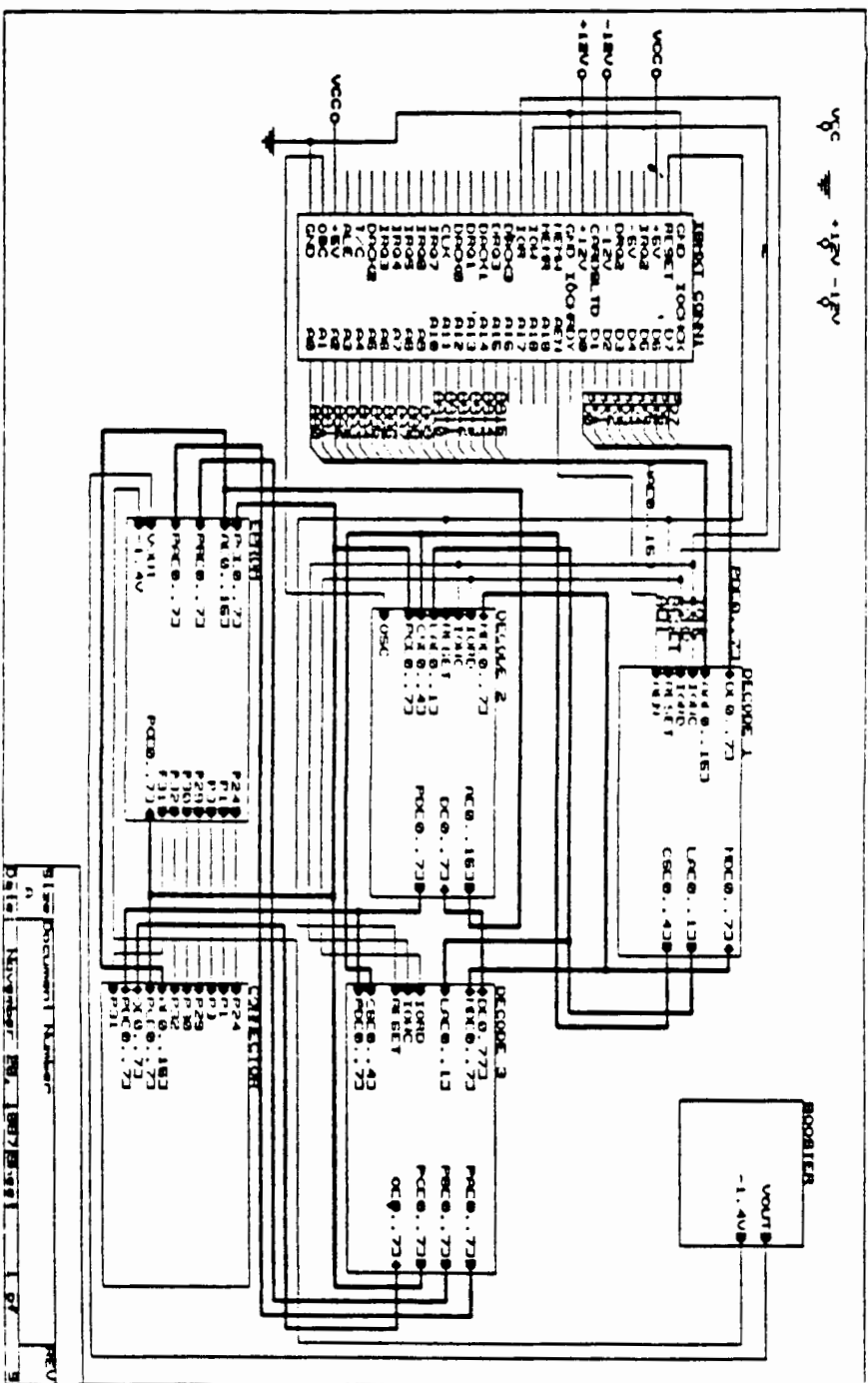
4) EPROM: this block has two subblocks.

EPROM1 and EPROM2 contains circuits to generate programming voltages for EPROM types.

5) BOOSTER: this block contains circuits to generate the supervoltage 16.50 Vdc for GAL.

6) CONNECTOR: this block contains the connector to the daughter board.

Figure 5.4. Zapagal block diagram



The following table shows the address map of the board and the LSI components on the board:

Base address = 130H.

Chip Select	Hex Address	Device Number	Device Name
CS0	130H - 13FH	U2	Timer i8254
CS1	140H - 14FH	U1	Parallel port i8255A
CS2	150H - 15FH	U22	Parallel port i8255A
CS3	160H - 16FH	U24	Parallel port i8255A

DECODE1: PC INTERFACE.

The circuit for PC interface is quite simple. The circuit is designed to respond to any I/O READ or WRITE cycle within the address range: 13XH to 17XH.

A PLD device is used to decode the internal /RD or /WR and the equation is as follows:

$$\begin{aligned}
 \text{RD} = & \text{/(IORC} * \text{/RESET} * \text{BDSLT} * \text{/AEN} * \text{/A7} * \text{/A6} * \text{A5} * \text{A4} \\
 & + \text{IORC} * \text{/RESET} * \text{BDSLT} * \text{/AEN} * \text{/A7} * \text{A6} * \text{/A5} \\
 & + \text{IORC} * \text{/RESET} * \text{BDSLT} * \text{/AEN} * \text{/A7} * \text{A6} * \text{/A4}); \\
 \text{WR} = & \text{/(IOWC} * \text{/RESET} * \text{BDSLT} * \text{/AEN} * \text{/A7} * \text{/A6} * \text{A5} * \text{A4} \\
 & + \text{IOWC} * \text{/RESET} * \text{BDSLT} * \text{/AEN} * \text{/A7} * \text{A6} * \text{/A5} \\
 & + \text{IOWC} * \text{/RESET} * \text{BDSLT} * \text{/AEN} * \text{/A7} * \text{A6} * \text{/A4});
 \end{aligned}$$

The IORC from the PC is used to control the direction control of the data bus transceiver, U10, 74LS245.

The address A0 and A1 are buffered and become LA0 and LA1 before being used to access specific registers in LSI devices.

Since the above design has already been prototyped, otherwise the PLD device can replace the U11, 74LS08, and U12, 74LS138 and save two ICs.

DECODE2: TIMER CONTROL

Each device requires different pulse duration for programming purposes. Thus, we must have a programmable timer source. One easy way is to use the software loop as a timer. However, this scheme will not work because different PCs run at different frequencies. For instance, a PC XT 8 mhz is running twice as slow as an PC AT 6 mhz. If we use the software loops, then the same software will have two different effects on two different machine. This will cause the Zapagal board not to work. Hence, we must have a fixed timer source on our board. The author chose the INTEL 8254 sixteen bit timer. The input frequency comes from the fixed oscillator, 14.2 mhz, on the mother board. This oscillator is used for TV monitor and is fixed on any PC XT or PC AT. This frequency was divided by 4 and then fed in to the counter timer. This worked out very well.

U1, i8255A-5, 24 bit parallel port, is used to control the GAL. Port A and B are fed through U3 and U5, 74LS244

tristate buffers, since GAL devices require many pins to be floated during entering Edit mode and exiting Edit mode. Port C is used to control SDIN, P/V, /STR pin.

DECODE3: EPROM DECODE.

This block contains two more i8255A-5 chips that are used for EPROM devices. Hence, it is not in the scope of this chapter and will not be discussed.

Port A of U24, i8255A-5, is used to sample data from the SDOUT pin.

BOOSTER: VOLTAGE CONVERTER.

A DC to DC converter chip, LM3578, from National Semiconductor, is used to perform the voltage conversion. This chip is a new product, 1987, and is very inexpensive and easy to use. It is configured in the fly back mode. The voltage gain is set by resistors R3 and R1.

$$V_{out} = R3/R1 + 1.$$

The combination of C5, C4, C3, and R2 sets the duty cycle (50%) for the squared wave at pin 6 of the LM3578. During the lower half of the pulse, the energy is released through the inductor and sustains the load. On the high half of the pulse, the energy is built up in the inductor and the capacitor, C1, supplies energy to the load. The Schottky diode D1, 1N5817, needs to be a fast switching diode to keep a good load regulation. The voltage V_{out} is set to

be around 22 Vdc because this voltage is then passed through another programmable voltage stage to generate VEDIT, 16.5 Vdc.

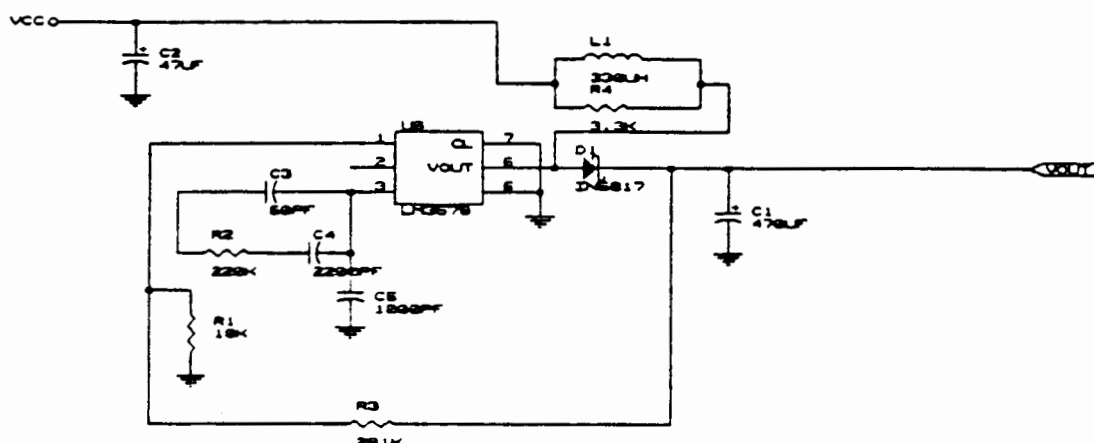


Figure 5.5. DC-DC converter

EPROM1 and EPROM2.

This block contains circuitry to implement programmable power supplies for GALs and EPROMs. In the following section the programmable voltage converter for the super voltage will be discussed in detail. The other programmable voltage converters work the same way.

In order to enter the Edit mode, we need to apply 16.50 Vdc to Edit pin. Normally it is at 0 or at 5 Vdc. Thus we have to have a way to set the voltage to three different values via programming the software. Normally,

this can be done using Digital-Analog converter chip. However, this design could be expensive. The following shows a very inexpensive way to implement the circuit.

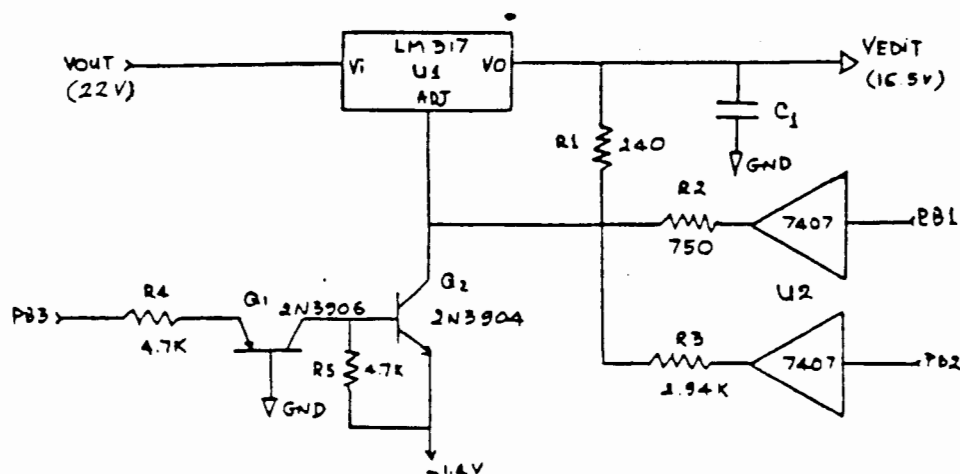


Figure 5.6. Programmable voltage converter

The inexpensive adjustable voltage regulator, LM317, is chosen for the design. This device is very easy to use. The input voltage comes from the VOLTAGE BOOSTER block, 22 Vdc, and goes to input pin Vi. The output voltage is determined by the following transfer function:

$$VEDIT = 1.25 * (1 + R_x/R_1).$$

In this case, our R_x is either R2 or R3. Additionally, if we apply a slightly negative voltage to the adjust pin, ADS, $V_{out} = 0$ Vdc.

The circuit which interfaces between digital and negative analog voltage is done via the transistor pairs

NPN/PNP Q1 and Q2. If PB3 is high "1" digitally, both Q1 and Q2 are turned on; the node ADJ will be pulled down to 0 Vdc and shut off Vout regardless of any gain network. Q1 is needed to absorb any negative voltage across its collector-emitter so that PB3 will not see any voltage below 0 Vdc. If PB3 is programmed "0", Q1 and Q2 are turned off. Effectively, Q2 is removed from node ADJ. Consequently, the Vout is now the function of the gain network of R1 and Rx.

The circuit which interfaces between digital and analog output voltage is done via U2, 7407 chip. The output of this chip is open-collector type and can operate from 0 Vdc upto 30 Vdc. So, if we want to turn on VEDIT, 16.5 Vdc, PB1 should be programmed high and PB2 low. With PB1 high and PB2 low, the upper path which consists of R2, 750 ohms, and the gate 7407 is off and considered disconnected from the circuit. On the other hand, the lower path which consists of R3, 2.2k ohms, and the gate 7407 is on. The current flows through R1, R2, and through the gate to constitute a complete path. Hence the VEDIT is equal to the gain network $= (1 + 2.94k/240) = 16.56$ Vdc. All the resistors must be 1% tolerance to stay within GAL's electrical specification. Similarly, if we want to set VEDIT to 5 volts, then program PB3, PB1 low and PB2 high. The rest of the programmable voltages for other pins function similarly.

