(NASA-CR-170759) DDL SYSTEM: DESIGN SYSTHESIS OF DIGITAL SYSTEMS Final Technical Report, 1 Oct. 1978 - 31 Jan. 1983 (Alabama Univ., Huntsville.) 103 p HC A06/MF A01 CSCL 09B G3/60

N85-16584

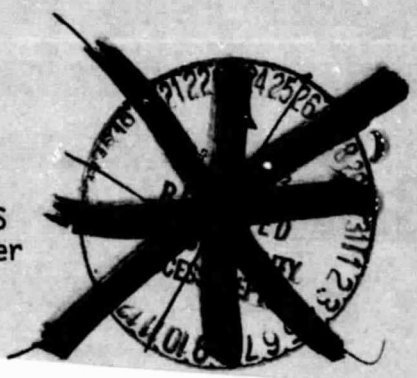
Unclas 14904

Prepared by

SAJJAN G. SHIVA
Computer Science Department
The University of Alabama in Huntsville
Huntsville, Alabama 35807

Final Technical Report
January 1983

for



NAS8 - 33096
DESIGN SYNTHESIS OF DIGITAL SYSTEMS
George C. Marshall Space Flight Center
Alabama, 35812

Date of general release January 1985

DDL SYSTEM - FINAL REPORT

Prepared by

SAJJAN G. SHIVA
Computer Science Department
The University of Alabama in Huntsville
Huntsville, Alabama 35807

Final Technical Report
January 1983

for

NAS8 - 33096
DESIGN SYNTHESIS OF DIGITAL SYSTEMS
George C. Marshall Space Flight Center
Alabama, 35812

FOREWORD

This is a technical summary of the research work conducted during October 1, 1978 to January 31, 1983 by The University of Alabama in Huntsville towards the fulfillment of the Contract NAS8-33096 from George C. Marshall Space Flight Center, Alabama.

The author gratefully acknowledges the numerous discussions with and helpful comments of Mr. John Gould, Mr. Robert Jones and Mr. Klaus Jurgensen during this research work.

Dr. Donald Dietmeyer provided the sources of several programs that are now part of the DDL system. Anil Shah, Jim Covington and Chitra Srinivas developed the majority of the other programs. Caryl Chandler and Jo Peddycoart provided the staff support. It is a pleasure to acknowledge the contributions of these individuals.

ABSTRACT

Digital Systems Design Language has been integrated into the CADAT system environment of NASA-MSFC. This document summarizes the major technical aspects of this integration. Automatic hardware synthesis is now possible starting with a high-level description of the system to be synthesized. The DDL system provides a high-level design verification capability, thereby minimizing design changes in the later stages of the design cycle. An overview of the DDL system covering the translation, simulation and synthesis capabilities is provided. Two companion documents (the user's and programmer's manuals) are to be consulted for detailed discussions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

As the integrated circuit technology advanced to the Very Large Scale Integration (VLSI) era, the complexity of a digital system that can be implemented on a chip has increased tremendously. Structured, top-down design methodologies [1,2,3] have evolved to "divide and conquer" this complexity. The design now is usually performed by a team of designers rather than an individual designer. Computer Hardware Description Languages (CHDL) [4] are designed to enhance the efficiency of communication between designers by enabling a precise yet concise description of the hardware structure and behavior. In addition to documentation , CHDLs have also been used for simulation, test-vector generation, design verification and synthesis. We will describe an automatic hardware synthesis system based on Digital Systems Design Language (DDL) [5]. The main reason for the development of the DDL system is to provide a high-level design/description/simulation environment to the traditional logic-net input oriented Computer Aided Design and Test System (CADAT) [6] of NASA-Marshall Space Flight Center.

Traditionally, logic diagrams or equivalent net-lists are used to input the design details into an automatic design system. This requires that the designer spend an enormous amount of time in generating the logic diagrams after the conception of the design. Further, the verification of the design is deferred to the logic simulation stage, after the logic diagrams are generated and input into the design system. This design environment is adequate for a Small/Medium Scale Integrated Circuit (SSI/MSI) design, but in Very Large Scale Integrated Circuit (VLSI) design, system complexities require that the design be verified as early in the design cycle as possible to prevent costly changes to the design at the low levels. Further, since a

proper bread board for a VLSI circuit is the circuit itself [6], a thorough computer verification of the design at the earliest stage in the design is mandatory.

The CADAT system is used in the design and fabrication of integrated circuits for inhouse use at NASA. It is a traditional computer-aided LSI design system used in the fabrication of PMOS, NMOS, PMOS/NMOS and CMOS circuits using single or double level interconnect metallization and in either random-logic (using standard cells) or more structured, standard transistor array logic technologies. Figure 1 shows the utility of DDL system in the CADAT design environment.

After a survey of the available CHDLs [4], DDL was chosen for the CADAT system. This report summarizes the major technical aspects of the research work conducted under NAS8-33096, since September 1978. Two companion reports are to be consulted for a detailed treatment of the DDL system:

DDL System User's Manual, December 1982.

DDL System Programmer's Manual, December 1982.

The following components of the DDL System were originally developed at the University of Wisconsin and were modified to suit the NASA-MSFC design environment:

Translator (DDLTRN)

Simulator (DDLSIM)

PLA Synthesizer (PLASYN)

Multiple-output Minimization Program (MOMIN)

A hardware synthesis algorithm was formulated and the logic minimization routines were interfaced during the contract period. Chapter 2 provides an overview of the current version of the DDL system. Chapter 3 summarizes the

hardware synthesis algorithm and provides some implementation details. A detailed synthesis example is shown in Chapter 4. PLA synthesis is discussed in Chapter 5. Logic minimization interface is summarized in Chapter 6. Chapter 7 concludesthe report. A complete list of publications under this contract is provided in the Appendix.

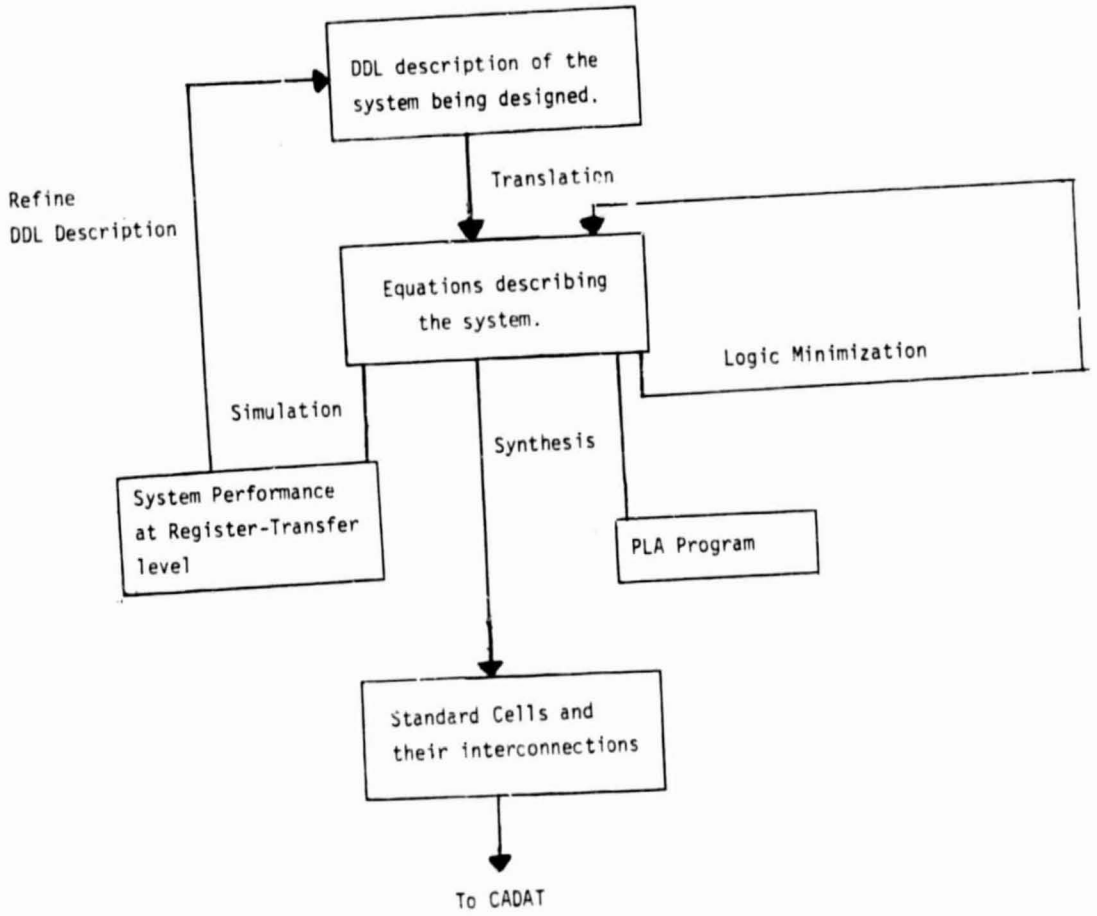


Figure 1: Utility of DDL system in CADAT environment.

The DDL software system consists of a translator, a simulator and two hardware synthesizers. The translator (DDLTRN) translates the DDL description into a set of Boolean (BE) and Register Transfer (RTE) equations. The Simulator (DDLSIM) provides a register-transfer level, two-value simulation capability. The synthesizer (DDLSYN) selects a set of standard cells from a cell library and provides an interconnect list of these cells to realize the BEs and RTEs. The PLA synthesizer (PLASYN) produces a PLA program for the combinational portion of the System. A brief description of the above components follows:

DDL

DDL was introduced in 1968 by Duley and Dietmeyer [5,7]. It is suitable for intermediate level of description of a digital system between the extremely abstract level and the fabrication level. All structural elements are explicitly declared in a DDL description. At the lower level of description, functional and structural elements correspond directly to the actual elements of the system. DDL is highly suitable for describing the system at the gate, register transfer and major combinational block level. The logical statements can be formed using the available primitive operators. The functional specification of the system consists of these logical statements, in blocks. The statements describe the state transitions of a finite state machine controlling the processes of the intended algorithm. The block then appears as an automaton. Parallel operations are permitted. Synchronous behavior is described by either identifying the pulses or by including delay elements described in terms of multiples of clock pulses. Asynchronous behavior is modeled by using conditional statements. Data paths can be explicitly declared by using terminal declarations.

Further details on the syntatic features of the language can be found in [11]. Two new constructs were included in the current implementation of DDL [11,13] to enable a modular description, simulation and synthesis. The new constructs are the MODULE and the DEFINE MODULE and are described in the following paragraphs.

The MODULE declaration is similar to the system declaration or automata declaration with the exception that all equations implied by the DDL description bounded by the MODULE are translated separately from the rest of the system. The module declaration also differs from the system and automata declaration in that the operations are not actually contained in the declaration, but are only called by the module declaration. The DEFINE MODULE (DM) declaration is used to actually contain the DDL operations. To tie the Input/Output information for the DDL description that will be used in the simulation and synthesis phases. Details of these language constructs are given below:

MODULE CALL

<MO> module name [:BE] [:csop]

[SYM_1$=VALUE$_1$,SYM$_2$=VALUE$_2$...$]

endstatement

where

[ ] - encloses an optional parameter

SYM$_n$ - is a symbolic parameter

VALUE - is a value to be substituted for sym$_n$

endstatement - may be '.' or '<ENDMO>'.

The module name names the block and associates it with a block of code that has been previously defined. The name may not be subscripted or contain parenthetical arguments. The BE and csop, if present, will be inserted in the head of the automata and will then serve the same purpose as in the automata declaration.

DEFINE MODULE AND INPUT/OUTPUT

<DM> **Module name**

<IO> (outputs: inputs) <ENDIO>

DDL statements

<ENDDM>

where the DDL statements may be a set of any allowable DDL declarations with the exception of another <DM> declaration. The define module declaration names the module and delimits the beginning and end of the DDL statements that make up a module. The define module declaration is required whether the module will reside in the temporary or permanent module library. One IO declaration is required for each module declaration and it must be the first declaration following the define module declaration. The purpose of this IO declaration is twofold: it makes the designer think about what the input/output interface of the module should be and gives the translator the capability of creating an element (EL) declaration in the main system at the point of call. (the ELEMENT declaration in DDL identifies a black box with only Input/Output signals defined). The inclusion of an element at this point gives the designer the capability of specifying values for the outputs of the blackbox at simulation time so that it is not necessary to have all components of the overall system designed at one time. This will allow a top-down approach to the hardware design process.

The scope of a module is defined in a manner that is consistent with the remainder of DDL, i.e., any declaration on a system level is considered global to any module within that declaration. Any declaration within a module is local to that module and may not be referenced by any declaration outside the module. If a module is contained within another module, then the higher level module declaration will be considered global to the lower level module.

DDLTRN

The DDL Translator [7,11] is a six-pass translator that compiles the DDL description into a Facility Table and a set of Boolean equations (corresponding to the combinational logic portion) and a set of Register Transfer equations (corresponding to the sequential logic portion). A seventh pass was added [14] to the translator so that the BEs and RTEs could be rearranged to eliminate duplicate expressions and Boolean constants. The current version of the translator [13] accepts the modular description constructs described earlier and translates each module independently from the others.

When a module declaration is encountered by the translator, the entire module declaration is parsed into name, BE, csop and symbolic parameters. The name field is then used to access an external file that contains the DDL statements that make up the module description. The [illegible] was defined. If it was defined using a define module declaration, then it will be found in a temporary file that the translator recreates each time it is executed; hence, the DM declared modules are temporary. If it is not found in the define module file, then the library file is searched for the description. These descriptions are permanent and available to

the designer each time he uses the translator. As soon as the description is located, an intermediate file is created and the state of the translator is saved (nesting level, global symbols, etc.) so that translation of the module can be done after translation of the present system is complete. The module description is now scanned and the IO declaration located and saved (must be the first declaration). The description is scanned and substitution is made for the csop and BE in the automata head, if the module is an automata. The description is now scanned for any symbolic parameters and the necessary substitutions made. At this point the module description is prepared for translation, so that translation of the main description may now resume. This process is repeated for each module that is encountered during translation. After translation of the main description is complete, the translation of each module proceeds in a sequential manner.

DDLSIM

The output of DDLTRN is the system description input to DDLSIM [8]. A simulation command language enables the designer to input and output various simulation parameters and control the simulation process. DDLSIM is a two-value, register transfer level simulator. The command language has the following capabilities:

- Declaration of new facilities (CLOCKS, DELAYS) and TRIGGER signals for simulation time.
- Initialization of the contents of various facilities.
- Read/Load data
- Output data
- Dump memory contents

Each MODULE can be completely translated by DDLTRN, thereby obtaining a single-level description of the system for a single-level simulation. The designer can choose to retain some modules at the element (black box) level and expand the others, at the translation phase. A multi-level simulation capability is thus provided. It is the designer's responsibility to provide the output information and verify the input information for the modules retained at the element level, during simulation.

DDLSYN [9,10,12,13]

DDLSYN is a hardware compiler. The BEs and RTEs output by DDLTRN are used by DDLSYN to compile a list of standard cells and their interconnections. A subset of the CADAT standard cell library (Table 1) was used. Two modes of synthesis are possible: modular and non-modular. For a non-modular synthesis, the designer commands DDLTRN to expand each module and generate one set of BEs and RTEs for the complete system. For a modular synthesis, each module is translated separately into a set of BEs and RTEs and synthesized individually by DDLSYN. The output of DDLSYN consists of:

- a list of standard cells chosen (Cell Table),
- an interconnection list, (Net Table),
- cross reference list (Identifier Table).

In addition to these, a module interconnection list will also be produced by DDLSYN, in the modular synthesis mode.

PLASYN [14]

The PLA Synthesizer uses the output of DDLTRN and produces a PLA program to implement the combinational logic portion of the system described in DDL. The RTEs and high fan-in gates are left for manual design. The PLA program is

simply a coded representation of the connections on the AND and OR array of a PLA. The PLA input limit, output limit and product-term limit are the parameters supplied by the designer.

LOGIC MINIMIZATION [15]

The BEs and RTEs produced by DDLTRN are not completely minimized. Although the minimization may not be required during the initial phases of the design cycle, it might be desirable to apply formal minimization techniques before the design is finalized. A multiple-output minimization program (MOMIN) is included in the DDL system. Due to the memory limitations, the number of variables (input and output combined) that can be accomodated by MOMIN is 16. The logic minimization interface partitions the BEs and RTEs to obey the above limit and minimized each partition of equitions. The use of minimization program is optional.

Figure 2 shows an overview of the DDL system.

