

AD-A185 289

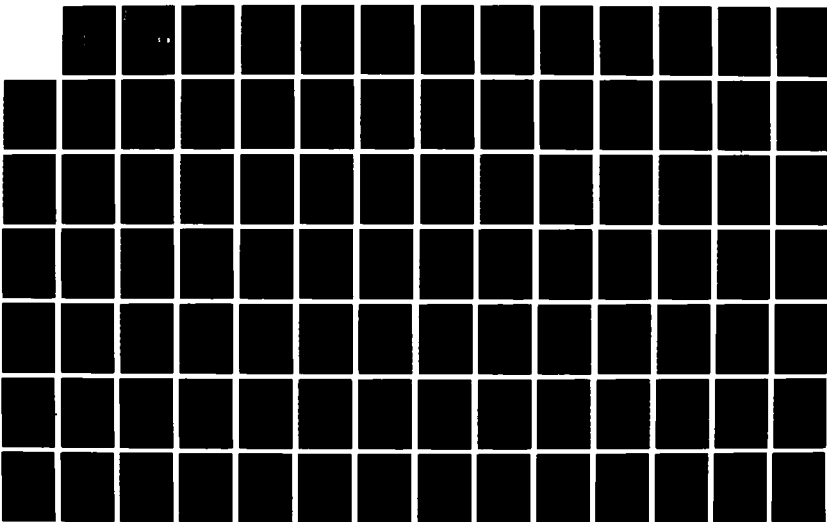
LOGIC CALC: A DESIGN TOOL FOR DIGITAL SYSTEMS(U) AIR
FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
G D ROSENBERGER AUG 87 AFIT/CI/NR-87-55T

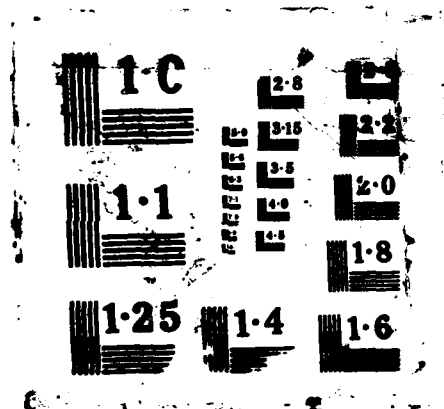
1/2

UNCLASSIFIED

F/G 12/5

NL





REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER AFIT/CI/NR 87- 55T	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER A195 289	
4. TITLE (and Subtitle) Logic Calc: A Design Tool For Digital Systems		5. TYPE OF REPORT & PERIOD COVERED THESIS/MASTER/DOCTORAL	
		6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) Glenn David Rosenberger		8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: University of Texas		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433-6583		12. REPORT DATE August 1987	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		13. NUMBER OF PAGES 131	
		15. SECURITY CLASS. (of this report) UNCLASSIFIED	
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE			
DISTRIBUTION STATEMENT (of this Report) APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1		DTIC ELECTE S OCT 26 1987 D <i>ck D</i> <i>Lynn E. Wolaver</i> LYNN E. WOLAVER 17 Aug 87 Dean for Research and Professional Development AFIT/NR	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) ATTACHED			

DC FILE COPY
 AD-A185 289

1

LOGIC CALC: A DESIGN TOOL FOR DIGITAL SYSTEMS

by

GLENN DAVID ROSENBERGER, BSEE

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

August 1987

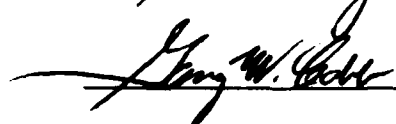


Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

LOGIC CALC: A DESIGN TOOL FOR DIGITAL SYSTEMS

APPROVED:





DEDICATION

To my Lord.

ACKNOWLEDGMENTS

I would like to express my deep appreciation for Professor Harvey Cragon for his sincere concern in my research efforts and for his guidance in the preparation of this thesis. I would also like to thank Doctor Gary Cobb for reading and making valuable, detailed suggestions to this thesis. I thank all the teachers at the University of Texas for their efforts to broaden my horizons. Lastly, I would like to thank my entire family for their love and support.

July 15, 1987

TABLE OF CONTENTS

1.	Introduction	1
2.	Data Structure	6
2.1	Spreadsheets and Object-Oriented Programming . .	6
2.1.1	Data Encapsulation	6
2.1.2	Special Purpose Procedure Encapsulation	7
2.1.3	Implementation of an Object-Oriented Programming Language	8
2.2	Lotus 1-2-3 Data Structure	9
2.2.1	Memory Allocation of Individual Lotus 1-2-3 Cells	10
2.2.2	External Representation of a Lotus 1-2-3 Cell . .	14
2.2.3	Internal Representation of a Lotus 1-2-3 Cell . .	14
2.2.3.1	Null Cell	16
2.2.3.2	ASCII String Cell	16
2.2.3.3	Fixed-Point Cell	19
2.2.3.4	Floating-Point Cell	19
2.2.3.5	Formula Cell	19
2.3	Logic Calc Data Structure	24
2.3.1	Logic Calc Spreadsheet Cell Description	25
2.3.2	Logic Calc Spreadsheet Column Description	29
2.4	Logic Calc Formula Cells	29
2.4.1	The "Cell" Function	30
2.4.2	Indirection Through Recursion of the "Cell" Function	31
2.4.3	Indirection Through "Cell-Indirect"	31
2.4.4	The "Cell-Offset" Function	32
2.4.5	The "Dual-Rank-Register" Function	33
2.4.6	Boolean Operations in Cell Formulas	33
3.	Logic Calc Program Development and Operation . . .	36
3.1	Programming Environment	36
3.1.1	Window Capabilities and Mouse Sensitivity	36
3.1.2	Incremental Compilation and Evaluation	38
3.1.3	Debugging Capabilities	39
3.2	Logic Calc Operation	40
3.2.1	Logic Calc Initialization and Termination	40
3.2.2	Spreadsheet Size	40
3.2.3	File Operations	41
3.2.4	Go-To Operations	42
3.2.5	Manual Spreadsheet Recalculations	42
3.2.6	Spreadsheet Editing Operations	43
3.2.6.1	Cell View	44

3.2.6.2	Cell Edit	45
3.2.6.3	Cell Bit Size	45
3.2.6.4	Cell Output Display	45
3.2.6.5	Cell Move and Cell Copy	46
3.2.6.6	Cell Erase	46
3.2.6.7	Column and Row Editing	46
3.2.6.8	Column Width Settings	46
3.2.7	Automatic Simulation	47
3.2.7.1	The "Print-Spreadsheet" Function	48
3.2.7.2	The "Calc" Function	49
3.2.7.3	The Load-Cell Function	49
3.2.7.4	The "Restart" Function	49
3.2.7.5	User-Defined Functions	50
4.	Digital System Simulation with Logic Calc	51
4.1	Logic Calc Simulation of Parallel Operations	51
4.1.1	Solution to the Recalculation Problem	54
4.1.2	Implementation of the Recalculation Solution	55
4.2	Circular References	56
4.3	Non-Converging Circuits	56
4.4	Clocking of a Digital System	61
4.5	Registers	66
5.	Demonstration of Logic Calc's Capabilities	71
5.1	Design of the Microprocessor	71
5.1.1	Preliminary Steps	72
5.1.2	Construction of Individual Components	77
5.1.2.1	Example Digital Component: The Program Counter	78
5.1.2.2	Example Digital Component: The Internal Data Bus	79
5.1.3	Writing Microcode	81
5.1.4	The Microcontroller	82
5.2	Final Testing of the Microprocessor	83
6.	Conclusion	87
6.1	Summary of Results	87
6.2	Future Research	89
Appendix 1		93
Appendix 2		124
Bibliography		129
Vita		131

INTRODUCTION

The object of this thesis is to develop a engineering-oriented spreadsheet that is capable of being used as a design tool for digital systems.

The design methodology of fully testing and validating a digital system at each level of abstraction before passing it on to lower levels is essential to ensure that no design flaws are passed on to the lower levels. There are several tools available to the designer for simulation of digital logic, but the tools to design and simulate a digital system at the architectural level, the highest level of design, are both limited in productivity and not easy to use. The architectural level needs a highly productive design tool to completely validate the architecture before pursuing the expensive lower levels of design. If the design tool is relatively simple to use, this validation process can be rapidly accomplished.

At the initial stages of design of a digital system, stages in which many changes are usually made, an ideal design tool would be interactive, flexible, highly visible, and self-documenting. Traditional design tools, such as Instruction Set Processor (ISP) and Register Transfer Level (RTL) languages, lack these characteristics. Spreadsheet programs, such as

Lotus 1-2-3*, have these characteristics, and have been proven to have the capability to simulate digital systems [4, 8, 11, 13]. Each cell within a spreadsheet can simulate fundamental components within a digital system (Figure 1). The cell's formula is used to describe the operation of the component. The spreadsheet offers an interactive means of editing cells through use of cursor positioning and menu selection techniques. Text entries may be made in other cells, aiding documentation. The cell's value, displayed on the screen, depicts the output state of the component in a highly visible manner. Because a spreadsheet cell can simulate computer "black boxes" at various levels of abstraction, a high degree of flexibility is offered with such a design tool.

Financial spreadsheets, such as Lotus 1-2-3, are widely available, but they have several drawbacks when used as digital design tools. These drawbacks stem from the design of such spreadsheets as financial tools rather than engineering tools. The primary shortcomings of financial spreadsheet programs are:

1. A maximum value for fixed-point integers. Financial spreadsheets typically do not allow fixed-point integers to be greater than preset limits. If integers become too large in a financial spreadsheet, they are automatically converted to floating-point numbers. In a digital system, however, many components, such as microstore, hold binary integer values in excess of financial spreadsheet's maximum integer value. When used to

* Lotus 1-2-3 is a registered trademark of the Lotus Development Corporation

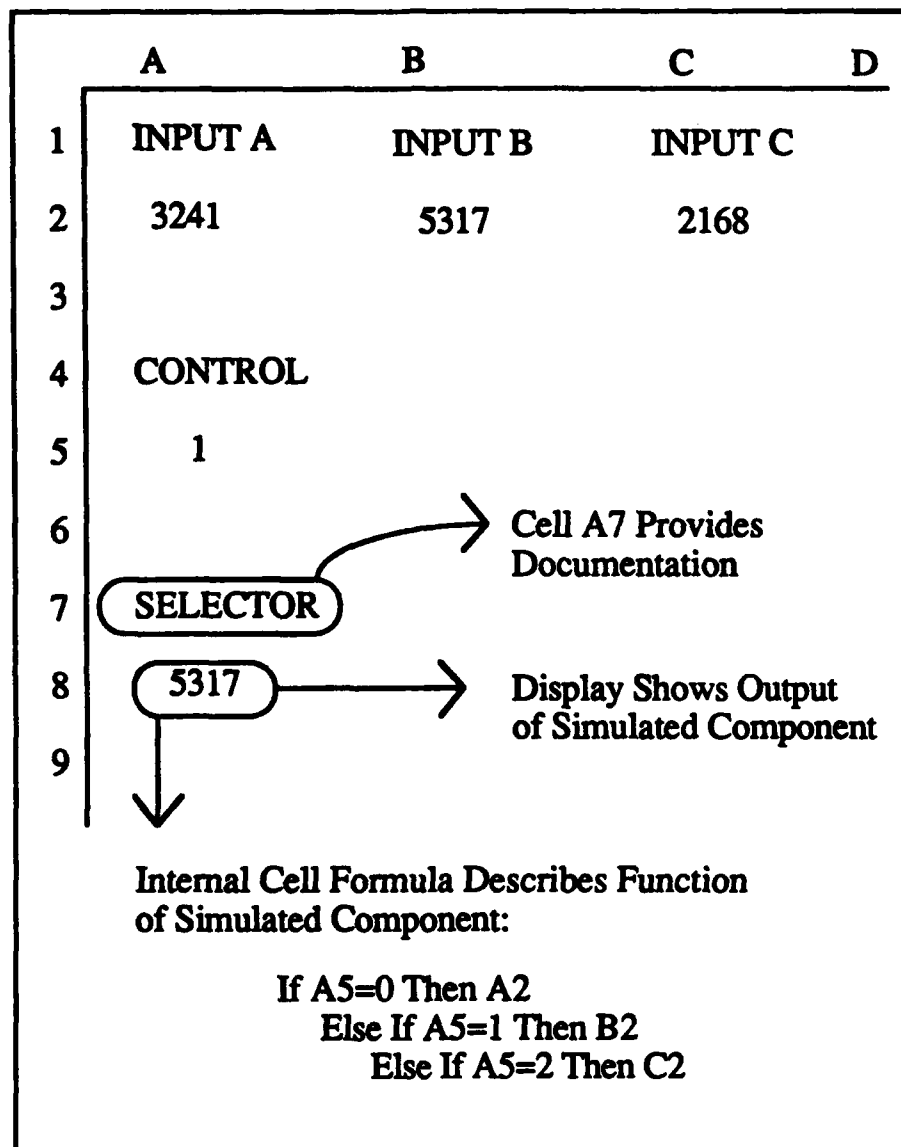


Figure 1. Spreadsheet Cell A8 Functioning as a Digital Selector

simulate such digital components, floating-point numbers are not acceptable due to a lack of precision.

2. Fixed size integers. Digital hardware is of various bit lengths. A flag register may consist of only a single bit, whereas microstore may consist of hundreds of bits. In simulation of digital systems, financial spreadsheets digital lack an ability to correctly specify the operations of hardware of various sizes because all cells have the same bit length in a financial spreadsheet.
3. Lack of Boolean operations. Boolean logic is the heart of all digital systems. In a financial spreadsheet, Boolean operations can only be accomplished with complicated if-then constructs. This method of simulating Boolean operations is cumbersome and error-prone.
4. Lack of binary and hexadecimal display. Interpreting decimal values in a binary system is also very cumbersome and error-prone.
5. Inability to simulate some key digital components in a single cell. To simulate some digital units, a financial spreadsheet requires several cells, cluttering the display.
6. Slow and cumbersome programmatic operation of the spreadsheet. Lotus offers "macros" to drive a spreadsheet, but these macros have an unusual syntax and are therefore difficult to write. Also, they are interpreted rather than compiled; therefore, their execution speed is slow.
7. No modification capabilities. The uncompiled, high-level-language source code is unavailable for most financial spreadsheets, making it difficult to modify the spreadsheet for specific applications.

An engineering spreadsheet, Logic Calc was developed with the goal to eliminate these shortcomings. A comparison of its data structure with the Lotus 1-2-3 data structure is presented in

Section 2. The design and operation of this spreadsheet is presented in Section 3. Digital system simulation with Logic Calc is discussed in detail in Section 4. To illustrate the capabilities of Logic Calc as a digital design tool, a simple microprocessor was designed using it. The techniques for this simulation are presented in Section 5. Conclusions and suggestions for future research in this area are stated in Section 6.



DATA STRUCTURE

2.1 SPREADSHEETS AND OBJECT-ORIENTED PROGRAMMING

Basically, a spreadsheet is a two-dimensional array of a data object called a "cell." An individual, unique cell can be referenced through use of the array indexes, typically called "rows" and "columns." Each cell has specific properties including a type, value, display characteristics, and possibly a formula used to derive its value from its relationships with data and other cells within the spreadsheet. The power in a spreadsheet is two-fold:

1. It has an ability to change and recalculate the values of every cell whenever changes are made to the data or the cell formulas.
2. It provides an interactive environment for editing data and cell formulas.

These two features allow rapid "what if?" analysis of a wide variety of problems. The array format of a spreadsheet provides both a structure for numeric data for easy recalculation and a format for displaying the data and calculations [11].