All the I/O pins come from the parallel ports, i8255A-5. Thus, programming the polarity of PB3, PB2, and PB1, is just the matter of programming the registers of the LSI i8255A-5 and considered to be easy.

GAL PIN DEFINITION WITH RESPECT TO THE DAUGHTER BOARD CONNECTOR

The following tabel shows the current pin out of the daughter board and the way the software maps the I/O pin for GAL16V8.

GAL PIN	CONNECTOR PIN
1	17
2	44
3	3
4	5
5	7
6	9
7	11
8	33
9	34
10	6,8,10
11	35
12	50
13	27
14	25

15	23
16	21
17	19
18	1
19	36
20	30,32,47

GAL PIN DEFINITION with respect to the connector name:

RAG0 - RAG7 = A0 - A7

VIL0 - VIL7 = A8 - A15

SDOUT = PD0

SCLK = 00

SDIN = 01

/STR = 02

P/V = 03

EDIT = P1

VCC = P30

ZAPAGAL SOFTWARE

OVERVIEW OF PROGRAMMING PLD DEVICES

Most of the commercial PLD devices are programmed as follows:

Step 1: - High level description of the problem.

The designer specifies the state diagram or the Boolean equations.

Step 2: - The compiler then translates the description into a binary format called JEDEC code. JEDEC format is the industry standard format to represent the information for PLDs which most companies follow to represent the binary code for their respective PLD devices. The following page shows an example of a JEDEC code for GAL16V8.

Step 3: - The programming device then uses this JEDEC file to program the device.

The rest of this chapter will concentrate on step 3 only.

OVERVIEW OF JEDEC FORMAT

The JEDEC FORMAT document defines a format for the transfer of information between a data preparation system and a logic device programmer. This format provides for, but is not limited to, the transfer of fuse, test, identification, and comment information in an ASCII representation. This format defines the "intermediate code" between device programmers and data preparation systems. A complete description of the JEDEC format can be found in the ABEL manual. Following is an example of a JEDEC file from ABEL output. The "*" character is a special character which is used to end a special field. The first part is the comment which is used for documentation purposes. It ends with an "*". The next field "QP20" means that this device

has 20 pins. The field "QF2194" means that the total number of fuses in the device is 2194. The field "L0000" designates fuse number 0. At the end, there are two checksums. The first one is the checksum of the content of total number of fuses transmitted, in this case is 2194 fuses. The other is the checksum of all the ASCII characters transmitted in the JEDEC file.

^BABEL(tm) Version 2.00b

JEDEC file for: P16V8C

Created on: 09-Sep-87 07:50 PM

86c38 arbiter

designer: Loc Nguyen

Intel corp Dec/1986 *

QP20* QF2194* L0000

1111111111111111111111111111111111

1010101110111011101110111111111111

0101011101110111011101111111111111

0000000000000000000000000000000000

0000000000000000000000000000000000

0000000000000000000000000000000000

0000000000000000000000000000000000

0000000000000000000000000000000000

0000000000000000000000000000000000

0000000000000000000000000000000000

0000000000000000000000000000000000


```

00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
* L2048  10000000
* L2056
0000000000000000000000000000000000000000000000000000000000000000
0000
* L2120  11111111
* L2128
1111111111111111111111111111111111111111111111111111111111111111
1111
* L2192  11
* C144A
* ^CD89C

```

The fuses which are numbered from L0000 to L2047 are compatible to those of 20-pin PAL devices. The fuses from L2048 to L2192 are specific to the OLMC cells of GAL16V8 devices.

HOW TO PROGRAM A GAL16V8

GAL16V8 from Lattice Semiconductor Corp has its unique way of programming the part. It requires a super voltage of 16.5 volt to bring the part into the programming mode (EDIT mode). In order to load the data (fuse map) into the device, it requires the data to be shifted into a special register

serially. Furthermore, it also has a special way to represent the address location of each bit in the JEDEC fuse map file.

Due to the NON DISCLOSURE AGREEMENT that the author signed with Lattice Corp. The author will not reveal all the information which are important to the programming aspects of the GAL in this thesis. Likely, most of the fuse address location and programming timing parameters are altered accordingly. However, the concept is still correct. If anyone is interested in building one, he or she can prototype one using the enclosed schematic and then write the author for the software. If that person wants to write his or her own software, then that person has to contact Lattice Semiconductor Corp for information.

PROGRAMMING ALGORITHM

A GAL16V8 is programmed as follows:

- 1) Enter the EDIT mode.

Within the EDIT mode, you can perform the following:

- a) Bulk Erase: erase the GAL.
- b) Erase Verification: verify that the device is blank after erase.
- c) Program/Verify Logic Array.
- d) Program/Verify UES Array.
- e) Program/Verify Architecture Array.

f) Program the Security Cell if desired.

2) Exit the EDIT mode.

To enter the EDIT mode, a supervoltage of 16.50 Vdc is applied to the EDIT pin.

Also, in the EDIT mode, a 64 bit Shift Register is active and provides the means to load and unload data from the device via pin SDIN and SDOUT respectively.

Also, in the EDIT mode, the GAL reconfigures itself to give the programmer the access to three arrays: 1) Logic Array, 2) the Users Electronic Signature (UES) array, 3) the Architecture array. Each of these arrays are broken into rows. An array can have several rows, as the Logic array does, or just one row, as the UES array does. To address the different rows in an array, Row Address Gates (RAGs) are used. There is a total of six RAGs on a GAL16V8 device. The RAGs are reconfigured to external pins when a device is in the EDIT mode.

Before any of the arrays in the device can be programmed the device must be erased. To erase a GAL device one procedure (Bulk Erase) is performed and all of the arrays in the device are erased.

An erase verification is performed to make sure that all of the cells in the device were erased and are functional. If a cell does not erase, the device is considered non-functional and should be discarded. If all

of the cells did properly erase the device is ready to be programmed.

The first array in the device to program is the Logic array. Data is loaded into the Shift Register to program into a row of the Logic array. With the data loaded into the Shift Register a row in the Logic array is addressed with the RAGs. With the RAGs set, a programming cycle is performed to the device which will transfer the data from the Shift Register into the addressed row. It is necessary to hold the RAGs constant throughout the programming cycle because they are not internally latched.

After the Logic array is programmed, it is verified that the correct data has been programmed.

The next array to program and verify is the Users Electronic Signature (UES) array.

The last array to program and verify is the Architecture array.

Once all three of the arrays are programmed and verified the user has the option to program the Security Cell. The Security Cell programs in the same fashion, using the same voltage and timing specifications as any cell in the device.

Once all of the arrays in the device have been programmed and verified, the device is ready to be taken out

of the EDIT mode. Upon exiting the EDIT mode the device will internally reconfigure itself back to perform logic operations.

EDIT MODE

To program a GAL16V8 device, it needs to be in the programming mode, called the Edit mode. To enter the Edit mode, one supervoltage of 16.50 volts is applied to the Edit pin of the device. In the Edit mode, the device is internally reconfigured to perform programming operations. When the device is internally reconfigured the external pins of the device are also reconfigured to operate: the Shift Register, the Row Address Gates (RAGs), and the Program/Verify control lines.

The Shift Register provides the means to load and unload data from the device. The Shift Register operates on standard TTL levels as do all the programming control signals.

In the Edit mode, the array of the device is broken down into three unique arrays: The Logic array, The Users Electronic Signature array and the Architecture array. These three arrays are broken down again into rows. The number of rows in an array is dependent on that array. The Logic array for a GAL16V8 consists of 32 rows, while the Architecture array consists of only one row. The RAGs

address all of the different rows in an array. There are six RAGs (RAG0-RAG5) which address all of the rows in an array.

Two more pins are configured to control the programming and verifying operations of the GAL16V8 device, Strobe-bar (/STR) and Program/Verify-bar (P/V). P/V determines if the device is to be programmed or verified. By applying a high signal (logic "1") to the P/V input, the device will enter the programming state. By applying a low level (logic "0") to the P/V input, the device will enter the verify state.

In the desired state, pulsing /STR low for the appropriate time produces a program or verify cycle. A programming cycle will transfer the data from the shift Register into the addressed row. A verify cycle will transfer the data from the addressed row into the Shift Register.

In the Edit mode, there are several pins that are unused, the pins must be connected to VIL or ground.

SDOUT is an open drain output that must be connected to VIH through a resistor (10K ohms).

Whenever in the Edit mode the P/V input should always be held at a logic "0", unless a programming cycle is to occur. /STR should be held at VIH at all times, except when

performing an actual program, verify, or load cycle.

The Edit mode pinout of the GAL16V8 is shown in figure 5.7 below.

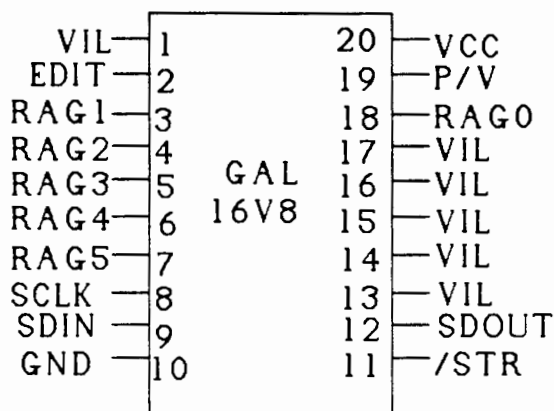


Figure 5.7. Edit mode pinout

ENTERING THE EDIT MODE PROCEDURE

When preparing to enter the Edit mode, all of the normal output pins on the device should be floated, or terminated through a high impedance of 10K ohms or greater to ground.

VIE, the Edit mode voltage is applied to pin 2 of the device, and the device will enter the Edit mode. The rise time of VIE is important. (Please contact Lattice Semi Corp for exact information about this timing).

In the Edit mode there are several unused pins on the device. These unused pins should be terminated to VIL or ground whenever in the Edit mode.

PROCEDURE:

1) Float all the normal output pins or terminate through a high impedance of 10K ohms or greater to ground.

GAL16V8 pins 12 - 19

2) Select the Edit mode by placing VIE (16.50 Vdc) on pin 2, the Edit mode pin.

3) Terminate all unused pins to VIL or ground, do not float.

4) Apply: VIH to /STR.

VIL to P/V.

EXITING THE EDIT MODE PROCEDURE

When programming is completed, the device needs to be taken out of the Edit mode. When preparing to exit the Edit mode all of the normal output pins on the device should be floated, or terminated through a high impedance of 10K ohms or greater to ground.

VIE, the Edit mode voltage is removed from pin 2 of the device, and the device will exit the Edit mode. Pin 2 should be connected to GND or VCC after exiting the Edit mode.

PROCEDURE:

1) Float all normal output pins through a high impedance of 10K ohms or greater to ground.

GAL16V8 pins: 12 - 19.

2) Remove VIE(16.50 Vdc) from pin 2, the Edit mode pin

SHIFT REGISTER OPERATION

The Shift Register is active in the Edit mode and three external pins are designated for its operation. These pins are: Serial CLock (SCLK), Serial Data Input (SDIN), and Serial Data Out (SDOUT). The SDIN is the input to the Shift Register, and SDOUT is the output of the Shift Register. Data is clocked into or through the Shift Register on the falling edge of the SCLK. It is possible to clock data straight through the Shift Register without performing a program or verify cycle.

The Shift Register operates on a first in first out format (FIFO). The first bit of data loaded into the device is located in the most significant bit of the array, product term 63 for a 16V8. Clocking the Shift Register 63 times will shift the data bit to the least significant bit location of the Shift Register, product term 0 for 16V8. The data in least significant bit of the Shift Register is always present on SDOUT.

When rows 60, and 63 are addressed the Shift Register is reconfigured to be different lengths. When row 60 is addressed the Shift Register reconfigures to 82 bits (Architecture array). When row 63 is addressed the Shift Register reconfigures to be transparent, the data applied to

SDIN will appear immediately on SDOUT.

The timing waveforms for loading and unloading data from the Shift Register are shown below in Figure 5.2.

