2miy

CR-134165

FINAL REPORT

THE CADSS DESIGN AUTOMATION SYSTEM

by

ERNEST A. FRANKE

The research for this report was supported by the National Aeronautics and Space Administration, Johnson Space Center, Houston, Texas.

CONTRACT NUMBER NAS 9-9214

DEPARTMENT OF ELECTRICAL ENGINEERING

TEXAS A&I UNIVERSITY

KINGSVILLE, TEXAS

(NASA-CR-134165) THE CADSS DESIGN AUTOMATION SYSTEM Final Report (Texas A&I Univ , Kingsville) ~~79~~ 145 p HC $6 00 CSCL 09B

N74-13883

Unclas
G3/08 25678

ABSTRACT

The purpose of this research was to implement and extend a previously defined design automation system for the design of small digital structures.

The report includes a description of the higher level language developed to describe systems as a sequence of register transfer operations. The system simulator which is used to determine if the original description is correct is also discussed.

The design automation system produces tables describing the state transitions of the system and the operation of all registers. In addition all Boolean equations specifying system operation are minimized and converted to NAND gate structures.

Suggestions for further extensions to the system are also given.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

In recent years digital computers have been applied, with great success, to the automation of an increasing variety of tasks in the design of digital systems, from the printing of wiring tables and the drawing of logical diagrams to the optimization, according to certain criteria, of the layout of components and wiring and even the actual computer controlled production of subassemblies such as printed circuit boards or integrated circuits.

More recently, languages have been developed which permit the simulation of a proposed system on an existing digital computer.

Examples of a few languages are given below to show recent work which has been done in this field.

The Logic Design Translator (LDT) System was developed as an aid in the automation of logic design of digital computer systems (Ref. 1). LOTIS is a formal language for describing the logical structure, the sequencing and the timing of digital machines (Ref. 3).

A register transfer language has also been developed for the Computer-Aided Digital System design and analysis (Ref. 4).

A computer program known as BLODI accepts for an input a source program written in the BLODI language, which corresponds closely to an engineer's block diagram of a circuit, and produces a machine program to simulate the circuit (Ref. 2).

Though the utilization of computers in designing digital systems is a recent innovation, a large amount of work has been done in this field. Computers are now being used to perform routine jobs: positioning the computer elements, providing the back panel wiring, printing part lists, checking and recording logical diagrams, doing a large part of the drafting, etc. (Ref. 5).

The reasons for design automation are a possible reduction in costs, increased problem analysis capabilities, an increase in man's creative ability through man-machine interaction, increased speed of design, change control and documentation, and in increase in the efficiency of available manpower.

Computer aid in the logical design of a digital system represents a natural extension of these automation procedures. This report is based on an attempt to develop a set of computer programs to produce a logic design from a conceptual (flow chart) design. The system developed has been named CADSS, an acronym for "Computer Aided Digital System Synthesis". CADSS has been developed to a point where a complete description of the operation of the system can be specified. The CADSS language has also been used as the input to a simulator, but the language has not yet been developed to specify different digital elements (like Flip-Flops, Gates, etc.) to be used for the whole digital system and to make a complete logic drawing of the system. As a whole CADSS represents a simple language which can be learned without having much background in computer programing.

The primary subject of this report consists of the description and use of the language to design a digital system and to develop a computer program to produce a table which will specify each operation and condition of each part of the digital system to be designed. The program was written in such a way that this table can further be used to assign various digital components to each block and finally be used in producing a complete logic design of the system.

1.1 An Introduction to the Design of Digital Systems

The development of the computer aided digital system synthesis (CADSS) program is an attempt to apply Computer-Aided design techniques to the design of Sequential Logic Systems. The recent development of integrated circuits and the anticipated development of large scale integration of logic elements allows a designer to use standard functional units such as registers, counters, or adders in the design of a System. A complete digital system may then be described in terms of the operations of the functional units and the control logic necessary to sequence the operations.

A digital system described in CADSS produces a computer output (giving each unit -- operation and condition.) This output can easily be combined to produce a complete logic design of the system.

Basically any set of sum-of-products Boolean equations can be said to represent a constructional description of a digital system. The digital system design problem can then be reduced to the problem of obtaining such a constructional description. Of course, before starting from Boolean equations, conceptual design and functional design are done, but the main concentration in the CADSS Programs is placed on logic design and its implementation in computer programs.

Starting from the Boolean equations it should be possible to automate a complete set of operations for each unit which would be implemented into complete logic diagrams of the system specifying each standard functional unit to be used.

Completion of this phase of the design process will result in a hardware layout description and a wiring list.

CHAPTER II

2. THE FLOW CHART DESCRIPTION LANGUAGE

The Computer Aided Digital System Synthesis (CADSS) language was developed without paying much attention to linguistic aspects. The most important factors taken into consideration are given below:

1. The operations of a digital system must be described by the CADSS language at the flow chart level.
2. The language must easily be learned by digital system engineers knowing the concepts and terminology of digital systems only, even without having much experience in computer programing.
3. The CADSS language must also serve as an input to a simulator. This requires the inclusion of simulation control commands such as print, read and pause.
4. The language must be capable of describing two or more concurrent interacting systems in order to avoid the problems arising from the interaction of separate machines.

DESCRIPTION OF THE CADSS LANGUAGE

2.1 GENERAL CONVENTIONS:

INPUT MEDIA: In order to facilitate preparation and modification of system descriptions, the input medium for CADSS is punched cards. No column restriction has been used and all 80 columns of a card may contain information.

COMMENT CARDS: A 'C" in the first column of a card designates a comment card. Comment cards are ignored by the processor and are simply printed out with the program listing.

TERMINATION CHARACTER ($): A special termination character ($) is used to indicate the end of a syntatic unit of the program such as a declaration or statement. Since a special termination character is used to indicate the end of a statement, a statement may thus be continued from one card to the next but the continuation must occur at a blank i.e. names, words, and numbers may not be divided. After a termination character, the remainder of the card is not processed, and additional comments may be placed after the terminator. The next statement must begin on a new card.

FORMAT: Cards are punched in free-format form. The only requirement is that at least one blank is necessary as a separator between syntactic elements such as words or numbers, but no restriction is placed on the number of blanks.

2.2 UNIT AND DECLARATIONS:

Hardware units such as memory elements or functionally connected groups of memory elements (counters or registers) correspond to CADSS 'Units.' All units are declared in the first section of the CADSS program. The unit declared format is:

```
UNIT TYPE:  NAME 1, NAME 2, ---------$
```

The unit types are:

1. INPUT: Input units are supplied to the system from outside. An input value cannot be changed by the CADSS program.

2. OUTPUT: The memory elements supplying information from the system are treated as output units.

3. REGIST: Groups of memory elements on which operations (shifting right or left, setting to any desired value, incrementing or decrementing in binary sequence) can be performed are declared as registers.

NAME: A name must start with an alphabetic character followed by a sequence of alphabetic and/or numeric characters. The first six characters of any two names may not be the same, but otherwise there is no restriction on the length of a name. If the unit declared consists of more than one memory element (or if it consists of several input/output lines considered as one unit), the name is subscripted with the number of memory elements in the unit. In the case of counters and registers the most significant bit is Bit 1, and it is located in the leftmost memory element of the unit. The names in a declaration list are separated by commas and the list is terminated by a dollar sign ($).

EXAMPLES

```
INPUT:  START, CLEARA, DATA (8) $

OUTPUT:  READY, DATAOUT (10) $

REGIST:  BUFFER (10), STATE (3), A (20), AREG (10) $
```

All declarations must appear in the heading of the CADSS description. The defined units of the System may be used as operands in three ways:

1. If the entire unit is to be used as the operand the unsubscripted name is used

```
        AREG
```

2. An operand consisting of a single bit of a unit is indicated by the name with a single subscript such as

```
        AREG (1)
```

3. If a section (several adjacent bits) of a unit is to be used as a subscript this is indicated by an extended subscript. For example, the first four bits of BUFFER would be referred as

AREG (1-4)

2.3 STATEMENTS:

A statement is a basic element of a CADSS program. The structure of a statement consists of a CONDITION LIST and an OPERATION LIST. These two lists will be described separately. Then their use in the formation of a statement will be considered.

2.3.1 CONDITION LIST:

The condition list is a well formed Boolean Expression of defined units of the System, Boolean literals, and the following operators:

1. - Logical NOT
2. + Inclusive OR
3. * Logical AND
4. = Logical Equality
5. ↑ Transition to True
6. ↓-Transition to false

<u>LOGICAL OPERATORS</u>: The logical negation (NOT) operator is used with a single one bit operand. The logical value of the negated variable is true when the variable is false and false when the variable is true. The logical AND and OR operators (+,*) are used in the conventional manner. Each operator requires two operands of one bit each. The logic Equality (=) operator also requires two operands, but at least one must be a defined unit. The other operand may be a binary value. Basically, the logic Equality operator is used to obtain a result by comparing bits of two operands. To illustrate the full use of logic Equality operators some examples are given below:

Examples: If we define OUTPUT: FINISH $

```
                          INPUT: AIN, BIN, SA, SB $

                          REGIST: R(20), AREG (10),
                                  BREG (10), ISIGN,
                                  PCTR (4) $
```

PCTR=0000 is true when all the four bits of PCTR are zero, and the statement is false for all other PCTR values.

Similarly:

```
     ISIGN = (SA*SB)+(-SA*-SB)
```

This states that the operation is true when the Boolean expression, (SA*SB)+(-SA*-SB), is true and false when the Boolean expression is false.

TRANSITIONAL OPERATORS:

The transitional operators ('and'-) are special unary operators which are true when the single one bit operand changes value. The use of the transition operators is illustrated in the following examples:

'CLEAR	True when CLEAR changes from false to true.
'-BUFFER(3)	True when the third bit of BUFFER changes from true to false.

The normal order in which operations are performed in a condition list is

1. = Equality operation
2. : Transition to true operation
3. :-Transition to false operation
4. - Negation operation
5. * And operation
6. + OR operation

By using parenthesis, the normal ordering of the operations may be modified.

All the condition lists must be well formed Boolean expressions i.e. each operator must be associated with the correct number of operands in the right order. The following examples illustrate the condition lists:

```
INPUT:  AIN (10), BIN (10), START, SA, SB, $

REGIST:  R(20), B(10), SCTR (4), PCTR (4), CARRY,
         SUM, NCARRY, ISIGN $
```

Examples of condition lists:

R(1)*B(1)+R(1)*CARRY+B(1)*CARRY

(SA*SB)+(-SA*-SB)

(R(1)+B(1))*-CARRY+(R(1)+CARRY)*-B(1)

(B(1)+CARRY)*-R(1)

2.3.2 OPERATION LISTS

The system operations during some increment of time are specified by a list of commands called the operation list. There are three types of commands available in the CADSS input language.

(a) Unit control command
(b) Sequence control command
(c) Simulation control command

(a) Unit control command:

Unit control commands can be grouped into 8 different commands as described below:

1. SET/UNIT NAME/

This specifies that the unit named is to be set to true (all ones)

Such as: Set AREG $

2. RESET/UNIT NAME/

This specifies that the unit named is to be reset to a logical false value (All zeros).

Ex. RESET PCTR $

3. RSHIFT/REGISTER NAME/ENTER/BIT NAME/

This specifies that the register named is to be shifted right one place. The value of the bit named is shifted into the left most position of the register.

Ex. RSHIFT R ENTER SUM $ (Shift register R to right one place and put the contents of SUM in R(1))

4. LSHIFT/REGISTER NAME/ENTER/BIT NAME/

This specifies that the register named is to be shifted left one position. The value of the bit named is placed in the right most position of the register.

5. INCREMENT/REGISTER NAME/

This specifies that the register named is to be incremented in binary sequence.

6. DECREMENT/REGISTER NAME/

This specifies that the named register is to be decremented in binary sequence.

7. LOAD/NAME 1/FROM/NAME 2/

This specifies that contents of NAME 2 are to be transferred to NAME 1 leaving the contents of NAME 2 unchanged. The dimensions of the named units must be the same.

8. LOAD/NAME 1/ FROM/ BINARY LITERAL/

This specifies that NAME 1 is to be assigned the value of the binary literal.

It has already been said that input values remain unchanged, and a command trying to make any change in the input value is illegal. As described in Section (2.3) a name used may be an unsubscripted name designating a complete unit, a name with a single subscript indicating a single bit of that named unit, or a name with an extended subscript denoting a section of a named unit.

(b) Sequence control command:

The second type of Command controls the sequence of the program. It has only one command, described below:

GO TO/ LABEL/

This command intercepts the normal sequence of operations and transfers control to the point indicated by the LABEL (See Section 2.4).

(c) Simulation control command:

These commands control the simulation of the system. Since the system is to be simulated, it is necessary for the user to specify when inputs are to be changed and when the contents of specified units are to be printed out.

The Simulation Control Commands described are:

1. ACCEPT

This command transfers the information of a data card to the Input units. The binary number in the first column of the card is transferred to the first bit of the input. The number in the second column of the card is transferred to the second bit of the input unit declared, and so on.

2. DISPLAY:

This controls the printing of the contents of the units named in the print list. The print list is a list of unit names entered during simulator initialization. This may be changed without recompiling the program.