2.1.1 DATA ENCAPSULATION

These fundamental aspects of a spreadsheet blend well with object-oriented programming. Construction of a spreadsheet through use of an object-oriented programming language allows use

of data abstraction to fully encapsulate the properties of the spreadsheet cell [16]. Data abstraction frees programmers from the details of the representation of the cell properties by hiding the details of lower-level data storage, modification, and retrieval primitives. Working on this higher level is accomplished by providing a collection of access procedures termed "data constructors," "data selectors," and "data mutators." Data constructors can be used to make a spreadsheet cell, data selectors can get information from the cell's property list, and mutators can change cell properties. Object-oriented programming also allows manipulation of data objects as a whole; therefore, spreadsheet cells can be created, moved, copied, or deleted as whole objects [16].

2.1.2 SPECIAL PURPOSE PROCEDURE ENCAPSULATION

Object-oriented programming also offers the encapsulation of special purpose procedures that operate on various properties of the object to reside within the property list of that object type [16]. This allows the programmer to use a general purpose procedure call on a variety of different types of object - the object itself invokes the special purpose procedure. A common example of this aspect of object-oriented programming can be given by defining two types of data objects: fixed-point numbers and floating-point numbers. The general purpose procedure call might be "Add". Since fixed-point addition differs from floating-point

addition, fixed-point data object types should have a special purpose fixed-point addition procedure listed under the "Add" entry of their property list while floating-point object types should have a special purpose floating-point addition procedure under their "Add" entry. The programmer can then invoke an "Add" procedure on an object of either type without consideration to its type. Similarly, a powerful spreadsheet can be developed by providing different type of cells: empty, constant, text and formula. Object-oriented programming allows general purpose operations on cells within a spreadsheet to be invoked without regard to the type of cell. The cell responds appropriately to the general purpose operation, selecting the special purpose operation according to its type. Thus, this aspect of an object-oriented programming language simplifies the development of the spreadsheet program by eliminating general purpose dispatch procedures that must determine the type of spreadsheet cell before issuing a special purpose procedure call [16].

2.1.3 IMPLEMENTATION OF AN OBJECT-ORIENTED PROGRAMMING LANGUAGE

The implementation of an object-oriented program generally involves a pointer-based access to the data structure. Several complex operations on data objects can be reduced to simple pointer manipulation with this type of access to the data structure. Associated with the pointer-based implementation is a

garbage collector which is used to recover the fragmented memory space that arises with the pointer-based access methods. Although a pointer-based implementation may slow computations on simple data objects, operations on complex data objects are easier to implement and may offer a performance advantage [16]. The pointer-based access to the data structure and the garbage collector of the popular Lotus 1-2-3 spreadsheet program is examined in the next subsection.

2.2 LOTUS 1-2-3 DATA STRUCTURE

A study of the Lotus 1-2-3 data structure was accomplished to reveal the operating principles and data encapsulation of the spreadsheet cells. This study was accomplished by temporarily exiting a Lotus session on an IBM PC AT through use of the /System command (Version 2.0 only). The current spreadsheet's data structure was left intact through this method, and it was examined with use of the MS-DOS debugger. The lessons learned from this study of Lotus 1-2-3 were used in the development of Logic Calc. While not a pure object-oriented program, certain aspects of the Lotus 1-2-3 data structure enable one to view the Lotus 1-2-3 spreadsheet program as a primitive form of object-oriented programming.

2.2.1 MEMORY ALLOCATION OF INDIVIDUAL LOTUS 1-2-3 CELLS

Like most spreadsheets, the Lotus spreadsheet is comprised of a two-dimensional array of cells. Columns of the spreadsheet are indexed with letters and rows are indexed with numbers [10]. Generally, Lotus allocates contiguous space in memory from the first active cell (a cell which is not empty) in each column to the last active cell in that same column. This can best be illustrated graphically. Figure 2 shows a portion of a Lotus 1-2-3 spreadsheet in which the active cells are marked by a shaded box. In this example, contiguous space in memory will be used to represent the cells outlined in Figure 3. Garbage collection of this portion of memory is invoked when cells are added or deleted in a column so as to change the row number of the first or last active cell in that column. In the preceding example, if cell D5 were erased, garbage collection would be invoked to reclaim the space allocated for cells D3, D4, and D5.

Each cell in the Lotus 1-2-3 spreadsheet may be viewed as a data object. A cell consists of both an internal value and a specification for displaying that value to the external world. A cell is described by four bytes (Figure 4). The first two bytes and the high order four bits of the fourth byte describe the internal properties of the cell. The third byte is used both to specify the cell's external representation and to identify the cell's protection property (ability of the user to edit the cell).

	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Figure 2. Active Cells in an Example Lotus 1-2-3 Spreadsheet

	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Figure 3. Cells from Figure 2 that are Given Space in Memory by Lotus 1-2-3

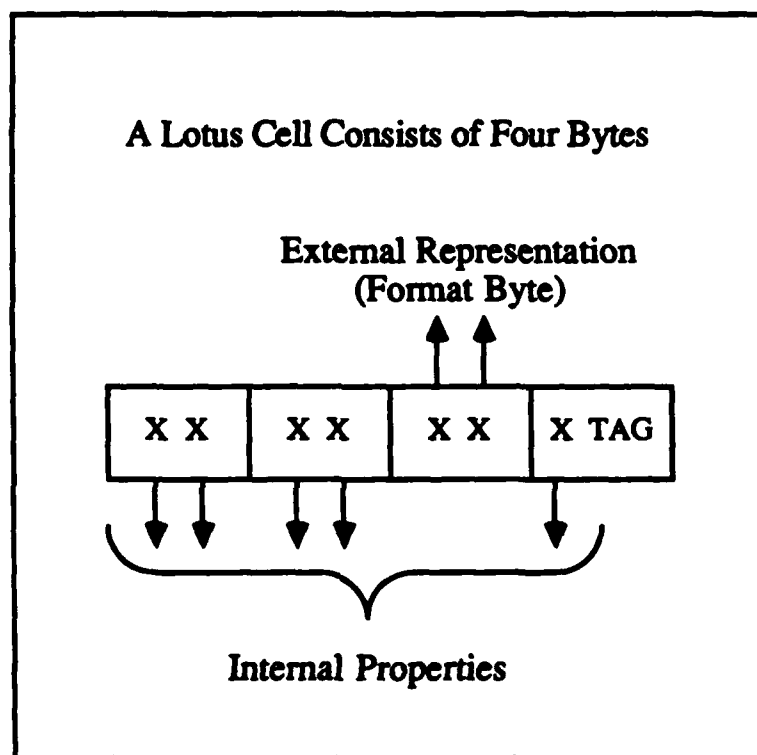


Figure 4. Lotus 1-2-3 Spreadsheet Cell

The low order four bits of the last byte identify the type of cell of which there are five possibilities: empty cell, ASCII string cell, fixed-point cell, floating-point cell, and formula cell.

2.2.2 EXTERNAL REPRESENTATION OF A LOTUS 1-2-3 CELL

The external representation and protection byte for each cell specifies how the internal value of the cell is displayed to the user. Seven bits are used to describe this external representation. An eighth bit is used as a cell protection flag. If both a separate global protection flag and this cell protection flag are set, editing operations on this cell are disallowed. With bit 7 as the high bit, Table 1 lists the possible values of the external representation and protection byte [15]. From this table, an example hex value of "93" would identify the cell's protection flag to be set and that its internal value is to be displayed in scientific notation format with three decimal places displayed.

2.2.3 INTERNAL REPRESENTATION OF A LOTUS 1-2-3 CELL

Lotus stores the internal value of a cell in various ways, depending on the type of cell. Null and fixed-point cells contain immediate, fixed-point internal values. Floating-point cells and formula cells utilize a pointer to obtain an IEEE standard floating-point value. String cells utilize a pointer to

<u>Bit</u>	<u>Description</u>	<u>Binary Value</u>	<u>Description</u>
7	Cell Protection	0	Unprotected
		1	Protected
6,5,4	Format type	000	Fixed Point
		001	Scientific Notation
		010	Currency
		011	Percent
		100	Comma
		111	Special
3,2,1,0	If the format type is 000 - 100, these bits are the number of dec- imal places displayed.	0000 to	
		1111	
	If the format type is 111	0000	+/-
		0001	General format
		0010	Day-Month-Year
		0011	Day-Month
		0100	Month-Year
		0101	Text
		1111	Default

Table 1. Cell Format Byte

obtain an ASCII string. Each of these cells is described separately in the following subsections.

2.2.3.1 NULL CELL

The first type of cell to be described is the null cell. Although the user has made no entry for this cell, it exists because of the memory allocation rule discussed in Section 2.2.1. It has a value of zero, but its value is not displayed. Rather, blanks are displayed at its location. Figure 5 shows the specification of the four bytes that describe a null cell.

2.2.3.2 ASCII STRING CELL

Figure 6 depicts the ASCII string cell, which corresponds to the cells identified as "labels" by Lotus [10]. The internal value of this type of cell for computational purposes is zero, but for string operations, it can be found by the use of a pointer to a buffer of ASCII strings for all string cells within the spreadsheet. This ASCII string has a length variable from 1 to 240 bytes and is terminated with a byte of zero [15]. Garbage collection tags are used to recover space created by editing existing string cells. This garbage collector simply allows space to be reused if possible, but does not compact the buffer. Internal fragmentation of this buffer may therefore occur, but fragmented space can be recovered by writing the spreadsheet to disk and reading it back into memory.

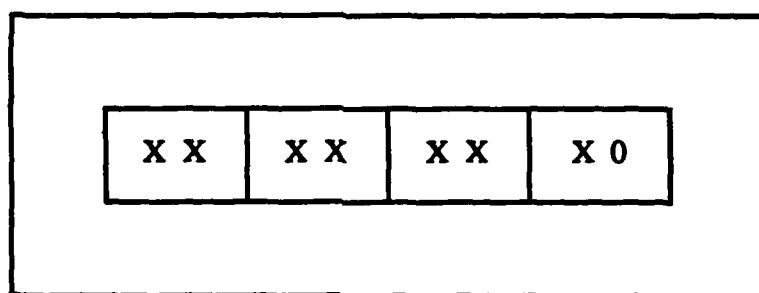


Figure 5. Lotus Null Cell

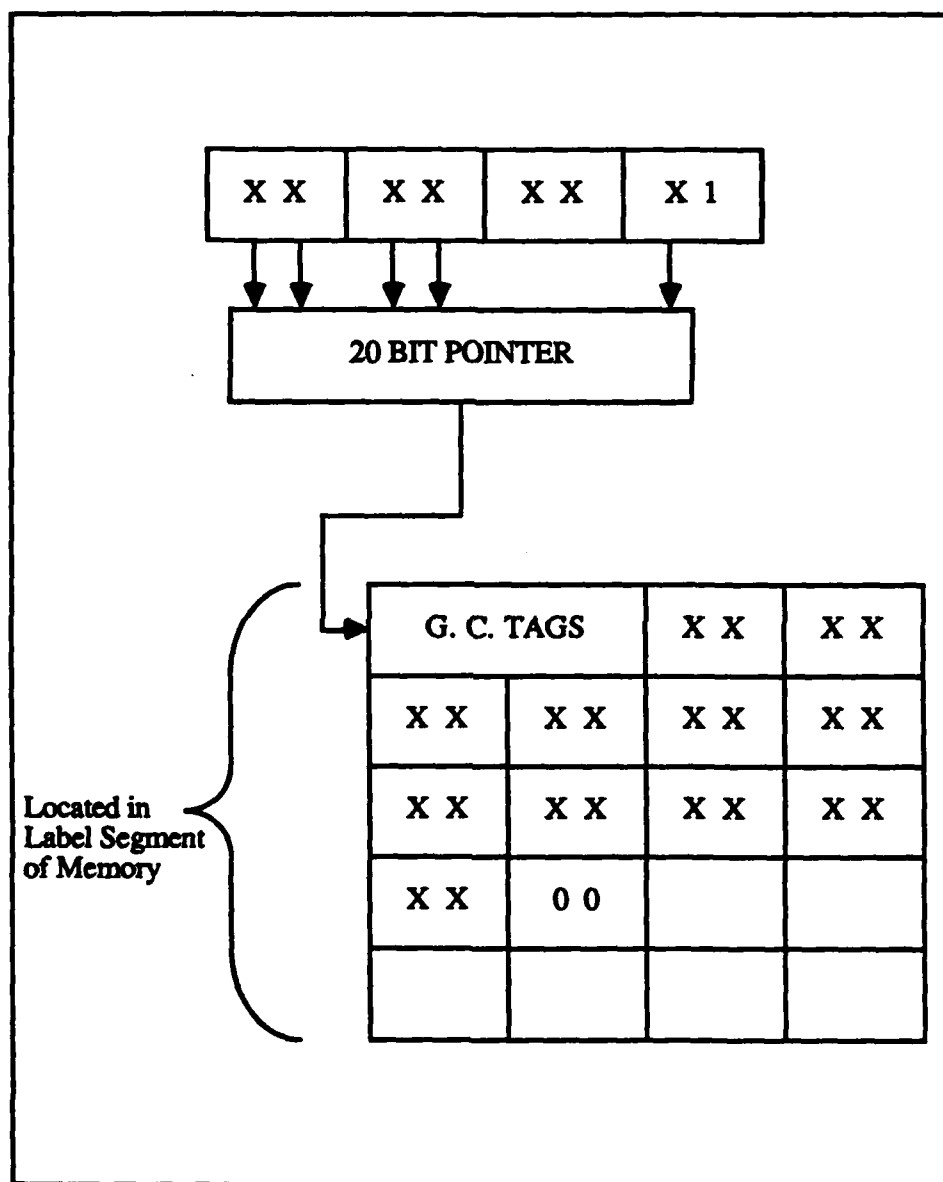


Figure 6. Lotus Label Cell

2.2.3.3 FIXED-POINT CELL

A much simpler cell is the fixed-point cell depicted in Figure 7. The internal value of a fixed-point cell is expressed as a constant two's complement integer. With 16 bits, constant decimal integer values from -32768 to 32767 can be represented by a fixed-point cell.

2.2.3.4 FLOATING-POINT CELL

If an internal value cannot be represented by a fixed-point cell, a floating-point cell, as shown by Figure 8, is automatically created. The internal representation bits form a pointer to a table of 64-bit IEEE standard floating-point numbers [1]. IEEE floating-point not-a-numbers are used to give the value "Error" or "NA" to a cell [1, 10]. Should a floating-point cell be edited so that its type is no longer a floating-point cell, a vacancy in the table will exist. This vacancy is marked with yet another IEEE floating-point not-a-number. This vacancy can be detected when creating a new floating-point cell, and the memory space reclaimed.