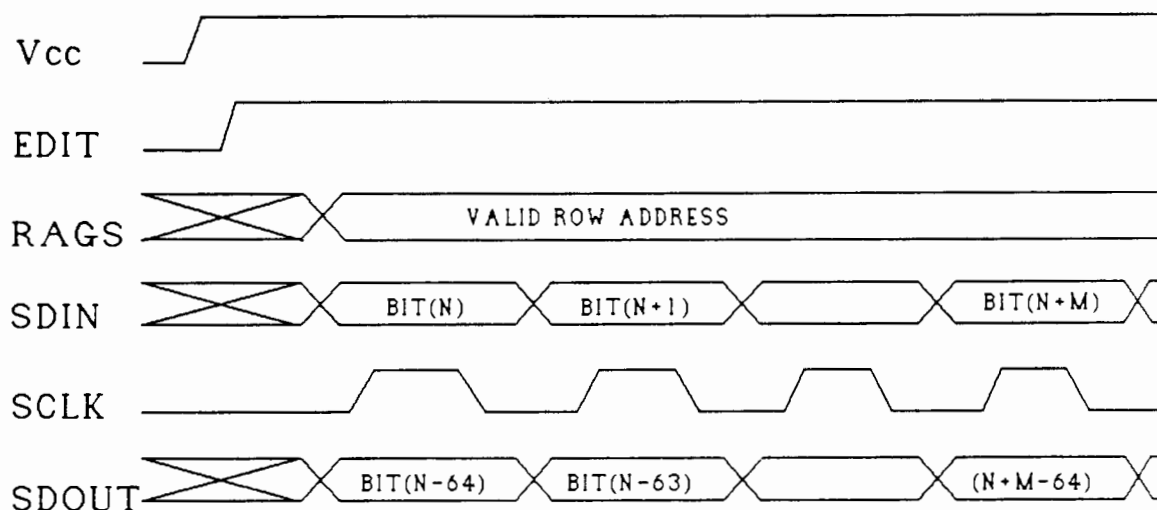


Figure 5.8. Shift register I/O timings

ADDRESSING ROWS

A GAL16V8 device is broken down into three array: the Logic array, the UES array, and the Architecture array. All three of these arrays consist of one or more rows. The relationship between the arrays and the rows is shown in figure 5.3. The picture shows the number of rows for GAL16V8.

To program data into a GAL16V8 device, a row in an array needs to be addressed. There are a total of 36

functional rows in a 16V8 device. To address a row in the device Row Address Gates (RAGs) are used; the GAL16V8 has six RAGs (RAG0 - RAG5).

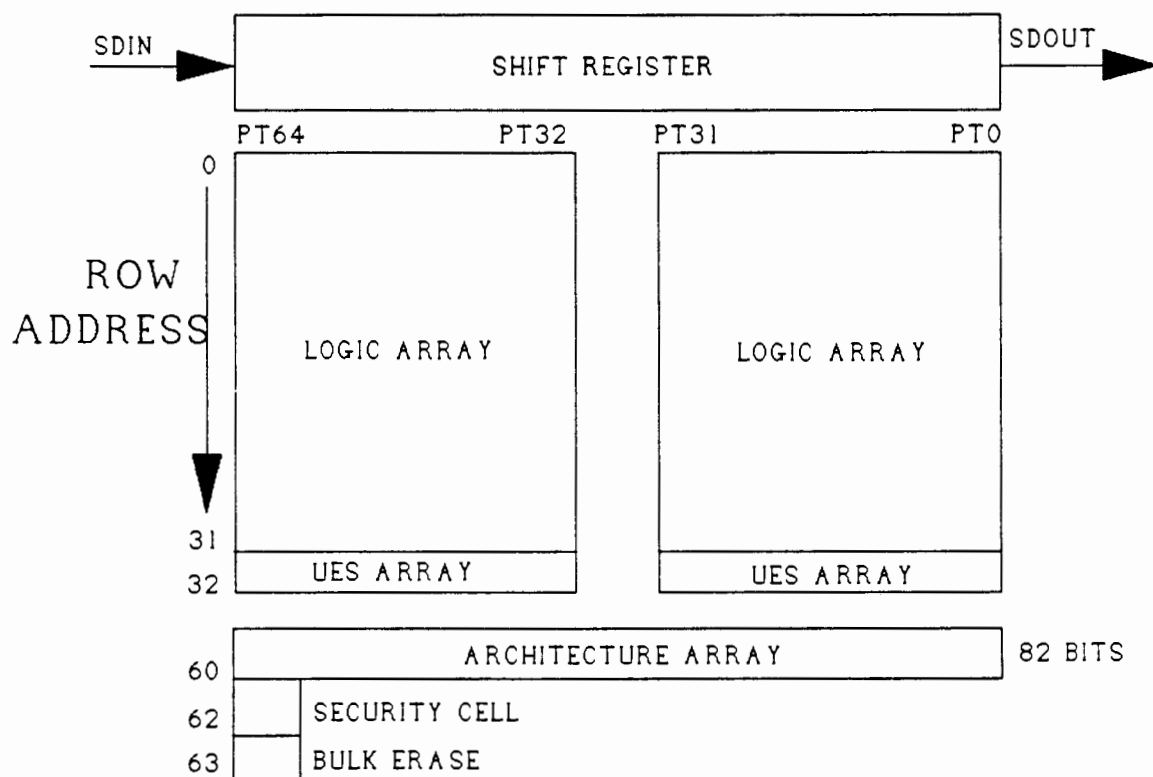


Figure 5.9. Array maps for GAL16V8

The RAGs are not internally latched so during a program or verify cycle the RAGs must be held constant. Only when a program or verify cycle is complete ($/STR = 1$) is it acceptable to change the RAGs.

BULK ERASE PROCEDURE

Before any of the E²CMOS cells in a GAL device can be programmed, they need to be erased. The reason why a GAL device must be Bulk Erased is as follows. A cell that is programmed equals a logic "0", and a cell that is erased equals logic "1". When a cell is programmed to a "0" from a "1", the charge on the floating gate is altered. It is only possible to change the charge on the floating gate of a cell through the Shift Register from a "1" to a "0". It is impossible to change the charge on the floating gate via the Shift Register from a "0" to a "1". The only way to change the charge on the floating gate from a "0" back to a "1" is to perform a Bulk Erase. The Bulk Erase procedure is, therefore, an initialization of all arrays in the device to a logic "1".

The following procedure shows how to perform a Bulk Erase on a GAL16V8 device.

- 1) In Edit mode
- 2) Address row 63 using RAG0 - RAG5 and hold constant
- 3) Apply: VIH to P/V
VIH to SDIN
- 4) Pulse /STR low for 50 ms
- 5) Apply VIL to P/V

Note: If the Security cell is set before performing a Bulk Erase, the programmer will not be able to edit any rows

in the device. The programmer needs to take the device out of the Edit mode and then back in. Exiting and reentering the Edit mode resets an internal latch, giving the programmer access to all the rows.

VERIFY PROCEDURE

The Verify procedure determines if a row has been programmed correctly, or if a Bulk Erase has properly occurred. If a Verify procedure is performed to verify that a row is correctly programmed, the original data programmed into the device is needed for comparison. If a Verify procedure is performed to verify that a Bulk Erase occurred properly, the data in the device needs to be verified it is all "1s".

In a Verify procedure a designated row in an array is addressed using RAG0 - RAG5. A Verify cycle performed and the data stored in the addressed row is transferred into the Shift Register. The data transferred into the Shift Register is now available to be shifted out through the Serial Data Output.

The following procedure shows how to perform a Verify procedure.

- 1) Select a row to verify using RAG0 - RAG5 and hold constant.
- 2) Pulse /STR low for 1 ms.

- 3) Shift the data out of the Shift Register.

- 4) Compare the data Programmed into the device to the original data.

PROGRAMMING PROCEDURE

A row of an array in a GAL is programmed as follows. First, the desired data to program is loaded into the Shift Register. Next, RAG0 - RAG5 are set to address the appropriate row and held constant. To perform the programming cycle, apply VIH to P/V and pulse /STR low for 10 ms. After the /STR pulse is complete, return P/V to VIL.

Programming a GAL16V8 is straight forward in that each row is read from memory and programmed into the device. The file in memory could have been a JEDEC file down loaded from a disk, or loaded into the programmer from another device.

The following procedure shows how to perform a programming cycle.

- 1) Load the Shift Register with the desired data.

- 2) Address a row to program using RAG0 - RAG5 and hold constant.

- 3) Apply VIH to P/V.

- 4) Pulse /STR low for 10 ms.

- 5) Return P/V to VIL.

SECURITY CELL PROCEDURE

All the Lattice GAL devices feature a Security Cell so

that it is impossible to copy or observe the Logic array in a GAL device. However it is always possible to observe the UES array and the Architecture array in a secured device. If a device is secured, programming and verification of the Logic array is impossible, until a Bulk Erase is performed.

To secure a GAL device, row 61 is addressed with the RAGs and held constant. VIH is applied to both SDIN and P/V. It is not necessary to clock a "1" into the Shift Register when row 61 is addressed because the Shift Register is transparent (SDIN = SDOUT). A programming cycle is performed on row 61 by pulsing /STR low for 10 ms. Upon completion of the program cycle, P/V is returned to VIL.

At this point in the process, the device is not yet secured. The device needs to exit and reenter the Edit mode to set the Security Cell latch. Exiting and reentering the Edit mode clocks the Security Cell latch and inhibits access to the Logic array. Further programming and verification of all arrays is allowed until the Edit mode is exited, at which time the device becomes secured. Once the Security Cell is latched, data read from the Logic array will be all "1s"; the device appears erased.

The following procedure describes how to secure a device.

- 1) Address row 61 using RAG0 - RAG5 and hold constant.
- 2) Apply VIH to P/V and SDIN.

3) Pulse /STR low for 10 ms.

4) Return P/V to VIL.

Note: The User Electronic Signature array and Architecture array can not be secured. This data is always available to the user to observer.

All the low level software is written in C language. Writing the low level software is easy but tedious. All one needs are the address locations and the specification of the boards and the LSI chips. One can obtain these information from INTEL data book and Lattice Semiconductor Corp for GAL programming details. However, there is a great deal of high level software that one needs to write to make the product marketable. One of the immediate needs is the way to download the JEDEC format file to the device and also the way to upload the content of the device to the standard JEDEC format. The JEDEC format can be obtained form IEEE standard committee.

At the moment, a minimum amount of software was written to use the product effectively. It consists of following screen menu:

Screen1:	GAL TYPE
	16V8 TYPE 1
	20V8 TYPE 2
	EXIT TO DOS X

Screen2:

GAL TYPE SELECT: 16V8 (If chosen)

MENU - LOAD <L>
- VERIFY <V>
- PROG <P>
- UPLOAD <U>
- DOWNLOAD <D>
- EDIT <E>
- EXIT <X>

Screen3:

GAL TYPE SELECT: 16V8

MENU - MAIN ARRAY <A>
- UES ARRAY
- ARCH ARRAY <C>

If you want to obtain a copy of this software, please
write to the author at:

LOC NGUYEN

1323 S.W. 213 AVE

ALOHA, OR. 97006

Note that for different devices like EPROMs or PLDs,
you may have to rewrite many pieces of code. It turns out
that the effort to build the hardware is very small compared
to the total time to spend for writing and maintaining
software.

CONCLUSION

The Zapagal board was successfully built and tested. It is used extensively for EPROM and GAL programming at home. At the moment, ABEL is used to compile the Boolean equations to JEDEC code. It is obvious that a compiler can be written to incorporate PALMINI to compile the Boolean equations to JEDEC code. When it is done, we will have a complete integrated tool from software to hardware.

At work, the author has some friends who are making fabs for this board. It is their opinion that the product is marketable and it will be a lowcost, useful tool to the lab bench.

Enclosed is the complete schematic of the Zapagal board. Again, due to the non-disclosure agreement with Lattice, the author can not disclose all the detailed analysis including timing parameters which are necessary to program the GAL. Anyone who builds the board according to the schematic and uses author's software will find that it works.

CHAPTER VI

CONCLUSION

Recently, the race in introduction of new PLD devices has become very hot on the market. Every manufacturer wants a piece of the vast, 1 billion dollar, market by 1990. The projection was made by Data Quest Source. We begin to see the emergence of new architecture like On-chip-programmable PLD like GAL16Z8 and those of Zilink and the gate array cell type PLD of Zilink as well as multiple layer NOR-NOR PLD of Erasic. Within the PLD technology, the CAD tool aspect is still behind the chip technology. Hence, the CAD tool provides a very good field to do further research on. Some of the hot topic and also the immediate needs for CAD tools are: functional and timing simulation, routing and fitting devices, functional logic partitioning of a design into multiple PLDs, automatic state assignment, and lastly logic synthesis.

In this thesis, we were concerned with two kinds of questions:

- The first one related to the theory and algorithms.
- The other related to the practical implementation of a system.

With respect to the first group of question, we have investigated a new approach to the Boolean Minimization. Almost all of the existing algorithms for exact minimization of Boolean functions solve in sequence two N-P complete problems. The first one is the generation of all prime implicants and the other is the set covering problem. In the present approach, we only have to solve one N-P complete problem; that is the graph coloring problem. We think that this approach is general and can be used in CAD. It permits us to use the existing graph coloring algorithms which have been optimized to very high extents. In addition, a lot of very sophisticated mathematical analysis have been done for these algorithms. The next contribution is a new method for designing hazardless-two-level networks.

The proposed rules for state assignment are based on new principles not found in literature. It should be an interesting topic for further research to formulate the given-by-me rules and see how they relate to the existing state assignment methods. Carefull analysis of the rules can perhaps lead to some theorems and properties that would prove that this algorithm will give efficient results for wide classes of state machines using D-flip flops.

The proposal of a front end chip for self-synchronized circuits is also introduced. This should ease the design task, lower the cost and the board space.

Another group of problems are related to the integration of PLD systems. This is a challenge and will require a lot of effort and time. Besides the hardware aspects, it requires a lot of software modules like language processor, user interface, etc. Lastly, it requires the integration of all the CAD software and hardware together. This can be a very good long term project for a group of students.