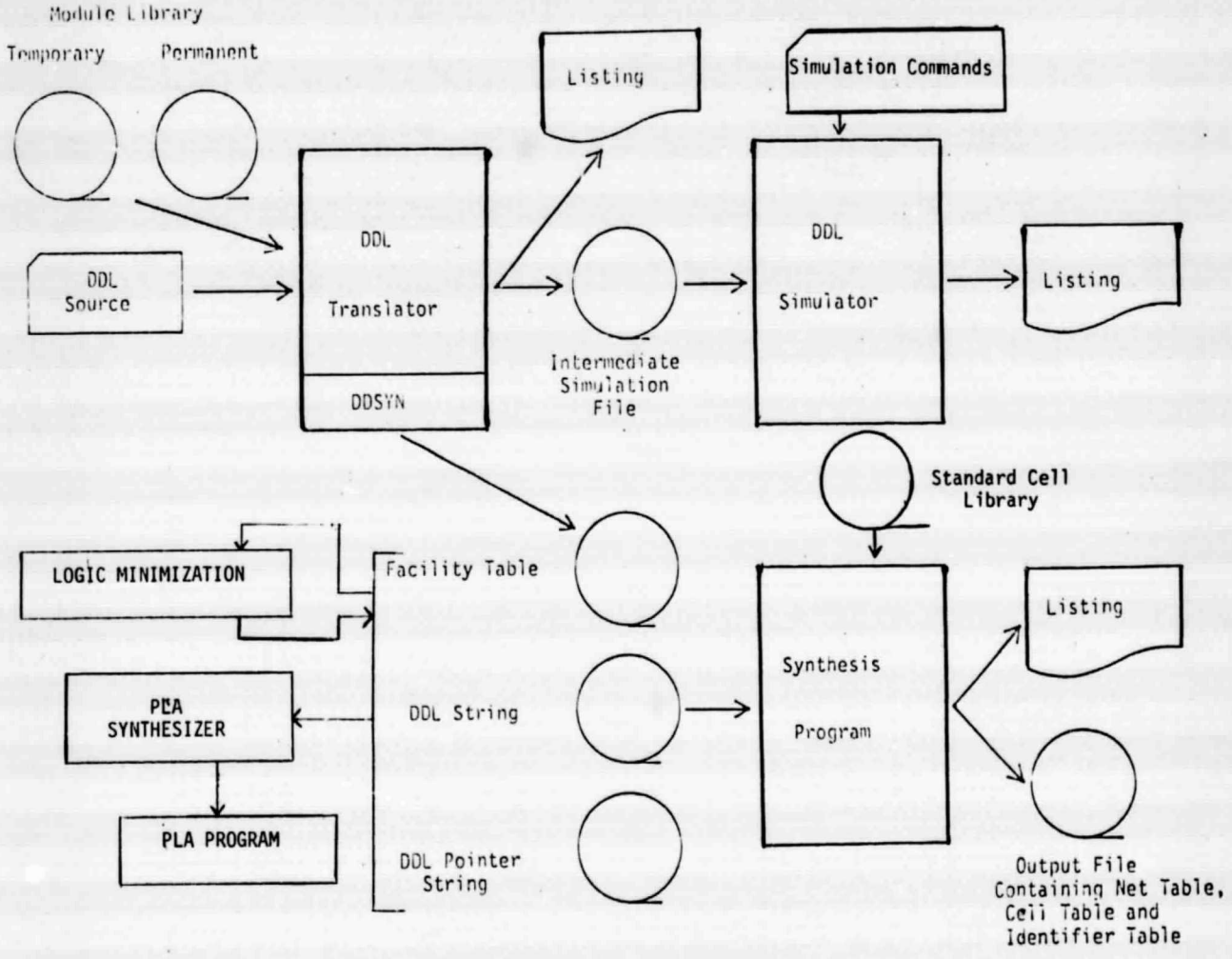


Figure 2: DDL System Overview

3. LOGIC SYNTHESIS ALGORITHM [9,10]

The BEs and RTEs are broken up into components and these are then matched to a standard cell library to choose a cell or set of cells that will realize the given function. Table 1 contains a partial list of the standard cells currently available with the number of devices for each cell and the cell width (as a measure of the silicon area needed). The last column shows the literals in each product term (LPP) of the function realized by the cells. The terms containing all 1's (11, 111, 1111) and those with one product term (1, 2, 3, 4) correspond to a single gate realization. Since it is desirable to realize a function by using the largest standard cell possible, more complex cells are included in the library (2222, 2112, 222, 22, etc). Note that the maximum number of product terms that can be accommodated in the largest cell is four, so that a function with more than four product terms is split into several 4-term units. An additional gate must then be used to combine these 4-term units into a single function.

The synthesis algorithm requires that the BEs be in the sum of products (SOP) form. Hence the BE output from the DDL translator must be changed to this form. The RTEs are synthesized by breaking the equations into two parts; the first corresponding to the condition part and the second to the transfer part. Each of these is in turn a BE so that the same synthesis algorithm may be applied to them. The overall synthesis algorithm is discussed next followed by the combinational logic synthesis algorithm.

3.1 SYNTHESIS ALGORITHM

The overall synthesis algorithm has the following five steps:

1) Memory references are reduced to memory READ and WRITE signals.

2) RTEs are broken into two BEs corresponding to the condition and transfer portions.

3) Equations with selection and reduction operators are reduced to SOP form.

4) Exclusive-OR operators, constants and parentheses are eliminated from the equation.

5) BEs in SOP form are now synthesized using the combinational logic synthesis algorithm.

The following sections cover these steps in greater detail.

3.1.1 Memory

The memory references in DDL are of the form M(MAR) where MAR must be the same register for all references to the memory M. A memory reference is interpreted as a read if it is on the right hand side of an equation or as a write if it is on the left hand side. In modeling the memory for synthesis, it is assumed that the memory module has an address decoder, a memory bus as wide as one word and read and write input signals. It is then only necessary to generate the correct input signals to synthesize the memory equation.

3.1.2 Selection and Reduction Operators

It is necessary to expand the selection and reduction operators to their true SOP form before they are synthesized. This is accomplished by performing the following steps:

1) If a selection operator is present, synthesize it by complementing the bits of its left operand if a zero appears in the corresponding position of the right operand.

2) Place the reduction operator between each bit of the selected left operand.

Example:

Assume that A is two bits wide for the following equations:

1) B = */ A ' 3D2
2) B = */ A ' 11 Expand constant
3) B = */ A(1) A(2) Apply step 1
4) B = A(1) * A(2)

3.1.3 Combinational Logic Synthesis Algorithm

The combinational logic synthesis algorithm consists of the following steps where the number of digits in the LPP is n and K_i is the ith digit of the LPP.

1) Scan the Boolean function to be implemented and count the number of literals in each product term to determine the digits of the LPP. If the product term contains more than two literals (function of the library), it must be reduced to a term with only one literal. This is accomplished by using one or more AND gates to realize the term individually.

2) Rearrange the LPP in descending order of its component digits.

3) If n is greater than four the LPP is split into two or more four digit units (the last unit may have less than four digits). Each of these four digit units is implemented separately so that the four digit unit may be replaced by a 1 in the original LPP.

If n is less than or equal to four it is compared to all the n digit standard cell LPPs until a standard cell is found that has a minimum number of mismatches. The mismatches are determined by the following criteria:

a) If the four digit unit is a sum term (K_i=1 for all i=1 to n), then the mismatches will be zero and the unit will be implemented using an OR gate with the proper number of inputs.

If the four digit unit is a sum term but is contained within a larger unit that contains at least one instance of K_i=2 then it will have a mismatch of zero and be implemented as a NOR cell. For example, in the LPP=22221111 the unit 1111 is implemented using a four input NOR gate.

b) If in the four digit unit there is at least one instance where K_i=2, then the mismatches shall be equal to the number of digits numerically less than its corresponding digit. The best match will then be found and the four digit unit implemented as this library cell.

Examples:

Four Digit Unit	Library Cell	Mismatches
2221	2222	1
2111	2211	1
221	222	1
211	222	2
21	22	1

4) The final implementation depends on the LPP as well as the library cell selected. The various options are explained below and summarized in Table 2.

a) The synthesis for K_i=1 for all i=1 to n where n is less than or equal to four is completed in step three and no further action is required.

b) The outputs from step three for all four digit units generated for equations in which K_i=1 for all i=1 to n when n is greater than four are combined into a single output by adding an OR gate.

c) The output of all LPPs in which K_i=2 appears one or more times must be inverted due to the nature of the more complex standard cells. This could possibly not have to be done if a standard cell was available that did not have an inverted output was available.

d) When K_i=2 for any i when n is greater than four and less than or equal to sixteen then a NAND gate is used to connect all the individual implementations of the four digit units. In this case, the inverter is not necessary since the NAND gate is used.

e) When K_i=2 for any i when n is greater than sixteen then an OR gate is used to connect all the individual groups of sixteen that have been synthesized as in part d.

f) If the LPP is a product term (K_i is greater than or equal to two for n equal to one) then it is implemented using one or more AND gates.

5) Compare any saved input and output identifiers from previous modules to the identifiers in the present module's identifier table for a match. If a match is found, this will be the second point in the point to point connection and the identifiers associated net must be saved. A comparison of identifiers in this fashion may be made since the translator forces all identifiers to be unique within a system even though they may be in two seperate modules.

6) Repeat steps one through five until all modules are synthesized and output the results.

The above procedures are implemented in DDLSYN and the resulting module interconnection list is output on both a cell level and on an identifier level.

The example below will serve to illustrate the above steps.

Example

Z = A + B * C + D * E * F + G * H * I

Step 1	1	2	3	3	
	1	2	1	1	3 to 1 by including 1630
Step 2	1	1	1	1	
Step 3					Violations = 1
Step 4	Must invert output of selected gate (1860)				

This Boolean equation can be implemented using one 1860, two 1630's and one 1310. The implementation is shown in Figure 3.

DDLSYN implements the preceding algorithm. The cell table, net table and identifier table are provided as the output. This information is complete enough to represent the logic that was implied by the DDL description.

Table 1: CADAT Standard Cell Library (Partial)

Cell No.	Type	No. of Devices	Cell Width (mils)	Function	Literals/Product Term
1120	2 input NOR	4	5.8	$\overline{A + B}$	1,1
1130	3 input NOR	6	7.7	$\overline{A + B + C}$	1,1,1
1140	4 input NOR	8	9.6	$\overline{A + B + C + D}$	1,1,1,1
1220	2 input NAND	4	5.8	$\overline{A \cdot B}$	2
1230	3 input NAND	6	7.7	$\overline{A \cdot B \cdot C}$	3
1240	4 input NAND	8	9.6	$\overline{A \cdot B \cdot C \cdot D}$	4
1310	Buffer Inverter	2	3.9	$\overline{A}$	1
1620	2 input AND	6	5.8	$A \cdot B$	2
1630	3 input AND	8	7.7	$A \cdot B \cdot C$	3
1640	4 input AND	10	9.6	$A \cdot B \cdot C \cdot D$	4
1720	2 input OR	6	5.8	$A + B$	1,1
1730	3 input OR	8	7.7	$A + B + C$	1,1,1
1740	4 input OR	10	9.6	$A + B + C + D$	1,1,1,1
1800	4 x 2 input AND + 4 x NOR	16	17.2	$\overline{(AB + CD + EF + GH)}$	2,2,2,2
1840	3 x 2½ input AND + 2 input NOR	10	11.6	$\overline{C(AB + DE)}$*	—
1960	2 x 2 input AND + 4 input NOR	12	13.7	$\overline{AB + E + F + CD}$	2,1,1,2
1870	2 x 2 input AND + 2 input NOR	8	9.6	$\overline{(AB + CD)}$	2,2
1880	2 bit carry Anticipate	10	14.9	$\overline{(CDE) + BE + A}$*	—
1890	3 x 2 input AND + 3 input NOR	12	16.9	$\overline{AB + CD + EF}$	2,2,2
2310	2 input EXOR	8	7.8	$A \oplus B$	1,1

* Special Functions

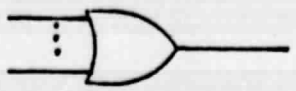
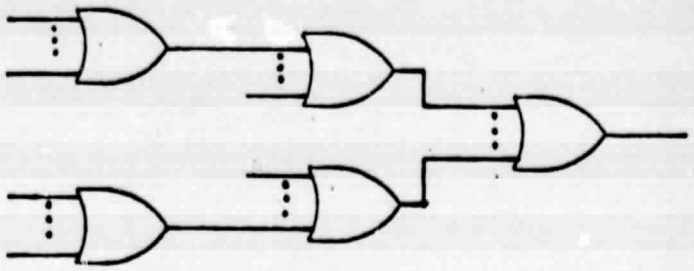
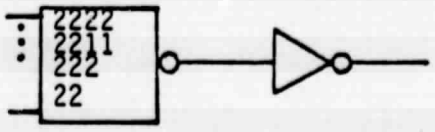
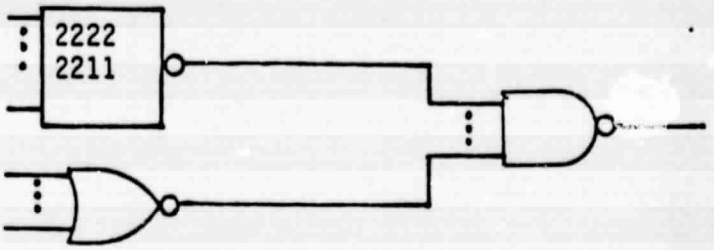
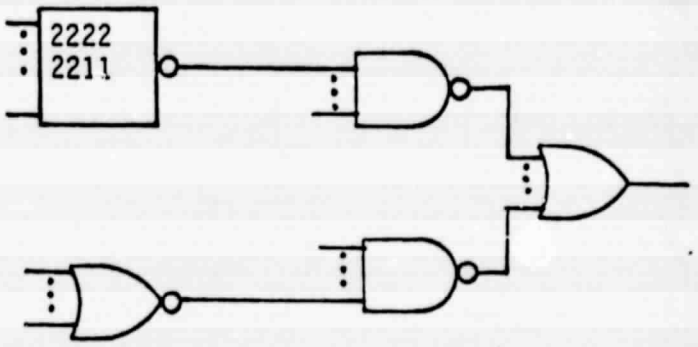
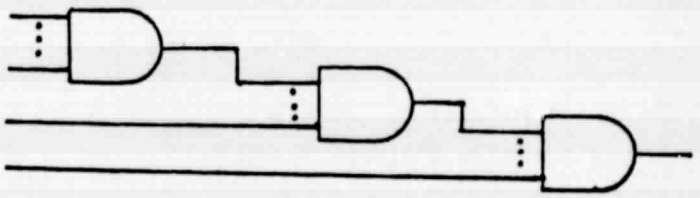
LPP configuration	Implementation
a) $K_i=1$, for all i=1 to n, $n \leq 4$.	
b) $K_i=1$, for all i=1 to n, $n>4$.	
c) $K_i=2$, for any i=1 to n, $n \leq 4$.	
d) $K_i=2$, for any i=1 to n, $4<n \leq 16$.	
e) $K_i=2$, for any i=1 to n, $n>16$.	
f) $K_i \geq 2$, for n=1.	

Table 2: Step 4 Implementations

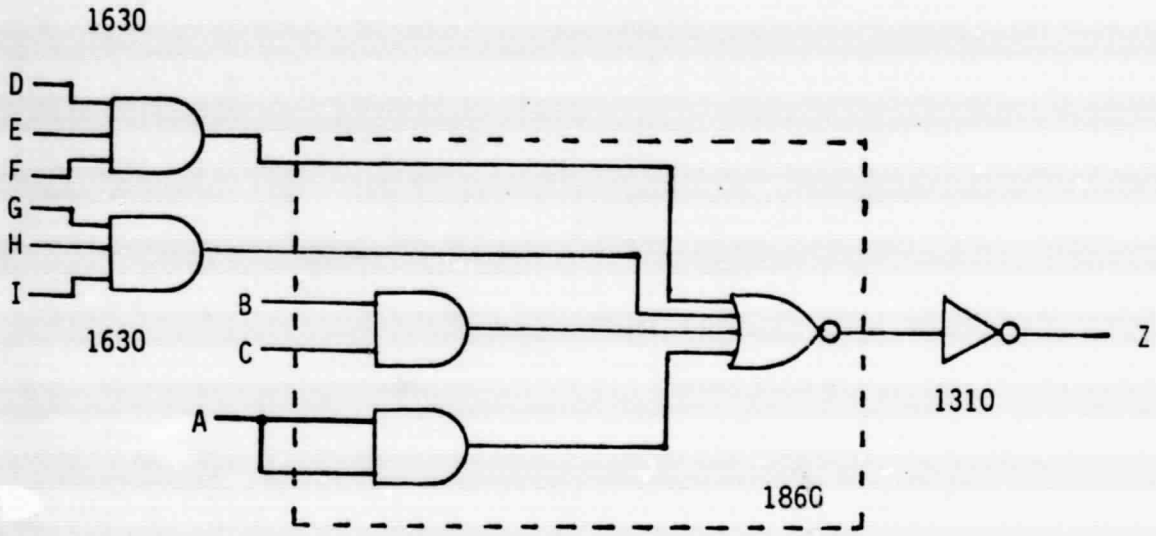


Figure 3: Implementation of Z=A+B*C+D*E*F+G*H*I

3.2 MODULAR SYNTHESIS

The synthesizer that was written to implement the algorithm described in the previous section was designed to synthesize only one group of equations at a time; nowever, a system design with modules contains at least two groups of equations and may contain many more. Synthesis of this type system requires the synthesizer to loop once for each module and to connect each individual synthesis output together to form an overall system connection list. The previously described synthesis program will lend itself fairly well to these modifications. This section will present an algorithm for connecting modules along with the implementation details of the algorithm.

3.2.1. Connection Algorithm

Conceptually the problem of connecting modules can be thought of as the point to point connection of a wire or of drawing a line from one point to another on a circuit diagram. However, when the hardware of the system is represented in a computer memory by an identifier table, a net table and a cell table and it is undesirable to retain more than one module's tables in memory at one time. The connection algorithm is not quite so straightforward. It is also desirable when connecting modules that both a connection list on an identifier level

[illegible] not only the identifiers, but also their associated cells. An algorithm to accomplish the above objectives is presented here and is represented diagramatically in Figure 4.

1) Read the output data files (facility table, DDL string and DDL pointer string) from the DDL translator to obtain the complete description for one module.