3. PAUSE:

This command halts the normal termination of the simulator until the computer 'run' switch is pressed. This may be used to provide time for the operator to observe the current condition of the system so that data cards may be prepared accordingly. This is particularly useful when the simulated system is to interact with an operator, process, or machine. An Operation list is a sequence of commands separated by commas. Given below is the order in which commands are to be executed, in case more than one command appears in a list.

1. All unit control commands (Simultaneously)
2. Accept command
3. Display command
4. Pause command
5. Sequence control (GO TO) command

Some examples are given in order to explain operation lists.
Assuming the units defined are:

```
OUTPUT: FINISH, ISIGN, A, B, C $
INPUT: START $
REGIST: R(20), B(10), SCTR(4), PCTR, AREG(10),BREG(4) $
```

Example:

```
LOAD SCTR FROM 0101,GO TO SKIP $
```

In this example SCTR is set to 0101, and control is then transferred to label Skip.

```
RESET AREG, GO TO HERE, $
```

AREG is reset, and control is transferred to label named HERE.

```
SET R, LOAD AREG (1-4) FROM BREG,LOAD BREG FROM AREG
(4-8), DISPLAY
```

R is Set, the first four bits of AREG are set to the contents of BREG, the contents of 4-8 bits of AREG are transferred to BREG, and the contents of the units specified by the print list are printed.

2.3.3 STATEMENT FORMATION:

The operation and condition lists described in the previous section now may be combined to form a statement. There are three different type of formats for statements:

(a) THEN/OPERATION LIST /$

This specifies that operation has to be done regardless of conditions existing in the system.

(b) If an operation is to be performed for a given condition, a conditional statement format can be used.

WHEN/CONDITION LIST/THEN/OPERATION LIST/$

Now the operation list is executed only when the condition list is true.

(c) Sometimes a system has to perform one set of operations when the condition is true and otherwise perform another set of operations when the condition is False. This is specified by the else statement format.

WHEN/CONDITION LIST/THEN/OPERATION LIST/ELSE/OPERATION LIST/$

In this case if the condition list is true, the first set of operations is executed. If the condition is false the commands in the second operation list are executed.

STATEMENT EXAMPLES ARE:

```
a.THEN LOAD BREG FROM AREG (1-4) $
  THEN SET FINISH $

b.WHEN 'START THEN RESET AREG $
  WHEN PCTR = 0000 THEN SET FINISH $

c.WHEN -R(11) THEN GO TO SKIP ELSE LOAD SCTR FROM 0101$
  WHEN SCTR = 0000 THEN GO TO SHIFT ELSE LSHIFT R ENTER
  SUM $
```

2.4 LABELS:

A label is a sequence of alphabetic and/or numeric characters starting with an alphabetic. No restriction has been placed on the length of the labels but the first six characters must not be same. A label may be used with any statement separated by a colon.

LABEL:STATEMENT

Any statement which is to be referenced by a sequence control command (GO TO) must be labeled. Labels may also be used with any statement to enhance the meaning of a program. Any statement which is to be the initial statement in the description of a system must be labeled so that it may be referred to during simulator initialization.

2.5 PROGRAM BLOCK STRUCTURE:

The statements of a CADSS program correspond roughly to the blocks making up the flow chart describing the system operation. In the case of a flow chart description of a system, the sequence of operation follows flow paths from block to block so that at a given time only the operations of one block may occur. This block may be said to be active at that particular time while all others are inactive.

In the same way, the sequence of operations in a CADSS program follows paths defined by the structure of the program. At a given time one (or perhaps several) of the statements are permitted to perform operations. The statements that may perform operations at a particular time are said to be potentially active at that time. The order in which statements become potentially active is defined by the block structure of the program and the sequence control commands.

For purposes of description it is useful to consider the statement or statements which are potentially active as being in control of the system. "Control" may be considered as a pointer (or pointers) which moves through the program, always specifying the potentially active statement (or statements). The transfer of control command (GO TO) may be interpreted as specifying the label of the next potentially active statement.

A block of a CADSS program consists of one statement which may be labeled followed by any number of unlabeled statements. Since three different types of blocks are defined, it is necessary to indicate a change in block type by means of a block type specification. The format of a block is:

```
BLOCK TYPE: LABEL: STATEMENT $
                   STATEMENT $
                   STATEMENT $
                       .
                       .
                       .
```

The block type specification may be emitted if the type is the same as that of the preceeding block. If the type specification is present the statement label may be omitted.
The end of a block (and the beginning of the following block) is indicated by a block type specification, a statement label, or both. One obvious result of this

method of specifying blocks is that no block may contain another.

2.5.1 BOOLEAN BLOCKS

The block type specification for a Boolean block is

BOOLEAN:

The only statements permitted in a Boolean block are Boolean statements having the format:

WHEN/CONDITION LIST/THEN SET/UNIT NAME/ $

Whenever the condition list is true, the unit named is set true and whenever the condition list is false, the unit named is set false. Boolean statements may be used to define signals to be used in the system or to define connections between registers and outputs.

Examples of Boolean Statements

```
  INPUT: START, AIN (10), BIN(10), SA, SA $

  REGIST: R(20), B(10), SCTR (4), PCTR(4), CARRY, SUM,
          NCARRY, ISIGN $

BOOLEAN:

WHEN (SA*SB)+(-SA*-SB) THEN SET ISIGN $

WHEN (R(1))*-CARRY+(R(1)+CARRY)*-B(1)+(B(1)+CARRY*-R(1)
THEN SET SUM $
```

2.5.2 SEQUENCED BLOCKS

The block type specification for a sequenced block is

SEQUENCED:

Only one statement in a sequenced block may be potentially active at a given time. If the condition list of a

potentially active statement is true, the statement becomes active and the commands in the operation list are executed.
If the condition list is false, the statement remains potentially active. After the operation list of a statement is executed, the statement becomes inactive and the next statement in the block becomes potentially active.
Thus, in the absence of any transfer of control, commands, statements become potentially active in sequence. When the last statement of a sequenced block is reached, control passes to the next block. This normal sequence of control will be interrupted when an operation list containing a transfer of control command is executed.

2.5.3 CONCURRENT BLOCKS

The block type specification for a concurrent block is

CONCURRENT:

When control is transferred to a concurrent block all statements in the block become potentially active. The condition lists of all potentially active statements are then checked to determine which of the statements are to become active. The commands in the operation lists of the active statements are executed concurrently. If a transfer of control command was not executed all statements in the block become potentially active and the process is repeated. Control remains at a concurrent block until a transfer of control command is executed.

2.6 CADSS CONTROL CARDS

The control cards necessary for running the CADSS programs may be divided into two types: those required for the computer operating system in order to execute the programs, and those used for control and choice of options in the CADSS programs themselves.

2.6.1 OPERATING SYSTEM CONTROL CARDS

The CADSS system currently consists of three computer programs:

COMPL - a complier which translates the English language CADSS description into lists and tables for later processing.

TABL - A table generator program which produces state transition tables and unit operation tables (specifying the conditions each operation on a unit is performed) from the compiler output.

SIMUL - A simulator program, operating from the compiler output, to verify that the original description is correct.

The control cards used for the IBM 360 implementation are:

360/DOS

```
// JOB CADSS
// EXEC  COMPL

     CADSS Program

/*
// EXEC TABL
/*
// EXEC SIMUL

     Simulation data

/*
/&
```

360/OS

```
//   CADSS    JOB
//S  EXEC   CADSS360
//PASS1.SYSIN  DD *

     CADSS Program

/*
//
//PASS2.SYSIN  DD *
/*
//PASS3.SYSIN  DD *

     Simulation data

/*
/&
```

2.6.2 COMPILER CONTROL CARDS

Due to error checking procedures during compilation it is necessary to separate the declarations (of units) from the statements of the program. This is accomplished by the control card $DEND$ which must begin in the first column of a card.

The end of a CADSS program is indicated by a $ in the first column of a card. This may be followed by two options (on the same card). The first option, TABLES, directs the compiler to list the system description produced from the input. These tables will be of limited value to users who are not familiar with the internal operation of the system.

The option STORE N directs the compiler to store the generated lists and tables on an assigned output file for use by the other programs. The tape produced is labeled by the 4 digit number N to prevent confusion of types of different systems. The STORE option must be specified if the Table generator or the simulator programs are to be run.

Storage of compiler results on tape allow many simulations to be performed without the need of recompiling. This also allows the tape to be saved for diagnostic use if problems arise during system construction.

CHAPTER III

3. SERIAL BINARY MULTIPLIER EXAMPLE

A complete design of a Serial Binary Multiplier was selected as an example of the use of CADSS language. The selection of various components and the complete logical design is described in detail.

3.1 A GENERAL DESCRIPTION OF A SERIAL, BINARY MULTIPLIER: (4 bit)

The design of a binary multiplier is simplified by the fact that a one-digit binary multiplication is very easily mechanized. The truth table for the binary multiplication of two digits is given below, and is seen to be the same as the logical operation "and."

Yi	0	1
Xi		
0	0	0
1	0	1

Therefore the multiplication of digit Xi by the number Y may be accomplished simply by storing Xi in a flip-flop for a word-time, and applying its output to an 'and' circuit whose other input is the serial number Y.

In general, if X is a n digit number, the multiplication may be looked upon and is often mechanized as a series of n simplified multiplications, each determining the product of the number Y by a single digit Xi. These subproducts must be shifted with respect to one another and added together. Very often the multiplication is carried out in a sequence of steps. Each of the steps determines what is called a 'partial product.' Each partial product requires a shift, a single-digit multiplication and an addition. n of these steps are required to form the product of two n-digit numbers.

The multiplication is then broken down into a sequence of operations which may be described by the following rules:

(a) Test the least significant bit of the multiplier register. If it is 'one' the multiplicand register is to be added to the left-hand end of the partial product register. Otherwise, go to step b.

(b) Right shift the multiplier register by one digit, putting the former least significant bit into the most significant bit position. This brings the next multiplier digit into a position when it can control the next one-digit multiplication. Saving the least significant bit in the multiplier register insures that the multiplier is not lost. (If it is not necessary to save the multiplier, the multiplication unit may be simplified by eliminating one half of the partial product register and shifting the product into the multiplier register, replacing the 'used' multiplier digits.)

(c) Right shift the partial product register by one bit, putting the least significant bit from the left hand half of the partial product register into the right hand half.

(d) If the previous three steps have been carried out four times, stop. Otherwise, return to step a.

As an example consider the multiplication of the two binary numbers 5 and 7. By hand the multiplication may be carried out as:

```
      0111 = 7
      0101 = 5
   ----------
      0111
     0000
    0111
   0000
   ----------
   0100011 = 35
```

The same multiplication carried out according to the sequence of steps described above is shown in Table 3.1.

Table 3.1

Multiplication Sequence

Step	Multiplier Register	Partial Product Register	
		Left-Hand Half	Right-Hand Half
	*	0 0 0 0	0 0 0 0
a	0 1 0 1.	0 1 1 1	0 0 0 0
b	1.0 1 0		
c		0 0 1 1	1 0 0 0
a	1.0 1 0	0 0 1 1	1 0 0 0
b	0 1.0 1		
c		0 0 0 1	1 1 0 0
a	0 1.0 1	1 0 0 0	1 1 0 0
b	1 0 1.0		
c		0 1 0 0	0 1 1 0
a	1 0 1.0	0 1 0 0	0 1 1 0
b	0 1 0 1.		
c		0 0 1 0	0 0 1 1
d	STOP		

* The dot indicates the position of the least significant bit of the multiplier.
The method described above has been used in the example of designing 10-bit serial multiplier, described in the next page.

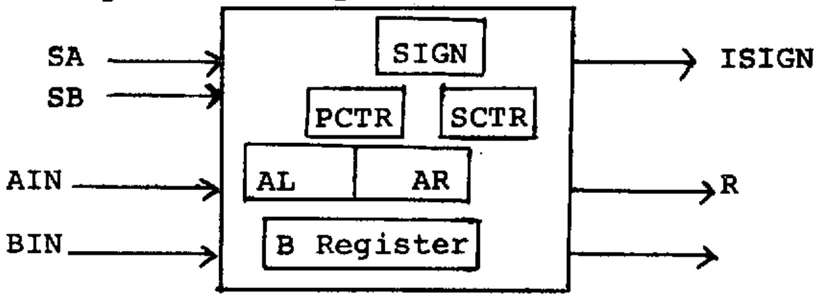
10-bit Serial Binary Multiplier:

3.2 SYSTEM ANALYSIS:

The multiplier is to be designed to accept two signed ten bit numbers and a start signal.

It is then to perform the multiplication and transmit a finish signal on completion. The multiplication is to be performed by repeated serial addition. Two internal counters are necessary for controlling the operation. One ten bit register is needed to hold the multiplicand. One twenty bit register is required to hold the multiplier and the partial product digits.

A simplified block diagram is given below to describe the system clearly.



3.3 LOGIC AND CIRCUIT DESIGN:

A flow chart describing the various operations for multiplication is drawn by considering the different sequences of operations that must be performed.

The following initialization operations are done at the start of multiplication:

(1) Reset finish signal

(2) Reset AL

(3) Accept the Signal on the input lines

AIN to AR

BIN to B Register

(4) Set the Sign bit to the Sign of the result

(5) Initialize the product counter to 10.

Since these operations occur at the same time, they are grouped in one block. Now initialization of the add operation is done by clearing the carry bit and decrementing the product counter. Bit 10 of the AR result register the least significant bit of the multiplier is tested and if it is 1, then the add operation is done. To perform addition the shift counter is first set to 10 and the sum and carry are formed. Register R is shifted entering the sum in the most significant bit position and the shift counter is decremented. This process is repeated until the contents of shift counter are zero.