BIBLIOGRAPHY

- [1] Bartholomeus, M., : "Prestol-II: Yet Another Logic Minimizer for Programmed Logic Arrays", Proc. Int. Symp. Circ. Syst., June 1985, p.58.
- [2] Brown, D.W. : "A State Machine Synthesizer-SMS", Proc 18th Design Automation Conference, p.301, June 1981.
- [3] Dietmeyer, D.L. : "Logic Design of Digital Systems", Allyn and Bacon, Boston, 1971.
- [4] Dagenais, M.R., Agarwal, V.K., Rumin, N.C. : "McBoole: A New Procedure for Exact Logic Minimization", IEEE Trans. on CAD, Jan. 1986, pp. 229 - 238.
- [5] Hong, S.J., Cain, R.G., Ostapko, D.L. : "MINI: A Heuristic Approach for Logic Minimization", IBM J. Res. and Dev., Vol.18, No.5, pp. 443-458, September 1974.
- [6] Perkowski, M.A., Nguyen, L.B., Goldstein, N.B. : "An Approach to Minimization, Decomposition, and Partitioning of PLA and PAL Circuits Based on Multi-Valued Algebra and Graph Coloring", Technical Report, Dept EE, PSU 1987.
- [7] Rudell, R., Sangiovanni-Vincentelli, A., : "Espresso-mv: Algorithms for Multiple-Valued Logic Minimization", Proc. 1985 Custom Integrated Circuits Conference, pp. 230 - 234, Portland, May 1985.
- [8] Rudell, R. : "Multiple-Valued Logic Minimization for PLA Synthesis", Research report, June 5, 1986, Univ. of California, Berkeley.

- [9] Nguyen, L.B, Perkowski, M.A., Goldstein, N.B. :
"PALMINI - Fast Boolean Minimizer for Personal Computers.", Proc. 24th Design Automation Conference, p. 615 - 621, 1987.
- [10] Nguyen, L.B, Perkowski, M.A., Goldstein, N.B. :
"Boolean Minimization for PALs Using Graph Coloring On Personal Computers", Northcon. Conf. Proc., 1986.
- [11] Sasao, T. : "An Algorithm to Derive the Complement of a Binary Function with Multiple-Valued Input", IEEE Trans on Comput., Vol. C-34, No.2, pp. 131 - 140, Feb. 1985.
- [12] Laarhoven, P.J.M., Aarts, E.H.L, Davio, M. :
"PHIPLA - A New Algorithm For Logic Minimization", Proc. 22th Design Automation Conference, p. 739 - 743, 1985.
- [13] Brayton, R.K., Hachtel, G.D., McMullen, C. T., Sangiovanni Vincentelli, A.L. : "Logic Minimization Algorithms for VLSI Synthesis." Kluwer Academic Publishers, 1984.
- [14] Ulug, M.E., Bowen, B.A. : " A Unified Theory of the Algebraic Topological Methods for the Synthesis of Switching Systems", IEEE Trans. on Comput., p. 255 - 267, March 1974.
- [15] Tracey, J.H., "Internal State Assignments for asynchronous sequential machines," IEEE Trans. Electronic Computers, vol. EC-15, pp. 551-560, Aug 1966.
- [16] Bredeson, J.G., and Hulina, P.T., "Generation of a clock pulse for asynchronous sequential machines to eliminate critical races," IEEE Trans Computer, vol. C-20, pp. 225-226, Feb. 1971.

- [17] Rey, C.A., and Vaucher, J., "Self-synchronized asynchronous sequential machines," IEEE Trans. Computer., vol. C-23, pp. 1306-1311, Dec. 1974.
- [18] Chuang, H.Y.H., and Das, S., "Synthesis of multiple input change asynchronous machines using controlled excitation and flip-flops," IEEE Trans. Computer., vol. C-22, pp. 1103-1109, Dec. 1973
- [19] Huertas, J.L., and Acha, J.I., "Self-synchronization of Asynchronous Sequential Circuits Employing a General Clock Function," IEEE Trans. Computer., pp. 297-300, March. 1976.
- [20] Unger, SH., "Self-synchronizing circuits and nonfundamental-mode operation," IEEE Trans. Computer., vol. C-26, pp. 278-281, March 1977.
- [21] Yenersoy. O., "Synthesis of Asynchronous Machines Using Mixed-Operation Mode," IEEE Trans. Computer., vol. C-26, pp. 325-329, Apr. 1979.
- [22] Kirkpatrick, D.C.; Powers, V.M., "An asynchronous design style to achieve ultimate operating speed," Fifth Annual International Phoenix Conference on Computers and Communications: PCCC'86. pp. 662-673, 1986.
- [23] PH.D Dissertation at OSU by Kirkpatrick, D.C; 1985. "Design of Self-Synchronized Aysnchronous Sequential State Machines Using Asymmetrical Delay Elements".
- [24] Robert Breuninger, William Thompson, "Metastability Evaluation of Logic Technologies", Texas Instruments Inc, 1986.
- [25] Lattice GAL Data Book, Spring 1988.
- [26] EPLD Hand Book, ALTERA, 1986
- [26] PAL Programmable Array Logic Handbook, MMI, 1981

- [27] Intel Memory Data Book, 1987, 1988
- [28] Intel Communication Handbook, 1988
- [29] Steve Garcia, " Build an intelligent serial EPROM programmer", BYTE magazine, p103 - p119, Oct. 1986.
- [30] DOS 3.1 User Guide, Microsoft Corp, 1987.
- [31] TUROBC, Borland International Corp, 1987.

APPENDIX A

MINIMAL BACTRACKING ALGORITHM FOR PROPER GRAPH COLORING

Algorithm 9.

Proper Coloring of the Graph, (Minimal Bactracking Algorithm)

A1. Create the initial node $N=0$ of the solution tree.

$NODE1 = SMI(f)[1]; NODE2 = SMI(f)[2];$

{ NODEX refers to a node in graph GIM }

$ALL-COLORS = \{1, 2, \dots, CARD(SMI(f))\};$

$N = 0; CF_{min} = CARD(SMI(f));$

if { $NODE1, NODE2\} \in RS$ then $CF(N) = 2$ else $CF(N) = 1;$

$QS(N) = \{ (NODE1, 1), (NODE2, CF(N)) \};$

if { $NODE1, NODE2\} \in RS$ then $M = 3$ else $M = 2;$

{ M is the number of next node of GIM }

$MI = SMI(f)[M];$

$GS(N) = ALL-COLORS - \{ COLF(MI_i) | \{MI_i, MI\} \in RS \};$

At this point $COLF(MI_i)$ may not be completely specified, we take only those MI_i that have been colored already.

if { $NODE1, NODE2\} \in RS$ then $COLORS(N) = \{1, 2\}$ else $COLORS(N) = \{1\};$

{ $COLORS(N)$ are the colors that have been already used }

$BT = \{ (QS(N), GS(N), COLORS(N), CF(N)) \};$

A2. Selection of new node of tree for extension.

```

if BT =  $\emptyset$  then
  begin
    print "OPTIMAL SOLUTION",
    print SOLUTION;
    return;
  end;

  ( BT =  $\emptyset$  when the tree has been searched completely.)
else begin
  FE = (QS(N), GS(N), COLORS(N), CF(N) );
  { FE = first element from list BT }

```

A3. Extension of the node.

```

a) if GS(N) =  $\emptyset$ 
  begin
    delete FE from BT;
    M = M - 1;
    go to A2;
  end;

  COLOR = first element from GS(N);
  N = N + 1;
  GS(N) = GS(N) \ COLOR; {deleting COLOR from set GS(N)}

b) QS(N+1) = QS(N)  $\cup$  {(M1, COLOR)};
  if COLOR  $\in$  COLORS(N)
  begin
    CF(N+1) = CF(N);
    COLORS(N+1) = COLORS(N);

```

```

end;
else begin
    CF(N+1) = CF(N) + 1;
    COLORS(N+1) = COLORS(N)  $\cup$  { COLOR };
end;
if CF(N+1)  $\geq$  CFmin then CUT-OFF
    go to A3;
c) if (CF(N+1) < CFmin) and (M = CARD(SMI(f)))
begin
    1) SOLUTION = QS(N+1);
    CFmin = CF(N+1);
    print ("solution found", SOLUTION, CFmin);
    2) for all nodes
        (QS(Ni), GS(Ni), COLORS(Ni), CF(Ni))  $\in$  BT
    do
        begin
            GS(Ni) = GS(Ni)  $\cap$  COLORS(N+1);
        end;
        GS(N+1) =  $\phi$ ;
        M = M + 1;
        go to A3;
    end;
end;
else begin

```



```

    { creation of new node }
    MI = SMI(f)[M];
    GS(N+1) = ALL-COLORS - {COLF(MIi) | (MIi, MI) ∈ RS}
    put 4-tuple (QS(N+1), GS(N+1), COLORS(N+1), CF(N+1))
    at the beginning of list BT;
    M = M + 1;
    go to A2;
end;
end algorithm;

```

Comments to Algorithm 9

1. Coordinate QS(N) of a node of the solution tree includes a partial coloring of the graph, i.e. a set of pairs (MI, COLFUN(MI)) where $MI \in SMI(f)$.

In the initial node of the tree two incompatible nodes of graph GIM are colored with different colors, 1 and 2, or two compatible nodes are colored with the same color 1.

2. GS(N) is a set of colors which can be used to color the currently selected node (minimal implicant) MI of GIM.

3. To make the execution of the program more efficient, the CUT-OFF rules are applied before calculating GS(N+1). As the possible colors for the minimal implicant, MI, we select colors which are different from the colors already assigned to the minimal implicants that have common edges with MI.

4. When solution QS(N+1) in node N+1 is found we know that the minimal solution is contained in the set of proper

coloring with at most $CARD(COLORS(N+1))$ colors. It is then sufficient to use only colors from the set $COLORS(N+1)$ for the next colorings. The colors not belonging to $COLORS(N+1)$ are then deleted from coordinates $GS(N_i)$ in nodes with numbers N_i that are in the branch leading from the node with number N_0 to the solution node with number $N+1$.

APPENDIX B

LISTING OF PALMINI

```
#include <time.h>
#include <types.h>
#include <timeb.h>
#include <STDIO.H>

int max,maxi,i,il,f,onsize,offsize,solsize,cost;
int sec1,sec2,min1,min2,hour1,hour2,time_flag;
long int *onpt,*offpt,*solpt;
char name[64],c,*pa;
int wcount,remainder,level[4],GIM[120][120],color[120];
long int cubel[1],cube2[2],cube3[3],cube4[4];

void gettime()
{ struct tm *foo;
  time_t *t1;
  *t1 = time(NULL);
  foo = localtime(t1);
  if (time_flag == 0)
  {
    sec1 = (*foo).tm_sec;
```

```

    min1 = (*foo).tm_min;
    hour1 = (*foo).tm_hour;
    time_flag = 1;
    /* printf("%d:%d:%dn",hour1,min1,sec1);*/
}
else if (time_flag == 1)
{
    sec2 = (*foo).tm_sn",hour2,min2,sec2); */
    sec2 = sec2 - sec1;
    min2 = min2 - min1;
    hour2 = hour2 - hour1;
    printf("nTOTAL TIME = %d:%d:%dn",hour2,min2,sec2);
}
}

/* function to compact inputs from name to cubes */
void compact_cube(name,pt,max,i)
    long int *pt;
    int max,i;
    char name[];
{
    int i1,ii,i2;

```

```

/* clear the storage first */
for (i1 = (wcount-1); i1 >= 0; --i1)
    *(pt+(i*wcount)+i1) = 0x0;

i2 = 0;                                /* keep track of index in
name[] */
for (i1 = (wcount-1); i1 >= 0; --i1)
{
    max = level[i1];
    for (ii = 0; ii <= (max-1); ++ii)
    {
        if (name[ii+i2] == '1')
        {
            *(pt+(i*wcount)+i1) = *(pt+(i*wcount)+i1) | 0x2;
            goto compact1;
        }
        else if (name[ii+i2] == '0')
        {
            *(pt+(i*wcount)+i1) = *(pt+(i*wcount)+i1) | 0x1;
            goto compact1;
        }
        else
        {
            *(pt+(i*wcount)+i1) = *(pt+(i*wcount)+i1) | 0x3;
            goto compact1;
        }
    }
}

```

```

    }
compact1:
    if (ii < (max-1))    /* last digit, do not shift left
twice */

        *(pt+(i*wcount)+i1) <= 2;

    }

    i2 = i2 + max; /* i2 will point to correct name[0] for
next wcount */

}

}

/* function to print out cubes from arrays */
void uncompact_cube (name,pt,max,i)

    long int *pt;
    int max,i;
    char name[];
{
    int ii,i1,mask,index;
    long int temp;

    /* process output */
    index = 0;
    for (i1 = (wcount-1); i1 >= 0 ; --i1)
    {
        max = level[i1];
        temp = *(pt+(i*wcount)+i1);
        for (ii = 1; ii <= max; ++ii)

```

```

{
    mask = temp & 0x3;          /* mask off but last 2
bits*/
    if (mask == 0x2)
    {
        *(name+index+max-ii) = '1';
        temp >>= 2;
    }
    else if (mask == 0x1)
    {
        *(name+index+max-ii) = '0';
        temp >>= 2;
    }
    else if (mask == 0x0)
    {
        *(name+index+max-ii) = 'e';
        temp >>= 2;
    }
    else
    {
        *(name+index+max-ii) = 'X';
        temp >>= 2;
    }
}

```

```

        index += max;
    }
}

void print_cube(name,max)
    int max;
    char name[];
{
    int i;
    char *pt;
    max = max;
    pt = &name[0];
    for (i=0; i<= (maxi-1); ++i)
        { printf("%c",*(pt+i));}
    printf("\n");
}

/* inclusion: this procedure will take each entry of ON
array
    and see if it is included in OFF array.
    A flag f is returned: 0 = included.
                        1 = not included.
*/
int include(onpt,offpt,i,f)
    long int *onpt,*offpt;
    int i,f;

```



```

{
    int i1,i2,i3;
    long int reg,mask;
    i1 = 0;
    f = 0;
    while (i1 != offsize)
    {
        f = 0;
        for (i3 = 0; i3 <= (wcount-1); ++i3)
        {
            reg = *(onpt+(i*wcount)+i3) & *(offpt+(i1*wcount)+i3);
/* A * Bi */
            max = level[i3];
            for (i2 = 0;i2 <= (max-1);++i2)
            {
                mask = reg;
                mask = mask & 0x3;                /* mask off but
last 2 bits */
                if (mask == 0)
                {
                    f = 1;
                    i2 = max;                /* A /[ Bi */
                    reg >>= 2;
                }
                else if (mask !=0)
                { reg >>= 2;}
            }
        }
    }
}

```

```

    }
}
if (f == 0)
    return(f);                /* A [ Bi return f = 0
*/
    ++i1;
}
return(f);                    /* A /[ B, return f = 1
*/
}

```

/* function absorbe: will check the array apt for subsumes.