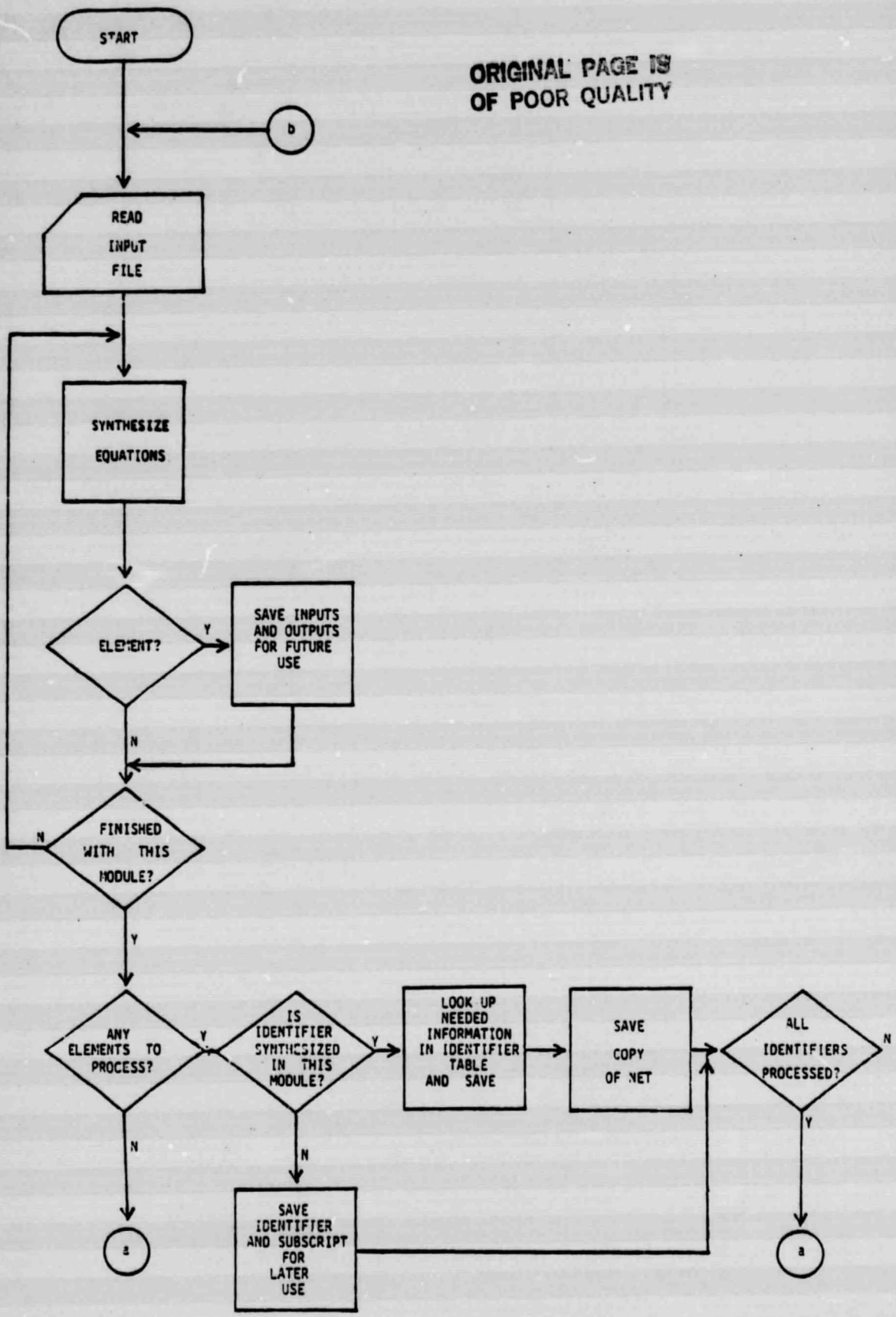


Figure 4: Connection Algorithm

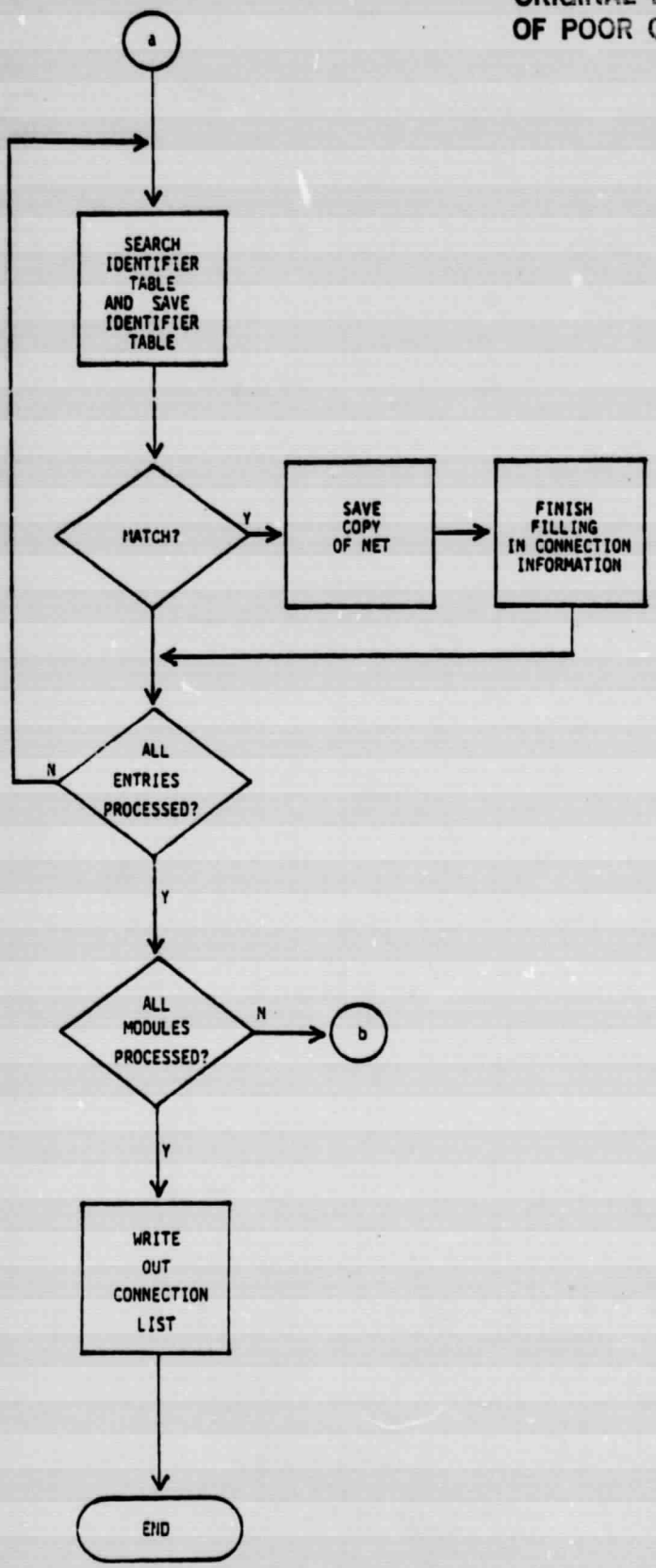


Figure 4: (Cont)

2) Begin synthesizing the DDL equations in a normal fashion
If an element declaration is encountered in the module, save all the declared inputs and outputs so that the connection process can be triggered.

3) Continue synthesizing until all equations have been processed for this module. At this point the identifier table, net table and cell table are complete and the declared inputs and outputs are known.

4) Look up each input and output identifier in the identifier table to find its associated net and save a copy of this net. At this time all information for one point of the point to point connection has been found.

4. A SYNTHESIS EXAMPLE

This chapter illustrates the complete synthesis process using one example description. Both modular and non-modular modes are illustrated. Typical outputs from DDLTRN, DDLSIM, and DDLSYN are shown.

Figure 5(a) shows a description of a serial twos complementer in DDL. This complementer uses the popular copy/complement algorithm:

1) Starting from the least significant bit, copy the bits as they are until the first non-zero bit is encountered.

2) After this bit, complement all remaining bits in the word.

The algorithm is implemented using a shift register that is right circulated while copying or complementing as required.

Four registers are used by the complementer and are declared in line two of Figure 7(a). R is a six bit register whose contents are to be complemented and placed back in R. The three bit register C is used to count the number of shifts necessary (six in this case since R is six bits wide). The register S is a state flip-flop to indicate the copy or complement state and T is a control flip-flop to indicate the RUN/STOP state of the complementer. The clock P is used to synchronize the state transitions of the complementer. In lines five through eight, an operator ADD is declared. This is a three bit adder to increment the contents of the argument register by one. Lines nine through twelve declare an automata COMP that has two states:

1) A waiting state I

2) A processing state S1.

Setting of switch SW is required for the transitions from state I to state S1. In the state S1, the register R is circulated right one bit

with the least significant bit copied or complemented, depending on S being a zero or one. If the register C has reached a value of five, the complementing is stopped by setting T to zero and returning to state I. If C is less than five, COMP stays in the state S1 and increments C.[1]

Figure 5(b) shows the output equations generated by the translator. It can be seen from this figure, that even though the input description has two clearly defined blocks (Operator and Automata declarations) the output equations show no distinction between these blocks.

Figure 6(a) shows the identical complementer but this time, the automata is contained within a module. Lines one through eight are the Define Module and Input/Output declarations which actually contain the previously described automata. The symbolic register REG is declared to be both input and output while the symbolic switch, SWITCH, is declared to be only input. In line seventeen the module is referenced by a Module declaration and the symbolic parameters SWITCH, REG and CON are all assigned values. The use of symbolic parameters allows the designer the flexibility of not being tied to those variable names that were assigned in the module, that is, he may assign them any name he chooses by the use of symbolic parameters. Lines eighteen and nineteen are the Element declaration that the translator generates for the module

Figure 5 (c) and 6 (d) show the identifier tables, input lists, net tables and cell tables that were generated from the synthesis of the two previously described systems. The input list shows any identifiers that were not internally generated within the system or module being synthesized. In the first synthesis (Figure 5 (c) SW, an on/off switch, and P, a clock, must be supplied from an external source. In the second synthesis (Figure 6(d) the first

module requires X(1), X(2) and C"1(3), which is equivalent to X(3), as its inputs while the second module requires SW, P, ADD(1), ADD(2) and ADD(3) as inputs. Looking back at the identifier tables it can be seen that ADD(1), ADD(2) and ADD(3) are generated from module one (DRIV) while X(1), X(2) and C"1(3) are generated in module two (MCMP). This information can be used to generate the IO declaration required by the translator if the inputs and outputs were not immediately obvious to the designer.

The identifier table contains all identifiers that were encountered in the synthesis and serves to associate the identifiers with the net table. In the identifier table, an entry with all Xs is an internally generated output of a net. All other entries are outputs of the corresponding net.

The [illegible] table contains the connectivity information on a cell level. For example, net one of Figure 5 (c) shows that cell 1000, pin 3 is the driver (the driver cell is always the first cell unless the signal is generated externally) and drives cell 1001 pin 3. Looking in the cell table, it can be seen that cell 1000 is a 1310 and cell 1001 is a 1620. From Table 1. it is found that cell 1310 is a buffer inverter while cell 1520 is a two input AND. Proceeding in this manner all nets could be expanded and a circuit diagram such as Figure 7 could be drawn.

These four tables describe all necessary connection information on a cell level and are sufficient for synthesis of a single module system. In multiple module systems it is necessary to add two additional tables: the identifier connection list and the cell

connection list. The tables showing the connection information for the two module system of Figure 6 (a) is shown in Figure 6 (f). The first table shows that the identifier X(1) is generated in module MCMP and is input to the module DRIV. The second table shows that the identifier X(1) is generated by cell 2049 pin 4 and drives cells 1000 pin 3 and 1002 pin 2. With this information and the identifier table, cell table and net table the circuit diagram of Figure 8 can be drawn.

To verify the designs of these two complementers, a simulation run was made with the input commands of Figure 9 (a). In line one, flags for DDLSIM are set for decimal data input (4) and binary output (6). In line two, SW is set to a one to begin the complementation process. Line three tells the simulator to read a value into R each time the complementer is in the state T. Since two values are specified (5 & 20), the simulator will perform two loops through the simulation. An output trigger, OUTTR, is declared to be on at the falling edge of clock P in line four. In line five, the values of COMP, R, S, C and T are output each time OUTTR is on and that of R when in state I. The simulation is started in line six.

Figure 9 (b) shows the simulation output that was produced by both complementers. Simulation of a system with more than one module is made by setting flag seventeen of the translator to a one. This flag tells the translator to expand all modules in-line at the point of call resulting in identical simulation results if both translations are valid. For this reason only one simulation output is reproduced here. At time zero, all registers are zeroed and the circuit is in

state I. On the next leading edge of the clock time advances to one and the switch is set to a one. At time two, R receives a 5. Twelve more time slots (6 clock pulses) are required for R to have its twos complement (time = 14). At time sixteen, the new value for R (20) is received and its twos complement is ready at time twenty-eight. Since all inputs are exhausted, the simulator stops at time twenty-nine.

```
      DIGITAL DESIGN LANGUAGE TRANSLATION

 1:   <S1>COMPLEMENTER.TEST:
 2:   <BE>R(1:8),C(2:0),S,I.
 3:   <L4>S..
 4:   <T1>P.
 5:   <LP> ADD(3)*XS
 6:        <IE> X(3),C(3).
 7:             <ID> CC=((C(2:3)||C1).
 8:        <MI> C=X*CC,A!C=>ALC..
 9:   <AL>CLAP:R:
10:         <S1>I(0):S..:I<-1,C<-C,S<-0,->S1.
11:           S1(1):T:]S) R(1)<-TN(E),R(2:8)<-R(1:5);S<-R(6),R<-R(E)||R(1:5)..
12:           JC(2)*TC(1)*C((0)]1<-0,->1;C<-A.B.C'.....
```

Figure 5(a) Serial Twos Complementer (Input Description)

```
DIGITAL DESIGN LANGUAGE TRANSLATOR

      PASS7--SIMPLIFICATION


<SY> ELEMENTS:
      1=T(CLK(1)),
      S1=(CLR(1)),
      "1=1*S1,
      "2=S1*1,
      "3="2*S,
      "4="2*1S,
      "5="2*C(2)*T(1)*C(0),
      "6="2*T((2)*1C(1)*C(0)),
      "7=F*"1 + F*"6,
      "8=F*"1 + F*"4,
      "9=F*"1 + F*"5,
      "10=F*"3 + F*"4,
      "11="4*R(6),
      "12="3*TR(6) + "4*R(6),
      C"1(1:2)=X(1:2)*C"1(2:3),
      C"1(3)=X(3),
      AD(1:2)=(X(1:2)*C"1(2:3)),
      AD(3)=(X(3)*1D1),
      )"9) T<-"1.,
      )"7) C<-"6*AND.,
      )"8) S<-"11.,
      )"9) CLR<-"1.,
      )"10) R(1)<-"12.,
      )"10) R(2:6)<-"3*R(1:5) + "4*R(1:5).,
      Y="6*C, .
```

Figure5(b) Serial Twos Complementer (Output Equations)

```
DIGITAL DESIGN LANGUAGE SYNTHESIZER
DESCRIPTION OF MODULE -     DRIV

    IDENTIFIER TABLE

NO.      IDENTIFIER    NO.      IDENTIFIER    NO.      IDENTIFIER
 1          I( 1)       2       COMP( 1)       3         S1( 1)
 4         *1( 1)       5         SW( 1)       6         *2( 1)
 7          T( 1)       8         *3( 1)       9          S( 1)
10         *4( 1)      11   XXXXXXXX( 1)      12         *5( 1)
13          C( 2)      14          C( 1)      15          C( 0)
16   XXXXXXXX( 1)      17         *6( 1)      18   XXXXXXXX( 1)
19   XXXXXXXX( 1)      20         *7( 1)      21          P( 1)
22   XXXXXXXX( 1)      23         *8( 1)      24   XXXXXXXX( 1)
25         *9( 1)      26   XXXXXXXX( 1)      27        *10( 1)
28   XXXXXXXX( 1)      29        *11( 1)      30          R( 6)
31        *12( 1)      32   XXXXXXXX( 1)      33   XXXXXXXX( 1)
34        C*1( 1)      35          X( 1)      36        C*1( 2)
37          X( 2)      38       C*1( 3)       39          X( 3)
40        ADD( 1)      41        ADD( 2)      42        ADD( 3)
43   XXXXXXXX( 1)      44   XXXXXXXX( 1)      45   XXXXXXXX( 1)
46          R( 1)      47          R( 2)      48   XXXXXXXX( 1)
49   XXXXXXXX( 1)      50          R( 3)      51   XXXXXXXX( 1)
52   XXXXXXXX( 1)      53          R( 4)      54   XXXXXXXX( 1)
55   XXXXXXXX( 1)      56          R( 5)      57   XXXXXXXX( 1)
58   XXXXXXXX( 1)      59   XXXXXXXX( 1)      60   XXXXXXXX( 1)
61   XXXXXXXX( 1)


    INPUT LIST

NET      IDENTIFIER
  5         SW( 1)
 21          P( 1)
```

Figure 5 (c). Synthesis Outputs

DIGITAL DESIGN LANGUAGE SYNTHESIZER
DESCRIPTION OF MODULE - DRIV

NET TABLE

NET	CELL	PIN	CELL	PIN	CELL	PIN	CELL	PIN	CELL	PIN
1	1000	3	1001	3						
3	1036	4	1000	2	1002	3				
4	1001	4	1011	4	1013	4	1015	4	1028	3
	1036	3								
5	1001	2								
6	1002	4	1003	3	1005	3	1007	5	1010	3
7	1028	4	1002	2						
8	1003	4	1017	4	1021	5	1038	5	1041	5
	1044	5	1047	5	1050	5				
9	1035	4	1004	2	1003	2				
10	1005	4	1013	2	1017	2	1019	3	1021	3
	1038	3	1041	3	1044	3	1047	3	1050	3
11	1004	3	1005	2						
12	1007	6	1015	2						
13	1030	4	1009	4	1007	4	1054	2		
14	1032	4	1008	2	1006	2	1055	2		
15	1034	4	1009	2	1007	2	1056	2		
16	1006	3	1007	3						
17	1010	4	1011	2	1029	3	1031	3	1033	3
	1054	3	1055	3	1056	3				
18	1008	3	1009	3						
19	1009	5	1010	2						
20	1012	3	1030	2	1032	2	1034	2		
21	1011	3	1011	5	1013	3	1013	5	1015	3
	1015	5	1017	3	1017	5				
22	1011	6	1012	2						
23	1014	3	1035	2						
24	1013	6	1014	2						
25	1016	3	1028	2	1036	2				
26	1015	6	1016	2						
27	1018	3	1037	2	1040	2	1043	2	1046	2
	1049	2	1053	2						
28	1017	6	1018	2						
29	1019	4	1035	3						
30	1053	4	1020	2	1021	2	1019	2		
31	1022	3	1037	3						
32	1020	3	1021	4						
33	1021	6	1022	2						
34	1023	4								
35	1054	4	1025	2	1023	3				
36	1024	4	1023	2	1025	3				
37	1055	4	1026	2	1024	3				