2.2.3.5 FORMULA CELL

The most interesting type of cell is the formula cell depicted by Figure 9. A formula cell is a type of cell that contains an internal value that is the result of the computation of its internally stored formula. This computation may have other

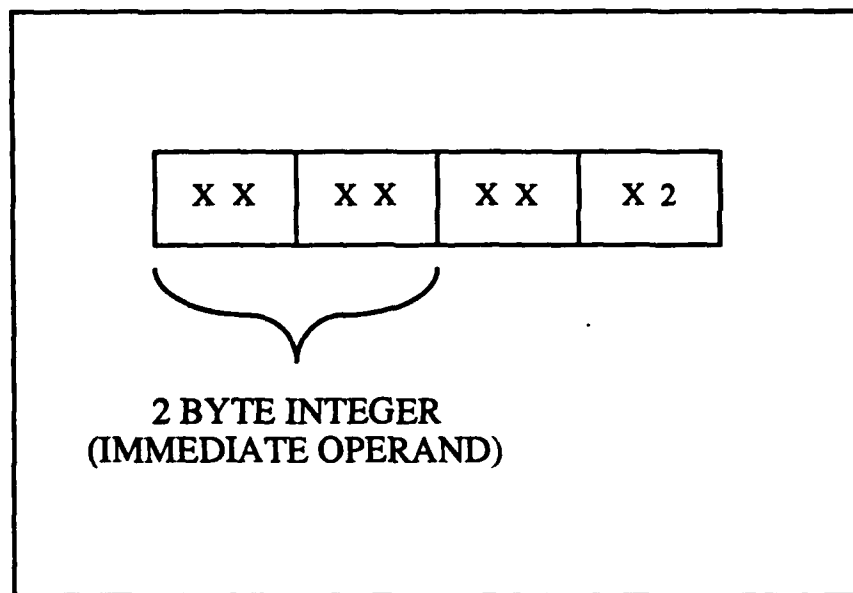


Figure 7. Lotus Fixed Point Cell

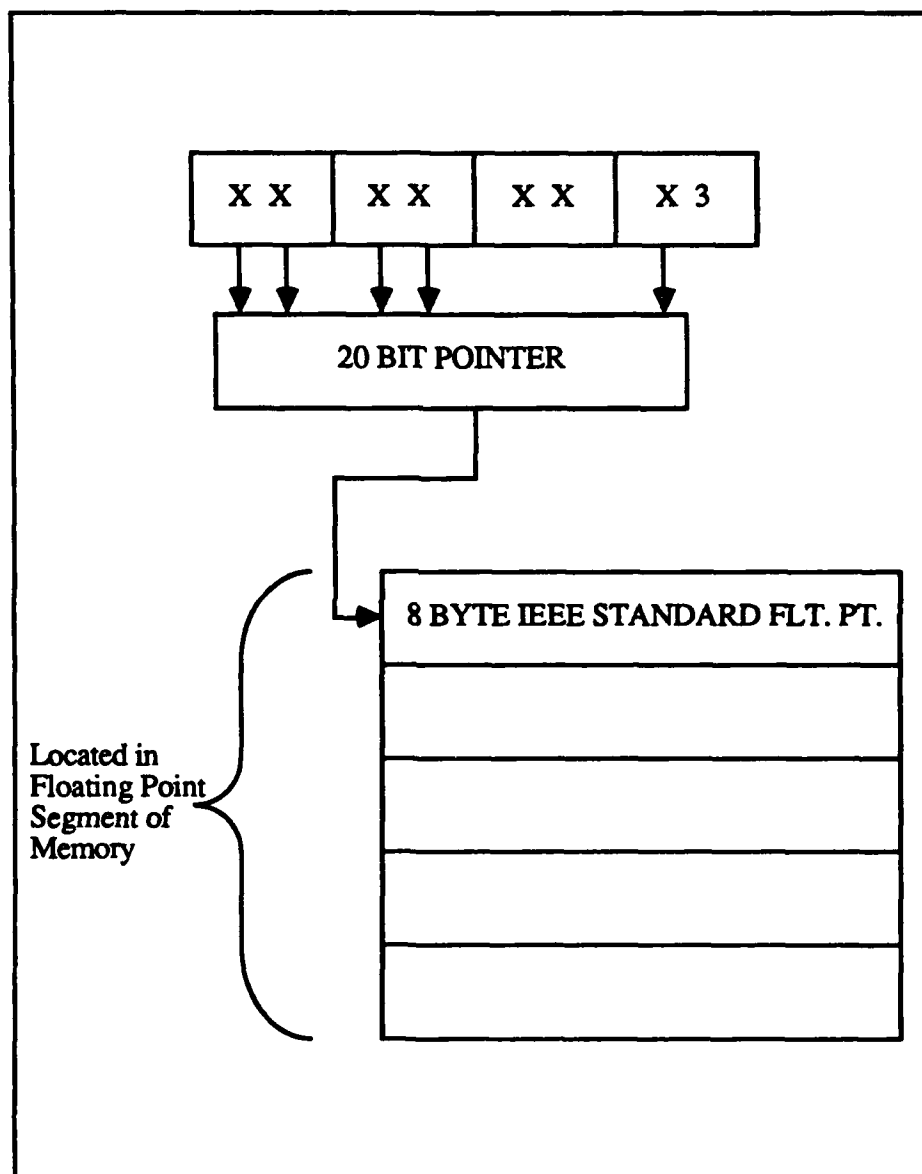


Figure 8. Lotus Floating Point Cell

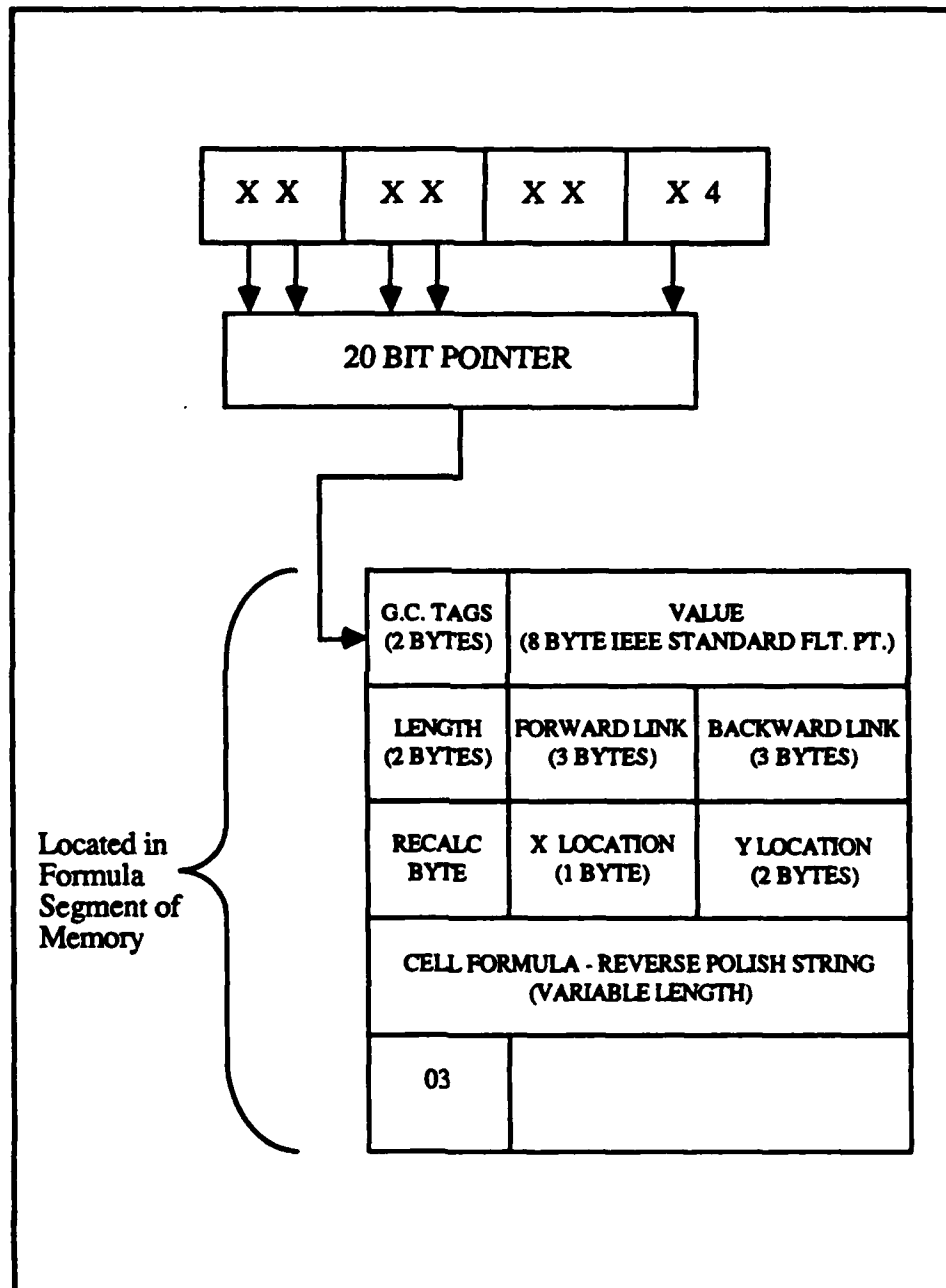


Figure 9. Lotus Formula Cell

cell values as its parameters. Here, the internal representation bits form a pointer to a buffer of various length entries that represent each formula cell. There are several items encapsulated in each entry of the formula buffer:

1. Garbage collection tags. Garbage collection is accomplished by reusing space in the same manner as for label cells.
2. Value. The same IEEE floating-point format of floating-point cells is used to represent the computed internal value of a formula cell.
3. Length of entry. These two bytes specify how many bytes are used to represent the formula. This value is useful when copying the cell to another location by specifying the number of bytes to be copied to another buffer entry.
4. Links. Two three-byte pointers link all formula cells together in a bidirectional manner. The formula cells are linked in the chronological order in which they were entered into the spreadsheet. The links facilitate spreadsheet recalculation. Lotus does not have to examine every cell to determine whether it is a formula cell during spreadsheet recalculation. Rather, it can step through all formula cells with use of this linked formula list. A double link is established so as to allow the arbitrary removal of an entry in the formula buffer. If a formula cell is edited so that its type is no longer a formula cell, the links of the two cells that point to it must be updated. A formula cell knows which cells are pointing to itself by examination of its own links.
5. Recalculation flag. A single byte is used as a flag during spreadsheet recalculation. During spreadsheet recalculation, this flag is initially cleared for all cells. Because formula cells may be evaluated out of sequence due to references of other cells, this flag is set to mark those cells that have been evaluated during the spreadsheet recalculation. Subsequent references to a marked formula cell during a spreadsheet recalculation need not evaluate its

formula. The net result is that each formula cell is evaluated only once during spreadsheet recalculation. The recalculation flag is also used to detect and correctly evaluate circular references during spreadsheet recalculation. Spreadsheet recalculation and circular references are discussed further in Section 4.

6. Location. The following three bytes identify the cell's location, one for the column, and two bytes for the row. A cell needs to know its location in the spreadsheet to enable it to utilize the relative cell addressing offered by Lotus [10].
7. Formula. The internal specification of the formula follows. This entry has a length variable from 2 to 2064 bytes [15]. In Lotus 1-2-3, a cell formula is both written by the user and displayed to the user in an infix notation with syntax and semantics unique to Lotus 1-2-3. When storing the formula, however, the Lotus 1-2-3 program converts the formula into a more compact representation using a reverse-Polish (postfix) notation that is terminated with a formula opcode of "03". A lengthy list of operations exist, giving Lotus the ability to compute scientific, financial, and database statistical functions. A cell may be a parameter in a cell formula and it may be specified either absolutely or relatively. Each time a formula cell is highlighted by the user for viewing, the Lotus 1-2-3 program reconstructs and displays the infix version of the formula. The internal representation of formula opcodes in memory is somewhat different than the representation when the spreadsheet is stored on disk, but publishings of disk formats can be used to unassemble cell formulas [15].

2.3 LOGIC CALC DATA STRUCTURE

The data structure of Logic Calc was patterned after that of Lotus 1-2-3. The differences center around the fact that Logic Calc is designed to be used as a digital design tool rather than a financial spreadsheet. Logic Calc was written in Common Lisp, a

popular object-oriented programming language used in artificial intelligence applications [5, 12, 16].

The spreadsheet indexes of the two spreadsheets are identical: columns are referenced by letters and rows are referenced by numbers. Logic Calc goes a step further, however, by offering a variable size spreadsheet array. This gives the design engineer the opportunity to tailor the size of the spreadsheet to a specific problem. A smaller spreadsheet will use less space in memory. This means that garbage collection will occur less frequently resulting in faster simulations. A large spreadsheet, however, may be necessary to simulate digital systems with many components.

2.3.1 LOGIC CALC SPREADSHEET CELL DESCRIPTION

The spreadsheet cells of Logic Calc are data objects defined by the use of Common Lisp flavors [5, 16]. While Lotus 1-2-3 has several types of data objects with different property lists, each cell within the Logic Calc spreadsheet is of the same type (identical property lists), but a value of an item of the cell's property list distinguishes the type of cell. (Figure 10) The single type of Logic Calc spreadsheet cell is patterned after the formula cell of Lotus 1-2-3: it has an internal representation in the form of a formula and a value, it has an external representation showing the output state of the cell, and it contains a recalculation flag to permit proper spreadsheet

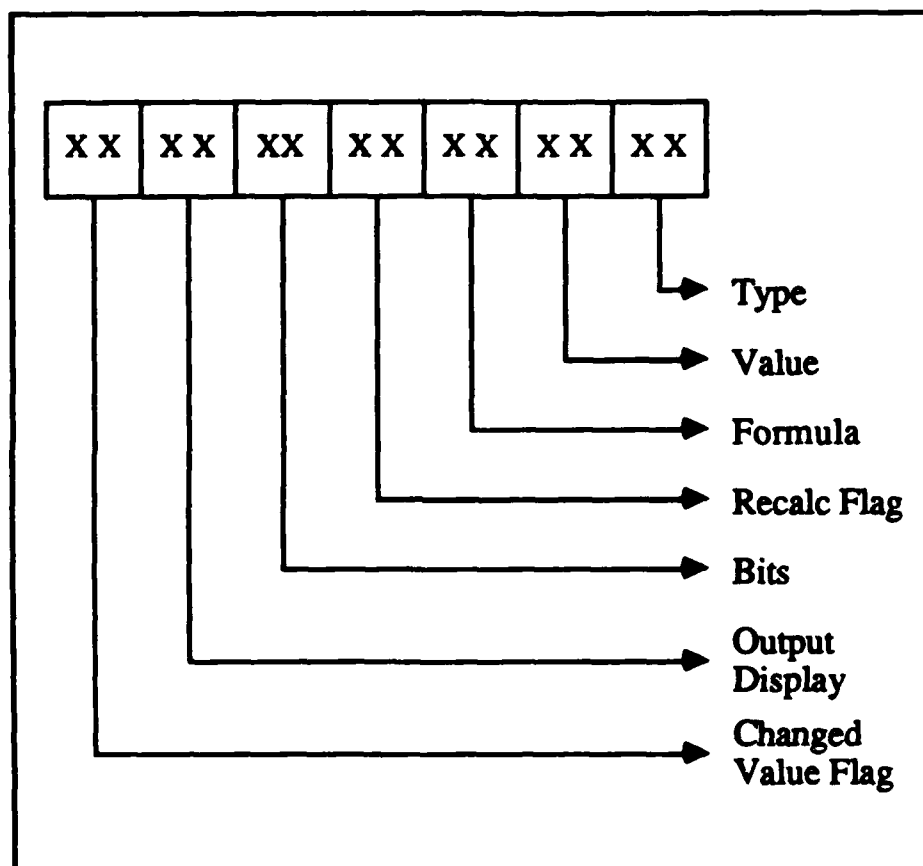


Figure 10. Logic Calc Cell

recalculation. Unlike Lotus 1-2-3, the Logic Calc cells are not linked together. Rather, a list of formula cells is maintained as a global list. Also, Logic Calc has two properties not found in a Lotus 1-2-3 cell: a bits property that allows the spreadsheet to more closely represent digital components of various lengths and a changed-value property that speeds up redisplay of a spreadsheet following a recalculation. The following is a list of the Logic Calc cell properties:

1. Type. The four types of cell objects are a null cell, which has a zero value but is displayed as blank characters; a constant cell, which has a constant integer value; a string cell, which has a constant ASCII string value; and a formula cell, which has either an ASCII string value, a constant value, a T (true) value, or a nil value (as defined by Lisp). The formula cell also contains a Lisp expression used to calculate the value. All cells are initially null cells, but change as the user edits the spreadsheet.
2. Value. As mentioned above, this property of the spreadsheet can be either an integer, an ASCII string, T, or nil, depending upon the type of cell. This value is displayed on the spreadsheet window in a format specified by the output-display property. If the cell is not a formula type, this value is constant. If the cell is a formula, this value is the result of the spreadsheet recalculation method described in Section 4.
3. Formula. Although all cells contain this property, only formula cells make use of it. This property is a Lisp form that is evaluated upon each spreadsheet recalculation to determine the cell's value. Unlike Lotus 1-2-3, formulas in Logic Calc are stored exactly how they are written and displayed: as a Lisp form. This Lisp form utilizes a prefix notation which is easy to read and evaluate. For example, the infix formula of $(3+5)/2$ would be written in prefix

form as "(/ (+ 3 5) 2)." This property is discussed further in the Section 2.4.