After each addition, it is necessary to restore the R register by 10 additional Shifts.

The Register R is shifted by one bit after each addition (or skipped addition), the product counter is decremented, and the AR(10) bit is tested in order to determine if an addition is required. When the product counter is zero, the finish signal is given (set), and the multiplier returns to unit for the next multiplication input. A complete flow chart describing the above given description is given in fig. 3.1.

Table 3.2 is the CADSS description of the multiplier written directly from the flow chart in following steps.

(a) All units of the system are defined such as Inputs, Outputs or Registers.

(b) The combinational logic used to generate sum, carry, sign and the output is defined in Boolean expressions in the Boolean block.

(c) Each operation of the system is then grouped in sequenced or concurrent blocks.

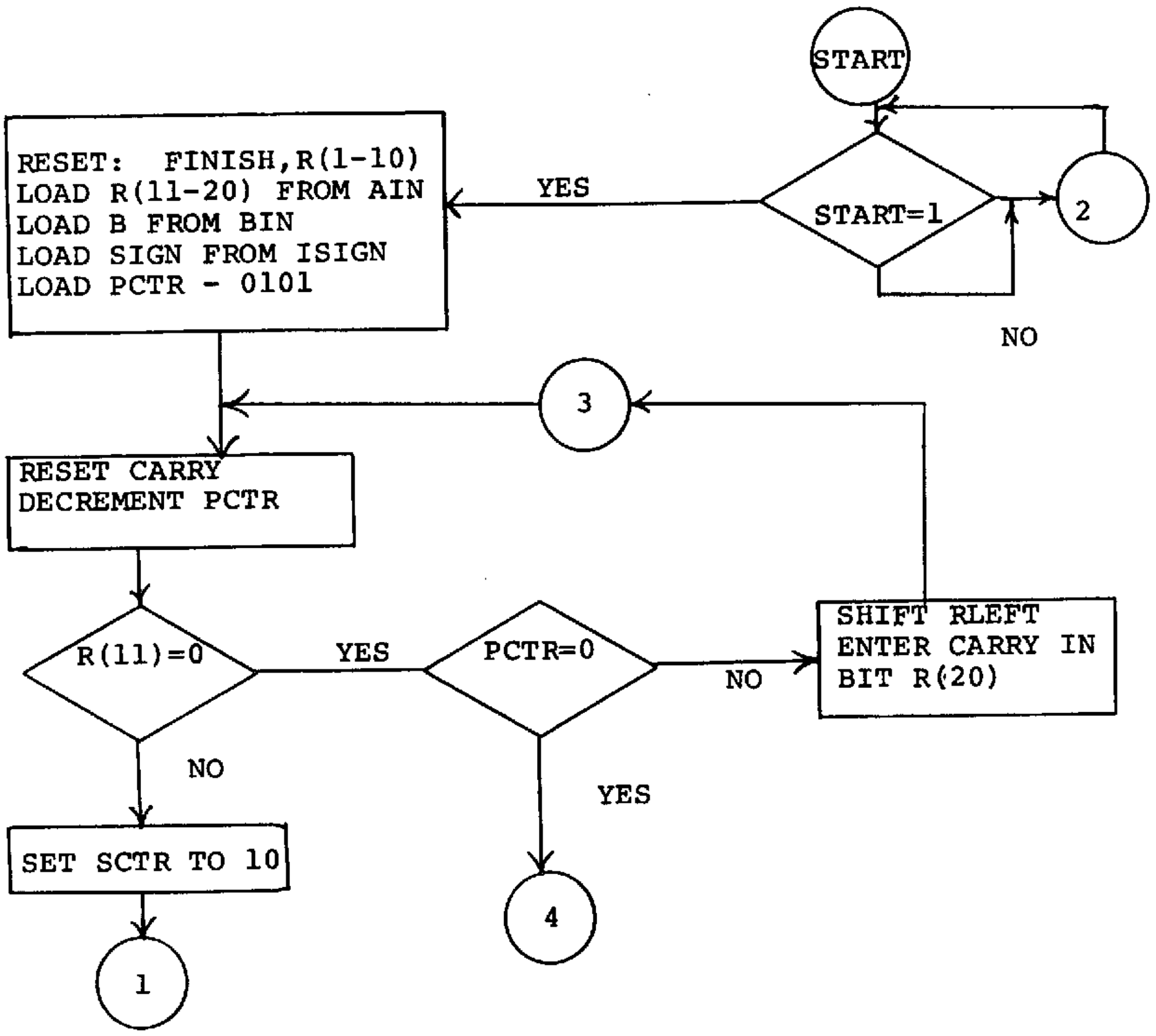
3.4 OUTPUT TABLE DESCRIPTION OF CADSS PROGRAM LEADING TO A COMPLETE LOGICAL DESIGN

The CADSS program shown in Table 3.2 was run on an IBM 360 Computer. The output obtained gives a detailed description of the logical design of the system.

Figure 3.1:

FLOW CHART OF 10 BIT MULTIPLIER

```
SUM=(R(1)+B(1))*-CARRY+(R(1)+CARRY)*-B(1)+(B(1)+CARRY)*-R(1)
NCARRY=R(1)*B(1)+(1)*CARRY+B(1)*CARRY
ISIGN=(SA*SB)+(-SA*-SB)
```



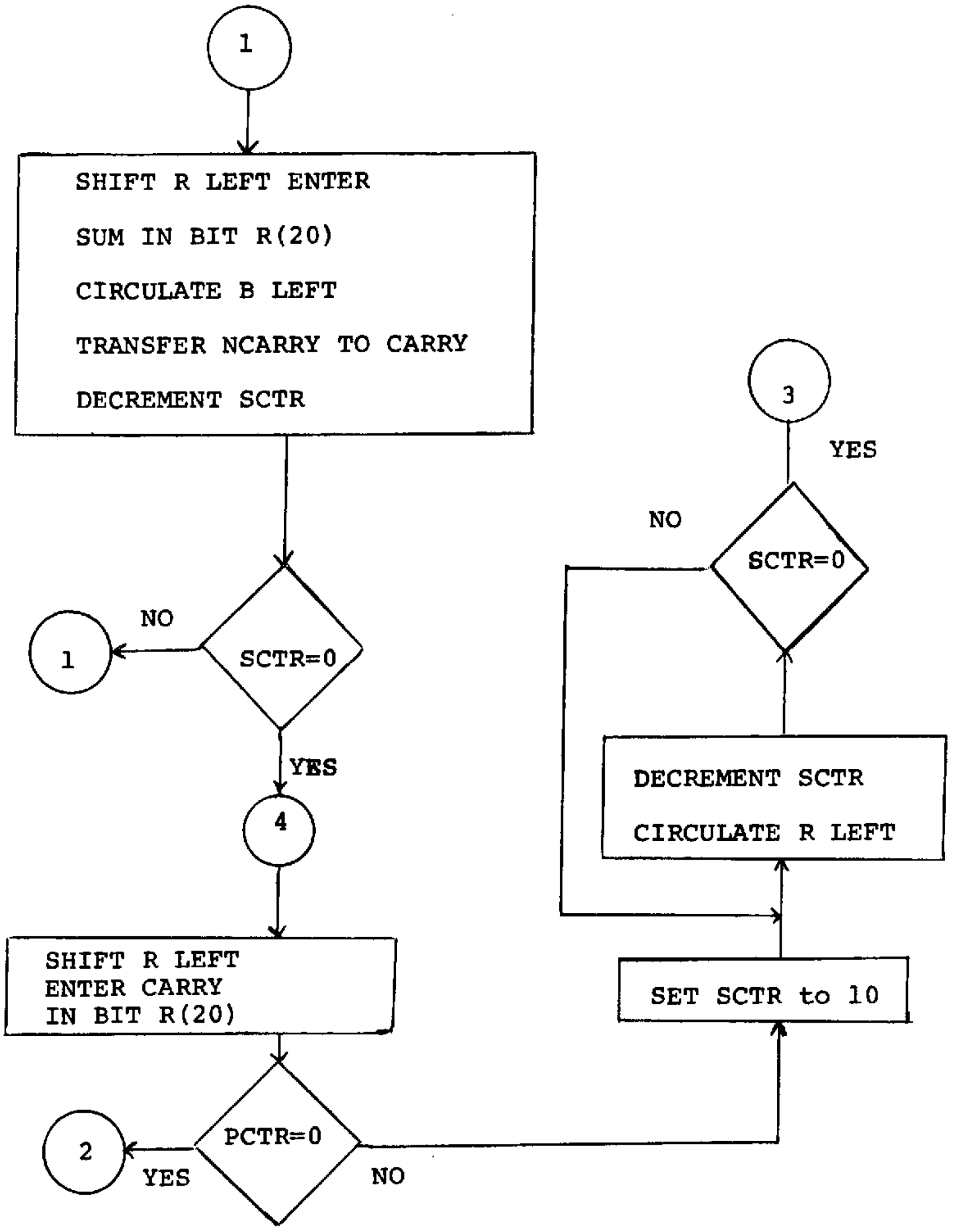

1
SHIFT R LEFT ENTER
SUM IN BIT R(20)
CIRCULATE B LEFT
TRANSFER NCARRY TO CARRY
DECREMENT SCTR
SCTR=0
NO
1
YES
4
SHIFT R LEFT
ENTER CARRY
IN BIT R(20)
PCTR=0
YES
2
NO
SET SCTR to 10
DECREMENT SCTR
CIRCULATE R LEFT
SCTR=0
NO
YES
3

Fig. 3.1 (Cont.)

```
 0     C          MULTIPLICATION BY REPEATED SERIAL ADDITION.
 0     C
 0     C    CADSS PROGRAM EXAMPLE ILLUSTRATING THE DESIGN OF
 0     C    A SIMPLE DECIMAL MULTIPLIER.
 0     C
 0     C          DEFINE MULTIPLIER INPUTS
 0            INPUT:  START,SA,SB,AIN(10),BIN(10)    $
 0     C
 0     C          DEFINE MULTIPLIER OUTPUTS
 0          OUTPUT:  FINISH, SIGN    $
 0     C
 0     C          DEFINE REGISTERS
 0          REGIST: R(20),B(10),SCTR(4),PCTR(4),
 0                CARRY,SUM,NCARRY,ISIGN    $
 0     $DEND  $
 1     C
 1     C          DEFINE COMBINATIONAL LOGIC
 1     BOOLEAN:
 1          WHEN (R(1)+B(1))*-CARRY + (R(1)+CARRY)*-B(1) +
 1                (B(1)+CARRY)*-R(1) THEN SET SUM    $
 2          WHEN R(1)*B(1) + R(1)*CARRY + B(1)*CARRY
 2                THEN SET NCARRY    $
 3          WHEN (SA*SB) + (-SA*-SB) THEN SET ISIGN    $
 4     C
 4     C          INITIALIZE AND ACCEPT DATA FOR SIMULATION
 4     C          TRANSFER DATA TO COUNTERS ON A START PULSE
 4       SEQUENCED:   START:
 4          THEN ACCEPT    $
 5          WHEN 'START THEN RESET FINISH, RESET R(1-10),
 5                LOAD R(11-20) FROM AIN, LOAD B FROM BIN,
 5                LOAD SIGN FROM ISIGN, LOAD PCTR FROM 0101,
 5                DISPLAY    $
 6     C    SKIP ADD CYCLE IF MULTIPLIER BIT IS A ZERO
 6       CONCURRENT:  SKIPCHECK:
 6                THEN RESET CARRY, DECREMENT PCTR    $
 7                WHEN -R(11) THEN GO TO SKIP ELSE
 7                     LOAD SCTR FROM 0101, GO TO ADD    $
 8          ADD:
 8                WHEN SCTR = 0000 THEN GO TO SHIFT ELSE
 8                     LSHIFT R ENTER SUM, LOAD CARRY FROM NCARRY,
 8                     DECREMENT SCTR, LSHIFT B ENTER B(1)    $
 9       SEQUENCED:   SKIP:
 9                WHEN -PCTR=0000 THEN LSHIFT R ENTER CARRY,
 9                     GO TO SKIPCHECK ELSE GO TO SHIFT    $
10          SHIFT:
10                THEN LSHIFT R ENTER CARRY    $
11                WHEN PCTR=0000 THEN SET FINISH,GO TO START,
11                DISPLAY ELSE LOAD  SCTR FROM 0101    $
12     C
12     C          THE REGISTER MUST BE RESTORED TO THE ORIGINAL
12     C          POSITION AFTER EACH ADDITION CYCLE
12       CONCURRENT:   RESTORE:
12                WHEN SCTR=0000 THEN GO TO SKIPCHECK ELSE
12                     LSHIFT R ENTER R(1), DECREMENT SCTR    $
13     $   TABLES,    STORE 50    $
```

Table 3.2
CADSS Program for Multiplier

Table 3.3 reproduces Page 2 of the CADSS output giving the number of flip-flops for the various elements.