Figure 5 (c). (Cont)

```
38 1056     4 1026     3 1027     2 1024     2
40 1025     4 1029     2
41 1026     4 1031     2
42 1027     3 1033     2
43 1029     4 1030     3
44 1031     4 1032     3
45 1033     4 1034     3
46 1037     4 1038     2 1038     4
47 1040     4 1041     2 1041     4
48 1038     6 1039     2
49 1039     3 1040     3
50 1043     4 1044     2 1044     4
51 1041     6 1042     2
52 1042     3 1043     3
53 1046     4 1047     2 1047     4
54 1044     6 1045     2
55 1045     3 1046     3
56 1049     4 1050     2 1050     4
57 1047     6 1048     2
58 1048     3 1049     3
60 1050     6 1052     2
61 1052     3 1053     3


DIGITAL DESIGN LANGUAGE SYNTHESIZER
DESCRIPTION OF MODULE -      DRIV


    CELL TABLE


CELL STD. CELL STD. CELL STD. CELL STD. CELL STD.
 NO  CELL  NO  CELL  NO  CELL  NO  CELL  NO  CELL
1000 1310 1001 1620 1002 1620 1003 1620 1004 1310
1005 1620 1006 1310 1007 1640 1008 1310 1009 1230
1010 1620 1011 1870 1012 1310 1013 1870 1014 1310
1015 1870 1016 1310 1017 1870 1018 1310 1019 1620
1020 1310 1021 1870 1022 1310 1023 1620 1024 1620
1025 2310 1026 2310 1027 1310 1028 1830 1029 1620
1030 1830 1031 1620 1032 1830 1033 1620 1034 1830
1035 1830 1036 1830 1037 1830 1038 1870 1039 1310
1040 1830 1041 1870 1042 1310 1043 1830 1044 1870
1045 1310 1046 1830 1047 1870 1048 1310 1049 1830
1050 1870 1051 1300 1052 1310 1053 1830 1054 1620
1055 1620 1056 1620
```

Figure 5 (c). (Cont)

```
 1:    <LA> COMP
 2:    <IO> (A,REG(1:6): ASYNC, A,REG(1:6)).
 3:    <AL>COMP:P:
 4:          <ST>I(6):ASYNC:I<-1,ALL.<-0,S<-0,->S1.
 5:            S1(1):I:JS) A,REG(1)<-1A,REG(P),A,REG(2:6)<-A,REG(1:5)
 6:                              ;S<-P,REG(P),P,REG<-A,REG(6)(A,REG(1:5).,
 7:            JACOM(2)*1A(CM(1)*ACOM(0))I<-0,->I;S(OM<-ADDSUB....
 8:    <ENLOM>
 9:    <SY>COMPLEMENTER:
10:    <HE>P(1:6),C(2:0),S,I.
11:    <LA>SA.
12:    <II>P.
13:    <LP> ADD(3)SXS
14:         <IE> X(3),C(3).
15:              <IO> CC=(C(2:3)!!C1).
16:         <EC> C=X*CC,ADD=A,CC..
17:    <PC> COMP:P:SYNC=SY,REG=A,COM=CA.
18:+   <PL>COMP
19:+         (A(1:6): SY, P(1:6)).
20:    <END SY>
```

Figure6 (a) Serial Twos Complementer With Modules (Input Description)

```
DIGITAL DESIGN LANGUAGE TRANSLATOR

      PASS7--SIMPLIFICATION

<SY> LINE*ITEM:
      C"1(1:2)=X(1:2)*C"1(2:3),
      C"1(3)=X(3),
      ALL(1:2)=(X(1:2)+C"1(2:3)),
      ALL(3)=(X(3)+1C1),
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 6(b) Serial Twos Complementer: Module 1 Output Equations

```
DIGITAL DESIGN LANGUAGE TRANSLATOR

      PASS7--SIMPLIFICATION
      DESCRIPTION OF MODULE - COMP

<M0> COMP: J=1CCMP"1(1),
      S1=COMP"1(1),
      "1=1*S1,
      "2=S1*1,
      "3="2*S,
      "4="2*1S,
      "5="2*C(2)*1C(1)*C(0),
      "6="2*1(C(2)*1C(1)*C(0)),
      "7=F*"1 + F*"6,
      "8=F*"1 + F*"4,
      "9=F*"1 + F*"5,
      "10=F*"3 + F*"4,
      "11="4*R(6),
      "12="3*1R(6) + "4*R(6),
      ]"9] 1<-"1.,
      ]"7] C<-"6*APL.,
      ]"8] S<-"11.,
      ]"9] (COMP"1<-"1.,
      ]"10] R(1)<-"12.,
      ]"10] R(2:5)<-"3*R(1:5) + "4*R(1:5).,
      X="6*C, .
```

Figure 6(c) Serial Twos Complementer: Module 2 Output Equations

```
DIGITAL DESIGN LANGUAGE SYNTHESIZER
DESCRIPTION OF MODULE -      DRIV


    IDENTIFIER TABLE

 NO.       IDENTIFIER   NO.       IDENTIFIER   NO.       IDENTIFIER
  1         C*1(  1)     2          X(  1)     3         C*1(  2)
  4          X(  2)     5         C*1(  3)     6          X(  3)
  7         ADD(  1)     8         ADD(  2)     9         ADD(  3)


    INPUT LIST

NET       IDENTIFIER
  2          X(  1)
  4          X(  2)
  5         C*1(  3)


    NET TABLE

  NET CELL  PIN CELL  PIN CELL  PIN CELL  PIN CELL  PIN
    1 1000    4
    2 1000    3 1002    2
    3 1001    4 1000    2 1002    3
    4 1001    3 1003    2
    5 1001    2 1003    3 1004    2
    7 1002    4
    8 1003    4
    9 1004    3


    CELL TABLE

CELL STD. CELL STD. CELL STD. CELL STD. CELL STD.
 NO  CELL  NO  CELL  NO  CELL  NO  CELL  NO  CELL
1000 1620 1001 1620 1002 2310 1003 2310 1004 1310
```

Figure 6 (d). Module 1 Synthesis Output

```
DIGITAL DESIGN LANGUAGE SYNTHESIZER
DESCRIPTION OF MODULE -      MCMP


    IDENTIFIER TABLE

 NO.       IDENTIFIER   NO.       IDENTIFIER   NO.       IDENTIFIER
  1           I(  1)     2         COMP(  1)    3          S1(  1)
  4          "1(  1)     5           SW(  1)    6          "2(  1)
  7           T(  1)     8          "3(  1)     9           S(  1)
 10          "4(  1)    11    XXXXXXXX(  1)    12          "5(  1)
 13           C(  2)    14           C(  1)    15           C(  0)
 16    XXXXXXXX(  1)    17          "6(  1)    18    XXXXXXXX(  1)
 19    XXXXXXXX(  1)    20          "7(  1)    21           P(  1)
 22    XXXXXXXX(  1)    23          "8(  1)    24    XXXXXXXX(  1)
 25          "9(  1)    26    XXXXXXXX(  1)    27         "10(  1)
 28    XXXXXXXX(  1)    29         "11(  1)    30           R(  6)
 31         "12(  1)    32    XXXXXXXX(  1)    33    XXXXXXXX(  1)
 34         ADD(  1)    35    XXXXXXXX(  1)    36         ADD(  2)
 37    XXXXXXXX(  1)    38         ADD(  3)    39    XXXXXXXX(  1)
 40           R(  1)    41           R(  2)    42    XXXXXXXX(  1)
 43    XXXXXXXX(  1)    44           R(  3)    45    XXXXXXXX(  1)
 46    XXXXXXXX(  1)    47           R(  4)    48    XXXXXXXX(  1)
 49    XXXXXXXX(  1)    50           R(  5)    51    XXXXXXXX(  1)
 52    XXXXXXXX(  1)    53    XXXXXXXX(  1)    54    XXXXXXXX(  1)
 55    XXXXXXXX(  1)    56           X(  1)    57           X(  2)
 58           X(  3)