Suppose

$A_i [B_i$ then A_i will be deleted.

The deleting method is as follows: the last entry in
array apt is

copied into A_i and asize is decreased by one.

```

*/
void absorbe(apt, asize)
    long int *apt;
    int *asize;
{
    int i1, i2, i3, flag;
    long int *regpt;
    /* printf("in absorben"); */
    regpt = (long int*) calloc(1, sizeof(cube4));

```

```

    if (regpt = NULL)
n");
        goto absorbe_exit;
    }
    i1 = 0;
    while (i1 <= (*asize-1))
    {
        flag = 0;
        /* flag is used to indicate if Ai is deleted. Flag =
1, Ai is.
        If Ai is deleted, update new value into Ai but keep
the same
        pointer and reset inside loop. If Bi is deleted,
keep same Ai
        and pointer and Bi pointer.
        If none is deleted, keep Ai and advance pointer
        */

        for (i3 = 0; i3 <= (wcount-1); ++i3)
            *(regpt+i3) = *(apt+(i1*wcount)+i3);      /* get Ai
        */

        /* Ai [ Bi ? */
        i2 = (i1+1);
        while (i2 <= (*asize-1))
        {

```

```

for (i3 = 0; i3 <= (wcount-1); ++i3)
{
    *(regpt+i3) &= *(apt+(i2*wcount)+i3);
    if (*(regpt+i3) != *(apt+(i1*wcount)+i3))
        goto step2;
}

/* here, Ai [ Bi */
for (i3 = 0; i3 <= (wcount-1); ++i3) /* delete Ai
*/
    *(apt+(i1*wcount)+i3) =
*(apt+((*asize-1)*wcount)+i3);
    --*asize;
    flag = 1;
    i2 = *asize;                                /* reset
inside loop */
    goto step5;
/* Bi [ Ai ? */

step2: for (i3 = 0; i3 <= (wcount-1); ++i3)
    *(regpt+i3) = *(apt+(i1*wcount)+i3);
for (i3 = 0; i3 <= (wcount-1); ++i3)
{
    *(regpt+i3) &= *(apt+(i2*wcount)+i3);
    if (*(regpt+i3) != *(apt+(i2*wcount)+i3))

```

```

        goto step5;
    }

    /* Bi [ Ai */
    for (i3 = 0; i3 <= (wcount-1); ++i3) /* delete Bi
*/
        *(apt+(i2*wcount)+i3) =
*(apt+((*asize-1)*wcount)+i3);

    --*asize;
    --i2;          /* to stay at the same pointer */
    /* Ai [/ Bi and Bi [/ Ai */
step5::
    ++i2;
} /* end of while i2 */
if (flag == 0)
    {++i1;}
} /* end for while i1 */
absorbe_exit::
}

```

/* function: make_graph_GIM will create graph GIM. The result

is stored at GIM. GIM is a two dimensional array
with row = column = onsize.

A 0 = no edge between that row and column.

A 1 = an edge exists between that row and column.

*/

```
void make_graph_GIM(onpt,offpt,GIM)
```

```
    long int *onpt,*offpt;
```

```
    int GIM[60][60];
```

```
    {
```

```
        int i1,i2,i3;
```

```
        long int *regpt;
```

```
        /* printf("in make_graph_GIMn"); */
```

```
        for (i1 = 0; i1 <= (onsize - 1); ++i1)
```

```
            GIM[i1][i1] = 0;  n");
```

```
            goto make_graph_exit;
```

```
        }
```

```
        for (i1 = 0; i1 <= (onsize-1); ++i1)
```

```
        {
```

```
            for (i2 = (i1+1); i2 <= (onsize-1); ++i2)
```

```
            {
```

```
                for (i3 = 0; i3 <= (wcount-1); ++i3)
```

```
                {
```

```
                    *(regpt+i3) = *(onpt+(i1*wcount)+i3) |
```

```
                    *(onpt+(i2*wcount)+i3);
```

```
                    /* Ai $ Ai+1 */
```

```
                }
```

```

        f = include(regpt, offpt, 0, f);
        if (f != 0)
        { GIM[i1][i2] = 0;
          GIM[i2][i1] = 0;
        }
        else
        { GIM[i1][i2] = 1;
          GIM[i2][i1] = 1;
        }
    }
}

make_graph_exit:
free(regpt);
}

/* compute_cost_of_GIM: this function computes the cost to
color graph
    GIM and also colors the graph and saves solution in array
COLOR[]
*/
void compute_cost_of_GIM(GIM)
    int GIM[120][120];
{
    int i1, i0, i2, f, tempcolor;
    long int *regpt;
    printf("in compute_cost_of_GIMn");

```

```

regpt = (long int *) calloc(1,sizeof(cube4));
if (regpt == NULL)
n");
    goto compute_cost_exit;
}
if (onsize < 2)
    cost = 1;
else
{
    color[0] = 1;    /* assign first color to first node
*/
    i0 = 1;
    while (i0 <= (onsize-1))
    {
        tempcolor = 1;
        i1 = 0;
        while (i1 <= (i0-1)) /* check against previous
nodes */
        {
            if (GIM[i0][i1] == 1)
            {
                if (tempcolor == color[i1])
                    ++tempcolor;
            }
            ++i1;

```



```

    }

    color[i0] = tempcolor; /* next node gets color */
    /* check and see if this color valid */

check1:

    printf("check1, tempcolor = %d\n",tempcolor);
    for (i1 = 0; i1 <= (wcount-1); ++i1)
        *(regpt+i1) = *(onpt+(i0*wcount)+i1); /* get
this cube */
    for (i1 = 0; i1 <= (i0-1); ++i1)
    {
        f = 0;
        if (color[i1] == color[i0])
        {
            for (i2 = 0; i2 <= (wcount-1); ++i2)
                *(regpt+i2) |= *(onpt+(i1*wcount)+i2); /*
match cubes */

                f = 1; /* set flag */
        }
        if (f == 1)
        {
            f = include(regpt,offpt,0,f); /* check cube

*/

            if (f == 0) /* cube overlaps offset */
            {
                /* search for another color */
                ++tempcolor;

```

```

        printf("overlap, tempcolor = %d\n",tempcolor);
        i2 = 0;
        while (i2 <= (i0-1))
        {
            if (GIM[i0][i2] == 1)
            {   if (tempcolor == color[i2])
                    ++tempcolor;
            }
            ++i2;
        }
        color[i0] = tempcolor;
        goto check1;
    }
}

    ++i0;
}
} /* end of else */
/* compute cost */
i0 = 0;
cost = 1;
while (i0 <= (onsize-1))
{
    if (color[i0] > cost)

```

```

        cost = color[i0];
        ++i0;
    }
compute_cost_exit:;

}

void graph_coloring()
{
    int i0,i1,i5,i2;
    /* printf("in graphcoloring\n"); */
    switch (wcount)
    {
        case 1:
            solpt = (long int *) calloc(cost+100,size-
of(cube1)); /* 16 vars */
            break;
        case 2:
            solpt = (long int *) calloc(cost+100,size-
of(cube2)); /* 32 vars */
            break;
        case 3:
            solpt = (long int *) calloc(cost+100,size-
of(cube3)); /* 48 vars */
            break;
        default:

```

```

        solpt = (long int *) calloc(cost+100,size-
of(cube4)); /* 64 vars */
        break;
    }

    if (solpt == NULL)
    {
        printf("Can not allocate memory for SOL array\n");
        goto graph_exit;
    }
    if (onsize == 1)
    {
        for (i1 = 0; i1 <= (wcount-1); ++i1)
            *(solpt+i1) = *(onpt+i1);
    }
    else
    {
        for (i1 = 0; i1 <= (cost-1); ++i1)
        {
            solsize = 1;
            for (i5 = 1; i5 <= cost; ++i5)
            {
                for (i0 = 0; i0 <= (onsize-1); ++i0)
                {
                    if (color[i0] == i5)

```

```

    {
        for (i2 = 0; i2 <= (wcount-1); ++i2)
            *(solpt+((solsize-1)*wcount)+i2) =
*(onpt+(i0*wcount)+i2);

        color[i0] = 0;          /* delete the used node
*/

        i1 = i0 + 1;
        while (i1 <= (onsize-1))
        {
            if (color[i1] == i5)
            {
                for (i2 = 0; i2 <= (wcount-1); ++i2)
                    *(solpt+((solsize-1)*wcount)+i2) |=
*(onpt+(i1*wcount)+i2);

                /* match cubes of same color */
                color[i1] = 0;  /* this step is extra */
            }
            ++i1;
        }
        ++solsize;
    }
}
}
}

```



```

        /*printf("temp = \n");
uncompact_cube(name,&temp,max,0);
print_cube(name,max);*/

mask3 = ~mask1;
temp |= mask1;          /* turn the bit into x
*/

f = include(&temp,offpt,0,f);
if (f == 0)    /* temp is included in offpt */
{
    temp &= mask3;    /* blank this bit */
    temp |= mask2;    /* restore this bit into
temp */

    mask1 <= 2;      /* shift to next bit */
}
else if (f == 1)    /* temp is not included in
offpt */

    mask1 <= 2;      /* shift to next bit */
}
else
    mask1 <= 2;

}

*(apt+(i1*wcount)+i2) = temp;

```

```

    }
}

}

void find_consensus(apt, asize)
    long int *apt;
    int *asize;
{
    long int *tempt, mask, reg;
    int i1, i2, i3, i4, ecount;
    tempt = (long int *) calloc(1, sizeof(cube4));
    if (tempt == NULL)
    {
        printf("Can not allocate memory for TEMPT in
find_consensus\n");
        goto consensus_exit;
    }
    for (i1 = 0; i1 <= (*asize-2); ++i1)
    {
        for (i2 = 1; i2 <= (*asize-1); ++i2)
        {
            for (i3 = 0; i3 <= (wcount-1); ++i3)
                *(tempt+i3) = *(apt+(i1*wcount)+i3) &
*(apt+(i2*wcount)+i3);

            /* star operator can be realized with AND

```



```

operator */
    ecount = 0;
    for (i3 = 0; i3 <= (wcount-1); ++i3)
    {
        max = level[i3];
        reg = *(tempt+i3);
        for (i4 = 0; i4 <= (max-1); ++i4)
        {
            mask = reg;
            mask &= 0X3;          /* check last two bits */
            if (mask == 0)
            { ++ecount;
              mask |= 0X3;        /* turn these bits into X */
              mask <<= 2*i4;
              *(tempt+i3) |= mask;
            }
            reg >>= 2;            /* shift to next Boolean
bit */
        }
        if (ecount > 1)
        {
            i1 = *asize;
            i2 = *asize;
            i3 = wcount;          /* no consensus exists
between A and B

```

```

                                so, exit */

        i4 = max;
    }
}

    if (ecount == 1)          /* create consensus if
ecount = 1 */
    {
        ++*asize;
        for (i4 = 0; i4 <= (wcount-1); ++i4)
            *(apt+((*asize-1)*wcount)+i4) = *(tempt+i4);
    }
}

consensus_exit;;

}

/* function scomp1: this function will find the complementa-
tion
    of cpt. The result is stored in bpt.
    method: disjoint sharp. */
void scomp1(aapt,asize,onpt,onsize)
    long int *aapt,*onpt;
    int *asize,*onsize;
{
    int i2,i3,i4,offset,cptx,bptx,cptr,bptr;

```

```

long int *regpt,mask,temp,temp2;
/* printf("in scomplementn"); */
regpt = (long int*) calloc(1,sizeof(cube4));
if (regpt = NULL)
n");
    goto scompl_exit;
}
/* fill apt[1] = xxxxx */
for (i2 = 0 ; i2 <= (wcount-1); ++i2)
    {*(apt+i2) = 0x0;
     *(apt+i2) = ~*(apt+i2);}
cptr = 0;
bptr = 0;
*asize = 1;
while (bptr <= (*onsize-1))
{
    cptr = 0;
    bptx = bptr*wcount;      /* bptx = offset into onpt */
    while (cptr <= (*asize-1))
    {
        cptx = cptr*wcount;  /* cptx = offset into apt */
        /* is A [ Bi ? */
        for (i2 = 0; i2 <= (wcount-1); ++i2)
        {
            *(regpt+i2) = *(apt+cptx+i2) & *(onpt+bptx+i2);