CADSS/SYSTEM 360/40

	NAME	TYPE	DIMENSION	INDEX
1	START	1	1	1
2	SA	1	1	2
3	SB	1	1	3
4	AIN	1	10	4
5	BIN	1	10	14
6	FINISH	2	1	24
7	SIGN	2	1	25
8	P	3	20	26
9	B	3	10	46
10	SCTR	3	4	56
11	PCTR	3	4	60
12	CARRY	3	1	64
13	SUM	3	1	65
14	NCARRY	3	1	66
15	ISIGN	3	1	67

Table 3.3
CADSS Element List

The dimensions specify the number of flip-flops required for each name or element. For example, to get a finish signal we need only one flip-flop. This gives us a complete list of flip-flops to be used in our logic design of the given system.

Elements used as inputs (Type 1) and outputs (Type 2) are listed in the table but do not require memory elements for implementation.

3.4.1 CADSS STATE TABLE

The CADSS Table Generator program constructs a state table for any described system from information implicit in the block structure of the system description. Table 3.4 shows the format of the generated

```
     CADSS STATE AND UNIT OPERATION TABLES


    TAPE ID    50

PRESENT STATE   NEXT STATE            CONDITION


        1
                         2
                              UNCONDITIONAL

        2
                         3        (!START)

        3
                         5        (-P(11))
                         4        -(-P(11))

        4
                         6        (SCTR=0000)

        5
                         2        (-PCTR=0000)
                         6        -(-PCTR=0000)

        6
                         7
                              UNCONDITIONAL

        7
                         1        (PCTR=0000)
                         8        -(PCTR=0000)

        8
                         2        (SCTR=0000    )
```

Table 3.4
Computer Output State Table

state table for the multiplier example. Table 3.5 shows the state table drawn in conventional form and Figure 3.2 shows a state diagram constructed from the computer output.

In addition to conditions derived by gating internal elements, "unconditional" and "persistant" conditions are also indicated.

"Unconditional" transitions occur whenever the system is in the first named state regardless of the conditions existing in the system. "Persistant" states are those having no exits. Once a system enter a persistant state no further state transitions occur.

3.4.2 CADSS UNIT OPERATION TABLE

The unit operation table for the multiplier example is shown in Table 3.6. This table essentially converts the initial sequence oriented description of a system to a block diagram description. The operations to be performed by each block and the conditions that must exist in order for each action to be performed are explicitly stated.

The CADSS Unit Operation Table provides a detailed description of operations to be performed on system units under different conditions. These steps almost complete the logical design of the given system.

The frist column in the table is the name of the element for which the operations are to be determined. The second column gives all operations performed by that element and the third column shows the conditions that are required to exist in the system in order for the particular operations to be performed. The system states are indicated by S1, S2, etc.

As an example the element FINISH (a single flip-flop) is to be set whenever the system is in state 7 and PCTR is zero. FINISH is reset whenever the system is in state 2 and a start pulse occurs.

With the generation of the State and Unit Operation Tables the first phase of the logical design of a system is complete. The next step in the design process is to choose a logic family, determine which memory elements are to be used, and implement the condition equations in combinational logic. Before these steps are considered in detail, the problem of verifying a system description by simulation is discussed.

CONDITIONS FOR NEXT STATES (STATE/CONDITION)

Present State	'Start	-R(11)	R(11)	SCTR=0	-PCTR=0	PCTR=0	Unconditional
1	X	X	X	X	X	X	2
2	3	X	X	X	X	X	X
3	X	5	4	X	X	X	X
4	X	X	X	6	X	X	X
5	X	X	X	X	3	6	X
6	X	X	X	X	X	X	7
7	X	X	X	X	8	1	X
8	X	X	X	3	X	X	X

Table 3.5
State Transition

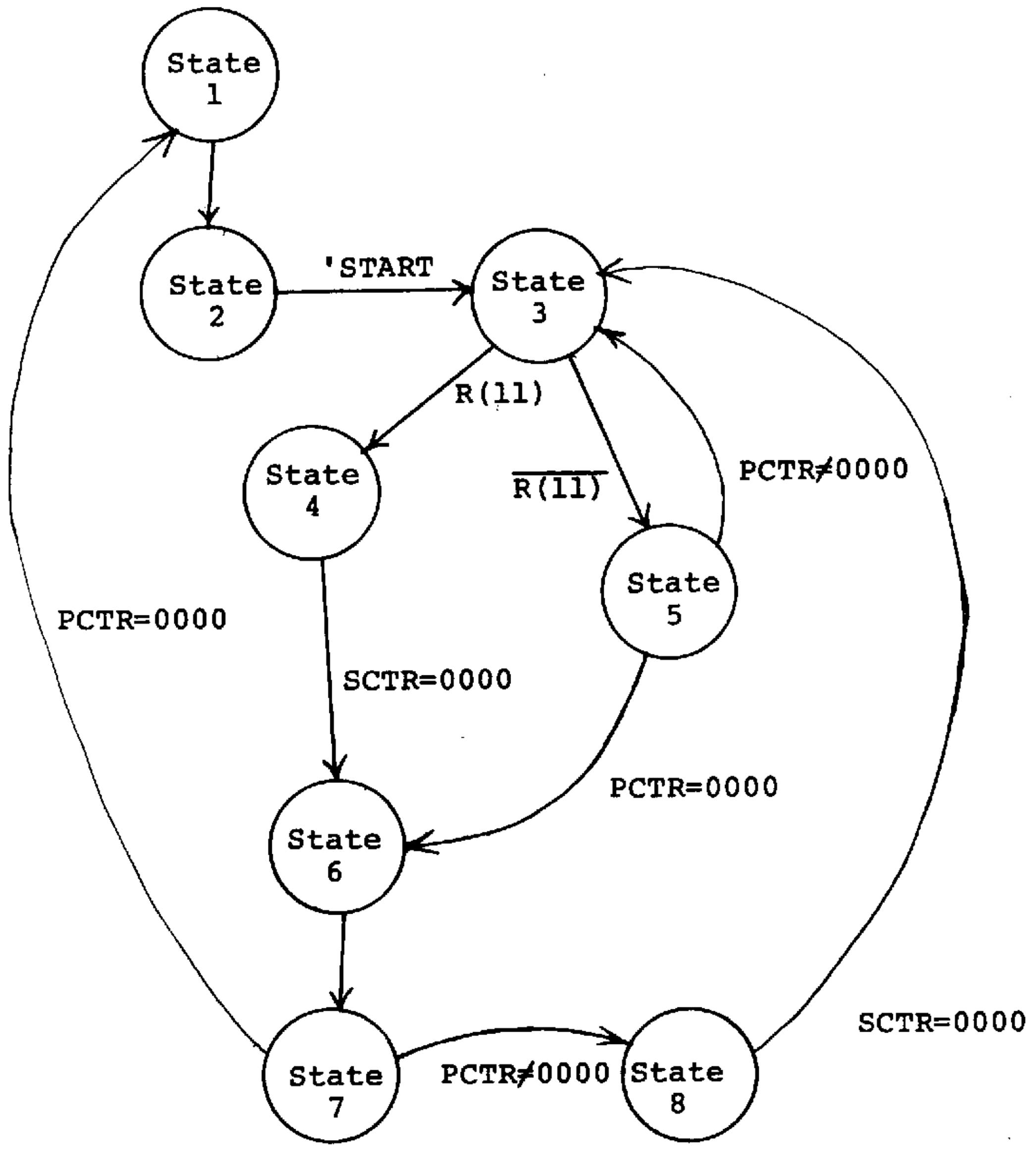


Figure 3.2
State Diagram for Multiplier Example

UNIT	OPERATION	CONDITION
FINISH		
	SET	S7*(PCTR=0000)
	RESET	S2*('START)
SIGN		
	LOAD FROM ISIGN	S2*('START)
R(1-10)		
	RESET	S2*('START)
R		
	LEFT SHIFT ENTER R(1)	S8*-(SCTR=0000)
	LEFT SHIFT ENTER CARRY	S5*(-PCTR=0000) +S6*UNCONDITIONAL
	LEFT SHIFT ENTER SUM	S4*-(SCTR=0000)
R(11-20)		
	LOAD FROM AIN	S2*('START)
B		
	LEFT SHIFT ENTER B(1)	S4*-(SCTR=0000)
	LOAD FROM BIN	S2*('START)
SCTR		
	DECREMENT	S4*-(SCTR=0000) +S8*-(SCTR=0000)
	LOAD FROM 1010	S3*-(-R(11))
	LOAD FROM 1010	S7*-(PCTR=0000)
PCTR		
	DECREMENT	S3*UNCONDITIONAL
	LOAD FROM 1010	S2*('START)
CARRY		
	RESET	S3*UNCONDITIONAL
	LOAD FROM NCARRY	S4*-(SCTR=0000)

Table 3.6
CADSS Unit Operation Table

CHAPTER IV.

THE CADSS SIMULATOR

4.1 INTRODUCTION

The object of all simulation is to eliminate errors on paper rather than in the laboratory. In this problem area the digital system designer has a distinct advantage. Properly interconnected building blocks eliminate individual component peculiarities and allow the designer to use these blocks, within practical limitations, to model only the system's functional performance. The approach to this model on the circuit level introduces macroscopic strains on the full system simulation. Bipolar circuitry and the device level approach using a finite number of nodes and devices increases the complexity of simulation. The great number of Boolean equations used by these methods increases the difficulty even more. Only the logic level approach allows functional operation to be modeled in terms of the bit pattern expressed only as "1's" or "0's". This convenience level allows such variables as rise times, fall times, and voltage levels to be ignored except when modifying previously built systems. The 1's and 0's can develop a logic simulation program which applies fan-out limits to flag every overload circuit or generate all printed circuit board interconnection lists. These attributes fulfill the major objective of proving that the system works before construction begins.

The anticipated use of the CADSS simulator (for testing system descriptions during design) led to several simplifications. Since the present CADSS programs are concerned with the design of synchronous systems, signal propagation delays are ignored. The result is a register transfer simulation that may be used to detect and locate logical errors in the system description. In addition, the value of each unit is monitored and more than one change during a simulation cycle causes an error message. This will detect some timing hazards in the system.

The simulator may be initialized in any state and with any desired initial unit values. As an aid to isolating logical errors, several trace options are available:

1. Statement trace--The number of every active statement is printed during simulation

2. Unit value trace--The values of the units specified are printed after each simulation cycle.

3. Potential hazards trace--Any timing hazards detected are listed during simulation

4.2 SIMULATOR CONTROL CARDS

The following control cards illustrate the sequence of cards necessary for proper execution of the CADSS Simulation.

```
// EXEC COMPL

     .

     .

     .

     (Program)

     .

     .

     .

/*

/&

// EXEC CADSIM

     (Title card of 80 characters)

     (Heading format to include the desired output variables

      with a length of 120 characters)
     (Initialization Card)

$DO  N

$START L,L1,L2

$RUN

 ITRACE,JTRACE,KTRACE    (format 3I5)

 BLANK CARD

/*

/&
```

In the previous example of the Simulator Control Cards, each has its own special meaning. The first control card // EXEC COMPL begins in column one of the input card. Spaces in the card indicate blank columns not punched by the key punch operator. This is inclusive of all control cards. This control card complies the program to be simulated. The /* and /& cards are key punched in columns one and two of their respective cards. These cards terminate the compilation process.

The control card // EXEC CADSIM initially begins the simulation of the desired program.

The TITLE CARD is an input card 80 characters in length which allows the user to place a title at the top of the simulated program.

The HEADING FORMAT card consists of two input cards totaling 120 characters in length which include the desired output variables to be printed on a page. Allocation for the number of variables to be printed must be determined by the individual format of each variable and its maximum size. The output names are compared with the name table to determine the data index and dimension of each name. The number of blanks necessary to format the output below the Print Title is computed and these values are then stored in the print list.

The INITIALIZATION CARD defines input variable names cited at the beginning of the compiled program. The name of the variable followed by an equal sign followed by the desired input number in binary defines an input variable. If more than one variable exists, a comma is placed after the previous number and the next variable is defined in the same way as the first. All unit values are set to zero before the data initialization cards are read. As each card is read, the name is matched to a name in the name table and the desired values are stored in the DATA1 location specified in the table. If the input variable list does not exist in the compiled program, then the initialization card may be omitted.

The $DO N control card sets the maximum number of simulation cycles. The integer N can be replaced by any other integer less than or equal to 999. When the integer is omitted, 100 simulation cycles are assumed.

The $START L,L1,L2 control card is next read. This card allows the system to be initialized in any state (or in several states) at the same time. The card lists the labels of all states that are to be initialized in the potentially active condition. The labels are matched to entries in the label table and the activities of the corresponding states are set to one (1) meaning potentially active.

The $RUN control card designates to the complier that all previous control cards are correct. Simulation begins after the next control card.

The control card ITRACE, JTRACE, KTRACE has a Fortran format of 3I5. Only the values of one (1) or zero (0) can be assumed by any one of the three variables listed. If zero (0) is assumed by any one of the three variables listed. If zero (0) is assumed, then the key punch operator need not punch any character in that integer field format. Blanks designate zeros. Also in the actual key punching operation, no commas are punched in this control card. These three variable names are trace options to aid the isolation of logical errors since the simulator may be initialized in any state with any desired initial unit values. The trace options available are as follows:

1. ITRACE specifies a STATEMENT TRACE. The number of every active statement is printed during simulation.

2. JTRACE specifies a UNIT VALUE TRACE. The values of the units specified are printed after each simulation cycle.

3. KTRACE specifies a POTENTIAL HAZARDS TRACE. Any timing hazards detected are listed during simulation.

By initializing a simulation to conditions existing before an error and tracing, these options make it possible to determine the exact statements causing the errors.