    INPUT LIST

NET       IDENTIFIER
  5         SW(  1)
 21          P(  1)
 34        ADD(  1)
 36        ADD(  2)
 38        ADD(  3)
```

Figure 6 (e). Module 2 Synthesis Output

DIGITAL DESIGN LANGUAGE SYNTHESIZER
DESCRIPTION OF MODULE - MCMP

NET TABLE

NET	CELL	PIN	CELL	PIN	CELL	PIN	CELL	PIN	CELL	PIN
1	2000	3	2001	3						
3	2031	4	2000	2	2002	3				
4	2001	4	2011	4	2013	4	2015	4	2023	3
	2031	3								
5	2001	2								
6	2002	4	2003	3	2005	3	2007	5	2010	3
7	2023	4	2002	2						
8	2003	4	2017	4	2021	5	2033	5	2036	5
	2039	5	2042	5	2045	5				
9	2030	4	2004	2	2003	2				
10	2005	4	2013	2	2017	2	2019	3	2021	3
	2033	3	2036	3	2039	3	2042	3	2045	3
11	2004	3	2005	2						
12	2007	6	2015	2						
13	2025	4	2009	4	2007	4	2049	2		
14	2027	4	2008	2	2006	2	2050	2		
15	2029	4	2009	2	2007	2	2051	2		
16	2006	3	2007	3						
17	2010	4	2011	2	2024	3	2026	3	2028	3
	2049	3	2050	3	2051	3				
18	2008	3	2009	3						
19	2009	5	2010	2						
20	2012	3	2025	2	2027	2	2029	2		
21	2011	3	2011	5	2013	3	2013	5	2015	3
	2015	5	2017	3	2017	5				
22	2011	6	2012	2						
23	2014	3	2030	2						
24	2013	6	2014	2						
25	2016	3	2023	2	2031	2				
26	2015	6	2016	2						
27	2018	3	2032	2	2035	2	2038	2	2041	2
	2044	2	2048	2						
28	2017	6	2018	2						
29	2019	4	2030	3						
30	2048	4	2020	2	2021	2	2019	2		
31	2022	3	2032	3						
32	2020	3	2021	4						
33	2021	6	2022	2						
34	2024	2								
35	2024	4	2025	3						
36	2026	2								
37	2026	4	2027	3						

Figure 6 (e). (Cont)

38	2028	2					
39	2028	4	2029	3			
40	2032	4	2033	2	2033	4	
41	2035	4	2036	2	2036	4	
42	2033	6	2034	2			
43	2034	3	2035	3			
44	2038	4	2039	2	2039	4	
45	2036	6	2037	2			
46	2037	3	2038	3			
47	2041	4	2042	2	2042	4	
48	2039	6	2040	2			
49	2040	3	2041	3			
50	2044	4	2045	2	2045	4	
51	2042	6	2043	2			
52	2043	3	2044	3			
54	2045	6	2047	2			
55	2047	3	2048	3			
56	2049	4					
57	2050	4					
58	2051	4					

DIGITAL DESIGN LANGUAGE SYNTHESIZER
DESCRIPTION OF MODULE - MCMP

CELL TABLE

CELL NO	STD. CELL	CELL NO	STD. CELL	CELL NO	STD. CELL	CELL NO	STD. CELL	CELL NO	STD. CELL
2000	1310	2001	1620	2002	1620	2003	1620	2004	1310
2005	1620	2006	1310	2007	1640	2008	1310	2009	1230
2010	1620	2011	1870	2012	1310	2013	1870	2014	1310
2015	1870	2016	1310	2017	1870	2018	1310	2019	1620
2020	1310	2021	1870	2022	1310	2023	1830	2024	1620
2025	1830	2026	1620	2027	1830	2028	1620	2029	1830
2030	1830	2031	1830	2032	1830	2033	1870	2034	1310
2035	1030	2036	1870	2037	1310	2038	1830	2039	1870
2040	1310	2041	1830	2042	1870	2043	1310	2044	1830
2045	1870	2046	1300	2047	1310	2048	1830	2049	1620
2050	1620	2051	1620						

Figure 6 (e). (Cont)

DIGITAL DESIGN LANGUAGE SYNTHESIZER
OVERALL CONNECTION INFORMATION

CONNECTION LIST BY IDENTIFIER

	IDENTIFIER	OUTPUT	INPUT
1	X(1)	MCMP	DRIV
2	X(2)	MCMP	DRIV
3	X(3)	MCMP	DRIV
4	ADD(1)	DRIV	MCMP
5	ADD(2)	DRIV	MCMP
6	ADD(3)	DRIV	MCMP

CONNECTION LIST BY CELL

	DRIVER		DRIVEN CELLS							
	CELL	PIN	CELL	PIN	CELL	PIN	CELL	PIN	CELL	PIN
1	2049	4	1000	3	1002	2				
2	2050	4	1001	3	1003	2				
3	2051	4	1001	2	1003	3	1004	2		
4	1002	4	2024	2						
5	1003	4	2026	2						
6	1004	3	2028	2						

Figure 6 (f). Connection Information for Module 1 and Module 2

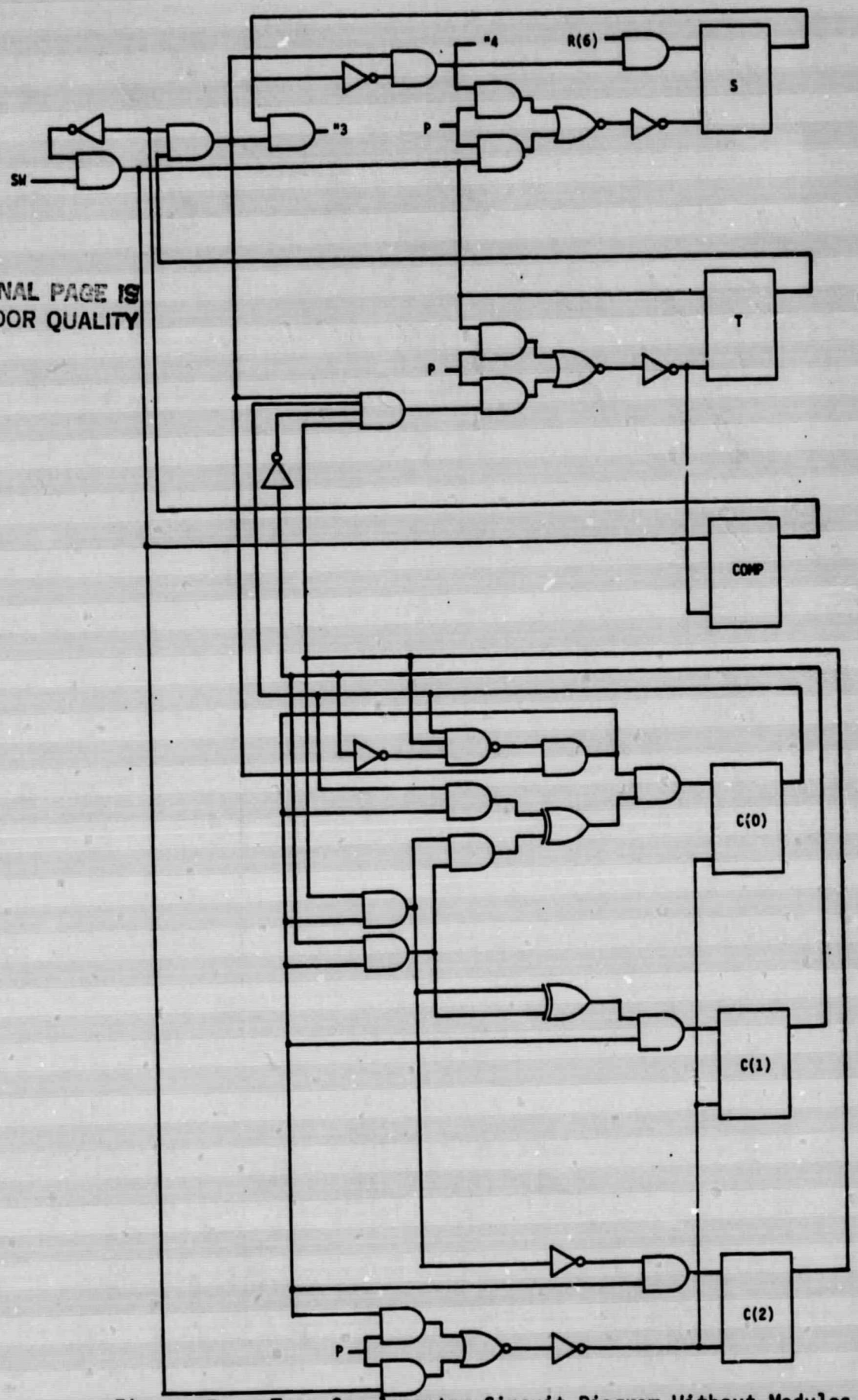


Figure 7 Twos Complementer Circuit Diagram Without Modules

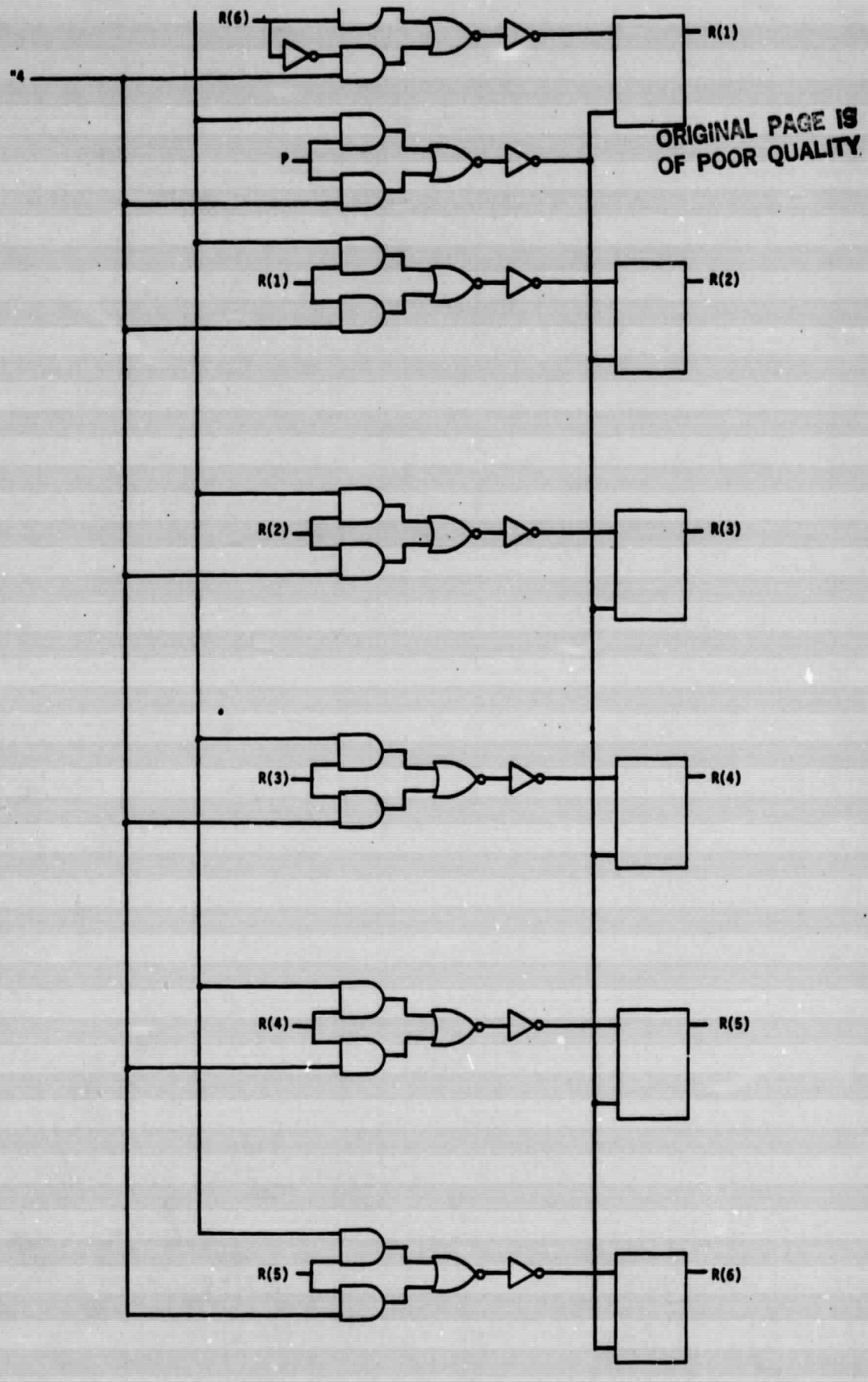


Figure 7 (Cont)

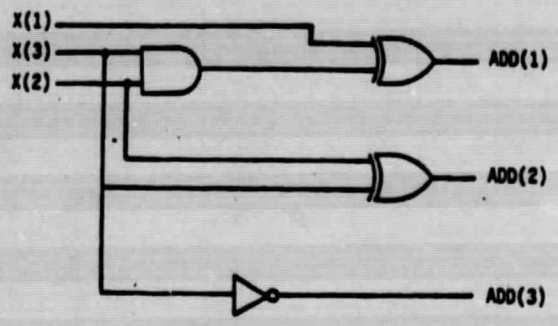


Figure 8 Twos Complementer Circuit Diagram With Modules

Module 1

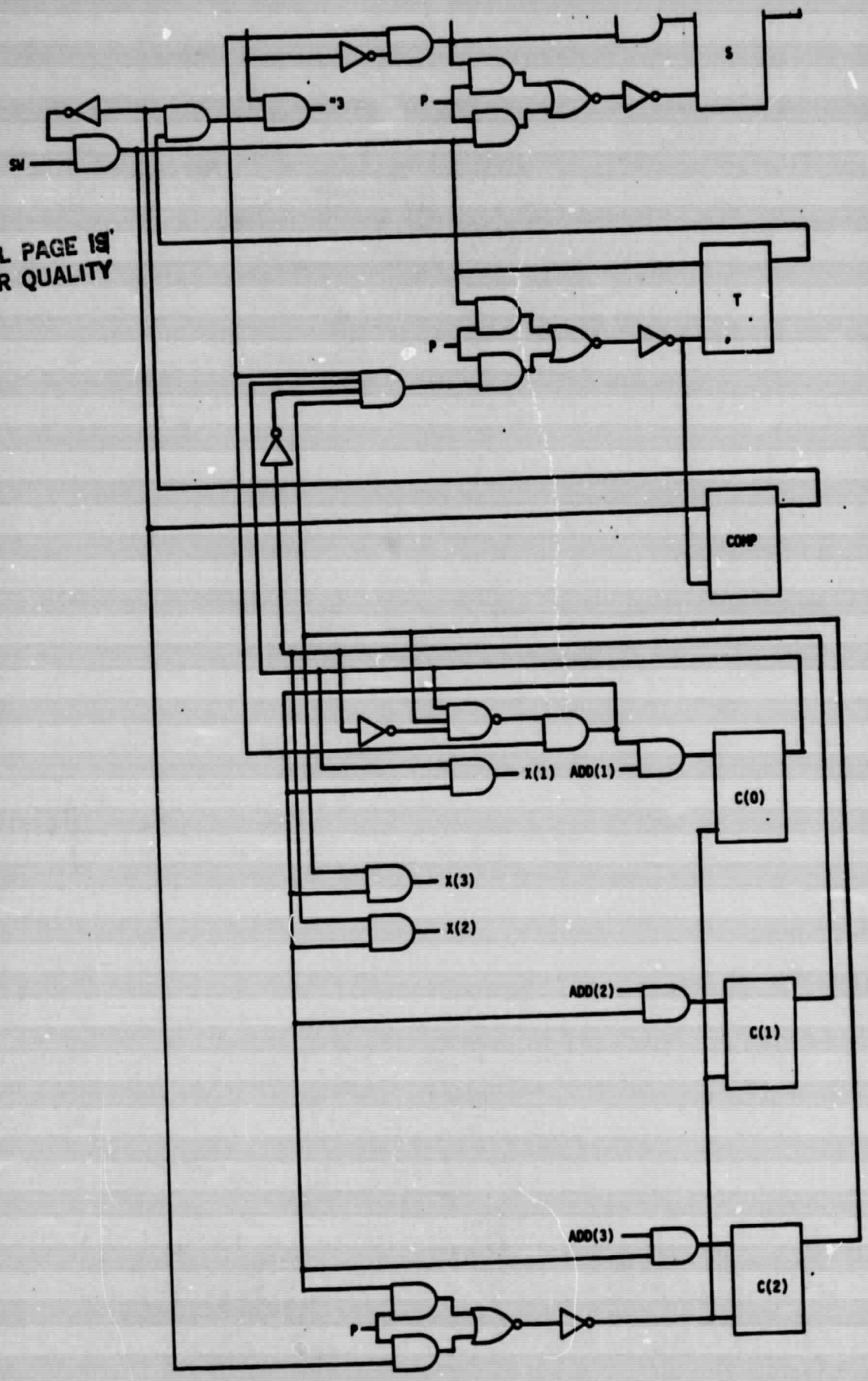


Figure 8 (Continued) Module 2

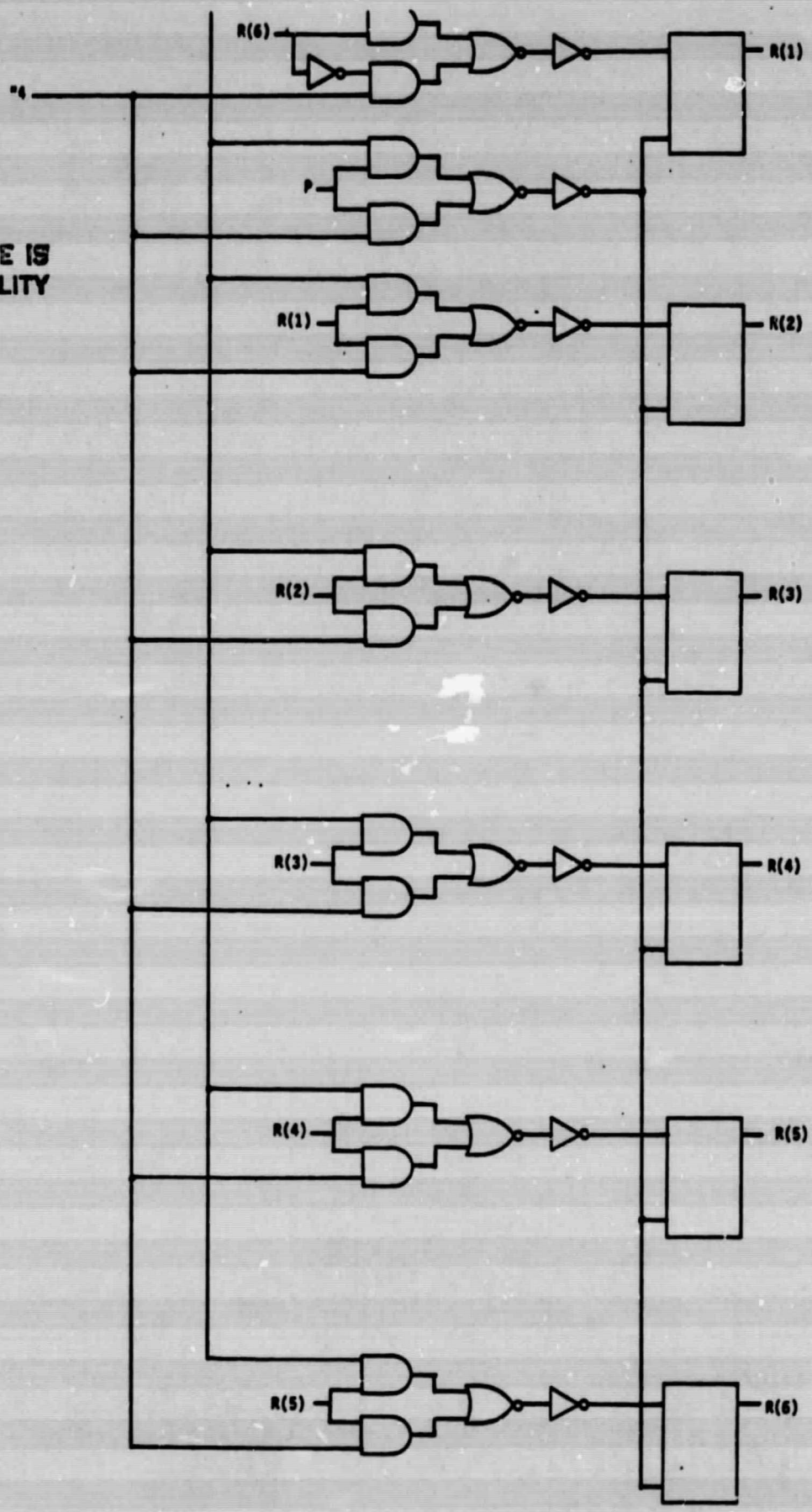


Figure 8 Module 2 (Cont)

```
DIGITAL DESIGN LANGUAGE SIMULATOR

        1:      <FL>4.6
        2:      <IN>SW/1
        3:      <RE>T/R/5,20
        4:      <TR>OUTTR/TP/
        5:      <OU>OUTTR/COMP,R,S,C,T/,T/R/
        6:      <SI>
```

Figure 9 (a). Simulation Input Commands

```
DIGITAL DESIGN LANGUAGE SIMULATOR

          C
          O
          M
TIME      P    R      S   C   T    R

   0      0  000000   0  000  0  000000
   2      1  000000   0  000  1
   4      1  100010   1  001  1
   6      1  110001   1  010  1
   8      1  011000   1  011  1
  10      1  101100   1  100  1
  12      1  110110   1  101  1
  14      0  111011   1  101  0  111011
  16      1  111011   0  000  1
  18      1  001010   0  001  1
  20      1  000101   0  010  1
  22      1  100010   1  011  1
  24      1  110001   1  100  1
  26      1  011000   1  101  1
  28      0  101100   1  101  0  101100
  30      1  101100   0  000  1

END     OF FILE REACHED ON INPUT
SIMULATION TERMINATED AT TIME =        31
```

Figure 9 (b). Simulation Output

5. PLA SYNTHESIS [14]

This chapter describes an algorithm and realizing program PLASYN that show the feasibility of automatically generating PLA realizations of much of the combinational logic of a system described in DDL. In brief, the description is translated to a set of Boolean equations and register transfer statements. Then the equations to be realized with PLAs are determined, and all other equations and register transfers are published for manual design. The equation set is partitioned to small subsets of equations that can each be realized with the PLAs to be used. Finally, a PLA program for each sub set of equations is reduced and published. PLASYN was developed at the University of Wisconsin [14] and is now implemented on SEL-32 at NASA-MSFC.

5.1 SYSTEM MODEL

Figure 10 shows the digital system model assumed by PLASYN. The PLAs are considered to provide AND array to OR array locic only. The Signetics 825100/101 16 input variable, 8 output variable and 48 product term devices are the sort of technology assumed, but PLA parameters are not fixed to these particular values. The following parameters charecterize the PLAs:

λ - PLA input limit

μ - PLA output limit

ν - PLA product term limit

Boolean terms that are naturally realized by high fan-in gates may be realized with PLAs, but they consume a great number of internal

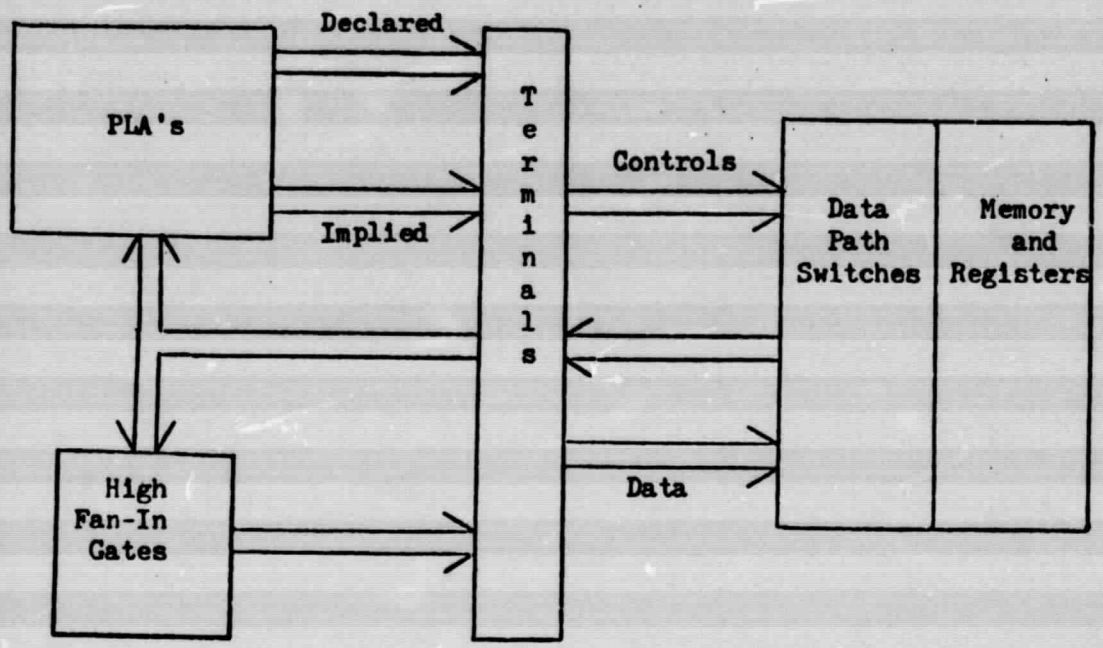


Figure 10. Digital System Model used by PLASYN.

AND gates, and hence some terms involving the DDL reduction operator are placed in a separate category for manual realization. The terminals of Figure 10 are those wires in a system (1) declared to be terminals by the author of a description, (2) essential control signals implied in a description, (3) memory and register output leads and (4) high fan-in gate leads. All but terminals of (2) are "primary input variables" to PLAs. Those variables of the equation set that are not terminals under this definition are "secondary variables." While declared terminals could often be treated as secondary variables to reduce the number of PLAs needed to realize a system, no attempt is made to guess which declared terminals are significant test points and which need not be physically realized.

Finally, the multiplexing of data paths preceding the flip-flops of registers is not realized with PLAs because we believe designers prefer to use MSI and LSI devices intended for this purpose or bus techniques.

5.2 TRANSLATION AND SYNTHESIS

PLASYN uses the output from DDLTRN as the input information for synthesis. The synthesis process is illustrated below with an example system.

Figure 11provides the DDL description of an 8-bit magnitude multiplier. The multiplicand resides in the R register; the multiplier is in the B register initially. The familiar selective add then shift algorithm is used with partial products accumulated in the A and B registers. Equations for COUT and SUM provide a ripple adder for forming partial products. Equations for CCOUT and CSUM provide the "add 1" logic to form register MCOUNT into an iteration counter.