```

```

    if (*(regpt+i2) != *(apt+cptx+i2))
    { /* printf(" A [/ B\n"); */
        goto step1;}          /* A [/ Bi */
    }
/* here A [ B, delete A */
offset = (*asize-1)*wcount;
for (i2 = 0; i2 <= (wcount-1); ++i2)
    *(apt+cptx+i2) = *(apt+offset+i2);
--*asize;
--cptr;
goto step3;
step1: /* is A overlapped Bi ? */

for (i3 = 0; i3 <= (wcount-1); ++i3)
{
    *(regpt+i3) = *(apt+cptx+i3) & *(onpt+bptx+i3);
    max = level[i3];
    for (i2 = 0; i2 <= (max-1); ++i2)
    {
        mask = *(regpt+i3);
        mask = mask & 0x3;          /* mask off but
last 2 bits */
        if (mask == 0)
        {goto step3;}              /* A /[ Bi */
        else if (mask !=0)

```

```

        { *(regpt+i3) >>= 2;}
    }
}

/* printf("A is overlapped B\n"); */

/* now regpt contains A * Bi. It then is sharpened
against A */
step2:
/* main body of sharp */
for (i3 = 0; i3 <= (wcount-1); ++i3)
    *(regpt+i3) = *(apt+cptx+i3) & *(onpt+bptx+i3);
for (i3 = 0; i3 <= (wcount-1); ++i3)
    *(regpt+i3) ^= *(apt+cptx+i3);
for (i3 = 0; i3 <= (wcount-1); ++i3)
{
    max = level[i3];
    for (i2 = 0; i2 <= (max-1); ++i2)
    {
        mask = *(regpt+i3);
        mask &= 0X3;                /* mask all but last
two bits */
        if (mask != 0)
        {
            temp = mask;
            if (mask == 0X1)
                temp2 = 0X2;

```

```

else if (mask == 0X2)
    temp2 = 0X1;
else if(mask == 0X3)
    temp2 = 0X0;

temp <<= 2*i2;
temp2 <<= 2*i2;
mask = ~temp;          /* to mask of these bits
*/

mask ^= temp2;
mask &= *(apt+cptx+i3); /* clear these bits
in A */

/* create new cube */
++*asize;
offset = wcount * (*asize-1);
for (i4 = 0; i4 <= (wcount-1); ++i4)
{
    if (i4 == i3)
        *(apt+offset+i4) = temp | mask;
    else
        *(apt+offset+i4) = *(apt+cptx+i4);
}
}
*(regpt+i3) >>= 2;
}
}

```

```

/* delete the entry Ai due to new created cubes */
for (i4 = 0; i4 <= (wcount-1); ++i4)
{
/* swap the last cube into current cube */
*(apt+cptx+i4) = *(apt+((*asize-1)*wcount)+i4);
}

--*asize;

--cptr;    /* decrement by one to remain at this
pointer

for next cube */

step3;;

++cptr;

if (*asize == 0)
{printf("asize = 0\n");
goto scompl_exit;}

} /* end for while cptr */

absorbe(apt,asize);

++bptr;    /* if no new cube is created, increment
cptr */

} /* end for while cptr */

scompl_exit;;

}

```

```

void create_disjoint(apt,asize)
    long int *apt;
    int *asize;
{
    int i1,i2,i3,i4,i5,f,xcount1,xcount2,cptx,bptx,offset;
    long int reg,mask,mask1,temp,temp2,*regpt;
    /* printf("in create_disjointn");*/
    regpt = (long int*) calloc(1,sizeof(cube4));
    if (regpt = NULL)
n");
        goto disjoint_exit;
    }
    i1 = 0;
    while (i1 <= (*asize-1))
    {
        i2 = i1 + 1;
        while (i2 <= (*asize-1))
        {
            f = 0;
            for (i3 = 0; i3 <= (wcount-1); ++i3)
            {
                reg = *(apt+(wcount*i1)+i3) &
*(apt+(wcount*i2)+i3);
                max = level[i3];
                for (i4 = 0; i4 <= (max-1); ++i4)

```



```

{
    mask = reg;
    mask = mask & 0X3;
    if (mask == 0)
    {
        f = 1;
        i3 = wcount;
    }
    else if (mask != 0)
    { reg >>= 2;}
}
}

```

```

if (f == 0)    /* A [ B, then find if A > B or B >

```

```

A */

```

```

{
    xcount1 = 0;
    xcount2 = 0;
    for (i3 = (wcount-1); i3 >= 0; --i3)
    {
        max = level[i3];
        mask = *(apt+(wcount*i1)+i3);
        mask1 = *(apt+(wcount*i2)+i3);
        for (i4 = 0; i4 <= (max-1); ++i4)
        {

```

```

        if ((mask & 0X3) == 0X3)
            ++xcount1;
        if ((mask1 & 0X3) == 0X3)
            ++xcount2;
        mask >>= 2;
        mask1 >>= 2;
    }

    if (xcount1 != xcount2)    /* check from most
significant bit */
        i3 = -1;
    }
    /* is A [ Bi ? */
    cptx = wcount*i1;
    bptx = wcount *i2;
    for (i3 = 0; i3 <= (wcount-1); ++i3)
    {
        *(regpt+i3) = *(apt+cptx+i3) & *(apt+bptx+i3);
        if (*(regpt+i3) != *(apt+cptx+i3))
            goto step0;          /* A [/ Bi */
    }
    /* here A [ B, delete A */
    offset = (*asize-1)*wcount;
    for (i3 = 0; i3 <= (wcount-1); ++i3)
        *(apt+cptx+i3) = *(apt+offset+i3);

```

```

        --*asize;
        goto step3;
step0:; /* is B [ A */
        for (i3 = 0; i3 <= (wcount-1); ++i3)
        {
            *(regpt+i3) = *(apt+cptx+i3) & *(apt+bptx+i3);
            if (*(regpt+i3) != *(apt+bptx+i3))
                goto step1;
        }
/* here B [ A, delte B */
        offset = (*asize-1)*wcount;
        for (i3 = 0; i3 <= (wcount-1); ++i3)
            *(apt+bptx+i3) = *(apt+offset+i3);
        --*asize;
        --i2;      /* to remain at the same pointer */
        goto step3;

step1: /* is A overlapped Bi ? */

        for (i3 = 0; i3 <= (wcount-1); ++i3)
        {
            *(regpt+i3) = *(apt+cptx+i3) & *(apt+bptx+i3);
            max = level[i3];
            for (i5 = 0; i5 <= (max-1); ++i5)
            {

```

```

        mask = *(regpt+i3);
        mask = mask & 0x3;                      /* mask off
but last 2 bits */

        if (mask == 0)
        {goto step3;}                          /* A /[ Bi */
        else if (mask !=0)
        { *(regpt+i3) >>= 2;}
        }
    }

    if (xcount2 > xcount1)
    {i1 = cptx;  /* if B > A, then delete B, else
delete A */

        cptx = bptx;
        bptx = i1;}

    /* printf("A is overlapped B\n"); */
    /* now regpt contains A * Bi. It then is sharpened
against A */
    step2:
    /* main body of sharp */
        for (i3 = 0; i3 <= (wcount-1); ++i3)
            *(regpt+i3) = *(apt+cptx+i3) & *(apt+bptx+i3);
        for (i3 = 0; i3 <= (wcount-1); ++i3)
            *(regpt+i3) ^= *(apt+cptx+i3);
        for (i3 = 0; i3 <= (wcount-1); ++i3)

```

```

{
    max = level[i3];
    for (i4 = 0; i4 <= (max-1); ++i4)
    {
        mask = *(regpt+i3);
        mask &= 0X3;                /* mask all but last
two bits */

        if (mask != 0)
        {
            temp = mask;
            if (mask == 0X1)
                temp2 = 0X2;
            else if (mask == 0X2)
                temp2 = 0X1;
            else if (mask == 0X3)
                temp2 = 0X0;
            temp <= 2*i4;
            temp2 <= 2*i4;
            mask = ~temp;           /* to mask of these
bits */

            mask ^= temp2;

            mask &= *(apt+cptx+i3); /* clear these
bits in A */

            /* create new cube */
            ++*asize;

```

```

        offset = wcount * (*asize-1);
        for (i5 = 0; i5 <= (wcount-1); ++i5)
        {
            if (i5 == i3)
                *(apt+offset+i5) = temp | mask;
            else
                *(apt+offset+i5) = *(apt+cptx+i5);
        }
    }
    *(regpt+i3) >>= 2;
}

/* delete the entry Ai due to new created cubes */
for (i4 = 0; i4 <= (wcount-1); ++i4)
{
    *(apt+cptx+i4) = *(apt+((*asize-1)*wcount)+i4);
}
--*asize;

/* if B is deleted, then adjust i2 to remain the
same pointer */
if (xcount2 > xcount1)
    --i2;

step3::

if (*asize == 0)
    {printf("asize = 0\n");

```

```

        goto disjoint_exit;}

    }

    ++i2;

} /* end of while i2 */

++i1;

} /* end of while i1 */

disjoint_exit: ;

}

/*****
*/

main ( )
{
    /* this program demonstrates the representation of Boolean
cubes

    as bits in registers.

    0 = 01
    1 = 10
    X = 11
    e = 00

    */

    int toffsize,out,i1,i2,i3,flag,static_hazard_flag;
    int delete_literal_flag,invert_output_flag,rsize;
    long int timp1,timp2,*tempt;

```

```

FILE *input_file,*output_file, *fopen ();

printf("nPALMINIn");
time_flag = 0;
pa = &name[0];
if ( (input_file = fopen ("texti.pas", "r") ) == 0)
{ printf("texti.pas can not be opened\n");
  goto exit;
}
/* skip comment lines */
start1:
    c = getc(input_file);
    if ((c == ';' ) || (c == ' '))
        { while ((c=getc(input_file)) != '\n'); /* skip a line
*/
          goto start1;
        }
    if ((c == 'i') || (c == 'I'))
        { fscanf(input_file,"%d",&max);
          printf("number of input variables = %dn",max);}
    else
        {printf("can not find in");
          goto exit;}
    while ((c=getc(input_file)) !=
'n'); /* skip i x line */

```



```

c = getc(input_file);
if ((c == 'o') || (c == 'On')); /* skip o x line */
c = getc(input_file);
if ((c == 'p') || (c == 'P'))
    {fscanf(input_file,"%d",&onsize);
    printf("number of input terms = %dn",onsize);}
else
    {printf("can not find pn");
    goto exit;}
while ((c=getc(input_file)) != '\n'); /* skip p x line */
c = getc(input_file);
if ((c == 'h') || (c == 'H'))
    {fscanf(input_file,"%d",&static_hazard_flag);
    if (static_hazard_flag == 1)
        printf("Static_Hazard_Check_Option = ON\n");
    else
        printf("Static_Hazard_Check_Option = OFF\n");
    }
else
    {printf("can not find hn");
    goto exit;}
while ((c=getc(input_file)) != '\n'); /* skip h x line n"
);

else
    printf("Delete_Literal_Option = OFFn");

```

```

    }
else
    {printf("can not find dn");
    goto exit;}

while ((c=getc(input_file)) != '\n'); /* skip d x line */
c = getc(input_file);
if ((c == 'e') || (c == 'E'))
    {fscanf(input_file,"%d",&invert_output_flag);
    if (invert_output_flag== 1)
        printf("Invert_output_flag = ON\n");
    else
        printf("Invert_output_flag = OFF\n");
    }
else
    {printf("can not find e\n");
    goto exit;}

while ((c=getc(input_file)) != '\n'); /* skip e x line */

maxi = max + out;
wcount = (maxi * 2) / 32;
remainder = (maxi * 2) % 32;          /* modulus operator
*/
if (remainder > 0)
    ++wcount;

switch (wcount)          /* setup level[i] for cube manipula-

```

```
tion */
{
    case 0:
        printf("error 1: number of variable = 0\n");
        goto exit;
        break;
    case 1:
        if (remainder == 0)
            level[0] = 16;
        else
            level[0] = remainder / 2;
        break;
    case 2:
        level[0] = 16;
        if (remainder == 0)
            level[1] = 16;
        else
            level[1] = remainder / 2;
        break;
    case 3:
        level[0] = 16;
        level[1] = 16;
        if (remainder == 0)
            level[2] = 16;
        else
```

```

        level[2] = remainder /2;
    break;
default:
    level[0] = 16;
    level[1] = 16;
    level[2] = 16;
    if (remainder == 0)
        level[3] = 16;
    else
        level[3] = remainder /2;
    break;
}
switch (wcount)
{
    case 1:
        onpt = (long int *) calloc(out*onsize+100,size-
of(cube1)); /* 16 vars */
        break;
    case 2:
        onpt = (long int *) calloc(out*onsize+100,size-
of(cube2)); /* 32 vars */
        break;
    case 3:
        onpt = (long int *) calloc(out*onsize+100,size-
of(cube3)); /* 48 vars */

```

```

        break;

default:
    onpt = (long int *) calloc(out*onsize+100,size-
of(cube4)); /* 64 vars */
        break;
    }

if (onpt == NULL)
{
    printf("Can not allocate memory for ON array\n");
    goto exit;
}

/* read in the on cubes */

rsize = 0;
for (il = 0; il <= (onsize-1); ++il)
{
    for (i = 0; i <= (max-1); ++i)
    {
        c = getc(input_file);
        if (c == '1' || c == '0' || c == 'x' || c == 'X')
            *(pa+i) = c;
        else
        { printf("error, data is not 0,1,x or X\n");
          printf("%c",c);