These previous control cards complete the CADSS initialization. The BLANK CARD is an input card containing 80 blanks which is used by the RDATA subroutine to terminate the simulated program. Otherwise, this card is a non-blank input data card containing no separation between individual data sets described by the "INPUT:" list of variables to be read in by the CADSS Simulator.

The /* and /& control cards are key punched in columns one and two of their respective separate input cards. These cards terminate the program.

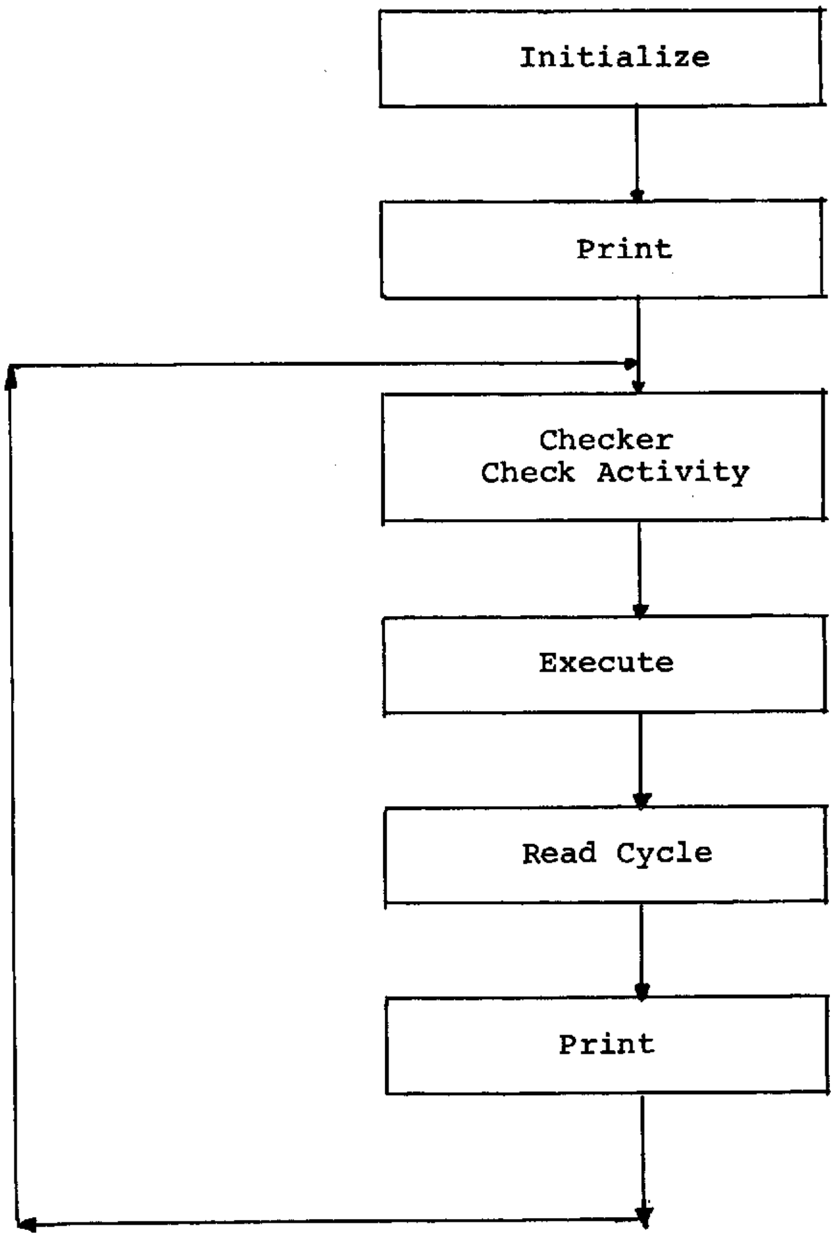
4.3 OPERATION OF SIMULATOR

To apply the tracing error aids and the ease in learning and use of this simulator, several phases of synchronous operations govern the actual processing of the simulated system. The pictorial description on the following page illustrates the sequenced operations being processed.

Figure 4.1

SEQUENCED OPERATION

OF THE CADSS SIMULATOR



During the initialization phase, the simulator program reads the tables and lists generated by the compiler. In order to simulate concurrent operations, two data arrays are used to store the values of the defined units. The DATA1 array contains the current values of the units and is initially set to values specified by cards read during this phase. The trace options, a cycle limit, and the initially active statements are also read from cards.

It is often desirable to observe different units during debugging runs. This is accomplished by reading a print list during simulation initialization to specify the units to be listed. In this way, any different units of the system may be observed without recompiling the system description.

After initialization, the simulator enters the check phase. At this time, the condition lists of all potentially active statements are executed interpretively. The statements having true condition lists are marked in the statement table.

During the operation phase, the operation lists of all active statements are executed. The current values of the units are not changed, but the new values are stored in the DATA2 array. Thus, all operations are performed using the same values of the defined units.

When the next values of all units are stored in the DATA2 array, the simulator enters the update phase. The values are transferred from DATA2 to DATA1 and transitions of units are noted in the DATA2 array. Any input or output operations specified are performed and the activities in the statement table are reset to potentially active. The program then returns to the check phase.

4.4 Simulation Example

The binary multiplier described in Chapter 3 was selected as an example of the simulator operation. In order to provide a more legible output the 20 bit R register was divided into two 10 bit registers, AR and AL. AR was used to contain the original multiplier and the least significant half of the result. AL is initially zero and contains the most significant half of the result upon completion.

After compilation the simulator was initialized as explained in Section 4.2. The data from the initialization cards used is:

```
MULTIPLICATION BY REPEATED SERIAL ADDITION
AL   AR   B   SCTR  PCTR  SUM  SA  SB  SIGN  FINISH
-
-

$DO
$START  STARTPOINT, LIST
$RUN
-
-

10000000000100000010000
/*
//
```

The simulator was run for several different trace options to verify the system description. The first example (Figure 4.2) illustrates a unit value trace in which the values of all units specified are printed after each simulation cycle. Note that at point A the initial values are read in to the AR and B registers and the counters are initialized.

Section B of Figure 4.2 illustrates an add cycle where registers B and AL are shifted and added with the sum going into AL. SCTR counts bit shifts during the add cycle. Section C shows the system shifting the multiplier in AR looking for a 1. Finally at point D the PCTR has been decremented to zero and the FINISH flag is set. The product (16) is then displayed in the AL and AR registers.

Figure 4.3 shows the results of several multiplications in which only the initial data and the result are displayed. This simulation of several binary multiplications verifies the system specified in the original description. If errors are noted in a simulation, a statement trace can be used to determine the particular statement of the system description that is in error. Correction of the error followed by additional simulation will quickly lead to a verified system description.

The simulation of 11 multiplications shown in Figure 4.3 required 694 clock cycles of the target system. Compilation and table generation required 17.6 seconds of CPV time on an IBM 360/50 operating in a multiprogramming sys-

```
AL          AR          B           SCTR  PCTR  SUM  SA  SB  SIGN  FINISH
0000000000  0000000000  0000000000  0000  0000  0    0   0   0     0
0000000000  0000000000  0000000000  0000  0000  0    0   0   0     0
0000000000  0000000001  0000010000  0000  1010  0    0   0   0     0        A
0000000000  0000000001  0000010000  1010  1001  0    0   0   0     0
0000000000  0000000001  0000001000  1001  1001  0    0   0   0     0
0000000000  0000000001  0000000100  1000  1001  0    0   0   0     0
0000000000  0000000001  0000000010  0111  1001  0    0   0   0     0
0000000000  0000000001  0000000001  0110  1001  1    0   0   0     0
1000000000  0000000001  1000000000  0101  1001  0    0   0   0     0
0100000000  0000000001  0100000000  0100  1001  0    0   0   0     0
0010000000  0000000001  0010000000  0011  1001  0    0   0   0     0        B
0001000000  0000000001  0001000000  0010  1001  0    0   0   0     0
0000100000  0000000001  0000100000  0001  1001  0    0   0   0     0
0000010000  0000000001  0000010000  0000  1001  0    0   0   0     0
0000010000  0000000001  0000010000  0000  1001  0    0   0   0     0
0000001000  0000000000  0000010000  0000  1001  0    0   0   0     0
0000001000  0000000000  0000010000  0000  1001  0    0   0   0     0
0000001000  0000000000  0000010000  0000  1000  0    0   0   0     0
0000000100  0000000000  0000010000  0000  1000  0    0   0   0     0
0000000100  0000000000  0000010000  0000  1000  0    0   0   0     0
0000000100  0000000000  0000010000  0000  0111  0    0   0   0     0
0000000010  0000000000  0000010000  0000  0111  0    0   0   0     0
0000000010  0000000000  0000010000  0000  0111  0    0   0   0     0
0000000010  0000000000  0000010000  0000  0110  0    0   0   0     0
0000000001  0000000000  0000010000  0000  0110  1    0   0   0     0
0000000001  0000000000  0000010000  0000  0110  1    0   0   0     0
0000000001  0000000000  0000010000  0000  0101  1    0   0   0     0
0000000000  1000000000  0000010000  0000  0101  0    0   0   0     0
0000000000  1000000000  0000010000  0000  0101  0    0   0   0     0        C
0000000000  1000000000  0000010000  0000  0100  0    0   0   0     0
0000000000  0100000000  0000010000  0000  0100  0    0   0   0     0
0000000000  0100000000  0000010000  0000  0100  0    0   0   0     0
0000000000  0100000000  0000010000  0000  0011  0    0   0   0     0
0000000000  0010000000  0000010000  0000  0011  0    0   0   0     0
0000000000  0010000000  0000010000  0000  0011  0    0   0   0     0
0000000000  0010000000  0000010000  0000  0010  0    0   0   0     0
0000000000  0001000000  0000010000  0000  0010  0    0   0   0     0
0000000000  0001000000  0000010000  0000  0010  0    0   0   0     0
0000000000  0001000000  0000010000  0000  0001  0    0   0   0     0
0000000000  0000100000  0000010000  0000  0001  0    0   0   0     0
0000000000  0000100000  0000010000  0000  0001  0    0   0   0     0
0000000000  0000100000  0000010000  0000  0000  0    0   0   0     0
0000000000  0000010000  0000010000  0000  0000  0    0   0   0     0        D
0000000000  0000010000  0000010000  0000  0000  0    0   0   0     1
```

Figure 4.2
Simulation with Unit Value Trace

```
AL          AR          B           SCTR  PCTR  SIGN  FINISH
0000000000  0000000000  0000000000  0000  0000  0     0
0000000000  0000000001  0000010000  0000  1010  0     0
0000000000  0000010000  0000010000  0000  0000  0     1     16X1 =16
0000000000  0000010000  0000010000  0000  1010  0     0
0000000000  0100000000  0000010000  0000  0000  0     1     16x16 = 32
0000000000  0100000000  0100000000  0000  1010  0     0     256x256 =
0001000000  0000000000  0100000000  0000  0000  0     1       65,536
0000000000  1000000000  1000000000  0000  1010  0     0
0100000000  0000000000  1000000000  0000  0000  0     1
0000000000  1111111111  1111111111  0000  1010  0     0
1111111110  0000000001  1111111111  0000  0000  0     1
0000000000  0000000111  0000000101  0000  1010  0     0     7x5
0000000000  0000100011  0000000101  0000  0000  0     1     = 35
0000000000  0000001000  0000001111  0000  1010  0     0
0000000000  0001111000  0000001111  0000  0000  0     1
0000000000  0000001010  0000001110  0000  1010  1     0
0000000000  0010001100  0000001110  0000  0000  1     1
0000000000  0000011110  0000001111  0000  1010  1     0
0000000000  0111000010  0000001111  0000  0000  1     1
0000000000  0001011010  0001110101  0000  1010  0     0
0000001010  0100100010  0001110101  0000  0000  0     1
0000000000  0000101010  1000011111  0000  1010  0     0
0000010110  0100010110  1000011111  0000  0000  0     1


  TERMINATED DUE TO LACK OF INPUT DATA
```

Figure 4.3
Simulation without Trace

tem under 360/OS. The simulation required 50.3 seconds
on the same system.

CHAPTER V

DEVELOPMENT OF COMPLETE DESIGN

In order to complete the design of a digital system using the data generated by the CADSS Table Generator program the following steps are necessary:

1. State Assignment.
2. Development of input equations to state Memory Elements.
3. Specification of particular devices for use as registrars, counters, etc.
4. Reduction of Conditon Lists to minimized Booleon equations.
5. Implementation of equations using specified logic elements.

Programs have been written for each of these steps but the operation of the complete design system is not yet considered satisfactory. In the following sections the progress made on each step is discussed.

5.1 STATE ASSIGNMENT

In any sequential system the operations to be performed depend on both the state of the system and the present inputs to the system. The state of the system is normally stored in a set of flip flops which are modified as the system changes state. The states of the system are assigned unique codes of flip flop combinations which can then be gated to determine the operations and next states of the system. The number of gates required in a system is very sensitive to the assignment used to relate states to memory element configurations. The problem of determining the coding of memory elements which will result in the least total cost of logic gates is normally referred to as state assignment.