```
DIGITAL DESIGN LANGUAGE TRANSLATOR
   1:   <CO> DESIGN OF A 8-BIT MULTIPLIER.
   2:   <SY> MULTPY : <TI> P. <RE> A(0:8), B(8),R(8),MCOUNT(3).
   3:                 <IE> START, BUS(8), DONE.
   4:                 <IE> SUM(8), COUT(8), CSUM(3), CCOUT(3).
   5:                 <ID> CIN=COUT(2:8)[0D1.
   6:                 <ID> CCIN=CCOUT(2:3)[1D1.
   7:                 <BO> COUT=R*A(1:8) + R*CIN + A(1:8)*CIN,
   8:                      SUM=R@A(1:8)@CIN,
   9:                      CCOUT=MCOUNT*CCIN,
  10:                      CSUM=MCOUNT@CCIN.
  11:                 <AU> MPY(2) :P:
  12:                 <SI> S1(0):START: R<-BUS, MCOUNT<-0, ->S2.
  13:                      S2(1): B<-BUS, A<-0, ->S3.
  14:                      S3(2): ]B(8)] A<-COUT(1)[SUM.,->S4.
  15:                      S4(3): A(1:8)[B<-A[B(1:7), A(0)<-0,
  16:                             MCOUNT<-CSUM, ]*/PCOUNT]DONE=1,->S1;->S3.....
```

Figure 11: A 8 Bit Multiplier

Figure 12 shows the multiplier after processing by DDLTRN. The first four equations in Fig. 12 provide the state decoder on automaton register MPY. Internally generated variables are identified with names "integer. Fourteen appear in Figure 12. Five of these, "5 through "9, provide control on register transfers. "11 through "14 provide right sides of transfers to single flip-flops. The other internal equations may be thought of as describing a multiple level hardware control of the example system. Note that most constants (carries into the adders, clearing registers, state assignments) have been eliminated in Fig. 12 by simplifying equations appropriately. The exclusive-OR of MCOUNT(3) and 1 appears in the CSUM(3) equation and an exclusive-OR with 0 appears in the SUM(8) equation.

Before partitioning equations to be realized, program PLASYN publishes the equations and transfers with which it will not be concerned. Figure 13 reveals that one high fan-in gate will not be realized for the example system. An AND reduction with fan-in of 3 is involved. It would not be unreasonable to extend DDLSYN to accept such reductions. Two identities were found in the equation set; clearly they do not require further synthesis. The example system did not involve a memory; eight register transfers are listed for synthesis by other means.

Twenty-four equations of Fig. 12 remain to be considered. Four of these equations have dimension greater than 1; the total number of variables of concern is therefore 38. Internal variables S2, S4, "1, "2, "5-"9 and "11-"14 appear on the right of published register transfers or as conditions on those transfers. Variables S1, S3, "3 and "4 do not so appear and need not be realized explicitly. They are secondary variables. Thus only 34 variables must be realized. This set of variables is identified by PLASYN.

```
DIGITAL DESIGN LANGUAGE TRANSLATOR
        PASS1--FACILITIES IDENTIFIED


DECLARED FACILITIES

      <SY> MULTPY
      <TI> P(1:1)
      <RE> A(0:8)
           B(1:8)
           R(1:8)
           MCOUNT(1:3)
      <TE> START(1:1)
           BUS(1:8)
           DONE(1:1)
           SUM(1:8)
           CCUT(1:8)
           CSUM(1:3)
           CCOUT(1:3)
      <ID> CIN(1:1)
           CCIN(1:1)
           <AU> MPY
           <ST> S1
                S2
                S3
                S4
```

Figure 12: DDLTRN Output For 8 Bit Multiplier

```
    DECLARED OPERATIONS

<SY> MULTPY:
     <BO> COUT=R*A(1: 8) + R*CIN + A(1: 8)*CIN, SUM=R@A(1: 8)@CIN, CCOUT=MCCUNT*CCIN, CSUM=MCCUNT@CCI

     <AU> MPY: P:
          <ST>
               S1: START: R<-BUS, MCOUNT<-0, ->S2.

               S2: B<-BUS, A<-0, ->S3.

               S3:
                    ]B(8)] A<-CCUT(1)[SUM., ->S4.

               S4: A(1: 8)[B<-A[B(1: 7), A(0)<-0, MCCUNT<-CSUM,
                    ]*/MCOUNT] DONE=1, ->S1; ->S3.....
```

Figure 12: (Continued)

```
DIGITAL DESIGN LANGUAGE TRANSLATOR
     PASS7--SIMPLIFICATION


<SY> MULTPY: S1=↑MPY(1)*↑MPY(2),
     S2=↑MPY(1)*MPY(2),
     S3=MPY(1)*↑MPY(2),
     S4=MPY(1)*MPY(2),
     "1=S1*START,
     "2=S3*B(8),
     "3=S4*"10,
     "4=S4*↑("10),
     "5=P*"1,
     "6=P*"1 + P*S4,
     "7=P*S3 + P*"3 + P*"1 + P*S2 + P*"4,
     "8=P*S2 + P*S4,
     "9=P*S2 + P*"2 + P*S4,
     "10=*/MCOUNT,
     "11=S3 + "1,
     "12=S3 + S2 + "4,
     "13=S2*BUS(1) + S4*A(8),
     "14="2*COUT(1),
     COUT(1:7)=(R(1:7)*A(1:7) + R(1:7)*COUT(2:8) + A(1:7)*COUT(2:8)),
     COUT(8)=(P(6)*A(8)),
     SUM(1:7)=(R(1:7)@A(1:7)@COUT(2:8)),
     SUM(8)=(R(8)@A(8)@0D1),
     CCOUT(1:2)=MCOUNT(1:2)*CCOUT(2:3),
     CCOUT(3)=MCOUNT(3),
     CSUM(1:2)=(MCOUNT(1:2)@CCOUT(2:3)),
     CSUM(3)=(MCOUNT(3)@1D1),
     ]"5] R<-"1*BUS.,
     ]"6] MCOUNT<-S4*CSUM.,
     ]"7] MPY(1)<-"12.,
     ]"7] MPY(2)<-"11.,
     ]"9] A(0)<-"14.,
     ]"9] A(1:8)<-"2*SUM + S4*A(0:7).,
     ]"8] B(1)<-"13.,
     ]"8] B(2:8)<-S2*BUS(2:8) + S4*B(1:7).,
     DONE="3,
     •
```

Figure 12 : (Continued)

```
    DIGITAL DESIGN LANGUAGE SYNTHESIZER

FOLLOWING EQUATIONS ARE NOT REALIZED IN PLAS:

  (1)  HIGH FAN IN GATES

     "10=*/MCOUNT,


  (2)  TERMINALS

     DONE="3,
     CCOUT(3)=MCOUNT(3),


  (3)  MEMORY

     NONE


  (4)  REGISTER TRANSFERS

     ]"5] R<-"1*BUS.,
     ]"6] MCOUNT<-S4*CSUM.,
     ]"7] MPY(2)<-"11.,
     ]"7] MPY(1)<-"12.,
     ]"8] B(2:8)<-S2*BUS(2:8) + S4*B(1:7).,
     ]"8] B(1)<-"13.,
     ]"9] A(1:8)<-"2*SUM + S4*A(0:7).,
     ]"9] A(0)<-"14.,
```

Figure 13 : PLASYN Output For 8 Bit Multiplier

DIGITAL DESIGN LANGUAGE SYNTHESIZER

PROGRAMMING CODE FOR

PLA 1

COLUMN	NAME
1	P(1)
2	MPY(1)
3	MPY(2)
4	"3(1)
5	"1(1)
6	"4(1)
7	"7(1)
8	"6(1)
9	"12(1)
10	"11(1)
11	"8(1)
12	"5(1)
13	S4(1)
14	S2(1)

```
101XXX 1-1-1--1
X01XXX --1----1
111XXX -1--1-1-
X11XXX ------1-
1XXX1X 11-1-1--
110XXX 1-11----
X10XXX --11----
XXXX1X ---1----
1XXXX1 1-1-----
XXXXX1 --1-----
1XX1XX 1-------
```

Figure 13: (Cont.)

```
DIGITAL DESIGN LANGUAGE SYNTHESIZER

 PROGRAMMING CODE FOR

      PLA    2

   COLUMN       NAME
-------------------------
       1          MPY(  1)
       2          MPY(  2)
       3          BUS(  1)
       4            A(  8)
       5          "10(  1)
       6            B(  8)
       7        START(  1)
       8            R(  8)
       9            0   D1
      10            P(  1)
      11          "2(  1)


      12          "13(  1)
      13           "4(  1)
      14           "3(  1)
      15           "2(  1)
      16           "1(  1)
      17         COUT(  8)
      18          SUM(  8)
      19           "9(  1)
-------------------------


X1XXXXXXX1X -------1
XXXXXXXXX11 -------1
XXX1XXX11XX -----11-
XXX0XXX01XX ------1-
XXX1XXX00XX ------1-
XXX0XXX10XX ------1-
XXX1XXX1XXX -----1--
00XXXX1XXXX ----1---
10XXX1XXXXX ---1----
11XX1XXXXXX --1-----
11XX0XXXXXX -1------
11X1XXXXXXX 1-------
011XXXXXXXX 1-------
```

Figure 13: (Cont.)

DIGITAL DESIGN LANGUAGE SYNTHESIZER

PROGRAMMING CODE FOR

PLA 3

COLUMN	NAME
1	R(7)
2	A(7)
3	COUT(8)
4	*2(1)
5	COUT(1)
6	MCOUNT(2)
7	CCOUT(3)
8	MCOUNT(1)
9	CCOUT(2)
10	MCOUNT(3)
11	1 01
12	SUM(7)
13	COUT(7)
14	*14(1)
15	CCOUT(2)
16	CSUM(2)
17	CCOUT(1)
18	CSUM(1)
19	CSUM(3)

```
XXXXXXXXX01 -------1
XXXXXXXXX10 -------1
XXXXXXX01XX ------1-
XXXXXXX10XX ------1-
XXXXXXX11XX -----1--
XXXXX01XXXX ----1---
XXXXX10XXXX ----1---
XXXXX11XXXX ---1----
XXX11XXXXXX --1-----
111XXXXXXXX 11------
11XXXXXXXXX -1------
1X1XXXXXXXX -1------
X11XXXXXXXX -1------
100XXXXXXXX 1-------
010XXXXXXXX 1-------
001XXXXXXXX 1-------
```

Figure 13: (Cont.)

PROGRAMMING CODE FOR

PLA 4

COLUMN	NAME
1	R(6)
2	A(6)
3	COUT(7)
4	R(5)
5	A(5)
6	COUT(6)
7	R(4)
8	A(4)
9	COUT(5)
10	R(3)
11	A(3)
12	COUT(4)
13	SUM(6)
14	COUT(6)
15	SUM(5)
16	COUT(5)
17	SUM(4)
18	COUT(4)
19	SUM(3)
20	COUT(3)

```
XXXXXXXX111 ------11
XXXXXXXX11X -------1
XXXXXXXX1X1 -------1
XXXXXXXXX11 -------1
XXXXXXXX001 ------1-
XXXXXXXX010 ------1-
XXXXXXXX100 ------1-
XXXXXX111XXX ----11--
XXXXXX11XXXX ----1--
XXXXXX1X1XXX ----1--
XXXXXXX11XXX ----1--
XXXXXX001XXX ----1---
XXXXXX010XXX ----1---
XXXXXX100XXX ----1---
XXX111XXXXXX --11----
XXX11XXXXXXX ---1----
XXX1X1XXXXXX ---1----
XXX11XXXXXXX ---1----
XXX001XXXXXX --1-----
XXX010XXXXXX --1-----
XXX100XXXXXX --1-----
111XXXXXXXXX 11------
11XXXXXXXXXX -1------
1X1XXXXXXXXX -1------
X11XXXXXXXXX -1------
100XXXXXXXXX 1-------
010XXXXXXXXX 1-------
001XXXXXXXXX 1-------
```

Figure 13: (Cont.)

PROGRAMMING CODE FOR

PLA 5

COLUMN	NAME
1	R(2)
2	A(2)
3	COUT(3)
4	R(1)
5	A(1)
6	COUT(2)
7	SUM(2)
8	COUT(2)
9	SUM(1)
10	COUT(1)

```
XXX111 --11
XXX11X ---1
XXX1X1 ---1
XXXX11 ---1
XXX001 --1-
XXX010 --1-
XXX100 --1-
111XXX 11--
11XXXX -1--
1X1XXX -1--
X11XXX -1--
100XXX 1---
010XXX 1---
001XXX 1---
```

Figure 13: (Cont.)

All right sides of all equations are converted to reverse Polish strings. This form facilitates back substitution to eliminate secondary variables and find the primary input variables of each terminal variable. We use infix notation here to find the primary input variables. For the example system

$$"7 = P*S3 + P*"3 + P*"1 + P*S2 + P*"4$$

$$S3 = MPY_1*\overline{MPY}_2$$

$$"3 = S4*"10$$

$$S4 = MPY_1*MPY_2$$

$$"1 = \overline{MPY}_1*\overline{MPY}_2*START$$

$$S2 = \overline{MPY}_1*MPY_2$$

$$"4 = S4*\overline{"10}$$

$$S4 = MPY_1*MPY_2$$

$$\therefore \text{Input set of } "7 = \{P, MPY_1, MPY_2, "10, START\}$$

Primary input variable sets are formed and stored in DDLSYN using the cube notation and operators of [5, Chapter 9 and Appendix 9.1]. In essence, a binary vector is formed for each equation with a position for each possible primary input variable. A 1 is used to indicate membership in the input set for the equation.

5.3 PARTITIONING

Let S be the set of equations E_i to be realized.

$$S = \{E_1, E_2, \ldots\}$$

The set of primary input variables for equation E_i is denoted E^i. Similarly partition block $P_j \subseteq S$ has input variable set P^j which is the union of all E^i for $E_i \in P_j$. We seek the minimum n such that:

$$\bigcup_{i=1}^{n} P_i = S$$

$$P_i \cap P_j = \phi \quad \text{for } i \neq j$$

$$|P_i| \leq \mu \quad \text{for } 1 \leq i \leq n$$

$$|P^i| \leq \lambda \quad \text{for } 1 \leq i \leq n$$

Where $|x|$ denote "size of set x". It is also necessary to be able to express the equations of a partition block with no more than ν product terms. This condition is ignored in the following partitioning algorithm and has not been violated in the example systems synthesized to date.

Partitioning Algorithm:

Step 1. Initialize i := 0 and S := $\{E_1, E_2 \ldots\}$

Step 2. Find an equation $E_j \in S$ for which $|E^j|$ is maximum.

$i := i + 1$

$P_i := \{E_j\}$

$P^i := E^j$

$S := S - E_j$

Step 3. Seek an equation $E_k \in S$ with $E^k \subseteq P^i$, and maximum $|E^k|$.

If none exists go to step 5.

Step 4. $P_i := P_i \cup E_k$

$P^i := P^i \cup E^k$

$S := S - E_k$

If $|P_i| < \mu$ go to step 3.

Otherwise, go to step 2.

Step 5. Seek an equation $E_k \in S$ for which $|P^i \cup E^k|$ is minimum and less than or equal to λ, and $|E^k|$ is maximum.

If an E_k exists, go to step 4.

Otherwise, go to step 2.

On the example system and equation set, "7 is selected as the seed equation of the first partition block since it has the largest input set.

$$|E^{"7}| = 5$$

$$E^{"7} = \{P, MPY_1, MPY_2, "10, START\}$$

The input set of "6 is a maximum subset of this set.

$$E^{"6} = \{P, MPY_1, MPY_2, START\}$$

Variable "5 has the same input set and hence is picked as the third member of P_1. A summary of the partitioning of the example system is presented later.

This algorithm fails if the input set of an equation has more than λ members. Such an equation cannot be realized with the 2-level logic of the available PLA. While it may be possible to realize it in terms of secondary variables, a simple algorithm for arriving at more suitable intermediate variables has been developed, but not programmed and included in PLASYN. While this algorithm is best implemented using the "cube" operators of[15], it is stated here in terms of sets using similar notation to that used to present the partitioning algorithm. This algorithm should be executed while finding the input sets of equations, i.e. before partitioning.

Input Set Partitioning Algorithm:

If $|E^i| > \lambda$:

Step 1. Express E_i in sum-of-products form with a reduced if not minimum number of product terms π_j.

$$E_i = \pi_1 \vee \pi_2 \vee \ldots$$

We will treat E_i as a set with members π_1, π_2, ... in the following steps. The set of primary input variables appearing in π_j is denoted π^j.

k := 1

Step 2. If $|E^i| \leq \lambda$, replace the right side of the original equation E_i with the sum of product terms in set E_i and exit. Otherwise, seek $\pi_j \notin_i$ for which $|\pi^j|$ is minimum. If $|\pi^j| > \lambda$, then a factoring algorithm such as[15, algorithm 11.6] must be used. Otherwise create an empty set A_k. (The input set of A_k is denoted A^k.)

Step 3. $A_k := A_k \cup \pi_j$

$A^k := A^k \cup \pi^j$

$E_i := E_i - \pi_j$

Step 4. Seek a $\pi_j \in E_i$ for which $|A^i \cup \pi^j|$ is minimum and less than or equal to λ.

If π_j exists, go to step 3

Otherwise, A_k provides a new terminal.

Form and enter into the data base a new variable (denoted v_k here) and equation:

$$v_k = a_1 \vee a_2 \vee \ldots$$

where all $a_i \in A_k$.

$E_i := E_i \cup v_k$

Go to step 2.

This algorithm is not needed in the example system of this paper, but was found to be efficacious in other system

5.4 PLA PROGRAM FORMATION

The technique used in PLASYN to form a program table for each PLA created by the partitioning algorithm is summarized below:

1. An ON-array is formed for each equation of a partition block using an extension of the algorithm of[15, Sec. 9.6] to eliminate secondary variables.

2. As each ON-array is completed, it is merged with previous ON-arrays to an approximate connection array that provides all of the information necessary to program a PLA. A product term appears once in this connection array, even if it is a member of several ON-arrays.

3. All logically valid AND-to-OR connections are formed and recorded in the connection array.

4. Redundant AND-to-OR connections are eliminated in an order that enhances the removal of all connections to an AND gate and hence its elimination. Certainly true AND gate minimization is not guaranteed, but compute time and memory requirements are modest.

Figure 13presents the PLASYN results for the first PLA of the example system. Neither the PLA input or product term limits are approached, but the PLA is "full" in the sense that all output terminals are utilized. Table 1 summarizes DDLSYN results for the example system. With $\mu = 8$, 34 equations may not be realized with fewer than 5 PLAs, the number listed in Table 2.

Table 3 summarizes results for a system of 117 equations. Again using $\mu = 8$, no fewer than 15 PLAs may be used. This minimum number was not attained by DDLSYN, because of $\lambda = 16$. PLAs 7 through 10 are input limited: they bit-slice multiplexers that drive adder-like networks. No partition of this equation set with fewer than 18 blocks has been found by manual means with $\lambda = 16$ and $\mu = 8$.

Table 3. Summary of PLASYN Realization of the Example Multiplier.

PLA	Input Set Size	Output Set Size	Product Term Set Size
1	5	8	9
2	11	8	13
3	12	8	21
4	12	8	28
5	3	2	7

Table 4. Summary of PLASYN Realization for a Larger Digital System.

PLA	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Input Set Size	18	15	13	13	12	15	15	15	15	15	16	10	10	14	12	12	15	8
Output Set Size	8	8	8	8	8	5	4	4	4	4	5	8	8	8	8	8	8	3
Product Terms	20	33	18	16	16	11	14	14	14	14	15	19	19	26	28	28	22	8

5.5 SUMMARY

The automatic synthesis of much combinational logic of a digital system described in a register transfer language is feasible and cost effective. DDL, DDLTRN and PLASYN are not necessarily optimum for practicing designers, however. DDL does not currently provide a means for the designer to distinguish terminals that must be realized and those that may be treated as secondary. DDLTRN has very weak syntax checking at the moment. Improvements to PLASYN are also possible. All reduction-selection terms could be realized with PLAs. Total removal of constants via equation simplification has been programmed; only additional memory is required. Factoring register transfer expressions would reduce the size and hence cost of data path switches. Then:

```
|"1|A← "2*B + "3*C + "4*C
```

would be realized:

```
"5 = "3 + "4

|"1| A← "2*B + "5*C
```

The elimination of equivalent logic generated from nonidentical Boolean expressions is possible. Finally, semiconductor manufacturers are now providing programmable multiplexers, PLAs with registers and a variety of PLAs with and without registers. A synthesizer that recognized such components could totally automate digital system synthesis.

6. LOGIC MINIMIZATION

The BEs and RTEs generated by DDLTRN are not minimized. Some simplification is performed during PASS7 by combining identical conditions on RTEs, by gathering identical right hand sides of BEs into a single occurrence and eliminating constants from the equations under the rules of Boolean Algebra. PASS7 looks for syntatic equivalence between equations rather than the functional equivalence. As such, it is possible to have two or more equations of different syntatic structures realizing the same logic function. Hence, logic minimization is required before entering the synthesis phase.

DDLSYN synthesizes one equation at a time. Further, it treats an RTE to be equivalent to 3 BEs to be synthesized. (i.e., the condition, the source expression and the destination expression). Hence, the following discussion on minimization does not distinguish between BEs and RTEs.

A multiple-output minimization program [15] (MOMIN) minimizes the equations generated by DDLTRN. Calling on MOMIN during the design cycle is an optional feature. Since MOMIN leaves the format of the DDLTRN output files unchanged, both DDLSYN and PLASYN can utilize the minimized set of BEs and RTEs for synthesis.

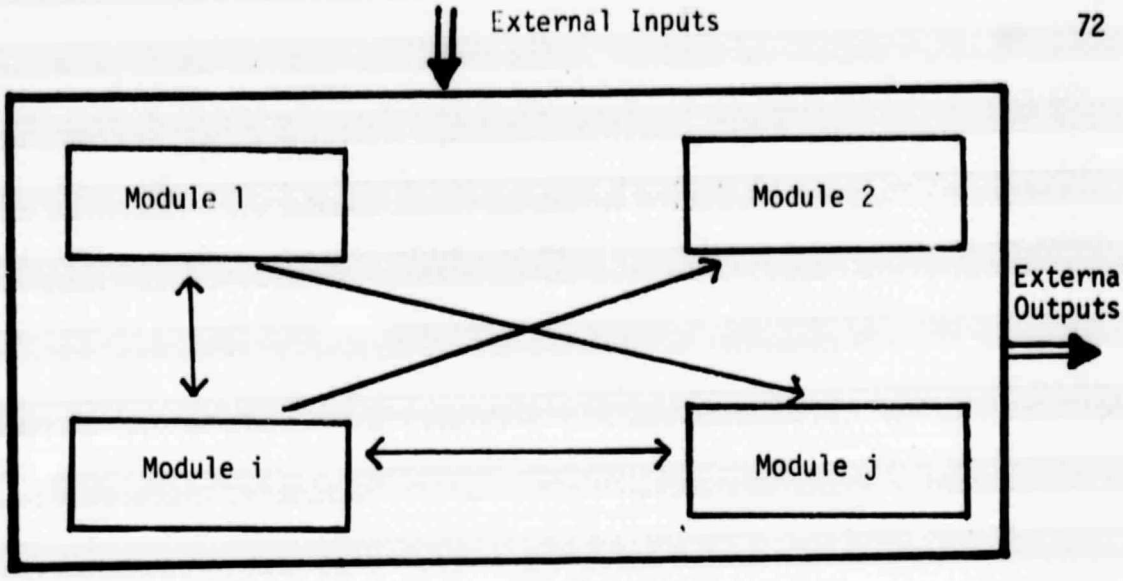
The memory requirements for the execution of MOMIN grow rapidly with the growth of the number of variables (input and output) involved in the set of BEs to be minimized. Hence the number of input variables is limited to n and the number of equations in the system is limited to m. (n + m) is now set at 16.

The logic minimization interface ensures that the order of each BE is less than or equal to n and partitions the equations in the DDLTRN output into partitions of m equations or less to satisfy the (n + m) $\leq$ 16 constraint. This interface also converts the equations from the linked list structure of the DDLTRN output into the cubic structure needed for MOMIN and reconverts them into the linked list format for DDLSYN processing. Figure 14 shows the logic synthesis model. If a non-modular synthesis mode is used, figure 14(a) will have just one module. Each nxm partition is minimized by MOMIN. If enough memory is available, n and m can be made large enough to include the complete set of equations in the DDLTRN output in a single partition.

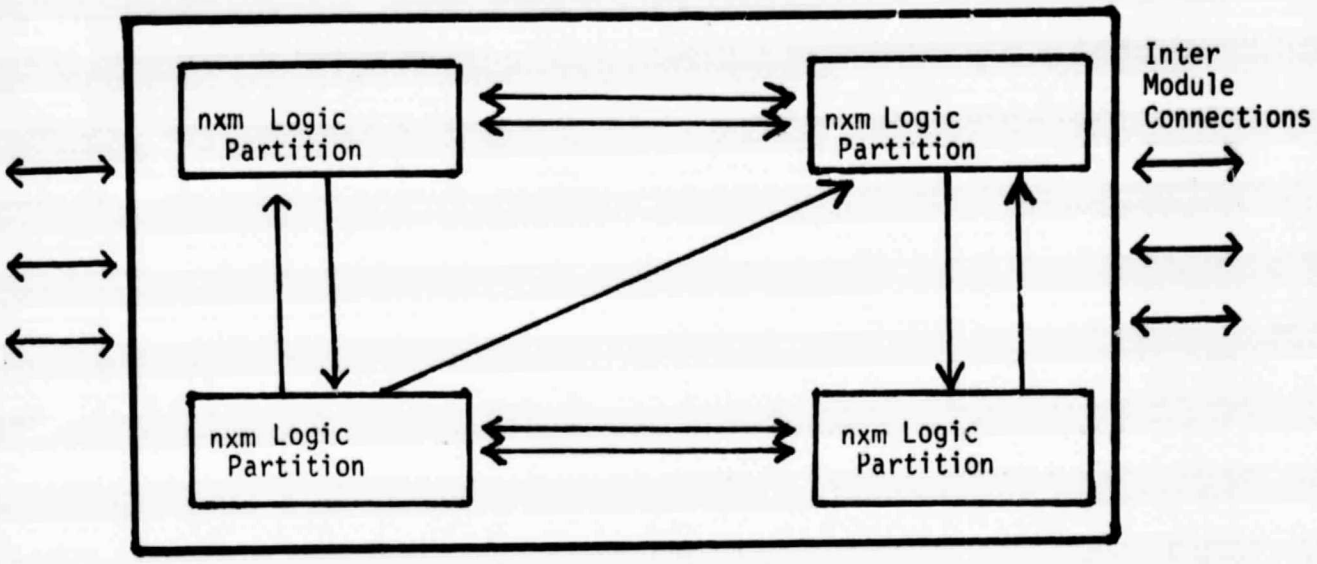
Sections 6.1 and 6.2 provide details of two other algorithms used in partitioning. Section 6.3 discusses the minimization theory along with an example. The implementation details are given in the Programmer's Manual.

6.1 SPLITTING AN EQUATION WITH LARGE NUMBER OF VARIABLES

To achieve the limit n, a function with a larger number of variables could be split into two or more subfunctions and each subfunction is minimized individually. These minimized subfunctions can be ORed to obtain the orginal function for synthesis. The 6 variable function for "7 from Figure 12 can be split into two subfunctions as shown below:

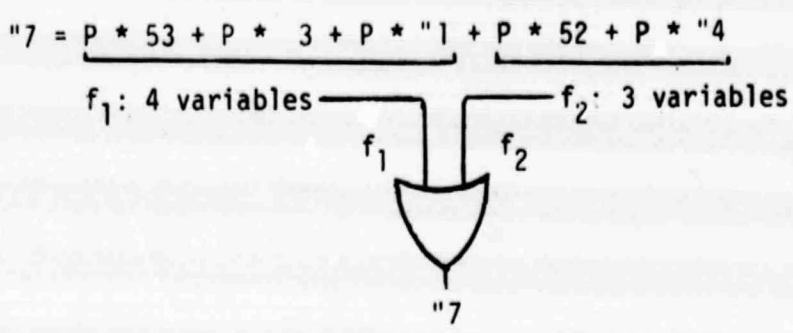


(a) System



(b) Module

Figure 14: Logic Synthesis hierarchy



The product terms to be included in each subfunction can be picked scanning the function left to right counting the number of variables, till the limit is reached. An algorithm that tends to select as many product terms of the BE as possible still keeping the number of variables in each subfunction within the limit is described below:

Algorithm: To partition a BE into subexpressions of lesser order.

let n = Limit on the number of variables (i.e. order) in the subexpression.
P_i (i = 1 to j) are the product terms of the original BE.
SE_k is the K^{th} subexpression.
V_k is the set of variables in SE_k.
S is the set of variables in BE.
V_i is the set of variables in P_i.
|x| denotes the number of elements in set x.

Step 1: If $|S| \leq n$. (no partitioning is needed) stop; else, K=1, go to step 2.

Step 2: $V_k = 0$, $SE_k = 0$, If j = 0, stop else go to step 3.

Step 3: Search for a P_i (i = 1 to j) such that $|V_i|$ is a maximum; Go to Step 4.

Step 4: If $|V_k| + |V_i| > n$, K = K + 1, go to step 2 else go to step 5.

Step 5: $V_k = V_k \cup V_i$, $SE_k = SE_k \cup P_i$, BE = BE - P_i, j=j-1,
If j=0, Stop ELSE go to Step 6.

Step 6: Compare P_i (i = 1 to j) with V_k to select a P_i Such that V_i has the most matching variables with V_k go to step 4.

This algorithm partitions the BE into k subexpressions each of order less than or equal to n. Each SE is minimized individually and combined to form:

$$BE = \bigcup_{i=1,k} SE_i.$$

The algorithm assumes that each of the product terms in the BE has less than n literals.

6.2 SUBSTITUTION TO ELIMINATE VARIABLES AND BEs

The variable names used in the DDL description by the designer are Primary Variables. The BEs corresponding to these variables are to be realized explicitly. DDLTRN generates Secondary Variables. These variables are identified with "integer in DDLTRN output. Some of these secondary variables are used either as conditions or on the right hand sides of RTEs. The BEs for such secondary variables need also to be realized explicitly. Any secondary or a primary variable that is not used as above, can be expanded in terms of the other primary variables and thus need not be realized explicitly. In figure 12, variables S1, S3, "3 and "4 do not appear either as conditions or on the RHS of any RTE. Hence, they can be replaced by the other variables. For example, "7 can be expanded as following:

6.3 MULTIPLE OUPUT MINIMIZATION [15]

This section contains a detailed description of minimization of multiple output switching functions. Minimization is the process of obtaining that expression of a switching function which is the cost of contructing the network specified by the available switching functions.

The switching functions are specified in the form of ON and DC arrays. Definitions of the terms and operators used in the algorithm are given in section 6.3.1. A brief description of the algorithm is given in Section 6.3.2. The use of the minimization algorithm is illustrated by means of an example in Section 6.3.3.

Details on the programming considerations are found in the programmer's manual.

6.3.1. DEFINITIONS

The terms and operator used in the algorithm are defined in this section. Examples to illustrate the definition are given.

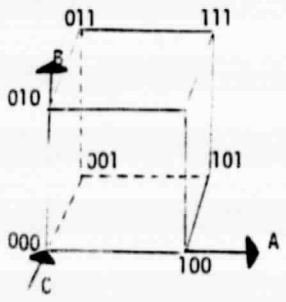
Switching Functions

A switching function of n input Variables x_1, x_2, ..., x_n is a rule that associates every n tuple of these valued variables with a m tuple of similar valued output variables z_1, z_2, ..., z_m.

The tuples are equivalent to the product terms of a boolean equation.

Example 1:

Consider a switching function F=AB+ABC. Here n=3; m=1. The cube representation of F is as follows:



Cube

A o-cube is defined as that cube of a switching function whose vertices are specified by combinations of o's and 1's only. If one X is included in the the combinations then the cube represented is a 1-cube.

The function F in Example 1 has one o-cube (ABC or 101) and one 1-cube (AB or 11X).

Cover Relation

As already mentioned, X can be either 0 or 1. Cube 11X can represent either cube 110 or cube 111, i.e., 11X 'cover' 110 and 111. In other words 110 and 111 are 'included in' 11X. The cover relation in represented as 110 $\sqsubset$ 11X or 111 $\sqsubset$ 11X.

Prime Implicants

The cubes of a switching function which are not covered by any other cubes are known as the prime implicants of the function. Example 1 has two prime implicants 11X and 101.

Base of a Function

The base of a switching function is that set of cubes of the function in which all the variables have either a 0 or 1 value and for which the function has a value 1.

The base of F (example 1) is $\left\{ \begin{array}{l} 101 \\ 110 \\ 111 \end{array} \right\}$

Extremal

Any prime implicant that is the sole cover of a member of the base of the function is known as an extremal.

The extremals of F (example 1) are AB and ABC i.e., 11X and 101.

Nonredundant Covers

A nonredundant cover of a switching function is a set of prime implicants in which no member is covered by the logical sum of two or more other members

of the set.

Less-Than Cubes

The prime implicants of a function are determined by comparing each cube of the set with the remaining cubes and determining if that cube is covered by any other cube of the set. The prime implicants which are less desirable than others in seeking a cover which needs the least number of comparisons, are called less-than cubes.

Arrays and Array Operators

An array is a set of cubes.

Example 2:

Consider the switching function of example 1.

The truth table representation of F is as follows:

A	B	C	F
0	0	0	X
0	0	1	X
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Types of Arrays

A switching function is defined by an array called the function array which is the set of all n cubes.

The set of cubes which cause the switching function to have a value 1 is referred to as the ON-array of the function.

$$\left\{ \begin{matrix} 101 \\ 110 \\ 111 \end{matrix} \right\}$$ is the ON-array of F. (example 2)

The set of cubes which make the function equal to 0 is called the OFF-array.

$$\left\{ \begin{matrix} 010 \\ 011 \\ 100 \end{matrix} \right\}$$ is the OFF-array of F.

The set of cubes for which the function is not defined to be 0 or 1 is called the DC (Don't Care) array.

$\left\{\begin{matrix}000\\001\end{matrix}\right\}$ is the DC-array of F.

ABSORB Operator (A)

The Unary ABSORB operator deletes from its operand array all cubes that are covered by othermembers of that array.

The covering cubes are found using the Co-ordinate covering relationship given by

$a_i \sqsubset b_i$		b_i = 0	1	X
a_i	0	ε	ϕ	ε
	1	ϕ	ε	ε
	X	ϕ	ϕ	ε

In other words, if a and b are two n-tuples of elements $a_i, b_i \in \{0,1,X\}$, then

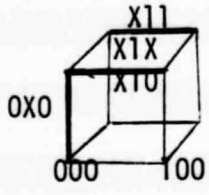
$a \sqsubset b$ if $(a_i \sqsubset b_i) = \varepsilon$ for all n.

$a \not\sqsubset b$ if $(a_i \sqsubset b_i) = \phi$ for any n.

ε indicates that a_i is included in b_i. i.e., $a_i = b_i$ or $b_i = X$.

ϕ indicates that a_i is not included in b_i. i.e., $a_i \neq b_i$ and $b_i \neq X$.

Example 3:



Let the function be represented by the cube shown as shown. Let the array C represent the set of cubes.

$$C = \left\{\begin{matrix}001\\100\\0X0\\X10\\X11\end{matrix}\right\}$$

If C^i is the i^{th} cube in array C and C^i_j is the j^{th} coordinate in the i^{th} cube then

$C^1 = 000$

$C^2 = 100$

$C_1^1 = -0-$; $C_2^1 = 1$; $C_1^1 \_\_ C_1^2 = \phi$

$\therefore$ C^1 does not cover C^2

$C^1 = 000$

$C^3 = 0X0$

$C_1^1 = C_1^3$ $\quad C_2^1 \_\_ C_2^3 = \varepsilon$

$C_3^1 = C_3^3$ $\therefore$ C^3 covers C^1 and C^1 may be absorbed.

Similarly, C^3 does not cover C^2, C^3 covers C^4 and C^4 may be absorbed.

C^6 covers C^5; C^5 may be absorbed.

The absorbed C = 100
0X0
X1X

$A(C) \equiv C$

Cube Union ($\cup$)

If A = $a^1, a^2, \ldots\ldots$ and B = $b^1, b^2, \ldots\ldots$ are two arrays of the same number of variables, the union of these arrays is the absorbed set A U B.

$A \cup B = A\,(A \cup B) = A\,(a^1, a^2 \ldots\ldots, b^1, b^2, \ldots\ldots)$

If $A = \left\{\begin{matrix}000\\X11\end{matrix}\right\}$ and $B = \left\{\begin{matrix}0X0\\X1X\end{matrix}\right\}$ then $A \cup B = \left\{\begin{matrix}000\\X11\\0X0\\X1X\end{matrix}\right\}$ $\left\{\begin{matrix}0X0\\X11\end{matrix}\right\}$

Cube Intersection ($\sqcap$)

The cube intersection of two n-tuples a_i and b is defined by the rules.

$a \sqcap b = \phi$ (empty) if any $a_i \sqcap b_i = \phi$

C otherwise, where $C_i = a_i \sqcap b_i$

and the co-ordinate intersection table

		b_i		
	$\sqcap$	0	1	X
a_i	0	0	ϕ	0
	1	ϕ	1	1
	X	0	1	X

000 0X0 = 000
000 100 = ϕ00 = ϕ
0X0 X1X = 010
X11 X1X = X11

The intersection of two arrays A and B is

$$A \sqcap B = \left\{ \left\{ A \sqcap b^1 \right\} \sqcup \left\{ A \sqcap b^2 \right\} \quad .... \right\}$$

The resulting array is to be absorbed using cube union operators.

Let $A = \left\{ \begin{matrix} 000 \\ X1X \end{matrix} \right\}$ and $B = \left\{ 0X0 \right\}$

The Common cubes in the two arrays are extracted and then absorption and cover relation concepts are applied.

Array A can be expanded as $\left\{ \begin{matrix} 000 \\ X1X \end{matrix} \right\} = \left\{ \begin{matrix} 000 \\ 01X \\ 11X \\ X10 \\ X11 \end{matrix} \right\} = \left\{ \begin{matrix} 000 \\ 010 \\ 011 \\ 110 \\ 111 \\ 010 \\ 110 \\ 011 \\ 111 \end{matrix} \right\}$ (repeated cubes are removed)

Array B is expanded as $\left\{ 0X0 \right\} = \left\{ \begin{matrix} 000 \\ 010 \end{matrix} \right\}$

The common cubes of A and B are $\left\{ \begin{matrix} 000 \\ 010 \end{matrix} \right\}$

$$\left\{ \begin{matrix} 000 \\ X1X \end{matrix} \right\} \sqcap \left\{ 0X0 \right\} = \left\{ \begin{matrix} 000 \\ 010 \end{matrix} \right\}$$

Sharp Product (#)

The sharp product of two cubes is defined by the co-ordinate sharp product table and the following rules:

a # b = a if $a \sqcap b = \phi$, i.e., $a_i \# b_i = \phi$ for some; (as in cover relationship)
if $a \sqsubseteq b$, i.e., $a_i \# b_i = \varepsilon$ for all i
$\bigsqcup_i (a_1, a_2, ..., \overline{b_i}, ..., an)$ otherwise where the union is
for all: for which $a_i \# b_i = \alpha_i \in \{0, 1\}$

a_i # b_i	b_i = 0	1	X
a_i = 0	ε	ϕ	ε
1	ϕ	ε	ε
X	1	0	ε

If X10 and 000 are two cubes