```

```

        goto exit;
    }
}
/* take care of number of output here */
for (i = 1; i <= out; ++i)
{
    while ((c = getc(input_file)) == ' '); /* skip
blank */
    if (c == '1')
    {
        for (i2 = 1; i2 <= out; ++i2)
        {
            if (i2 == i)
                *(pa+(max-1)+i2) = '0';
            else if (i2 != i)
                *(pa+(max-1)+i2) = '1';
        }
        compact_cube(name,onpt,max,rsize);
        ++rsize;
    }
}

while((c=getc(input_file)) != '\n'); /* skip to next
line */

```

```

}
fclose(input_file);
onsize = rsize;

/* start counting time */
gettime();

toffsize = 600;
switch (wcount)
{
    case 1:
        offpt = (long int *) calloc(toffsize, size-
of(cube1)); /* 16 vars */
        break;
    case 2:
        offpt = (long int *) calloc(toffsize, size-
of(cube2)); /* 32 vars */
        break;
    case 3:
        offpt = (long int *) calloc(toffsize, size-
of(cube3)); /* 48 vars */
        break;
    default:
        offpt = (long int *) calloc(toffsize, size-
of(cube4)); /* 64 vars */

```

```

        break;
    }
    if (offpt == NULL)
    {
        printf("Can not allocate memory for OFF arrayn");
        goto exit;
    }

printf("Complementation using Disjoint Sharp methodn");

/* check for special cases of all xxxxxx */

for (i1 = 0; i1 <= (onsize-1); ++i1)
{
    for (i2 = 0; i2 <= (wcount-1); ++i2)
    {
        timp1 = *(onpt+(i1*wcount)+i2);
        max = level[i2];
        flag = 0;
        for (i3 = 0; i3 <= (max-1); ++i3)
        {
            timp2 = timp1;
            if ((timp2 &= 0X3) != 0X3)        /* check last two
bits */
                {i3 = max;                    /* check next cube
*/

```



```

        flag = 1;}
    else
        timpl >>= 2;
}
if (flag == 0)
/* here, the cube is all xxxx */
{
    offsize = 0;
    printf("nComplementation of f is emptyn");
    goto print_result;
}
}
}

scomplement(offpt,&offsize,onpt,&onsize);
absorbe(offpt,&offsize);
print_result::
printf("number of MAXTERMS = %d\n",offsize);
if (invert_output_flag == 1)
{
    tempt = onpt;
    onpt = offpt;
    offpt = tempt;
    il = onsize;

```



```

if ( (output_file = fopen ("texto", "w") ) == 0)
{
    printf("texto can not be openedn");
    goto exit;
}

fprintf(output_file,"i %dn",maxi);
fprintf(output_file,"o %dn",out);
fprintf(output_file,"p %dn",solsize);

for (il = 0; il <= (solsize-1); ++il)
{
    uncompact_cube(name,solpt,max,il);
    for (i = 1; i <= out; ++i)
    {
        if ( (*(pa+(max-1)+i) == '0') || (*(pa+(max-1)+i) ==
'X') )
            *(pa+(max-1)+i) = '1';
        else
            *(pa+(max-1)+i) = '-';
    }
    for (i = 1; i <= out; ++i)
        *(pa+max+out-i+1) = *(pa+max+out-i);
    *(pa+max) = ' ';
    for (i = 0; i <= (maxi-1); ++i)
        fprintf(output_file,"%c",name[i]);
    fprintf(output_file,"\n");
}

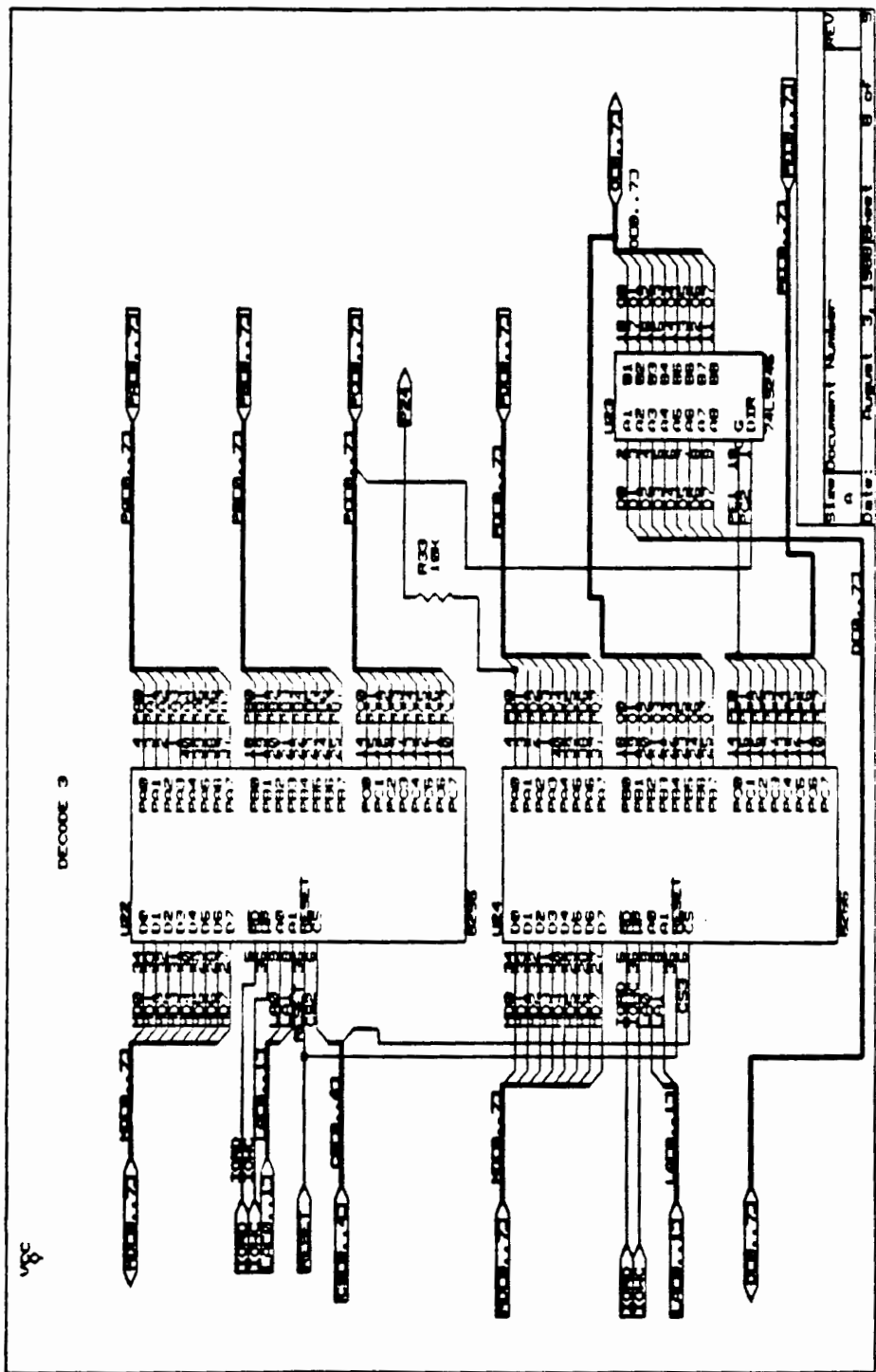
```

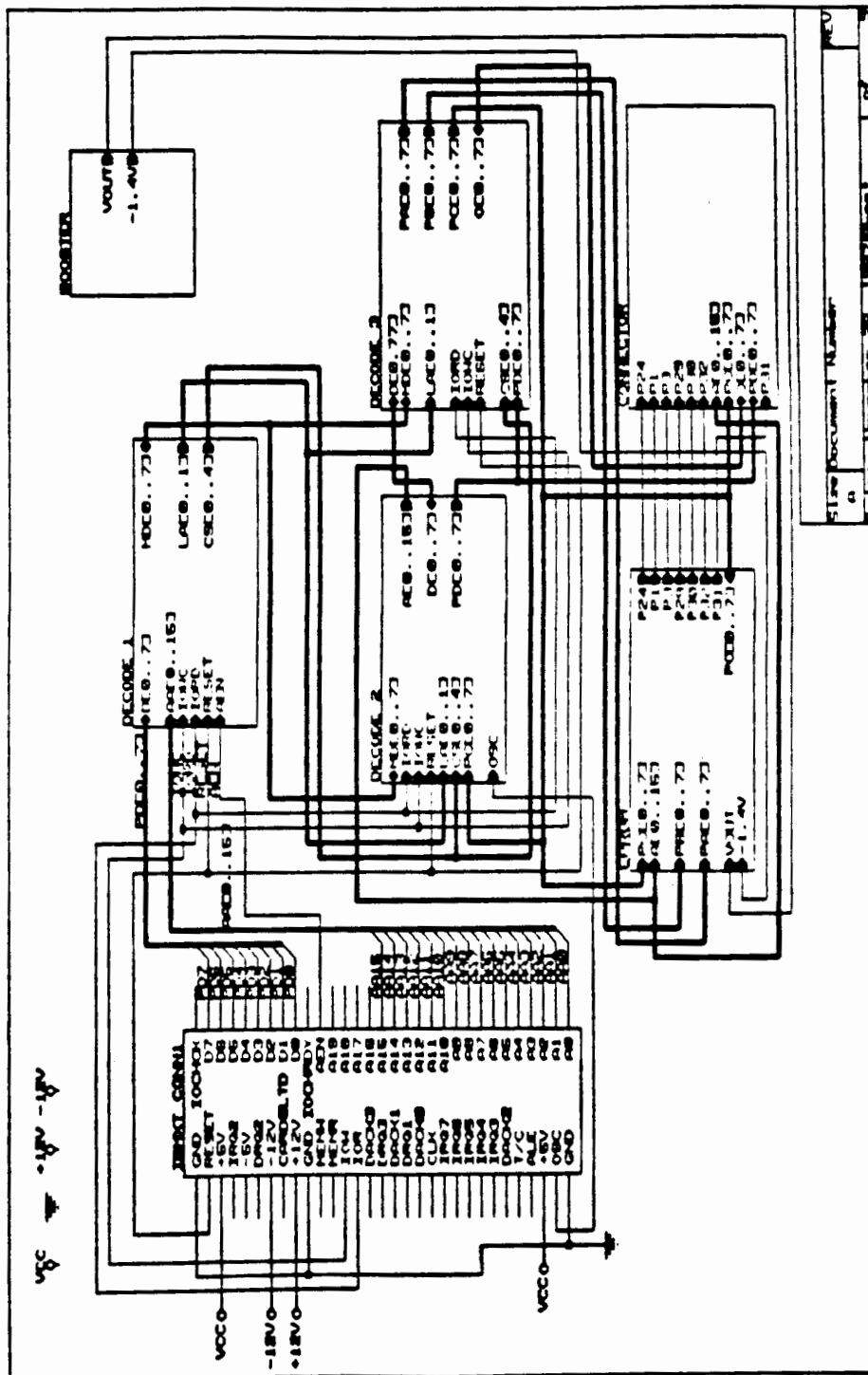
```
    }  
    fclose(output_file);  
    gettimeofday();  
  
exit::  
}
```