Several alogrithms for generating "good" state assignments are known. Most require a large amount of computer storage and time and are limited in the number of states and system inputs which can be used. After considering several methods a modified form of Dolotta's alogrithm for large state tables was programmed in Fortran IV (MS Thesis, M. Boutte, Texas A&I, 1971). The program was tested and finally used as a subroutine for generation of state assignments for systems described in the CADSS language. (MS Thesis, P. Santhanam, Texas A&I, 1973)

When running the state assignment program in a 64K byte partition on an IBM 360/50 the largest system which could be accomodated was composed of 10 ineternal states and 2 inputs. Run times for 10 state systems were on the order of 10 to 15 seconds.

The limited number of states and inputs which could be used limited the CADSS programs to artifical examples. One such example is shown in figure 5.1. In order to display the operation of the system an artificial state was added to the system description. As shown in the state table this state (5) in not connected to the other 4 states of the system.

The state assignment program was run for the system described and the results are shown in figure 5.2. The program correctly assumed that 3 memory elements would be necessary in order to encode 5 states. In analyzing the state transitions for the system the program recognized that the fifth state was not connected to the other 4 and therefore produced a state assignment requiring only 2 memory elements.

Although the programs developed will find a good state assignment for systems having a limited number of states and inputs they proved inadequate for typical system designs where the number of states and inputs is an order of magnitude greater than could be accomodated. In an effort to resolve this difficulty several other state assignment alogrithms were acceptable in terms of computer memory and run time requirements.

Reconsideration of the basic problem of state assignment produced several conclusions. First, early research in this area was performed at a time when memory elements (constructed from vacumn tubes) were unquestionably much more expensive than gates (constructed from diodes). The first step of all early researchers was to first minimize the number of memory elements required and then develop a method for coding the states to minimize the amount of logic required. At the present time integrated circuit memory elements are comparable in cost to gates. It seems possible that a strategy using more memory elements than required might prove less expensive if the amount of logic could be reduced. A preliminary study in this area has been initiated but no definite results are available at this time.

The second conclusion reached was that if a method of generating the next state functions for the memory elements could be found which was not sensitive to the particular state assignment used then the state assignment could be made arbitrarily. Such a method was proposed and is discussed in the next section.

```
                         CADSS/SYSTEM   360/40

     1         0         INPUT: X1, X2 $
     2         0         OUTPUT: Z $
     3         0       $ DEND $
     4         1         CONCURRENT:
     5         1         S1:  WHEN -X1*-X2 THEN GO TO S2 $
     6         2         S2: WHEN -X1*X2 THEN SET Z, GO TO S3 $
     7         3             WHEN X1*-X2 THEN GO TO S1 $
     8         4         S3: WHEN -X1*-X2 THEN GO TO S4 $
     9         5         S4: WHEN -X1*X2 THEN GO TO S3 $
    10         6             WHEN X1*-X2 THEN GO TO S1, RESET Z $
    11         7         CONCURRENT:  LIST:  THEN DISPLAY $
    12         8         THEN ACCEPT $
    13         9       $ TABLES, STORE 201 $
```

```
      CADSS STATE AND UNIT OPERATION TABLES

      TAPE ID  201

PRESENT STATE   NEXT STATE          CONDITION

        1
                       2        (-X1*-X2)

        2
                       3        (-X1*X2)
                       1        (X1*-X2)

        3
                       4        (-X1*-X2)

        4
                       3        (-X1*X2)
                       1        (X1*-X2     )

        5
                               PERSISTANT
```

Figure 5.1
State Assignment Example Description

```
THE ARRANGED TABLE FOR BOUTTE'S PROGRAM IS
    1    2   -1   -1   -1   -1   -1   -1   -1   -1   -1
    2   -1   -1    1   -1    3   -1   -1   -1   -1   -1
    3    4   -1   -1   -1   -1   -1   -1   -1   -1   -1
    4   -1   -1    1   -1    3   -1   -1   -1   -1   -1
   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1
   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1
   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1
   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1
   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1
   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1
THE NO. OF ROWS IS     4
     STATE                    MEMORY ELEMENTS

      1  2-1  -1-1  -1-1  -1-1

      2 -1-1   1-1   3-1  -1-1

      3  4-1  -1-1  -1-1  -1-1

      4 -1-1   1-1   3-1  -1-1
```

STATE	MEMORY ELEMENTS		
	1	2	3
1	0	0	0
2	0	1	1
3	0	0	1
4	0	1	0

Figure 5.2
Computer Generated State Assignment

Since the state assignment programs developed were not satisfactory and since two attractive lines of research are being followed, the state assignment programs were not incorporated into the CADSS system.

5.2 DEVELOPMENT OF MEMORY ELEMENT INPUT EQUATIONS

Due to the lack of a satisfactory solution to the state assignment problem no programs were written to obtain the input equations for the state memory elements. It is believed at this time that the use of arbitrary state assignments with programmable logic arrays or read only memories will provide a very attractive design automation system. Programmable logic arrays (PLA) or programmable read only memories (PROMs) are os recent commercial introduction and offer a convenient and relatively inexpensive method of implementing complex logic functions. The information stored in these meories is introduced into the memory such that the information is semi-permanent or permanent. These memories are classified according to tye type of construction. There are three major classifications, as follows.

1. Transformer core type. In this type, given word line either threads through or bypasses a core. One of these conditions is designated as a binary 1 whereas the other condition is a 0.

2. Card-capacitor type. A set of words are etched on one printed circuit board. Due to formation of capicators at the junctions, a selected word line is capacitively coupled with the sense line (output line).

3. Integrated circuit type. In this type the coupling between the word line and the putput line is by means of diodes or transistors. Sometimes only the diodes (or transistors) at the 1 junction are manufactured. Sometimes Read-only Memories are manufactured with diodes at all the junctions. The o - output or no-diode positions in such case are obtained by breaking or burning the input conneciton.

Out of all these types it can be seen that the integrated circuit type is the most convenient, as it can be programmed as desired by the user. PROMs are available from several different manufacturers as off the shelf items. A typical device having 5 address lines and 8 outputs is represented in figure 5.3. This memory is organized in 32

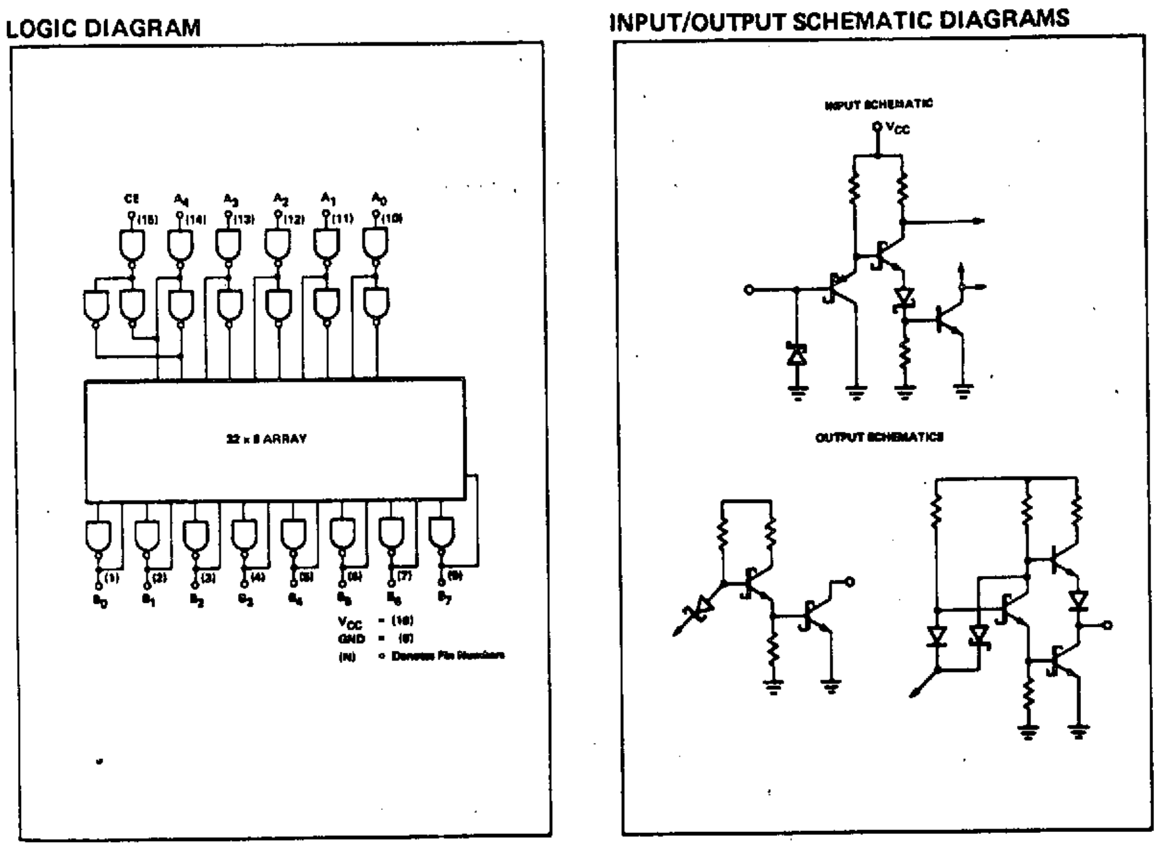


Figure 5.3
Programmable Read Only Memory

words of 8 bits/word. PROMs are supplied by the manufacturer with all bits set to either 1 or 0. Programming (or initial storage of information) is accomplished by setting the address of the word to be programmed on the address lines and then supplying excess current or voltage to the data lines to burn out the diodies, transistors, or fuse links for selected bits. Commercial programmers are available for this operation.

At the present time PROMs are priced between \$20.00 and \$30.00 each in lots of 10 to 100. Using current costs PROMs are somewhat more expensive than NAND/NOR combinational logic. There are however, several related advantages in using PROMs for implementation of next state functions.

1. Gate density is much higher in PROMs. A 16 pin DIP may conatin 32 eight bit words of memory. To provide the same potential amount of logic using unminimized NAND/NOR gates would require 32 five input gates and up to 256 inverters. This amounts to 30 to 60 IC packages.

Obviously minimization of the switching function will reduce the number of NAND/NOR gates drastically, but even if a 10 to 1 reduction is obtained the PROM implementation will require 1/3 to 1/6 the number of IC packages.

2. Construction time of prototype or limited production systems is much less for PROMs. When systems are to be constructed by manual methods a PROM implementation will require only 20 to 30% of the number of connections required for NAND/NOR technology. The programming or storage of information in the PROM may be accomplished very rapidly using a computer controlled programmer.

3. The system design time will be reduced by using PROMs since minimization of functions is not necessary.

4. System reliability will be increased due to the smaller number of interconnections.

5. System troubleshooting and repair will be easier since the logic equations implemented will not be in minimized form.

6. The required parts inventory would be decreased since the normal variety of NAND and NOR gates would not be required.

In view of these factors as well as an anticipated drop in PROM prices as the technology and demand develop, this method of design appears to deserve future study.

5.3 SPECIFICATION OF PARTICULAR DEVICES

The CADSS language allows the designer to designate general purpose registers and then specify the operations these registers are to perform. The Unit operation table (described in Chapter 3) lists all operations (such as shift right, reset, load, increment, etc.) each register is to perform. From this table a designer can specify a particular register design which will perform the desired functions.

By including a library of standard registers together with a description of the functions each will perform it is possible to program the computer to compare the functions each register in a system is required to perform with the functions specified in the library. The program can then choose the least expensive standard register which will perform the required functions.

A program to abstract the functions required for each register was written and tested (MS Thesis, A. Sharma, Texas A&I, 1970). During testing using several CADSS program descriptions it was found that in many cases registers must be separated into several subregisters which have different requirements. For example a portion of a register may require parallel loading while the remainder of the same register will not. The program was modified to specify the requirements of each bit of a register in order to accomodate this situation.

The final result of this portion of the research project was a program which accepts the CADSS input description of a system and produces tables specifying the operations each bit of each register is to perform. A program to match these tables with library descriptions of standard registers is being developed but is not yet complete.

5.4 Reduction of Condition Lists to Minimized Boolean Equations

The condition lists of a CADSS description specify the conditions required in order for particular operations to be performed. As written by the designer they are already in Boolean form but they must be minimized and converted to a form that can be used more easily by the computer. Programs for performing logic reduction of generalized switching functions were written (MS Thesis, B.J. Patel, Texas A&I University, 1970). The method programmed was a modified form of McCluskey's tabular method.

This program was later modified to read the tables produced by the CADSS compiler and produce output tables to be used by a NAND/NOR implementation program (MS Thesis, G.I. Mehta, Texas A&I University, 1973). This program produces lists describing each condition list in standard sum of products form. The format of the internal minterm list is shown in figure 5.4. An example of the output lists is shown in figure 5.5.

At the present time each condition list is minimized independantly. Consideration was given to development of a method of minimizing all condition lists as a multiple output function. Examination of a large number of CADSS descriptions indicates that there are very few cases where minimization as a multiple output function would reduce the logic requirements significantly. Since both memory and computer run time requirements would be greatly increased no further efforts were made in this direction.