4. Recalculation flag. Again, only formula cells make use of this property. This flag is used to ensure that each cell is evaluated only once during spreadsheet recalculation in a manner similar to that employed in Lotus 1-2-3. Logic Calc recalculation methods and circular reference capabilities are described in Section 4.
5. Bits. This is a property that is unique to the Logic Calc spreadsheet when compared to other spreadsheets, and it stems from the design goal of making Logic Calc a design tool for digital systems. The bits property, set by the user or defaulted to 32, allows a cell to closely simulate a digital component by providing a specification for the number of bits of the component. Cells which have integer values are restricted to a value that can be specified with the prescribed number of bits given by the cell's bit property. Because internal values are adjusted by this property, negative values show a true two's complement form of leading 1's in the left-most bit positions and are displayed without a minus sign. For example a 32-bit cell subtracting four from three with a formula of "(- 3 4)" will yield a value of FFFFFFFF hexadecimal. Overflow of a cell is also possible due to this property. For example, a three-bit cell adding two and seven with a formula of "(+ 2 7)" will yield a result of 1. This property has no effect on the cells with ASCII string, true, or nil values.
6. Output display. This property, set by the user during spreadsheet editing, determines whether the value of the cell is displayed in binary or hexadecimal. This property has no effect on ASCII string, true, or nil values.
7. Changed value flag. During editing or spreadsheet recalculation, this flag is set whenever a cell changes its value. The function "Print-Spreadsheet-Changed-Items" rapidly updates the display by redisplaying only those cells whose value has changed since the last display update. The redisplay time can be significantly reduced by this function, allowing a faster program operation during all facets

of spreadsheet programming for digital design simulation. When the user wishes to view a different region of the spreadsheet, all values of the spreadsheet need to be redisplayed. In this case, Logic Calc uses the function "Print-Spreadsheet" which is significantly slower than "Print-Spreadsheet-Changed-Items." Fortunately, this total redisplay is usually required only during spreadsheet editing operations where speed is not critical.

2.3.2 LOGIC CALC SPREADSHEET COLUMN DESCRIPTION

Besides the cell, the other significant data object in the Logic Calc spreadsheet is the column. This object has three properties, all of which are used to generate the display:

1. Letter. This property is equal to the one or two uppercase letters that identify the column index. This ASCII string is printed as a mouse-sensitive item whenever the entire spreadsheet is redisplayed. The user can highlight each column index with the mouse and select from a menu various column editing operations.
2. Width. This value, set by the user or defaulted to 20, is the amount of horizontal space on the screen that the column occupies. This value is measured in number of characters of the fixed-width font that is used in the spreadsheet window.
3. Position. This value is used to remember the left edge of each column. This information must be maintained in order to rapidly update the display after editing or recalculated.

2.4 LOGIC CALC FORMULA CELLS

Formula cells form the heart of the spreadsheet - without them, a spreadsheet would be merely a data storage and display device. Because Logic Calc cell formulas are both written and stored as a Lisp form, a user needs to learn only one syntax to

utilize Logic Calc directly or to interface to Logic Calc from other programs. When used to design digital systems, the Lisp syntax and semantics prove far more powerful than that used by Lotus 1-2-3 due to Lisp's straight-forward, unambiguous syntax, its ease of editing, and its ability to perform operations not offered by Lotus - specifically, Boolean operations. Also, because Logic Calc cell formulas are written in the same language that Logic Calc is written in, the user will find it easy to make modifications to the Logic Calc source code once he learns how to write simple cell formulas. Thus, he can easily tailor Logic Calc to his own needs.

2.4.1 THE "CELL" FUNCTION

A few functions have been provided to reference the values of other cells within the spreadsheet. These functions provide the necessary relationships between spreadsheet cells for the cells to simulate discrete digital components. The most common is the "Cell" function. Because this is the basic building block for cell references within the spreadsheet, particular attention was given to the operating speed of this function when the source code was developed. There can be only a single parameter to the cell function - the cell location. An example best illustrates the use of the cell function. The formula entry (+ (Cell A2) (Cell A3)) will return a value equal to the sum of

the values in cells A2 and A3. The cell function will accept either upper or lower case column indexes. If the cell location does not exist or typographic errors exist in the parameters, this function returns nil. This type of error will generally cause the Explorer debugger to be entered because the value nil cannot be used with integer or string operations. The Explorer debugger prints out highly informative error messages that will aid the user in detecting the source of his error.

2.4.2 INDIRECTION THROUGH RECURSION OF THE "CELL" FUNCTION

To simulate digital systems, it was deemed that indirection would be useful. Any level of indirection is available through recursion of the cell function. For example, if the value of cell A6 is the ASCII string "B52", an entry of (Cell (Cell A6)) would return the value of cell B52.

2.4.3 INDIRECTION THROUGH "CELL-INDIRECT"

An alternate means to obtain a single level of indirection is provided through the "Cell-Indirect" function. For example, if the value of cell F16 is the ASCII string "RF4", then the formula entry (Cell-Indirect F16) would return the value of cell RF4. As a final example of indirection consider the values of the following cells:

A4: "B17"

A10: "B29"

B17: 34 (Decimal)

B29: 12 (Decimal)

A formula entry of (+ (Cell (Cell A4)) (Cell-Indirect A10)) would return a decimal value of 46.

2.4.4 THE "CELL-OFFSET" FUNCTION

A lookup function was also deemed useful in order to simulate digital systems. This function is most useful when simulating a component that needs the capability to index into a block of data such as a memory system. The function that provides this is "Cell-Offset." This function takes three parameters: the anchor cell location, an x-offset from the anchor location, and a y-offset from the anchor location. An example best describes its function: (Cell-Offset C5 3 10) returns a value equal to the value of cell F15. The offset values may be any integer or a Lisp expression which returns an integer. If a parameter for an offset does not evaluate to an integer, or the offset cell does not exist, this function returns nil. For another example, consider the following cell values and formulas:

F4: -6

F5: (Cell F4)

F18: 3

F20: 5

The formula (cell-offset L23 (cell F5) (cell F18)) will return a decimal value of 5.

2.4.5 THE "DUAL-RANK-REGISTER" FUNCTION

The ability of Logic Calc to describe digital systems is most apparent with the use of the function "Dual-Rank-Register." This function allows a user to describe a dual-rank register in a single cell. A dual-rank register is an elementary digital module used in many digital circuits [2, 3, 9]. Its use is described in Section 4.4.

2.4.6 BOOLEAN OPERATIONS IN CELL FORMULAS

As a digital systems design tool, a key advantage of Logic Calc over financial spreadsheets such as Lotus 1-2-3 is its ability to perform Boolean operations. Logic Calc cell formulas can utilize Lisp Boolean operators to perform logical operations at the cell, byte, or bit levels [5, 12].

Boolean operations at the cell level are best illustrated with the "Cell" function. For example, a cell formula of (Logand (Cell A1) (Cell A2)) will return a value equal to the bitwise logical "anding" of cells A1 and A2. The user must consider the bits property of each cell when writing Boolean operations at the cell level. When used as parameters of Boolean operations, cells of different lengths are right-aligned with leading zeros placed on the left for cells with lower bit values. The final result is left-trimmed as per the formula cell's bit property. For example, consider the following cell formulas, values, and bit specifications:

Cell A1: 0110101001 Binary Value (10 Bits)

Cell A2: 01111 Binary Value (5 Bits)

Cell A3: (Logand (Cell A1) (Cell A2))

If cell A3 has a bit property of 10, its formula will yield a binary value of 0000001001. If cell A3 has a bit property of 5, its formula will yield a binary value of 01001.

As a contrasting example, consider a change to the cell formula for cell A3 to specify a logical inclusive-or:

Cell A1: 0110101001 Binary Value (10 Bits)

Cell A2: 01111 Binary Value (5 Bits)

Cell A3: (Logior (Cell A1) (Cell A2))

Now, if cell A3 has a bit property of 10, its formula will yield a binary value of 0110101111. If cell A3 has a bit property of 5, its formula will yield a binary value of 01111.

For byte operations, Lisp provides several functions for dealing with an arbitrary-width field of contiguous bits appearing anywhere in a cell [12]. In the following example, cell B2 uses the Lisp functions "Ldb" and "Byte" to extract a 4-bit byte from cell B1 starting at bit #2:

Cell B1: 1111010110 Binary (10 Bits)

Cell B2: (Ldb (Byte 2 4) (Cell A1))

With a value of 10 for its bit specification, the resulting value of cell B2 is 0000000101 Binary.

To perform operations at the bit level, Lisp provides the capabilities to test any bit of any cell [5, 12]. For example, the cell formula (Logbitp 3 (Cell F16)) returns a value of true if bit 3 of cell F16 is 1. Other bit operations can be performed by extracting a 1-bit byte from a cell as described above.

LOGIC CALC DEVELOPMENT AND OPERATION

3.1 PROGRAMMING ENVIRONMENT

The programming environment used for the development of Logic Calc was the Texas Instruments Explorer workstation. This proved to be an ideal tool for the development and implementation of the spreadsheet. A combination of extensive window capabilities, mouse sensitivity, incremental compilation and evaluation capabilities, and extensive debugging capabilities allowed rapid development of a user-friendly, highly interactive spreadsheet.

3.1.1 WINDOW CAPABILITIES AND MOUSE SENSITIVITY

One key feature of the Explorer that aided this research was a sophisticated window capability which resulted in a easy-to-develop user interface for the spreadsheet [6]. The user interface of Logic Calc consists of a team of three windows that share a common input buffer. An Explorer constraint frame determines the sizes and positions of the three windows. Figure 11 depicts the team of three windows.

The largest window, the spreadsheet window, depicts the row and column indexes and the values of the cells that can fit within the window. This window is simply an Explorer truncating window mixed with a mouse-sensitive typeout window [6]. This type

LOGIC CALC									
	A	B	C	D	E	F	G	H	I
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									
20									
21									
22									
23									
24									
25									
26									
27									
28									
29									
Interaction Window									
File	Size	Main Menu		GoTo	Calc	Exit			

Figure 11. Logic Calc User Interface

of window allows certain items to be output so that they are highlighted when the mouse cursor is positioned over them. The spreadsheet program was designed to respond to menu choices when the user clicks the mouse upon the highlighted mouse-sensitive item. This gives the user the capability of using the mouse to select spreadsheet cells, columns or rows for viewing, editing, copying, moving, inserting or deleting.

The second window of the constraint frame is the Interaction Window. This typeout window [6] is used to output information requested by the user, to display certain messages, and to input user responses to various program prompts. Essentially, anything that must be typed by the user to edit the spreadsheet is generally input through this window.

The third window of the constraint frame is the Logic Calc Main Menu. This is an Explorer command-menu window which allows the spreadsheet programmer use of the mouse to select items of the Main Menu [6]. The Main Menu consists of choices that can initiate file operations, change the size of the spreadsheet, select the region of the spreadsheet to be viewed, recalculate the spreadsheet, or exit the spreadsheet to terminate editing operations.

3.1.2 INCREMENTAL COMPILATION AND INTERPRETATION

Another feature that allowed rapid program development was the flexibility of the Explorer environment to accommodate

various degrees of compilation and interpretation [7]. During development of Logic Calc, it was possible to select between a recompilation of the entire Logic Calc program, the changed sections of that Logic Calc program, or just a specified region of the Logic Calc program. This allowed an incremental compilation ability which aided in the incremental, design and test, bottom-up software development methodology that was used. The Explorer environment also supported an interpretive mode with similar degrees of freedom as to the amount of code to be evaluated. This allows testing of various segments of code without recompiling, thereby speeding up testing of the Logic Calc program.

3.1.3 DEBUGGING CAPABILITIES

The final element of the Explorer environment that stood out as an aid during Logic Calc program development was an extensive interactive, on-line debugger. When program errors arose during testing of Logic Calc, the debugger printed out highly informative error messages and offered the ability to examine program variables at the time of the error. Depending on the type of error, the debugger often offered the capability to proceed from the error, reading a replacement value for the one in question. With this extensive debugging capability, it was easy to locate the source of errors, and sometimes corrections could be made and the test continued from the point of the error, without

exiting the test. Coupled with the capability to accomplish incremental compilation, the debugger was the primary reason for the ability to rapidly develop a sophisticated user interface to Logic Calc.

3.2 LOGIC CALC OPERATION

The user interface to Logic Calc was designed to utilize menus and mouse selection whenever possible. Appendix A is the source code for Logic Calc. The rest of this section serves as a users manual for Logic Calc operations.

3.2.1 LOGIC CALC INITIALIZATION AND TERMINATION

Logic Calc is stored in SEL-5 and SEL-6 Explorer workstations of the University of Texas Symbolic Engineering Laboratory in the local Logic-Calc directory under the filename "Logic-Calc.Xfasl." The easiest way to start the program is to enter the ZMACS editor and select "Load File" from the Explorer Suggestions Menu. Then simply respond to the prompt with "Logic-Calc;Logic-Calc". Logic Calc will automatically be loaded and initialized. At this point, the program is an edit mode. Selection of items from the Logic Calc Main Menu and editing of columns, rows and cells are possible. The program is simply looping, waiting for a command from the user.

To terminate a Logic Calc session, simply click on the "Exit" option from the Logic Calc Main Menu.

Subsequent reentries into Logic Calc after the initial load can be accomplished by evaluating "(Restart)" from a Lisp Listener or ZMACS buffer.

3.2.2 SPREADSHEET SIZE

The spreadsheet size is adjustable, limited primarily by the amount of virtual memory, the length of strings contained in string cells, the bit size of constant and formula cells, and the complexity of formula cells. With 115 Megabytes of virtual memory available, it was possible to create a spreadsheet of 50,000 null cells; 50,000 constant cells of 32 bits; 50,000 string cells, each containing a ten character ASCII string; and 50,000 32-bit formula cells, each containing a simple cell formula.

The user can change the size of the spreadsheet by selecting the "Size" item from the Logic Calc Main Menu. He will then be prompted for the number of rows and columns in the spreadsheet. The program currently limits the user to a spreadsheet of 400 rows by 300 columns, but this could easily be changed by adjusting the program constants "Max-Number-of-Rows" and "Max-Number-of-Columns" at the beginning of the source program. If more than 702 columns are specified for Max-Number-of-Columns, changes will have to be made to accommodate a three-letter column indexes in the "Row&Col" function and the "Column-String" function.

3.2.3 FILE OPERATIONS

Because it is necessary to save spreadsheets for later work, file operations were deemed necessary for Logic Calc. Although these file operations were kept at a minimum, they are sufficient for proper operation of the spreadsheet. Basically, a spreadsheet can be written to, or retrieved from, the Logic Calc directory of the local machine. To accomplish other file operations, such as deleting, renaming or copying spreadsheets, it is necessary for the user to exit Logic Calc and use the tools provided by the Explorer workstation. To initiate a file write, the user should click upon the "File" menu item on the Logic Calc Main Menu. He should then select "Save" from the next menu. The user will be prompted for the file name through the Interaction Window. If the name given already exists as a saved file, a new copy is saved, with a higher version number. Read operations are accomplished in a similar manner. If more than one copy of the specified file exists during read operations, the file with the highest version number is selected. If the specified file does not exist, the user is informed with an error message through the Interaction Window.