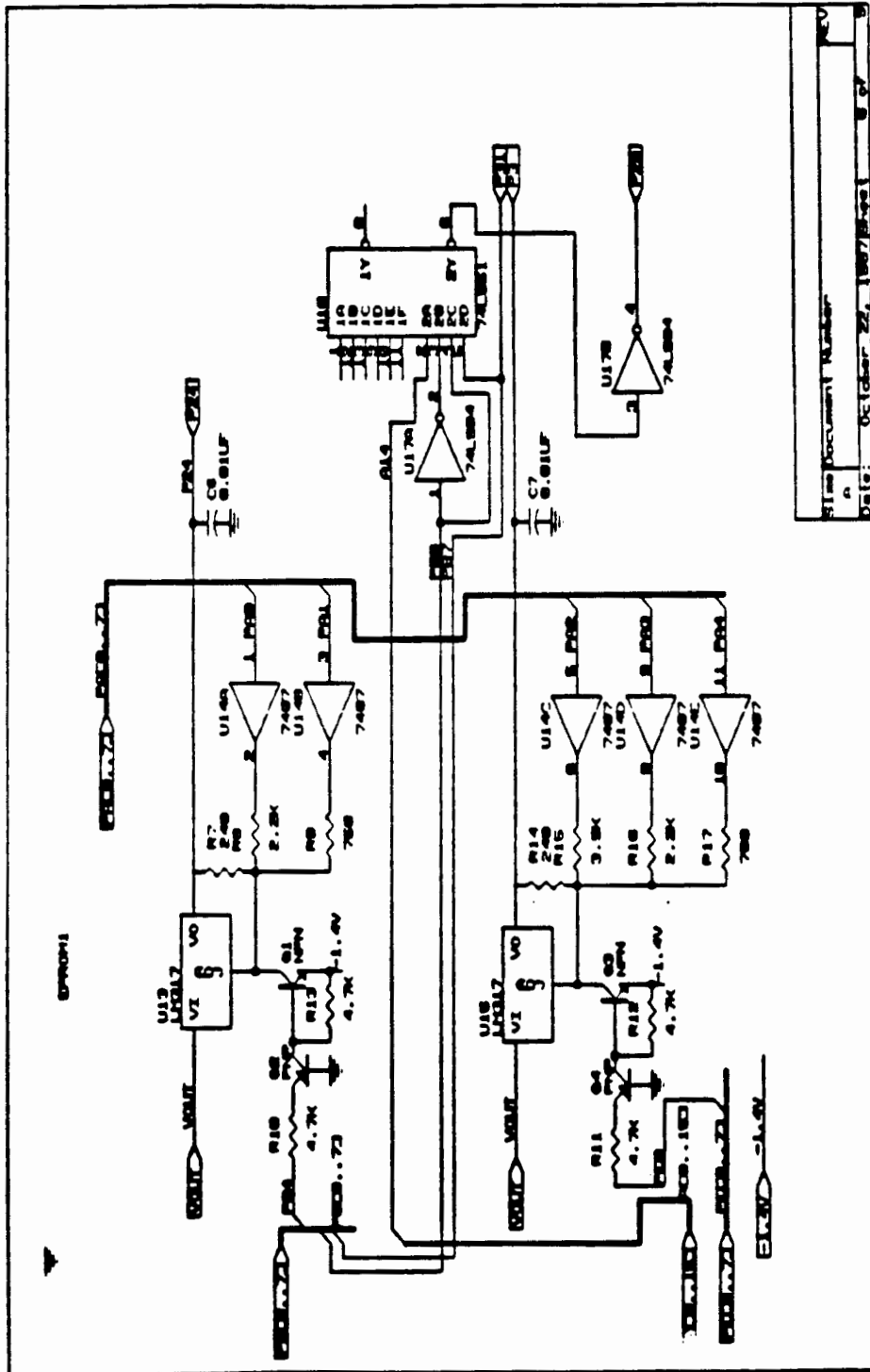
APPENDIX C

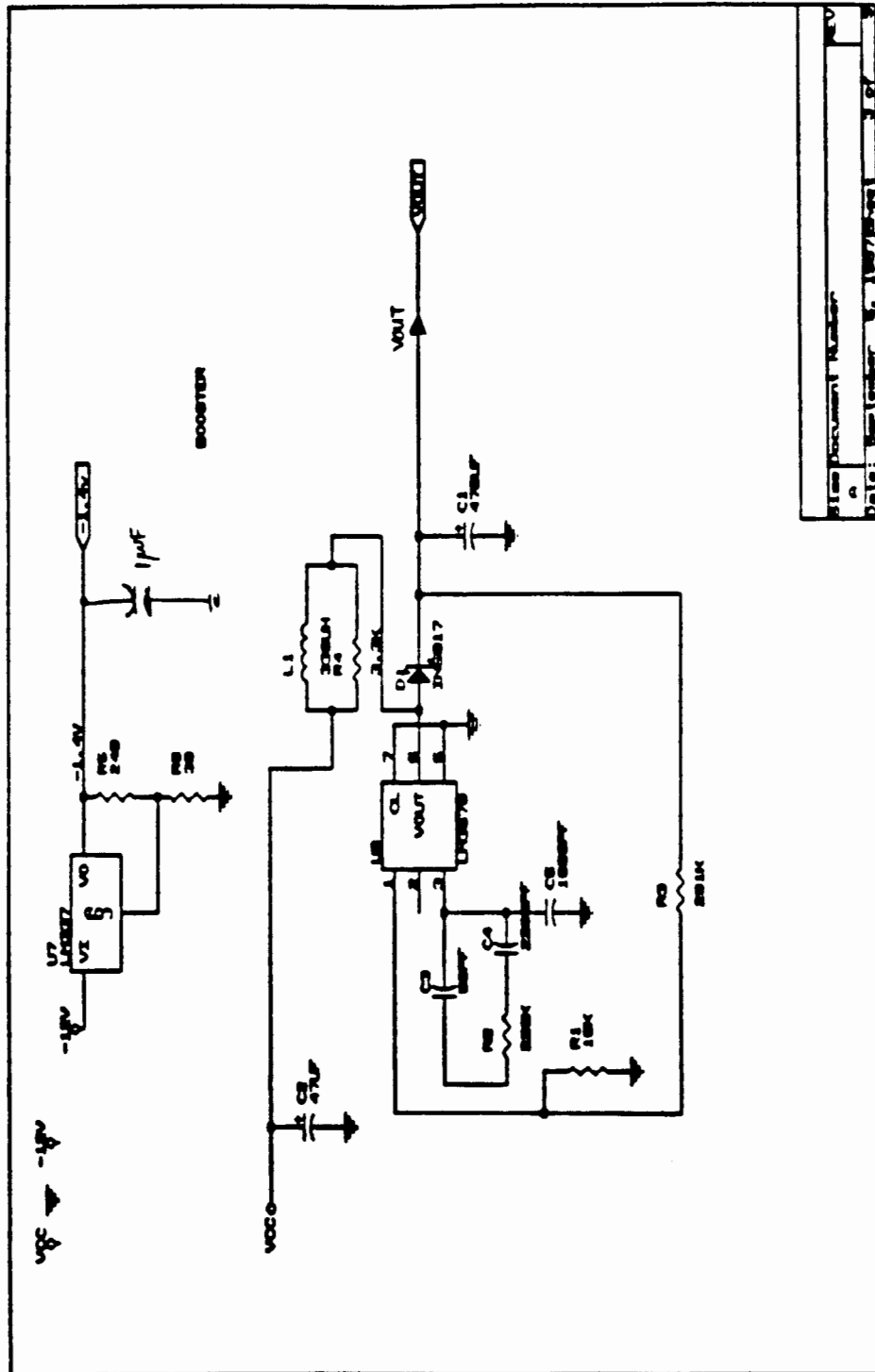
SCHEMATIC OF ZAPAGAL BOARD

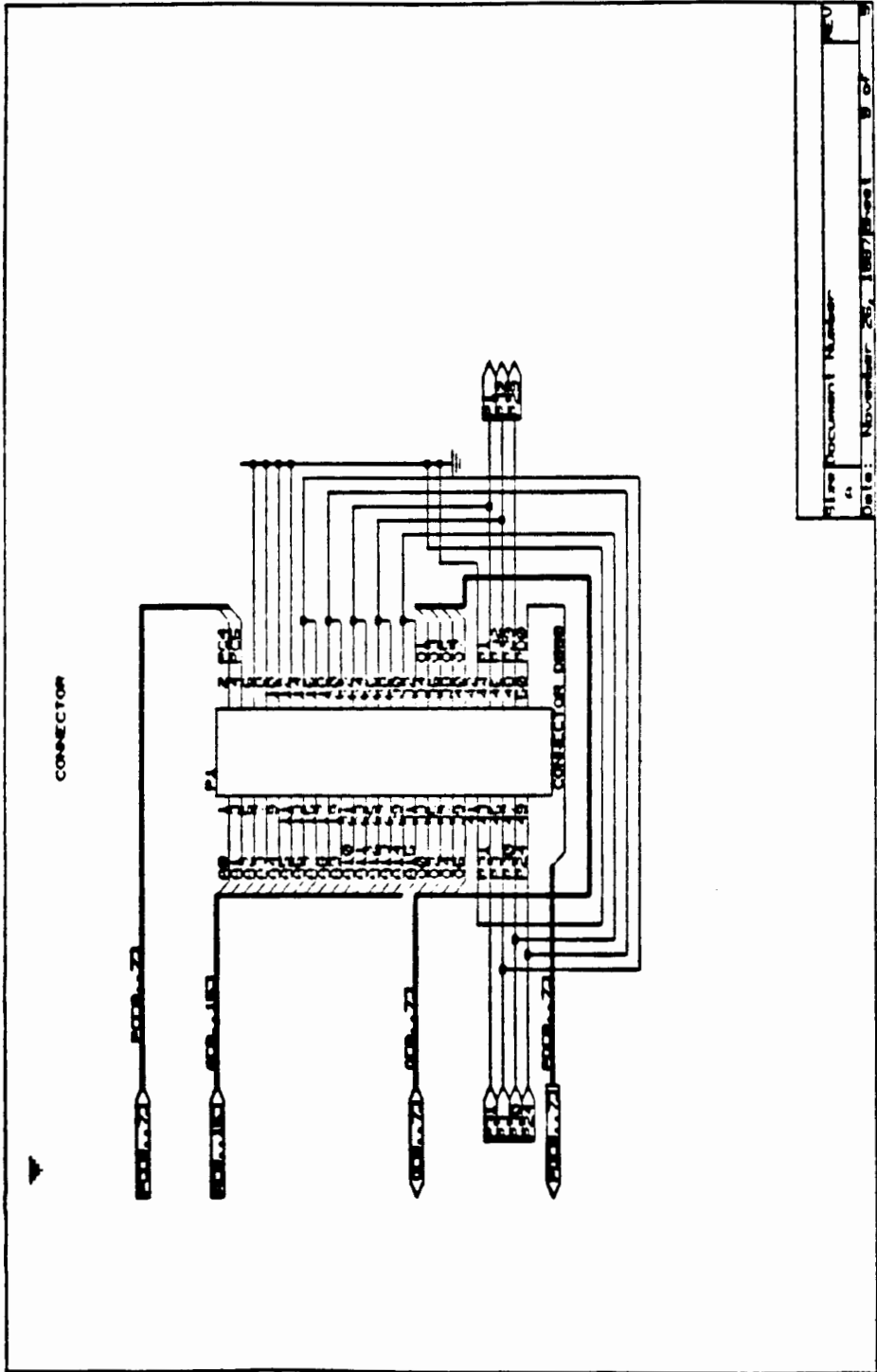
Enclosed is the complete schematic of the Zapagal Board.



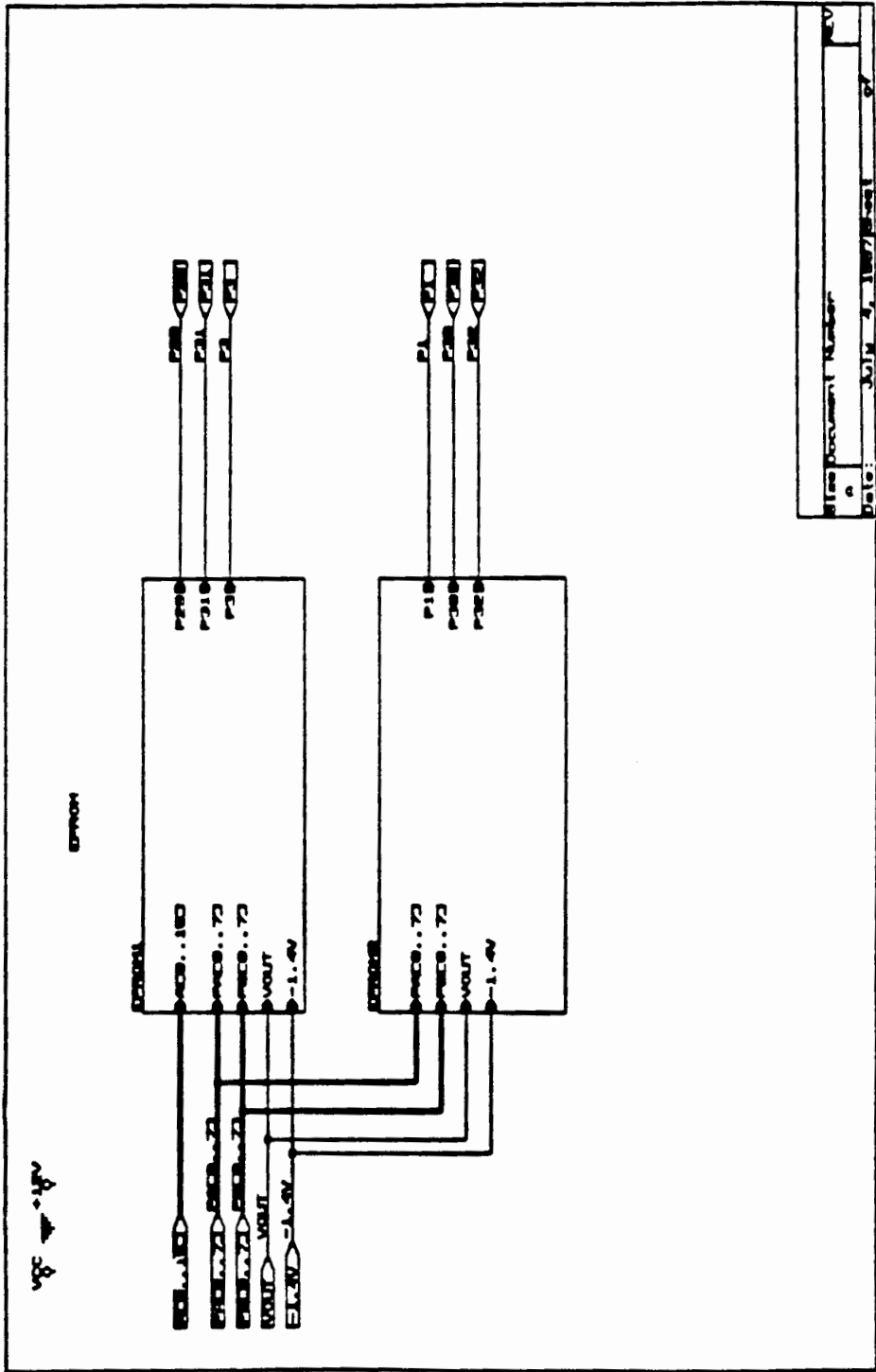








File Document Number
4
Date: November 26, 1987/Sheet 9 of 9



Size	Document Number
a	
Date:	July 4, 1997/Sheet 1 of 1