```
   X10
#  000        X10 # 000 = X10
-------
   1ϕε
```

- There is nothing in common between X10 and 000.

Similarly,

```
   X10
#  X1X        X10 # X1X = ϕ
-------
   εεε
```

- Cube X1X covers X10.

```
   X1X
#  010        X1X # 010 = { 11X, X11 }
-------
   1ε1
```

If A and B are Two arrays. A # B is defined as

$$A \# B = \{\{ \ldots \} \ \{A \# b^1\} \ \# b^2 \} \ \ldots \} \quad \text{or}$$

$$A \# B = \{a^1 \# B\} \cup \{a^2 \# B\} \ \ldots \}$$

The first cube from array B is considered and the sharp product of that cube with all the cubes of array A is computed. The sharp product of the resultant array and the next cube of array B is computed. All the cubes of array B are considered thus, one by one and the final sharp product A # B is computed. The roles of arrays A and B may be interchanged.

SPLIT Operator (S)

For the use of a split operator a special mask cube is needed. A mask cube is a special (n + m) tuple which has X's in all positions except one position in which a 0 or 1 appears.

A split operator is one which identifies and transfers to another array all cubes of a given array that are covered by a given mask cube.

If F is a function array and μ is a mask cube $FS\mu$ represents the array of cubes removed from F under mask μ.

Let $F = \begin{Bmatrix} 101\ 01 \\ 110\ 10 \\ 111\ 01 \end{Bmatrix}$ and $\mu = X^0$ i.e. Only the 4^{th} position from the left handside of the mask cube is 0.

Then $FS\mu = \begin{Bmatrix} 101\ 01 \\ 111\ 01 \end{Bmatrix}$

Star Product (*)

A star product of cubes a and b is defined by the rules

$$a * b = \begin{cases} \phi & \text{if } a_i * b_i = \phi \text{ for more than one n.} \\ C & \text{where } C_i = \begin{cases} a_i * b_i \neq \phi \\ X \text{ when } a_i * b_i = \phi \end{cases} \end{cases}$$

and the co-ordinate star product table

		b_i		
	$a * b$	0	1	X
a_i	0	0	ϕ	0
	1	ϕ	1	1
	X	0	1	X

If 11X and X01 are two cubes then from the above table

```
   11X
 * X01        By applying the rules     11X * X01 = 1X1
-------
   1φ1
```

Similarly

```
   X1X              X1X * X00 = XX0
 * X00
-------
   Xφ0
```

Consensus

The consensus of two cubes or implicants is the product term of those variables which do not have different values in the two cubes. The variables may not appear in both the cubes.

If AB and $\bar{B}C$ are two implicants, it can be seen easily that B has different values in AB and $\bar{B}C$. If B and $\bar{B}$ are removed the remaining variables are A and

C. Then the consensus of AB and $\bar{B}C$ is AC.

Similarly A is said to be the consensus of implicants AB and $A\bar{B}$.

6.3.2. MINIMIZATION ALGORITHM

Multiple output switching functions may be treated either as many single-output functions, or as a single many-input, many-output function. The second approach is taken on the minimization algorithm as it leads to better results than the first one.

The minimization algorithm followsthe six steps detailed below:

(1) A function array is formed from the input ON_i and DC_i array corresponding to each output.

(2) An array of prime implicants is formed from the function array.

a. Consensus techniques are used to find the multiple-output prime after each '1' in the output of each cube of the function array is replaced with an 'X' ('-' in the example). The output parts will then

(i) Never prohibit the formation of a * product.

(ii) Keep account of the output variables to which each input part of the cube applies, and

(iii) Prevent the loss of multiple-output prime implicants through absorbing.

b. The distinction between ON_i and DC_i entries which would be lost due to this transformation is restored later by retaining a copy of the original ON-arrays.

c. The number of trivial cubes formed is substantially reduced by removing all the cubes with an all 0-output part at each step. This is done by forming a mask cube with an all 0-output part, and then removing the undesirable cubes with the split (S) operator.

(3) A separate array of extremals or essential prime implicants is formed.

a. The cube intersection of each of the prime implicants with the ON-array is determined.

b. If the result A is null (ϕ) then that prime implicant is discarded because it covers no active members of any ON_i-array.

c. If the result A is not null ($\neq \phi$), then the external test is applied to that prime implicant.

d. The sharp product B of the result A with the array of prime implicants except the prime implicant under consideration is determined.

e. If B is not null (ϕ) then the prime implicant is an external and it is included in the array of extremals.

f. All the prime implicants are considered one by one.

(4) Non-essential prime implicants (MOMINS) are picked.

a. Even after the extraction of all the extremals, if the ON-array is not empty a complete cover has not yet been found and a less-than test is performed.

b. The less-than cubes are removed from the array of prime implicants.

c. Another extremal test is performed.

d. A branching procedure is resorted to and the prime implicant which covers the greatest number of elements of the ON-array is picked.

e. The above prime implicant is added to the set of extremals to get the final extremal array.

(5) A connection array is formed from the final extremal array. That is, the 'X''s in the output part of the extremal array are replaced by '1''s.

(6) Redundant connections are eliminated from the connection array.

a. One output is selected.

b. The cube with the selected output is extracted by applying the

split operator with a mask cube having a '1' in that output column only.

c. Each of the cubes is tested for extremals.

d. If the result of the test is not empty then the cube is not redundant.

AN EXAMPLE

FUNCTION OUTPUT ARRAY FORMED WITH THE INPUT CUBES.

```
0101
0111
1000
1001
1010
1011
1101
EOF
EOF
0001
0111
1011
1111
EOF
EOF
0001
0110
0111
1000
1001
1010
1011
EOF
EOF
```

ARRAYS FORMED IN THE INTERMEDIATE STEPS.

F-ARR
1011 111
1111 010
10X0 101
0111 111
100X 101
1101 100
0110 001
X100 000
001X 000
11X0 000
0001 011
00X0 000
0X00 000
0101 100

ON-ARR
01X1 100
1X01 100
10XX 100
0001 010
X111 010
1X11 010
X001 001
10XX 001
011X 001

F-ARR WITHOUT ALL 0S OUTPUT
1011 111
1111 010
10X0 101
0111 111
100X 101
1101 100
0110 001
0001 011
0101 100

PI OF F-ARR
1011 ---
X001 00-
01X1 -00
0111 ---
1X01 -00
X111 0-0
10XX -0-
0001 0--
X101 -00
1X11 0-0
011X 00-

```
EXTREMAL
10XX -0-
0001 0--
011X 00-

NON-EXTREMAL
1011 ---
X001 00-
01X1 -00
0111 ---
1X01 -00
X111 0-0
X101 -00
1X11 0-0

LEFT ON
01X1 100
1101 100
X111 010
1X11 010

EXTREMAL
10XX -0-
0001 0--
011X 00-
X101 -00
1X11 0-0

NON-EXTREMAL
01X1 -00
0111 ---
X111 0-0

LEFT ON
0111 100
0111 010

W/T LESS-THAN
01X1 -00
0111 ---
X111 0-0

ON LEFT
0111 100
0111 010
```

```
MOMIN PICKED
0111 ---


E-ARR
10XX 101
0001 011
011X 001
X101 100
1X11 010
0111 111
```

ARRAY REPRESENTING THE SET OF MINIMIZED CUBES.

```
E-ARRAY WITHOUT REDUNDANT CONNECTIONS
X101 100
1X11 010
0111 110
011X 001
10XX 101
0001 011
```

7. CONCLUSIONS

A high-level synthesis and design verification interface for an automatic LSI design system has been described. The output of DDLSYN is compatible with the CADAT system input. The most significant utility of the DDL system to CADAT, is that it enables an early verification of the design and automatically produces the net list. This would save design time and cost.

The modular description simulation and synthesis capabilities enable a true top down design methodology in the sense that the modules of a system can be individually designed and verified. The designer thus can associate the chip floor plan with the modules of the DDL description.

The quality of the synthesis output produced compares with that of the manual design. Due to the finite state machine model dependency of DDL, some extra flip-flops are introduced to realize state transitions. Only D-flip flops are used in the synthesis. The complement output of flip-flops are not utilized in the synthesis. Table 5 compares the automatic and manual designs for several circuits.

The designer can control the logic produced by varying the DDL description and judicious use of IDENTIFIER and BOOLEAN declarations in the description. However, DDLSYN tries to minimize the silicon area required by selecting a standard cell that realizes the majority of the BE first, followed by the selection of other standard cells to complete the synthesis. Table 6 shows a cost comparison of various implementation techniques.

Some simple logic simplification is performed by DDLTRN during its last pass. The multiple-output logic minimization interface provides an additional logic minimization option.

The PLA synthesis is limited to a portion of the combinational logic of the DDL description.

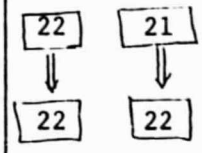
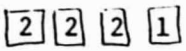
Table 5: Comparison of Automatic Design to Manual Design

Circuit	Extra Gates Needed For Automatic Design	Comments
A simple sequential circuit	9 2-Input NAND Gates	Duplicate subexpressions in RTEs were not eliminated, resulting in these extra gates
	4 Inverters	The available inverted output of the D-Flip-Flops is not used by DDLSYN.
Serial Twos Complementer	1 Inverter and 1 3-Input AND	The DDL translator does not recognize and eliminate all duplicate Boolean equations. The user may force this condition to not occur by the use of an explicit Boolean declaration.
	10 Inverters	Restrictions of available standard cells (only inverted output was available so must invert to be able to use such cell).
	2 Inverters	Inverted output of the D-Flip-Flops were not used.
Variable Timer Circuit	18 2-Input AND 5 Inverters 1 4-Input NOR 2 2-Input NOR	The finite state machine model required by DDL can cause gates to be added.

Table 6: Implementation Cost Comparison for AB + CD + EF + G

AB + CD + EF + G - Function to be implemented

2 2 2 1 - Pattern

	Implementation	Cells Needed	No. of Devices	Area (Mils)
1	2 2 2 1 2 2 2 2	1800	16	17.2
		1220	4	5.8
	*Total Cost		20	23.0
2		1870	8	9.6
		1870	8	9.6
		1220	4	5.8
	Total Cost		20	25.0
3		4 x 1220	16	23.2
		1240	8	9.6
	Total Cost		24	32.8

* Least Cost Implementation

REFERENCES

[1] S. W. Director, et.al., "A Design Methodology and Computer Aids for Digital VLSI Systems." IEEE Trans. CAS, Vol. CAS-28, No. 7, July 81, pp. 634-644.

[2] D. E. Thomas, "The Automatic Synthesis of Digital Systems," Proc. IEEE, Vol. 69, No. 10, October 1981, pp. 1200-1211.

[3] S. G. Shiva, "Automatic Hardward Synthesis," Proceedings IEEE, February 1983.

[4] S. G. Shiva, "Computer Hardware Description Languages - A Tutorial," Proceedings IEEE, Vol. 67, No. 12, Dec. 1979, pp. 1605-1615.

[5] D. L. Dietmeyer and J. R. Duley, "A Digital Systems Design Language (DDL)," IEEE Trans. Comput., Vol. C-17, pp. 350-361. Sept. 1968.

[6] J. Gould, "The Large Scale Microelectronics Computer-Aided Design and Test System," NASA TM-78202, Oct. 78.

[7] D. L. Dietmeyer, "Translation of DDL Description of Digital Systems," Dept. Elec. Comput. Eng., University of Wisconsin-Madison.

[8] D. L. Dietmeyer, "DDLSIM - Users Guide," University of Wisconsin.

[9] A. M. Shah, "Automatic Hardware Synthesis from DDL Description," Master's Thesis, University of Alabama in Huntsville, 1981.

[10] S. G. Shiva, "Combinational Logic Synthesis from an HDL Description," Proceedings 17th Design Auto. Conf., 1980, pp. 550-555.

[11] D. L. Dietmeyer and J. R. Duley, "Register Transfer Languages and Their Translation," in Digital Systems Design Automation (Vol. II), M. A. Breuer, Editor; Woodland Hills, California: Computer Sciences Press, 1975, pp. 117-218.

[12] S. G. Shiva and J. A. Covington, "Modular Description/Simulation/Synthesis Using DDL," Proceedings 19th Design Auto. Conf., 1981, Las Vegas, Nev., pp. 321-329.

[13] J. A. Covington, "Modular Logic Synthesis From a DDL Description," Master's Thesis, The University of Alabama in Huntsville, 1982.

[14] M. H. Doshi and D. L. Dietmeyer, "Automated PLA Synthesis of the Combinational Logic of a DDL Description," ECE-78-17, University of Wisconsin, November 1978.

[15] D. L. Dietmeyer, Logic Design of Digital Systems, ed. 2, Allyn and Bacon, Boston, MA, 1978.

APPENDIX

PUBLICATIONS UNDER NAS8-33096

Reports

[1] S. G. Shiva, "DDL Software System," Final Report, December 1982.

[2] S. G. Shiva, "DDL Software System - Users Manual," December 1982.

[3] S. G. Shiva, "DDL Software System - Programmers Manual," December 1982.

[4] S. G. Shiva, "Hardware Synthesis from DDL - Extensions and Logic Minimization," October 1981, NASA CR-161912.

[5] S. G. Shiva and A. M. Shah, "Hardware Synthesis Using DDL Description," October 1980.

[6] S. G. Shiva, "Digital Systems Design Language," October 1979, NASA CR-162032.

[7] S. G. Shiva "A Comparison of Hardware Description Languages," NASA, NASG-8057 Final Technical Report, Alabama A & M University, October 1978, NASA CR-157762.

[8] S. G. Shiva, "Hardware Design Languages - A Bibliography," NASA, NSG-8057 Status Report, March 1978.

Theses

[9] C. Srinivas, "Logic Minimization Interface for DDL System," the University of Alabama in Huntsville, 1983.

[10] J. Covington, "Modular Logic Synthesis from a DDL Description," The University of Alabama in Huntsville, 1982.

[11] A. M. Shah, "Automatic Hardware Synthesis from a DDL Description," The University of Alabama in Huntsville, 1981.

Publications

[12] S. G. Shiva and J. Covington, "An Automatic Logic Synthesis System," submitted to IEEE Trans. CAD-ICS.

[13] S. G. Shiva, "Automatic Hardware Synthesis," Proceedings of IEEE, January 1983 (Invited).

[14] S. G. Shiva, "Hardware Description Languages - A Tutorial," Proceedings of IEEE, December 1979, pp. 1605-1615.

[15] S. G. Shiva and J. Covington, "Modular Description/Simulation/Synthesis Using DDL," IEEE Design Automation Conference, June 1982, Las Vegas, Nevada, pp. 321-329.

[16] A. M. Shah and S. G. Shiva, "Hardware Synthesis Using DDL," IEEE Southeast Conference, Huntsville, AL, April 1982, pp. 517-520.

[17] J. Covington and S. G. Shiva, "Modular Hardware Synthesis Using HDLs," IEEE Southeast Conference, Huntsville, AL, April 1982. pp. 715-718.

[18] S. G. Shiva, "Combinational Logic Synthesis From an HDL Description," 17th Design Automation Conference, Minneapolis, Minn., June 1980, pp. 550-555.

[19] S. G. Shiva, "Use of DDL in an Automatic LSI Design and Test System," International Symposium on CHDLs and their Applications, Palo Alto, California, October 1979, pp. 28-32.