3.2.4 GO-TO OPERATIONS

Depending on the selected widths of columns, Logic Calc automatically selects the number of columns displayed. The number of rows that are displayed is 36. Because scrolling operations do

not exist in Logic Calc, the user must select the "Go To" option from the Logic Calc Main Menu to view various regions of the spreadsheet. After selecting this option, the user must respond to the prompt with the cell location that he wishes to be placed in the upper left corner of the screen. Logic Calc will then change the display as specified. If the specified cell does not exist, Logic Calc will issue a beep. The "Go To" item from the Logic Calc Main Menu may also be used during Move and Copy operations on rows, columns and cells as discussed in Section 3.2.6.

3.2.5 MANUAL SPREADSHEET RECALCULATION

To accomplish incremental design and testing of a digital system on Logic Calc, it is often desired to recalculate the spreadsheet during editing operations. This can be accomplished by selecting "Calc" from the Logic Calc Main Menu. The spreadsheet will then be recalculated, and the screen will be altered to display then updated results. This action is very similar to pressing the "Calc" function key when using Lotus 1-2-3 [10]. If there are numerous formulas within the spreadsheet, this recalculation process may take a few seconds.

3.2.6 SPREADSHEET EDITING OPERATIONS

When first activated, the spreadsheet is in an edit mode. Cells, rows or columns may be edited. In this edit mode, the spreadsheet program simply loops continuously, awaiting mouse clicks from the user. Upon detection of a mouse click, the main program loop calls the appropriate function to accomplish the requested operation.

Editing operations on cells are initiated by the use of mouse. The user locates the cell with the mouse cursor. The cell selected by the mouse is highlighted with a box-like cursor. To simply view the highlighted cell's contents, the user should click left on the mouse. If the user click's right, he is presented with a menu to view the cell, edit (rewrite) the cell, change the cell's bit specification, set the cell's output display to binary or hexadecimal, copy the cell to another location, move the cell to a new location, or erase the cell. When the user selects an item from this menu, Logic Calc will then prompt the user for the necessary information to accomplish the editing operation that he selected.

3.2.6.1 CELL VIEW

If the user selects to view a cell, the following information is displayed in the Interaction Window: cell location, cell type, cell value (both decimal and hexadecimal for

integer values), cell bit size, and cell formula (if a formula cell).

3.2.6.2 CELL EDIT

If the user selects to edit a cell, Logic-Calc presents another menu to the user so that he can create a null, constant, string, or formula cell. The cell's current contents are then displayed, and the user is prompted through the Interaction Window for the necessary information to complete construction of the cell. If errors are made during cell editing operations, Logic Calc beeps and the cell construction is aborted.

3.2.6.3 CELL BIT SIZE

If the user selects to change the bit specification of a cell, he is prompted to enter the size in bits for the specified cell. If the user does not respond with a positive integer less than 127, Logic Cell simply beeps to signify an error has occurred.

3.2.6.4 CELL OUTPUT DISPLAY

If the user selects to change the output display of a cell, he is presented with another menu to choose between binary or hexadecimal with the mouse. He simply needs to click the mouse on the desired choice of cell output display.

3.2.6.5 CELL MOVE AND CELL COPY

If the user selects to move or copy a cell, Logic Calc prompts the user to select a target location with the mouse. Before selecting a target cell, the user may make use of the "Go To" option from the Logic Calc Main Menu to display a different portion of the spreadsheet.

3.2.6.6 CELL ERASE

If the user selects to make the selected cell empty, the cell is erased and the display updated appropriately.

3.2.6.7 COLUMN AND ROW EDITING

Other editing operations exist for entire rows and columns, and are accomplished in a similar manner. An entire row or column of null cells may be inserted. Similarly, rows and columns can be moved, copied or deleted. The user simply needs to highlight the row or column index, click right, and choose from the items presented. Because a large portion of the spreadsheet is altered during this operations, they are accomplished only after the user confirms his intentions through a second confirmation menu.

3.2.6.8 COLUMN WIDTH SETTINGS

The width of a column will affect the display of values within that column. The value of each column width specifies the maximum number of characters that may be displayed horizontally

for each cell within the column. Cells with ASCII string values will have their output display right-trimmed to fit within the column width. Cells with numeric values will be displayed by a string of asterisks if their bit size generates a display in the current output-display format that requires more characters than the column width.

To change a column width, the user needs to highlight the column index with the mouse and click left. The user is then prompted for the requested width of the column. The display will then be updated as requested. If the user does not respond with a positive integer, Logic Calc simply beeps to signify the error.

3.2.7 AUTOMATIC SIMULATION

In Lotus 1-2-3, a program written with Lotus macros was executed to initiate and drive an automatic simulation of a digital system [10, 13]. The Lotus 1-2-3 macros can be difficult to read, write, and edit because they are stored in individual spreadsheet cells and are often hidden by the contents of other cells. Also, Lotus macros utilize awkward semantics that can be difficult to understand.

In Logic Calc, an automatic simulation is initialized and driven by a Logic Calc driving program. To write and execute a Logic Calc driving program, Logic Calc should be exited normally. The driving program can then be written and executed

from the ZMACS editor. ZMACS offers extensive full-screen editing operations, file operations, and incremental compilation and evaluation capabilities [7]. Another advantage that Logic Calc has over Lotus is that the Logic Calc driving program is written in Lisp, the same language used for cell formulas, the same language used for the development of Logic Calc, and the the same language used for the operating system of the Explorer workstation. This greatly simplifies the initial learning process of Logic Calc and provides for easy modifications to Logic Calc for specific applications.

The Logic Calc driving program can simulate controlling digital components at several levels of abstraction. During the simulation, the driving program can function as a controlling processor, a microcontroller, or it can simply drive a clock. Section 4.4 details the development of a driving program used to drive a clock. For the microprocessor design presented in Section 5, the driving program functions as the microcontroller and clock driver. Several Lisp functions have already been provided by Logic Calc to aid in the development of the driving program.

3.2.7.1 THE "PRINT-SPREADSHEET" FUNCTION

The Logic Calc function "Print-Spreadsheet" should be one of the first Lisp functions evaluated. This function will simply display the spreadsheet on the screen.

3.2.7.2 THE "CALC" FUNCTION

The Logic Calc function "Calc" should be used to recalculate the spreadsheet. This is identical to selecting "Calc" from the Logic Calc Main Menu. If the Lisp keyword "no-redisplay" is included as an argument and set to "t", then the display will not be updated after a recalculation. Use of the "no-redisplay" construct speeds up simulations where interim results are not checked.

3.2.7.3 THE "LOAD-CELL" FUNCTION

The Logic Calc function "Load-Cell" can be used to load an integer value into any cell. This function simply updates the "value" and "changed-value" items in the cell's property list. It does not change the type of cell. This function can be used to simulate any external input (Load, Reset, Databus Input...) or loading of registers controlled by the driving program. The two parameters for this function are:

1. Cell Location (i.e. "A12").
2. Value to be loaded.

3.2.7.4 THE "RESTART" FUNCTION.

This function can be evaluated at anytime during a driving program to return to the editing mode. This gives the user an opportunity to temporarily stop an automatic simulation to

examine values or to return to circuit construction after an automatic simulation.

3.2.7.5 USER-DEFINED FUNCTIONS

The user can write his own functions to tailor Logic Calc to specific applications. Also, the Lisp function "Defun" in a driving program can be used to name cell, greatly increasing documentation of cell formulas. For example, the following Lisp functions can be used to name two cells as data buses and a third cell as a control signal:

```
(Defun DatabusA () (Cell F4)
(Defun DatabusB () (Cell F5)
(Defun Control  () (Cell C5)
```

A cell formula, using a Lisp conditional construct, that simulates a selector can now be written in a self documenting form:

```
(Cond ((Equal 0 Control) (DatabusA))
      ((Equal 1 Control) (DatabusB)))
```

This method of naming cells through use of a driving program was used extensively in the microprocessor presented in Section 5. A more complicated driving function can simulate a microcontroller by checking the values of the cells that function as clocks and cells that simulate the status register and instruction register. Based on the contents of these cells, the function "Load-Cell" may be utilized to load the microinstruction register. The microprocessor presented in Section 5 utilizes this feature in its driving program, which is listed in Appendix 2.

DIGITAL SYSTEM SIMULATION WITH LOGIC CALC

Generally, the components in a digital system operate in parallel with a common clock synchronizing operations. A digital system also includes numerous combinational logical circuits that form interconnections between the parallel components. These interconnections often form circular references: the output of Unit A is an input to Unit B, and the output of Unit B is the input to Unit A. Figure 12 depicts this concept of a digital system. A spreadsheet serves as a useful tool to simulate a digital system because of its fundamental abilities to map parallel operations and their circular interconnections onto a single, serial processor. The methods that Logic Calc utilizes to accomplish this mapping are presented in this section.

4.1 LOGIC CALC SIMULATION OF PARALLEL OPERATIONS

In a digital system, the input into each register must be resolved before that register is clocked. This input is usually a logical combination of other registers, memory, and control signals. To simulate a digital system, a spreadsheet must maintain the proper relationships between components. For example, if a Logic Calc spreadsheet was made to model the circuit in Figure 13, six cells would be needed to simulate the

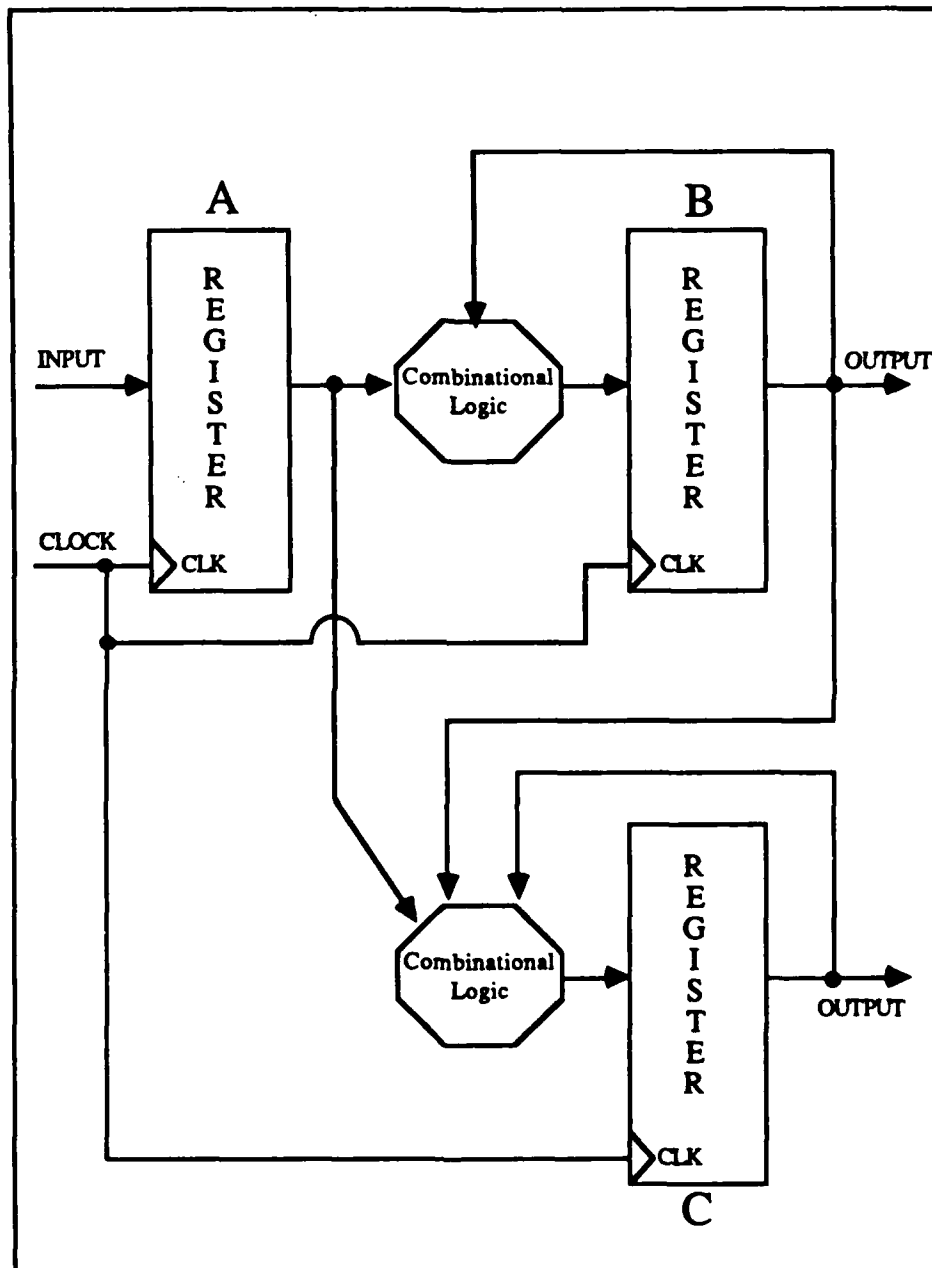


Figure 12. Concept of a Digital System

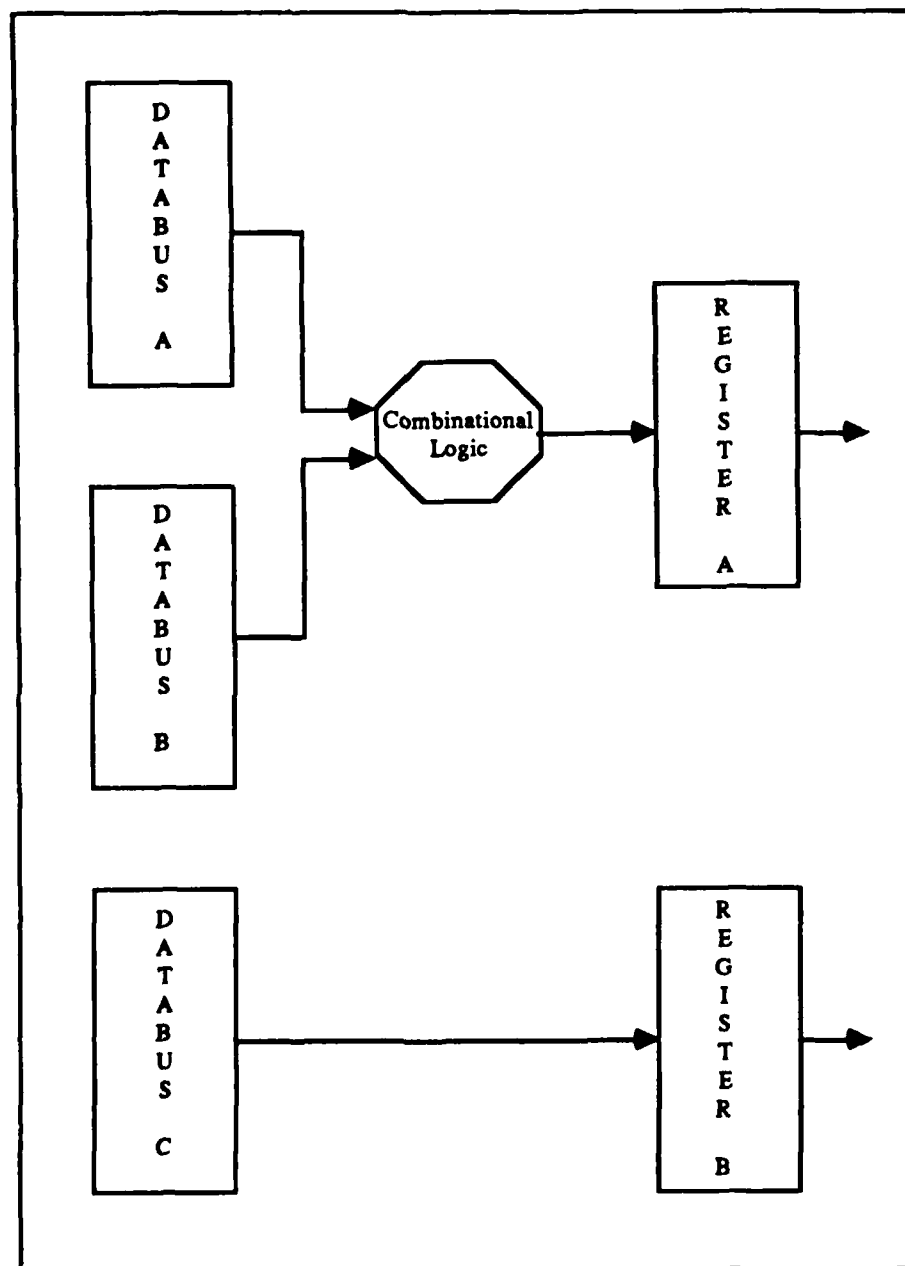


Figure 13. Example Digital Circuit

components: one for each data bus, one for each register, and one for the combinational logic circuit. In this example digital system, the output of the combinational logic circuit must be resolved before clocking of Register A. Similarly, in Logic Calc, the equation for the cell representing the combinational logic circuit must be calculated before the equation for the cell representing Register A. Also, in the example digital system, the clocking of both registers occurs in parallel. In Logic Calc, the order of calculation for the cells representing the two registers is not important: independent parallel operating components can be calculated one at a time without regard to order.