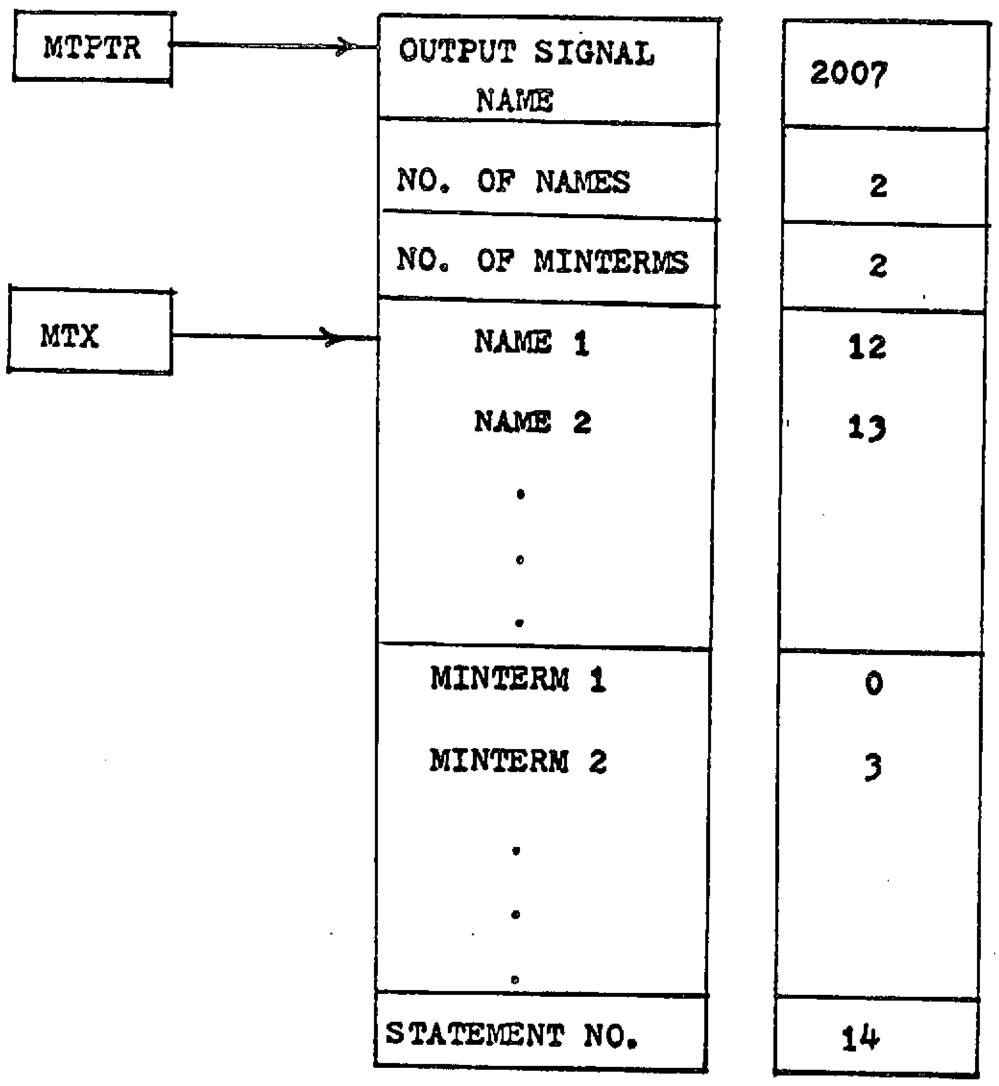


Figure 5.4

Minterm List (Program Format)

```
          TAPE ID   201

        SRIBL VALUES ARE AS FOLLOWS.

        3    4    2    4

        TSIG VALUES ARE AS FOLLOWS.

        5    3    4    6    5    2    6    7    6    7

        4    8    7    8

        TSIG VALUES ARE AS FOLLOWS.

        2    2    3    4    2    1    4    5    3    5

        3    6    4    6

        MINTERMS      NUMBER OF ONES IN EACH TERM
          4             1
          5             2
          6             2
          7             3

        THIS IS A MINTERM CHART.

     2001    3    4    2    3    4    4    5    6    7

        2
```

Figure 5.5

Minterm List (Printed Format)

During the examination of the system descriptions it was noted that some system conditions (such as a counter being equal to zero) occur in many condition lists. In order to eleminate duplication of logic these conditions were identified as special signal names and generated only once. For example in processing a condition list:

(SCTR = 100*SA)

SCTR = 100 is identified as a special signal. The table of special signals is searched and if this signal has been entered and assigned a signal name, that name is used in the minimization program. If the signal has not been entered in the table, it is entered and assigned a name for use in later processing. The signal names are assigned in sequence beginning at 1001 as shown in Figure 5.5.

5.5 Implementation of Equations in NAND/NOR Logic

The final step in producing a system design is the implementation of the Boolean equations describing the system in terms of a standard logic family. Several methods were considered and a procedure proposed by D.T. Ellis (IEEE Transactions on Electronic Computers, Vol. EC-14, October, 1965, pp. 701-705) was selected. A computer program using this method was written and tested (M.S. Thesis, C. S. Hou, Texas A&I University, 1969). This program was later expanded and interfaced to the tables produced by the CADSS system.

A set of NAND logic circuits is included in Figures 5.6 and 5.7 to illustrate the format of the generated tables. Since the NAND logic is generated only for the condition lists of the description an example consisting of several Boolean equations was prepared. This example description is shown in Figure 5.6. This system description was run on the CADSS compiler, the minterm program and the NAND logic program.

Figure 5.7a shows the output produced by the minterm and NAND programs. The statement is first identified and the signals used as inputs are identified. The decimal values of the minterms in a standard sum-of-products form are given. In example 5.7a the standard form of the minterm would be:

$$f(A,B,C) = m_4 + m_5 + m_6 + m_7$$

Below the minterm description is a list specifying the NAND logic required to implement the expression. Since the system will eventually include NORs and other logic elements,

```
          CADSS/SYSTEM   360/40

   0      C THIS IS A CADSS EXAMPLE PROGRAM TO TEST FEASIBILITY OF MINT
   0      C
   0        INPUT: A, B, C, D, E  $
   0        OUTPUT: T $
   0      $ DEND $
   1      C
   1        SEQUENCED: STARTPOINT:
   1            WHEN A+ A*B*C THEN SET T   $
   2            WHEN -A + A*B*D*B   THEN SET T    $
   3            WHEN -A*-B*-C*-D THEN SET T $
   4            WHEN -A*-B*-C*D  THEN SET T $
   5            WHEN -A*-B*C*-D  THEN SET T $
   6            WHEN -A*-B*C*D   THEN SET T $
   7            WHEN -A*-D*-C*B  THEN SET T $
   8            WHEN B*-A*D*-C   THEN SET T $
   9            WHEN B*-D*C*-A   THEN SET T $
  10            WHEN -A*B*C*D    THEN SET T $
  11            WHEN -D*-B*A*-C  THEN SET T $
  12            WHEN A*B*-C + A*-B*D + B*D*-E + E*C*-D THEN SET T  $
  13      $ TABLES, STORE 201 $
```

Figure 5.6

Description for NAND Example

```
( 1)           STATEMENT NUMBER:  1

                INPUT(S):

                          A

                          B

                          C

                MINTERM(S):          1   3   5   7


  NAND GATE STRUCTURE


     1          1      2001
        A

     1          1      2002
        2001
```

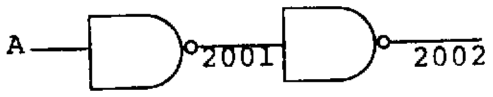


Figure 5.7a

Minterms, NAND Structure and Logic Diagram
for Statement 1

```
( 2)              STATEMENT NUMBER:  2

                  INPUT(S):

                            A

                            B

                            D

                  MINTERM(S):        0   2   4   6   7


   NAND GATE STRUCTURE


      1          1      2001
        A

      1          1      2002
        2001

      1          2      2003
        D
        B

      1          2      2004
        2002
        2003
```

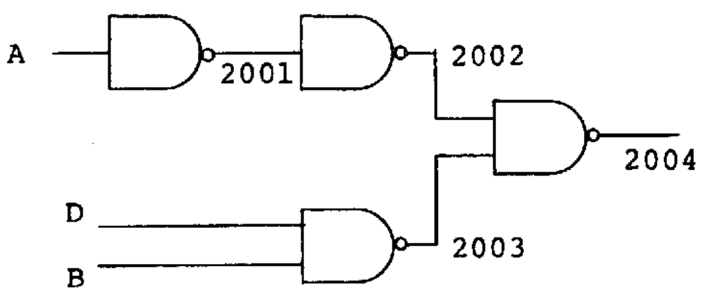


Figure 5.7b

Minterms, NAND Structure, and Logic Diagram for Statement 2

```
                    INPUT(S):
                                    A
                                    B
                                    C
                                    D
                    MINTERM(S):          0

    NAND GATE STRUCTURE

    1            1         2001
       D
    1            1         2002
       C
    1            1         2003
       B
    1            1         2004
       A
    1            4         2005
       2001
       2002
       2003
       2004
    1            1         2006
       2005
```

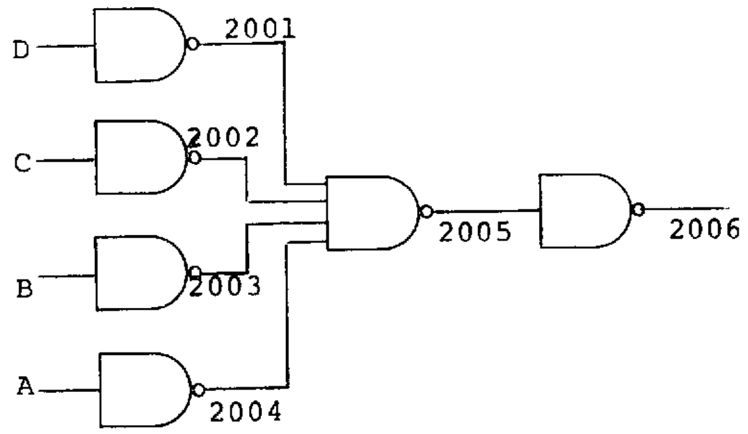


Figure 5.7c

Minterms, NAND Structure and Logic Diagram for Statement 3

```
                         INPUT(S):
                                      A
                                      B
                                      C
                                      D
                         MINTERM(S):        8


NAND GATE STRUCTURE


   1           1         2001
      C
   1           1         2002
      B
   1           1         2003
      A
   1           4         2004
      2001
      2002
      2003
      D
   1           1         2005
      2004
```

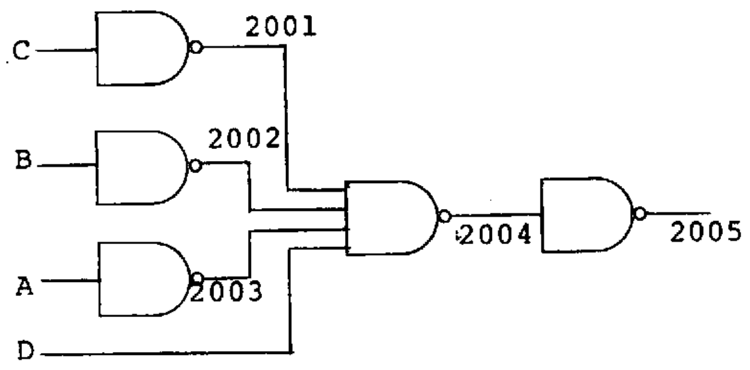


Figure 5.7d

Minterms, NAND Structure and Logic Diagram for Statement 4

```
(12)              STATEMENT NUMBER: 12

                     INPUT(S):

                               A

                               B

                               C

                               D

                               E

                     MINTERM(S):         3   9  10  20  11  13  14  19  21  22

                     MINTERM(S):        25  15  23  27  29


 NAND GATE STRUCTURE


       1          1        2001
          C

       1          3        2002
          2001
          B
          A

       1          1        2003
          E

       1          3        2004
          2003
          D
          B

       1          1        2005
          D

       1          3        2006
          2005
          E
          C

       1          3        2007
          2003
          D
          A
```

Figure 5.7e

Minterms, NAND Structure and Logic Diagram
for Statement 12

1		3	2008
	2001		
	D		
	A		
1		1	2009
	B		
1		4	2010
	2009		
	E		
	C		
	A		
1		6	2011
	2002		
	2004		
	2006		
	2007		
	2008		
	2010		

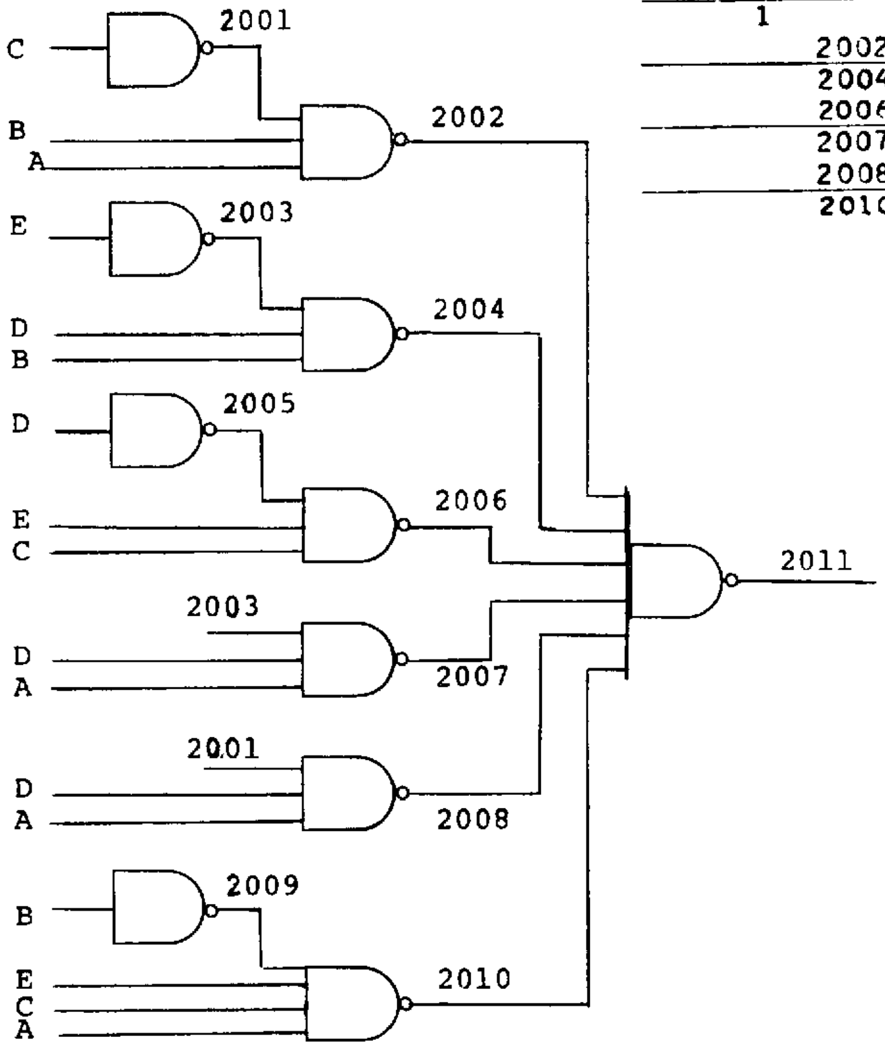


Figure 5.7e (continued)

Minterms, NAND structure and Logic Diagram for Statement 12

the first entry on each now identifies the type of gate with 1 representing a NAND. The second entry is the number of inputs to this gate and the final entry in the line is a signal name (beginning with 2001(assigned to the output of the gate. Below this line the signal names of the gate inputs are listed. Following this, with the same format, are descriptions of any other gates required to implement the desired function. The logic diagram specified has also been drawn on each figure.