4.1.1 SOLUTION TO THE RECALCULATION PROBLEM

In order to correctly map the parallel components with their interconnections onto a spreadsheet, two solutions to handle the order of cell recalculation are possible. The first would be to maintain a list during spreadsheet editing that describes dependency relationships and the order of cell recalculation and utilize this list during spreadsheet recalculation. This method was not chosen in the design of Logic Calc due to the complex data structure that would arise and the problems with circular references discussed in Section 4.2. Rather, the solution used in Logic Calc is patterned after that used in Lotus 1-2-3. This solution utilizes a check of the parameters to each cell formula during spreadsheet recalculation. If a parameter of a cell formula

is another formula cell, recalculation of the first cell is postponed until the second cell is recalculated. For example, during recalculation of cell F4, if a parameter of the cell F4's formula is found to be cell F16 and F16 is also a formula cell, the recalculation of cell F4 is postponed and the recalculation of cell F16 is begun. When cell F16 has been recalculated, the recalculation of cell F4 may be continued.

4.1.2 IMPLEMENTATION OF THE RECALCULATION SOLUTION

To implement this solution in Logic Calc, a global list of all formula cells is maintained in the chronological order in which they were entered into the spreadsheet. This global list is stored in the variable "Formula-List." The recalculation flag in the cell property list is also used to prevent a formula cell from being recalculated more than once during a single spreadsheet recalculation. A spreadsheet recalculation is accomplished by first setting all formula cell's recalculation flag to nil. Next, each cell listed in "Formula-List" is sequentially recalculated. The first step during recalculation of a cell is to test the recalculation flag. If the flag is set to "true," the recalculation of that cell is terminated. If the flag is "nil," the flag is set to "true" and the recalculation of that cell's formula is initiated. During recalculation of the cell's formula, if another formula cell is found to be a parameter of the formula,

the current recalculation is postponed until recalculation of the cell used as a parameter is accomplished. Tables 2, 3, and 4 show that this method of recalculation produces identical results regardless of the ordering of of "Formula-List." The asterisk in the "Processing" column of these tables identifies which cell is being processed in each step.

4.2 CIRCULAR REFERENCES

Digital systems of appreciable size will have circular references. Logic Calc's method of recalculation handles circular references of cells with predictable results. The simplest example of a circular reference is a cell functioning as a counter. If cell A1 were to have the formula (+ 1 (Cell A1)), a circular reference is established because this cell refers to itself. The Logic Calc program will increment this cell by one for each spreadsheet recalculation. Table 5 shows the steps taken when recalculating a spreadsheet with this example circular reference. Logic Calc also handles more complex circular references, involving several cells, regardless of the ordering of the entries under "Formula-List." The microprocessor described in Section 5 presents several examples of circular references.

4.3 NON-CONVERGING CIRCUITS

If a circuit that does not converge is simulated, a circular reference must be present and the results will be

<u>Ordering</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
1	B1	(+ 1 (Cell B17))	XXX	Nil
2	B17	(+ 1 (Cell B29))	XXX	Nil
3	B29	(+ 3 4)	XXX	Nil

- 1) All formula cell's recalculation flag set to nil.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
*	B1	(+ 1 (Cell B17))	XXX	T
	B17	(+ 1 (Cell B29))	XXX	Nil
	B29	(+ 3 4)	XXX	Nil

- 2) Cell B1's recalculation flag set but evaluation postponed.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	B1	(+ 1 (Cell B17))	XXX	T
*	B17	(+ 1 (Cell B29))	XXX	T
	B29	(+ 3 4)	XXX	Nil

- 3) Cell B17's recalculation flag set but evaluation postponed.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	B1	(+ 1 (Cell B17))	XXX	T
	B17	(+ 1 (Cell B29))	XXX	T
*	B29	(+ 3 4)	7	T

- 4) Cell B29's recalculation flag set and value set to 7.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	B1	(+ 1 (Cell B17))	XXX	T
*	B17	(+ 1 (Cell B29))	8	T
	B29	(+ 3 4)	7	T

- 5) Cell B17's evaluation continued resulting in a value of 8.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
*	B1	(+ 1 (Cell B17))	9	T
	B17	(+ 1 (Cell B29))	8	T
	B29	(+ 3 4)	7	T

- 6) Cell B1's evaluation continued resulting in a value of 9.

TABLE 2. ORDER OF EVALUATION (FORMULA LIST: B1, B17, B29)

<u>Ordering</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
3	B1	(+ 1 (Cell B17))	XXX	Nil
1	B17	(+ 1 (Cell B29))	XXX	Nil
2	B29	(+ 3 4)	XXX	Nil

- 1) All formula cell's recalculation flag set to nil.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	B1	(+ 1 (Cell B17))	XXX	Nil
*	B17	(+ 1 (Cell B29))	XXX	T
	B29	(+ 3 4)	XXX	Nil

- 2) Cell B17's recalculation flag set but evaluation postponed.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	B1	(+ 1 (Cell B17))	XXX	Nil
	B17	(+ 1 (Cell B29))	XXX	T
*	B29	(+ 3 4)	7	T

- 3) Cell B29's recalculation flag set and value set to 7.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	B1	(+ 1 (Cell B17))	XXX	Nil
*	B17	(+ 1 (Cell B29))	8	T
	B29	(+ 3 4)	7	T

- 4) Cell B17's evaluation continued resulting in a value of 8.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
*	B1	(+ 1 (Cell B17))	9	T
	B17	(+ 1 (Cell B29))	8	T
	B29	(+ 3 4)	7	T

- 5) Cell B1's recalculation flag set and value set to 9.

TABLE 3. ORDER OF EVALUATION (FORMULA LIST: B17, B29, B1)

<u>Ordering</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
3	B1	(+ 1 (Cell B17))	XXX	Nil
2	B17	(+ 1 (Cell B29))	XXX	Nil
1	B29	(+ 3 4)	XXX	Nil

- 1) All formula cell's recalculation flag set to nil.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	B1	(+ 1 (Cell B17))	XXX	Nil
	B17	(+ 1 (Cell B29))	XXX	Nil
*	B29	(+ 3 4)	7	T

- 2) Cell B29's recalculation flag set and value set to 7.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	B1	(+ 1 (Cell B17))	XXX	Nil
*	B17	(+ 1 (Cell B29))	8	T
	B29	(+ 3 4)	7	T

- 3) Cell B17's recalculation flag set and value set to 8.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
*	B1	(+ 1 (Cell B17))	9	T
	B17	(+ 1 (Cell B29))	8	T
	B29	(+ 3 4)	7	T

- 4) Cell B1's recalculation flag set and value set to 9.

TABLE 4. ORDER OF EVALUATION (FORMULA LIST: B29, B17, B1)

<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
A10	(+ 1 (Cell A10))	9	Nil

- 1) All cell's recalculation flag's are set to nil

<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
A10	(+ 1 (Cell A10))	9	T

- 2) Cell A10's recalculation flag is set but evaluation postponed in order to evaluate its paramater (which is also A10).

<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
A10	(+ 1 (Cell A10))	9	T

- 3) Evaluation of Cell A10 is begun recursively, but this time the check of the recalculation flag ends the recursive evaluation with no changes made.

<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
A10	(+ 1 (Cell A10))	10	T

- 4) The original evaluation of Cell A10 is continued resulting in a value of 10.

TABLE 5. LOGIC CALC'S METHOD OF HANDLING A CIRCULAR REFERENCE

dependent on the ordering of the entries under "Formula-List." An example of such a circuit is shown in Figure 14. The output of each adder is used as the input to the other adder. This circuit will never converge to constant values for both adders. Such a circuit can be simulated by two cells. Tables 6 and 7 demonstrate that different orderings of "Formula-List" will produce different results for a spreadsheet recalculation. When designing a digital system, care must be taken to avoid non-converging circuits because Logic Calc will not identify non-converging circuits as errors and will give misleading results.

4.4 CLOCKING OF A DIGITAL SYSTEM

A fundamental aspect of a digital system is that many components utilize a clock to load registers and synchronize operations. Logic Calc presents a simple way for simulating the clock in a digital system. Every recalculation of the spreadsheet can be made to simulate one of four states to a clock: the rising edge, the high level, the falling edge, and the low level. To accomplish this, one of the cells within the spreadsheet can be made to iterate between four values with each value describing one of the four states of the clock. Other cells, functioning as clocked digital components, can use conditional constructs in their cell formulas to test the value of the "clock" cell. This will enable the cells functioning as clocked digital components to act differently for each state of the simulated clock.

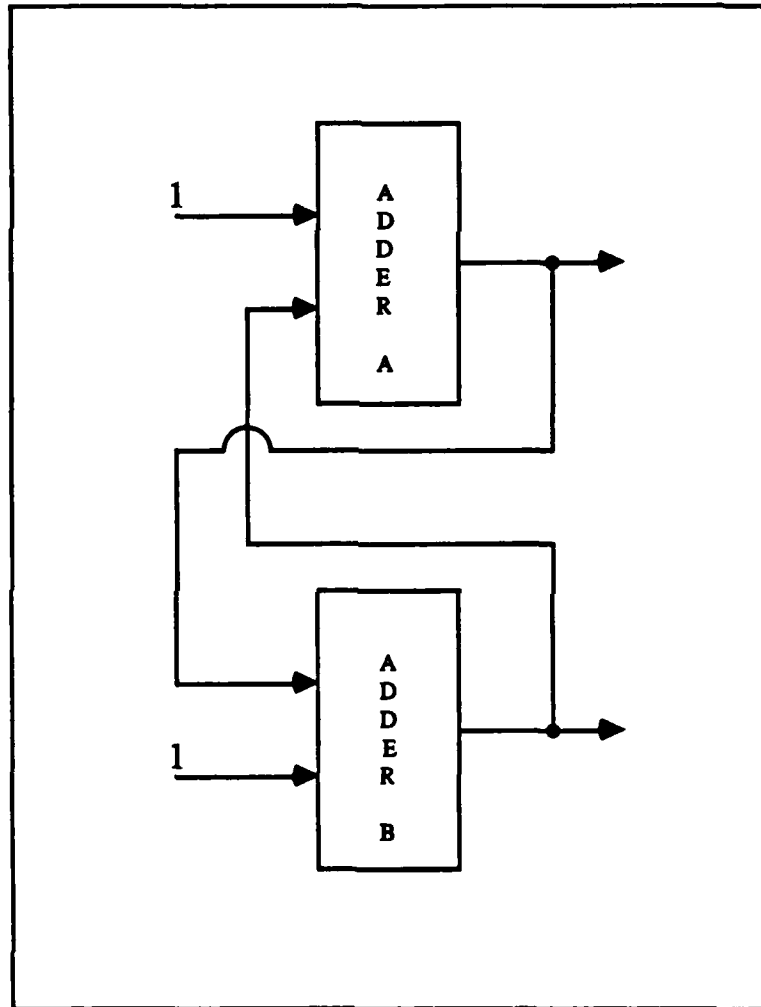


Figure 14. Non-Converging Circuit

<u>Ordering</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
1	F15	(+ 1 (Cell F16))	0	Nil
2	F16	(+ 1 (Cell F15))	0	Nil

- 1) All cell's recalculation flag's are set to nil.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
*	F15	(+ 1 (Cell F16))	0	T
	F16	(+ 1 (Cell F15))	0	Nil

- 2) Cell F15's recalculation flag is set but evaluation postponed.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	F15	(+ 1 (Cell F16))	0	T
*	F16	(+ 1 (Cell F15))	1	T

- 3) Cell F16's recalculation flag is set and value set to 1.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
*	F15	(+ 1 (Cell F16))	2	T
	F16	(+ 1 (Cell F15))	1	T

- 4) Cell F15's evaluation is continued yielding a value of 2.

TABLE 6. NON-CONVERGING CIRCUIT OF FIGURE 14.
(FORMULA LIST: F15, F16)

<u>Ordering</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
2	F15	(+ 1 (Cell F16))	0	Nil
1	F16	(+ 1 (Cell F15))	0	Nil

- 1) All cell's recalculation flag's are set to nil.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	F15	(+ 1 (Cell F16))	0	Nil
*	F16	(+ 1 (Cell F15))	0	T

- 2) Cell F16's recalculation flag is set but evaluation postponed.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
*	F15	(+ 1 (Cell F16))	1	T
	F16	(+ 1 (Cell F15))	0	T

- 3) Cell F15's recalculation flag is set and value set to 1.

<u>Processing</u>	<u>Cell</u>	<u>Formula</u>	<u>Value</u>	<u>Recalc Flag</u>
	F15	(+ 1 (Cell F16))	1	T
*	F16	(+ 1 (Cell F15))	2	T

- 4) Cell F16's evaluation is continued yielding a value of 2.

TABLE 7. NON-CONVERGING CIRCUIT OF FIGURE 14.
(FORMULA LIST: F16, F15)

To illustrate this concept of clocking, consider a spreadsheet with the following specification for cell A2:

Type: formula

Bits: 2

Value: 0

Formula: (+ 1 (Cell A2))

Because Cell A2's bit specification restricts its value to 0, 1, 2, and 3, it can define four states of a clock with unique integer values depicting each state:

0: Low

1: Rising Edge

2. High

3. Falling Edge

The bit specification and the circular reference of cell A2 cause its value to cycle during multiple recalculations of the spreadsheet. For each recalculation of the spreadsheet, the value of cell A2 will iterate: 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, ...

Other cell's functions may include conditional constructs to check the value of cell A2 to determine the phase of the clock. A D flip-flop triggered on the rising edge of the clock can be simulated by cell C5 with the following cell formula:

```
(Cond ((Equal 0 (Cell A2)) (Cell A3)) ;A3 is the input to
      (T (Cell C5))) ; the flip-flop
```

When testing a digital system on Logic Calc, clocking of the digital system can be initiated in two ways. The user can

simply click the mouse on the "Calc" item from the Logic Calc Main Menu. This would initiate a single spreadsheet recalculation and change the state of the clock. Four clicks on the "Calc" item would be necessary to generate a full clock cycle. An alternative method to generate clock signals would be to exit Logic Calc and execute a driving program that generates clock pulses through the "Calc" function. For example, the following driving program will generate twenty five cycles of the clock:

```
(Print-Spreadsheet)
(Loop for i from 1 to 100 do
  (Calc))
(Restart)
```

4.5 REGISTERS

In a digital system, several registers can be clocked at the same time. If these registers are constructed as simple flip-flops, the output of these registers must not be used as inputs into other registers using the same clock edge because the output of the first register may or may not change during the clocking edge. Figure 15 depicts this type of circuit with registers constructed with a single D flip-flop (ie. single-rank registers). In this example, the output of the Y Register is unpredictable because both registers are using the same clock edge to load their inputs and it cannot be determined whether the output of the X Register will change during the clocking edge. This type of circuit construction must be avoided.

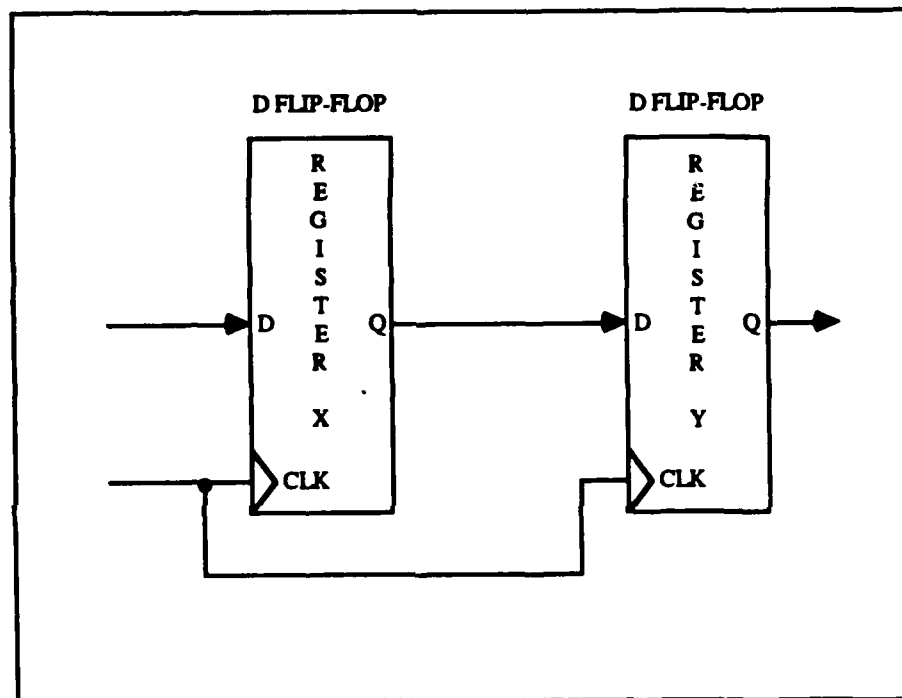


Figure 15. Improper Register Construction

A simple method of circuit construction which avoids this clocking problem can be accomplished by replacing each of the single-rank registers of Figure 15 with dual-rank registers. A dual-rank register is an elementary digital module comprised of a collection of edge-triggered master-slave D flip-flops (Figure 16). Used as a register, it is useful in digital systems because of its ability to feed its output into another digital component using the same clock, even while loading new data [1, 2, 9]. Use of dual-rank registers in a digital system eliminates the problems of circular references. By clocking all inner ranks of the registers with the same clock edge and all outer ranks of the registers with the opposite clock edge, a digital system containing circular references is reduced to two separate, independent systems, each containing no circular references.

In Lotus 1-2-3, several cells would be required to simulate a dual-rank register [13]. Logic Calc, however, provides a special function for simulating a dual-rank register in a individual cell. The "Dual-Rank-Register" function, introduced in Section 2, is intended to stand alone as a cell's formula and takes four parameters:

1. Name. The first parameter must be a unique name. This parameter creates a variable which internally stores the output of the A flip-flop.
2. Clock A. The second parameter is a logical expression that defines when to clock the A flip-flop in Figure 16.

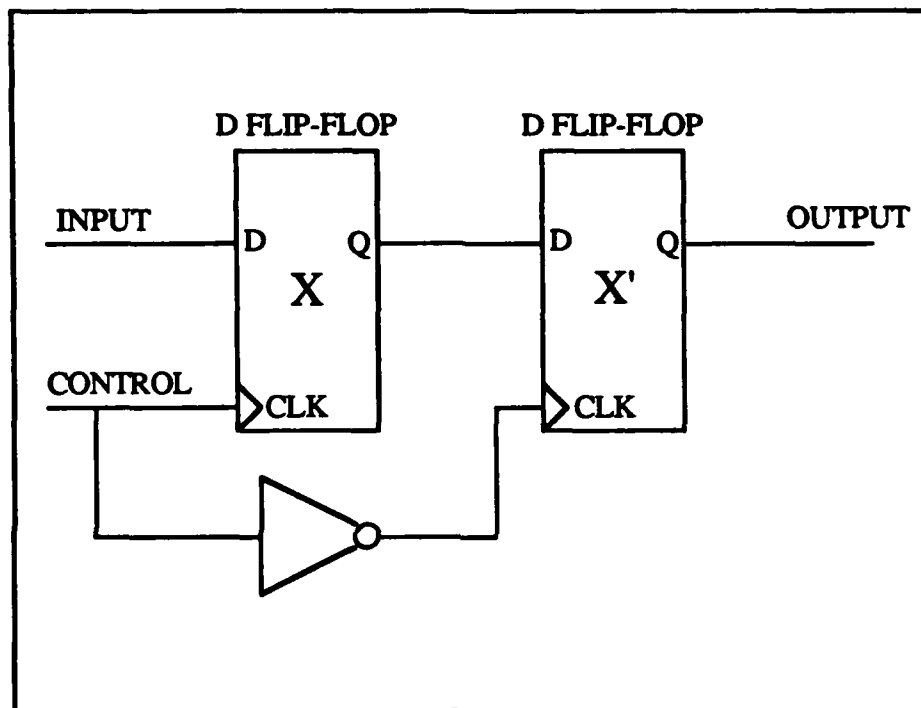


Figure 16. Dual Rank Register

3. Clock B. The third parameter is a logical expression that defines when to clock the B flip-flop in Figure 16.
4. Input A. The final parameter describes the input of the A flip-flop in Figure 16.

The cell's value is constantly set to the output of the dual rank register, the output of the B flip-flop in Figure 16. As an example, consider the following function specification:

```
(Dual-Rank-Register RegisterA (Equal 0 (Cell A2))  
(Equal 3 (Cell A2)) (Cell D2))
```

The first parameter, RegisterA, defines a global variable that is used to hold the output of the A flip-flop. The second parameter, (Equal 0 (Cell A2)), specifies that the value of cell D2, the fourth parameter, should be clocked into the register if the value of cell A2, which may be functioning as a clock, is equal to 0. The output of the A flip-flop is gated into the B flip-flop (and available to other components) whenever the value of cell A2 is 3. Further examples of the use of the "Dual-Rank-Register" function are given in Section 5.

DEMONSTRATION OF LOGIC CALC'S CAPABILITIES

In order to demonstrate the capabilities of Logic Calc, a simple microprocessor was designed using Logic Calc. The microprocessor was patterned after the Motorola 6800 family of microprocessors [9, 14]. It features two 32-bit general purpose registers, two 32-bit index register, and a 32-bit stack register. The arithmetic logic unit (ALU) consisted of only an adder/subtractor. The status register had flags for zero, negative, carry, and overflow, which were set/cleared depending upon the macro instruction. Forty-eight macro instructions were developed, including a full range of load/store instructions, stack operations, conditional branching, and basic arithmetic instructions including add, subtract, increment, and decrement. The microprocessor is fully microcontrolled. This section presents the steps taken to design and test this microprocessor with Logic Calc.

5.1 DESIGN OF THE MICROPROCESSOR

Six steps were utilized to design the microprocessor. The first step was to decide on the number, type and size of registers, arithmetic units, and memory units. Next, the data paths between each unit were designed. Also, a listing of macro instructions was created. These preliminary decisions were made

without Logic Calc; a simple scratch pad was used. The hardware design at this level is presented in Figure 17. The list of macro instructions is listed in Table 8. Logic Calc was then used to construct each component of the microprocessor. Paralleling construction of the individual components, fields within the microcode instruction were assigned to control the components. Next, the microinstructions were written and stored in the microstore. Finally, the microcontroller was designed.

5.1.1 PRELIMINARY STEPS

With Logic Calc in the edit mode, text cells were placed above the locations of all components to aid in documentation. A portion of this layout is shown by Figure 18.

A clock was constructed exactly as described in Section 4.4 in Cell A2.

Logic Calc was then exited and a short driver program was written and executed to provide self-documentation capabilities for the cell formulas. This driver program consisted of four parts. The first part gives names to each of the cells simulating a digital component. A partial listing of this portion of the driver program follows:

(Defun PC	() (Cell A9))	;Program Counter
(Defun RegA	() (Cell A15))	;A Register
(Defun RegB	() (Cell A18))	;B Register
(Defun RegX	() (Cell A21))	;X Register
(Defun RegY	() (Cell A24))	;Y Register
(Defun SP	() (Cell A27))	;Stack Pointer
(Defun IDB	() (Cell B2))	;Internal Data Bus

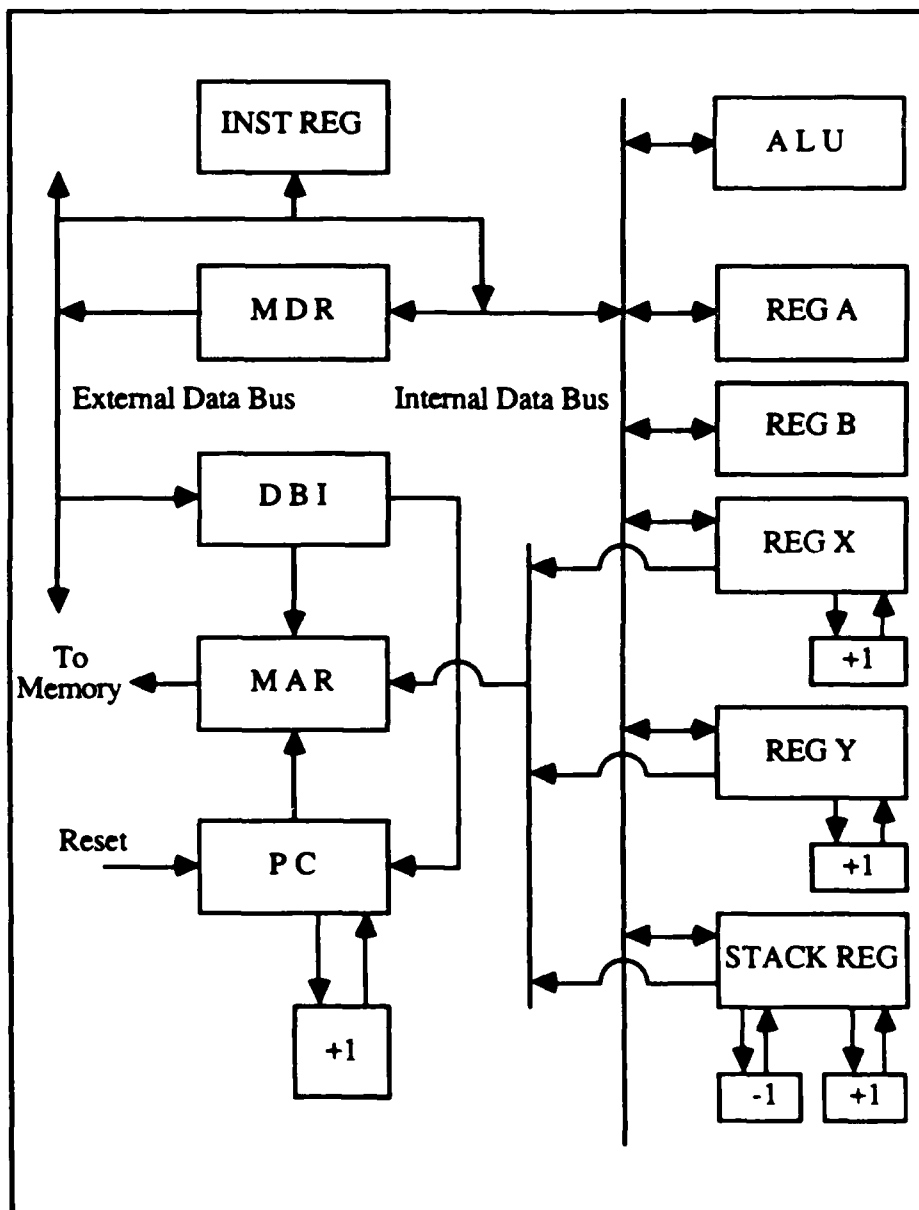


Figure 17. 32-Bit Microprocessor Designed with Logic Calc

<u>Opcode</u>	<u>Instruction</u>	<u>Opcode</u>	<u>Instruction</u>
Load Immediate		Stack Operations	
01	LDAI	18	PUSHA
02	LDBI	19	PUSHB
03	LDXI	1A	PUSHX
04	LDYI	1B	PUSHY
05	LDSI	1C	POPA
		1D	POPB
		1E	POPX
		1F	POPY
Load Memory Location		Arithmetic Instructions	
06	LDAM	20	ADD A:- A + B
07	LDBM	21	SUB A:- A - B
08	LDXM	22	INCA
09	LDYM	23	DECA
0A	LDSM	24	INCB
		25	DECB
Load Memory, Indexed		26	INCX
0B	LDA, X	27	DECX
0C	LDA, Y	28	INCY
0D	LDB, X	29	DECY
0E	LDB, Y		
Store Memory Location		Branching Instructions	
0F	STAM	2A	BSR (Subroutine Call)
10	STBM	2B	RET (Subroutine Return)
11	STXM	2C	BRA (Branch Always)
12	STYM	2D	BZ (Branch if Zero)
13	STSM	2E	BM (Branch if Minus)
		2F	BC (Branch if Carry)
Store Memory, Indexed		Control Instruction	
14	STA, X	00	HALT
15	STA, Y		
16	STB, X		
17	STB, Y		

TABLE 8. LIST OF INSTRUCTIONS FOR MICROPROCESSOR

LOGIC CALC			
	A	B	C D
1	CLOCK	INTNL DBUS	ALU REG1 MICROINSTRUCT REG
2	4	FF239821	FFFFFFF 1000110000000001010000
3	Low		
4		MEM DAT REG	ALU REG2 NEXT MICRO INSTRUCT
5	RESET	FF239821	FFFFFFF 0AD
6	1		
7		MEM ADD REG	ALU CODE MICROSTORE
8	PRGM CNTR	000000AC	1 1110100000000000001101
9	000000AD		1000110000000001001001
10		DBI REGISTER	ALU OUT 1000110000000010001001
11	INSTRT REG	00000015	FFFFFFF 1000100000000100001001
12	00000015		1000100000010000001001
13		EXTNL DBUS	STAT REG 1000100001000000001001
14	REGISTER A	FF239821	0101 1110100000000000001001
15	FF239821		1000110000000001010000
16		MEMR	1000110000000010010000
17	REGISTER B	0	1000100000000100010000
18	0023C3D3		1000100000010000010000
19		MEMW	1000100001000000010000
20	INDX REG X	1	1000110000000001100000
21	0000022B		1000110000000001101000
22			1000110000000010100000
23	INDX REG Y		1000110000000010101000
24	0000022C		0001010000000000010000
25			0011010000000000010000
26	STACK REG		0101000000000000010000
27	00000045		0111000000000000010000
28			1011000000000000010000
29			0001010000000000010000
Interaction Window			
File		Size	Main Menu
		GoTo	Calc
			Exit