The program runs in a 64K partition on an IBM 360/50. Execution time for the example of figure 5.6 was 8.9 seconds for the compilation and 10.6 seconds for generation of minterms and the NAND gate structure. Execution time for compilation and logic generation is directly proportional to the number of condition list statements in the CADSS description.

One oversight in the NAND generation program is illustrated in figure 5.7a. The method does not recognize that two inverters in series can be combined and eliminated. A correction has been found but has not yet been incorporated in the program.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

In retrospect, the project to produce a complete computer-aided design system was much too ambitious for the resources available. Industrial programming support groups have spent tens of man-years producing portions of design automation systems. Six Master's Theses as well as several student projects and small amounts of faculty time have produced several parts of a system which should prove of assistance to a digital designer. Several important areas must be completed before a completely automated system will be available. Several interesting research areas have been identified and are currently under study.

6.1 Current Applications

At its present stage of development the CADSS system will provide assistance to a digital designer. A system can be easily described provided it does not use some of the recently developed integrated circuits now available such as decoders, arithmetic elements or special types of counters. A system description can be simulated at a functional level to determine if any errors exist in the logic of the design.

After a correct description has been obtained and verified the state and unit operation tables provide the first step toward logic implementation. The designer must perform the state assignment. The generation of NAND/NOR logic for control of the defined registers of the system may be performed by the CADSS system although it will not provide an absolute minimum circuit. The selection of standard registers, counters, etc. must be done by the designer with the aid of a CADSS table specifying the functions each bit of an element is to perform.

6.2 Use of New Functions

The types of digital semiconductors available as stock items have increased rapidly in recent years. Single IC elements are now available to perform arithmetic functions, decode values for display, count in various codes, and many other functions needed in digital systems. These new integrated circuits can significantly reduce both the cost and the design time for many systems. Any design automation system should provide a means of including new functions in the list of available devices.

The CADSS compiler, table generator and simulator programs are written as syntax directed programs. The register operations (shift, load, count, etc.) are defined by the Fortran program itself. In order to add new operations it is necessary to rewrite significant portions of both the compiler and the simulator. The amount of reprogramming required makes it impractical for a designer to add new elements to the CADSS description.

In order to modify the CADSS programs to allow a user to easily add new elements it will be necessary to change the method of compilation and simulation and rewrite the compiler as a table directed system.

In essence, a table directed system contains the operations to be performed in a table which can be easily modified or augmented by the user. For the CADSS programs this will require a table with an entry for each type of element defined. The table entry will contain the number and name of all inputs and outputs of the element and the input/output relationship for the device. For example, a 4 bit serial in/parallel out shift register might be defined as:

Input: Shift, Data
Output: A0, A1, A2, A3

$$A3 = \text{Shift} * A2 + \overline{\text{Shift}} * A3$$

$$A2 = \text{Shift} * A1 + \overline{\text{Shift}} * A2$$

$$A1 = \text{Shift} * A0 + \overline{\text{Shift}} * A1$$

$$A0 = \text{Shift} * \text{Data} + \overline{\text{Shift}} * A0$$

Once this information defining the operation of the shift register has been entered by the user any number of shift registers can be used in the system description.

Both the compiler and the simulator programs must be rewritten in order to use the tables defining the operation of elements. Different methods for implementing a table driven version of the CADSS programs are under consideration at this time.

6.2 Extension of Logic Generation

Three extensions of the NAND generation program are under consideration. The first is to idenfity particular logic devices by number rather than specify a gate with a given number of inputs. For example, rather than printing 1 1 2001 indicating a NAND inverter the program could print 7404 2001 identifying the part number for a hex inverter. In addition the particular IC device could be given a code and the inputs and outputs of the gate could be identified by pin number. Thus:

```
A7:  7404     2001  (14)
          C (1)
```

would specify 1C A7 (a 7404 hex inverter) with pin 1 connected to signal C and pin 14 to signal 2001.

This output format could be obtained easily by adding a table of logic gates to the NAND program and searching for the gates with the correct number of inputs. Assigning signal names to particular IC devices without regard to the location of the devices may lead to excessive connection lengths in some cases. However since the gate structure generated by one statement would be assigned together there should be a grouping of gates involved in each equation.

The second extension of the logic generation program involves eliminating the series inverters which are generated. The method which has been selected is to generate gates as before but whenever an inverter is generated check the source of the input. If the input is found to come from another inverter then both inverters will be removed from the gate list.

Addition of NOR, AND and OR gates to the system is the third desirable extension of the logic generation program. Several methods have been considered which will allow implementation using specified logic elements but no programming has been done as yet.

6.3 State Assignment

As discussed earlier no reasonable solution to the state assignment program has been found. The most attractive alternative appears to assign states to arbitrary codes and generate the next-state input functions using PROM elements.

As a first step in testing the validity of this concept, the memory available to the state assignment program will be expanded to the maximum permitted on the computer system in use (512 Kbytes). A number of typical systems will be described in the CADSS language and complete state assignment and logical design will be performed for each. The same systems will be designed using a PROM structure and the costs of the systems will be compared.

If it is found that the hardware costs are comparable then the other advantages of PROM systems will be very attractive. In this case a program to specify PROM contents will be added to the CADSS system. The output of this program will be a deck of cards that can be read on a minicomputer with digital output capability. This minicomputer can then be used to automatically store the desired information in the PROM.

6.4 Final Conclusions

Even at the present stage of development the CADSS system offers considerable advantages to the digital designer. It has been used (in several stages of development) for several years in logic design classes and projects at Texas A&I. The designs produced usually require more logic than would be needed by an experienced designer. On the other hand the simulation locates errors in logic which are occasionally made by even the most experienced designer. Inexperienced students can easily design a system, eliminate logical errors and obtain a correct (though not completely minimized) system in less time than an experienced designer who uses no computer aids.

Particularly where limited production of a system is anticipated and design costs are higher than component costs, use of the CADSS system can reduce total development cost as well as time.

APPENDIX A

ABSTRACTS OF TEXAS A & I UNIVERSITY M.S. THESES RELATING TO CADSS

ATODARIA, J. I.

Data Structure for CADSS Display
(May, 1971)

Thesis Advisor

Ernest A. Franke

Abstract

The CADSS (Computer Aided Digital Systems Synthesis) programs are intended to form a complete computer aided design system. The complete system consists of many computer programs such as a compiler, table generator, simulator, state assignment, logic reduction, and implementation programs. Some of the basic concepts of "Man-Computer Graphics" for the computer aided digital system design are discussed in this paper.

This paper deals with the problem of how the particular graphic information of digital system synthesis will be represented and processed inside the computer. Two computer programs written in FORTRAN IV and run on an IBM 360/44 computer are presented here.

BOUTTE, Milton D.

Computer Aided State Assignment by Dolotta's Algorithm
(August, 1971)

Thesis Advisor

Ernest A. Franke

Abstract

The purpose of this thesis was to develop a computer program that would select a minimum cost state assignment for sequential circuits.

A computer program was written using Dolotta's algorithm. A set of desirable codes were generated and then mapped into the state table to form a scoring array. The scoring array was then scored for such features as all zero, adjacency, all one and other entries which will reduce the cost of the circuit. The codes that are codable and with the highest score are selected as the state assignment. Output logic simplification and don't cares are also taken into consideration.

HOU, Chen-Seng

Computer Reduction of Logic Equations and NAND or NOR Implementation
(May, 1969)

Thesis Advisor

Ernest A. Franke

Abstract

The purpose of this paper is to test the feasibility of using a computer program to reduce and implement logic systems from given Boolean Equations.

The prime implicants of the logic equations are found by the McCluskey Method. By grouping the prime implicants into the basic patterns of all NAND or all NOR structure, a reduction in size of the equation will be realized.

By handling such a problem with a computer program, the simplified resultant stracture can be obtained in a short time.

MEHTA, Girishchandra I.

Minterm Generation Using CADSS DATA
(December, 1973)

Thesis Advisor

Ernest A. Franke

Abstract

The research reported in this thesis consists of an extension to the set of CADSS programs previously developed for computer aided design. The CADSS (Computer Aided Digital Systems Synthesis) language developed for the design of digital systems is the basic source of this thesis project. For complete design of a combinational logic system and for its NAND implementation, a NAND program written previously by Mr. Hou is used.

The inputs to the NAND program are the number of input variables, the number of minterms, the names of the input variables and the actual minterms. This data to the NAND program is supplied through punched cards.

In this thesis a program called minterm is developed which reads and interprets some of the tables generated by the CADSS system. The program determines the input variables used in the CADSS system and generated the logical minterms associated with these input variables. An attempt is also made to prepare a data set for the NAND program which can be interfaced later with the CADSS system and the minterm program of this thesis.

The thesis includes the discussion of the program and flow charts. The flow charts are drawn to an extent to make the program self explanatory and easily understandable. As an illustrative example, Booth's algorithm for binary multiplication is selected which tests the feasibility of the minterm program.

PATEL, Bipin J.

Computer Aided Logic Minimization
(August, 1970)

Thesis Advisor

Ernest A. Franke

Abstract

The purpose of this paper is to present a computer oriented method of obtaining minimal logic functions from original switching functions so as to minimize the cost of the combinational logic. A computer oriented algorithm is presented using McCluskey's technique to generate the minimal switching functions. The flow chart for the program and complete details of program operation including results are presented.

The state assignment problem in the design of synchronous circuits is also described briefly.

SANTHANAM, Parthasarathy

State Assignment for the CADSS System
(December, 1973)

Thesis Advisor

Ernest A. Franke

Abstract

The CADSS language simplifies the design of digital machines from the flowchart level to the state assignment level. In this paper an attempt is made to save the design engineers' time further by minimizing the state assignment produced by the CADSS system.

Both the CADSS language and a number of state assignment Algorithms were studied in detail for the preparation of this thesis. Dolotta's Algorithm for state assignment was chosen due to the availability of programs written by Mr. M. D. Boutte (M.S. Thesis, Texas A&I, 1971).

This thesis describes the linkage between the table generator of the CADSS system and the state assignment programs. Program flow charts and descriptions of the CADSS data tables, several state assignemtn algorithms and Boutte's programs are also included.

SHARMA, Anil K.

Computer Aided Logic Selection
(May, 1970)

Thesis Advisor

Ernest A. Franke

Abstract

The purpose of this thesis is to illustrate the use of the Computer Aided Digital System Synthesis (CADSS) language in the design of a digital system. This is achieved by first describing the CADSS language and secondly by designing a 10 bit multiplier. The design of a 10 bit multiplier includes a complete logic design description and details of cost and characteristics of each logic unit.

Since the CADSS language has not yet been developed to its full extent, a computer program has been included in this paper to generate a table to be used as a next step in the development of CADSS language.

REFERENCES

1. Gorman, D. F. and Anderson, J. P., "A Logic Design Translator," Proceedings of Fall Joint Computer Conf., pp. 251-261; Fall 1962.

2. Kelly, Jr., J. L.- Lochbaum and Vyssotsky, V. A., "A Block Diagram Compiler," The Bell System Technical Journal, pp. 669-676; May 1961.

3. Schlaeppi, H. P., "A Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)," Trans. IEEE, Vol. EC-13, pp. 439-448; August 1964.

4. Schorr, H., "Computer-Aided Digital System Design and Analysis Using a Register Transfer Language," Trans. IEEE, Vol. EC-13, pp. 730-737; December 1964.

5. -- ADD, Philco-Ford Application Manual, Section 7, Philco-Ford.