Figure 18. Logic Calc Display of Microprocessor

```
(Defun EDB    () (Cell B15)) ;External Data Bus
(Defun Adder  () (Cell C12)) ;Adder/Subtractor
```

The second part of the driver program was developed to test the state of the clock, simulated by Cell A2. These functions were used by the registers and returned a value of "T" if the clock was in the corresponding state:

```
(Defun Rising () (Zerop (Cell A2)))
(Defun High   () (Equal (Cell A2) 1))
(Defun Falling () (Equal (Cell A2) 2))
(Defun Low    () (Equal (Cell A2) 3))
```

A third set of user-defined functions were used to test the cells that simulated the Reset signal, Memory Read, and Memory Write:

```
(Defun Reset () (Zerop (Cell A6)))
(Defun MemR  () (Zerop (Cell B21)))
(Defun MemW  () (Zerop (Cell B24)))
```

The last part of the driver program consisted of a set of user-defined functions that were used to test individual bits within the microcode instruction register, cell D2. Up to five bits of the microcode instruction register could be tested. The function "Microbitp" returned a value of "T" if all bit positions listed as parameters were 1. The function "Microbitn" returned a value of "T" if all bit positions listed as parameters were 0. These two functions were extremely valuable for two reasons. First, they provided documentation of cell formulas. Second, and perhaps more important, is that they provided a simple method of

masking and testing for any combination of bits in the microinstruction register. Such Boolean operations are lacking in Lotus 1-2-3. The listings for these two functions follow:

```
(Defun Microbitp (A &Optional B C D E)
  (And (Cond (E (Logbitp E (Cell D2))) (T))
    (Cond (D (Logbitp D (Cell D2))) (T))
    (Cond (C (Logbitp C (Cell D2))) (T))
    (Cond (B (Logbitp B (Cell D2))) (T))
    (Logbitp A (Cell D2))))

(Defun Microbitn (A &Optional B C D E)
  (Not (Or (Cond (E (Logbitp E (Cell D2))) (T))
    (Cond (D (Logbitp D (Cell D2))) (T))
    (Cond (C (Logbitp C (Cell D2))) (T))
    (Cond (B (Logbitp B (Cell D2))) (T))
    (Logbitp A (Cell D2)))))
```

5.1.2 CONSTRUCTION OF INDIVIDUAL COMPONENTS

Constructing a digital component was a three step process. First, microcode instruction fields were assigned to control the component. Second, the cell formula was written. The "Dual-Rank-Register" function was used in the cell formulas for all registers in the microprocessor. Other components, such as the data buses and control signals that formed interconnections between the registers, were constructed as a Boolean combination of the registers. Although these components were not registers, they were still controlled by bits within the microinstruction register. The last step in constructing a digital component was to individually test it. This was accomplished by loading dummy values into registers and databuses, setting the microinstruction register with a suitable value, and manually clocking the digital

system by clicking on the "Calc" item from the Logic Calc Main Menu. It was very easy to determine if the component was functioning correctly. By testing individual components in this manner, the design of the individual components within the digital system was validated.

The next two subsections detail the construction of two example components: the program counter and the internal data bus.

5.1.2.1 EXAMPLE DIGITAL COMPONENT: THE PROGRAM COUNTER

The first component that was developed was the program counter. A program counter must have the ability to be reset, increment by one, stay the same, or load a value from the internal databus for branching. Because its output is used by other registers using the same clock edge, a dual-rank-register must be used. The reset signal, set externally, and the last two bits of the microcode instruction register are used to control the program counter. With the logic shown by Table 9, the program counter's cell formula is:

```
(Dual-Rank-Register Program-Counter (Rising) (Falling)
(Cond ((Reset) 0)
      ((Microbitp 0) (+ 1 (PC)))
      ((Microbitp 1) (IDB)))
(T (PC)))
```

In this example, the first parameter to dual-rank-register, "Program-Counter," simply provides a variable name for

<u>Reset Signal</u>	<u>Microcode Bit 1</u>	<u>Microcode Bit 0</u>	<u>Function</u>
0	X	X	Reset to Zero
1	0	0	Remain the Same
1	0	1	Increment By One
1	1	X	Clock in Internal Data Bus

TABLE 9. CONTROL OF THE PROGRAM COUNTER REGISTER

Logic Calc to store the output of the first flip-flop of the register as described in Section 4.5. The next two parameters, "(Rising)" and "(Falling)," describe when to clock the two flip-flops used to construct the register. The final argument, the conditional construct, uses the user-defined function "Microbitp" and the value of the "Reset" cell to determine what the program counter should clock in.

To test the program counter, the internal data bus was loaded with a dummy variable. The microinstruction register was then set with the decimal values 0, 1, 2, and 3. For each setting of the microinstruction register, the system was manually clocked with the "Calc" item on the Logic Calc Main Menu. It was quite simple to check that the program counter was working correctly. Thus, its design was validated very rapidly.

5.1.2.2 EXAMPLE DIGITAL COMPONENT: THE INTERNAL DATA BUS

The microprocessor has a single internal data bus as depicted by Figure 17. The internal data bus was controlled by

bits 19, 20, and 21 of the microinstruction register. These bits simply selected which components output was to be gated onto the internal databus. A slight problem was encountered here in the construction of the cell formula. The cell formula can easily be described with a Common Lisp "Case" macro [5,12]. Common Lisp, however, expands its macros upon evaluation into a form that optimizes speed and restores this expansion [12]. This results in a cell formula that is difficult to read and edit. An alternative cell formula is simply a large conditional statement. Although the alternative cell formula is detailed and lengthy, its construction was relatively easy due to the self-documenting features of the formula. With the logic given in Table 10, the internal data bus cell formula is:

```
(Cond ((Microbitn (19 20 21)) (RegA))
      ((And (Microbitp 19) (Microbitn 20 21)) (RegB))
      ((And (Microbitp 20) (Microbitn 19 21)) (RegX))
      ((And (Microbitp 19 20) (Microbitn 21)) (RegY))
      ((And (Microbitp 21) (Microbitn 19 20)) (EDB))
      ((And (Microbitp 19 21) (Microbitn 20)) (SP))
      ((And (Microbitp 20 21) (Microbitn 19)) (PC))
      (T (Adder)))
```

It was simple to test the internal data bus. Because all registers had been constructed at this point, no dummy values were needed. The microcode instruction register was simply loaded with proper values and the spreadsheet manually recalculated to update the contents of the cell simulating the internal data bus. The contents of the internal data bus could then be checked against the register identified by bits 19, 20, and 21 of the

Microcode Instruction			Internal Data Bus
Bit 21	Bit 20	Bit 19	Contents
0	0	0	Register A
0	0	1	Register B
0	1	0	Register X
0	1	1	Register Y
1	0	0	External Data Bus
1	0	1	Stack Register
1	1	0	Program Counter
1	1	1	Adder/Subtractor

TABLE 10. CONTROL OF THE INTERNAL DATA BUS

microinstruction register. With these tests, the design of the internal data bus was validated.

5.1.3 WRITING MICROCODE

After all components were constructed and individually tested, the microcode was written. This too was an easy process because of Logic Calc's ability to display a cell's contents in binary. The microcode consisted of forty-eight instructions, each twenty-two bits wide. The microcode was stored in cells D9 through D58 as constant cells.

Microinstructions were developed to fetch an individual macro instruction from memory and to execute each of the macro instructions. To execute most macro instructions, only one or two microinstructions were needed. A few macro instructions, however, required as many as four microinstructions for execution. As each set of microinstructions was developed for a particular macro

instruction, they were manually loaded into the microinstruction register, and the system was manually clocked. This provided an easy method for testing of the microcode. These tests validated the design of both the macro instruction and the set of microinstructions used to execute the macro instruction.

5.1.4 THE MICROCONTROLLER

The final element to be developed for the microprocessor was the microcontroller. A Logic Calc driving program was utilized to simulate the microcontroller and drive the clock. The code for this program was simply added to that already written for cell naming and testing described in Section 5.1.1. The code simply initializes the program counter to zero by loading a value of zero into Cell A6 which acts as a Reset signal. Next, the fetch-execute cycle of the microprocessor is initiated. The microcontroller simply loads microcode to fetch a macro instruction from memory and then, based on the contents of the macro instruction register and the status register, loads microcode to execute the macro instruction. Between each portion of this fetch-execute cycle, the driving program cycles the clock by evaluating the Logic Calc function "Calc" four times. This non-overlapped fetch-execute cycle continues until the instruction register is loaded with a macro instruction of 0. This is the opcode for Halt. At this point, the driving program terminates.

The complete Logic Calc driving program for this microprocessor is listed in Appendix B.

5.2 FINAL TESTING OF THE MICROPROCESSOR

Due to the stepwise-refinement design techniques offered by Logic Calc, a good bit of testing and validation had already been accomplished during the design of the microprocessor; therefore, final testing was approached with confidence.

First, the driving program was modified slightly to allow various settings of the stepping of the microprocessor's clock. With this modification, it was possible to select from a menu single step, multiple steps, or full speed operations of the microprocessor clock. If full speed operations was selected, interim results are not displayed. This is accomplished with the "no-redisplay" keyword in the "Calc" function. By controlling the stepping of the clock in this manner, testing could be accomplished at various levels of detail.

Next, single instructions were loaded in cells that functioned as memory locations. The driving program was then executed to drive the clock in a single step mode and function as the microcontroller. Interim results were checked between each machine cycle.

With every macro instruction checked individually, memory was next loaded with simple sequences of instructions. The driving program was executed to drive multiple steps of the clock.

This method of testing was similar to setting a breakpoint with a debugger. With this method of testing, it was easy to follow the progress of the instruction sequence by watching key components such as the program counter, but only the final result could be checked in detail. Still, this was an effective means of testing.

Finally, a few programs were written that used a full spectrum of the macro instructions. These programs were run at full speed so interim results were not checked. The programs consisted of simple multiplication by addition, division by subtraction and squaring integers. In addition to the single-stepped and multiple-stepped tests, these full-speed programs validated the design of the microcontroller and the overall performance of the microprocessor.

Because of the high amount of looping in the full-speed programs, these tests were quite long, sometimes requiring hours to complete. For a program to square the numbers from one to ten, the microprocessor utilized 781 clock periods. As described in Section 4.4, Logic Calc requires four spreadsheet recalculations to simulate one clock period; therefore, Logic Calc performed this simulation by performing 3124 spreadsheet recalculations (781×4). Logic Calc used 25 formula cells to simulate the microprocessor and 24 formula cells to simulate 24 locations in RAM memory resulting in a total of 49 formula cells. It accomplished the simulation in one hour and three minutes

giving it a simulation speed of .21 clock-periods/second and a formula recalculation rate of 40.5 formulas/second. The simulation speed could be improved by eliminating the twenty-four RAM formula cells and running programs which used only ROM memory. The speed for these ROM-only simulations was .42 clock-periods/second with a similar formula recalculation rate of 41.8 formulas/second.

Although a direct comparison between simulation speeds between Lotus 1-2-3 and Logic Calc was not made, it was noted that Lotus 1-2-3 has a significantly higher formula recalculation rate for identical cell formula entries. This apparent performance advantage of Lotus 1-2-3 is offset by the fact that Lotus 1-2-3 lacks the Boolean operations and specialized functions of Logic Calc. Lotus 1-2-3 therefore requires more formula cells than Logic Calc to design a digital system, resulting in simulation speeds similar to Logic Calc. As a crude comparison, a numeric coprocessor was designed with Lotus 1-2-3 during previous research at the University of Texas [13]. The complexity of this coprocessor was similar to that of the 32-bit microprocessor designed with Logic Calc. The Lotus 1-2-3 design, however, required 107 formula cells and ran on an IBM PC AT with a simulation speed of .61 clock-periods/second. The Lotus 1-2-3 simulation speed for the coprocessor is faster than that of Logic Calc because the Lotus 1-2-3 design utilized a "clock" cell with

only two states: high and low. If the Lotus 1-2-3 design was simulated with a more precise four-state clock such as that used in the Logic Calc design, the simulation speeds of the two spreadsheets would be approximately the same.

CONCLUSION

6.1 SUMMARY OF RESULTS

Logic Calc was developed to be used as a design tool for digital systems. It was intended to eliminate the shortcomings of financial spreadsheets when used as design tools. Each of the improvement areas listed in Section 1 is summarized:

1. A Maximum value for fixed-point integers. The Lisp programming language eliminated this shortcoming. Lisp allows integers to be any size by providing software to represent and operate on integers that are larger than one data word. Integers larger than 32 bits are called expressed as "Bignums" in Lisp. Software arithmetic routines are provided by Lisp to allow bignums to maintain the same precision as fixed-point integers.
2. Fixed size integers. Logic Calc eliminated this shortcoming by providing a bit property for each cell. This provided for a more precise mapping of digital hardware onto a spreadsheet.
3. Lack of Boolean operations. Lisp eliminated this shortcoming. In Lisp, a full set of Boolean operations is available, including operations at the bit level.
4. Lack of binary and hexadecimal display. Logic Calc eliminated this shortcoming by providing both of these display formats.
5. Inability to simulate some key digital components in a single cell. Logic Calc offers much improvement in this area, by providing the "Dual-Rank-Register" function and allowing the user to write his own functions. Many of the components that require several cells in a financial spreadsheet can be expressed as a single cell in Logic Calc.

6. Slow and cumbersome programmatic operation of the spreadsheet. Logic Calc's method of using a separate driving program is superior to Lotus macros in that it offers full screen editing features, is highly flexible, and can be compiled. A direct comparison of speed for programmatic operation of Logic Calc and Lotus 1-2-3, however, shows that Lotus 1-2-3 is slightly faster. This is probably due to the fact that Logic Calc does not compile its cell formulas whereas Lotus 1-2-3 partially compiles its cell formulas. This item is addressed in the next subsection.
7. No modification capabilities. Logic Calc eliminated this shortcoming. High-level-language source code is obtainable for Logic Calc. It should also be relatively easy to modify because Logic Calc is written in Common Lisp, the same language that the user utilizes to write cell formulas, the same language that the user utilizes to write a Logic Calc driving program.

The design of the microprocessor presented in Section 5 was both easy and fast. The ease of testing was due to Logic Calc's interactive nature, self-documenting capabilities, and highly visible results. It took only three days to design and partially test the microprocessor. This design process was a simple, straight-forward stepwise-refinement of the graphical depiction of the design. Digital components were individually designed, tested, and validated using dummy variables for components not yet designed. As the number of validated components grew, the number of dummy variables used to represent them decreased. Eventually, all components were individually designed, tested, and validated, and there were no dummy variables. Full scale testing was next and was also quite

rapid: the full scale tests took several hours, but could be run overnight. These test were sufficient to validate the design of the microprocessor. The resulting design, completely validated, could be passed down to the next level of design with sufficient detail and documentation to ensure that no design errors would be passed down.

Logic Calc, therefore, was proven to be an effective and efficient design tool for digital systems. It presents an ability to completely validate a design at the architectural level, the highest level of design. Its use of well-proven computer science concepts of interaction, visibility, self-documentation, and stepwise-refinement of the problem enable it to be used with the greatest of ease.

6.2 FUTURE RESEARCH

Logic Calc, being a prototype, is not perfect. A few changes to Logic Calc could improve its use as a digital design tool:

1. Currently in Logic Calc, in order to enter a formula cell, it must be entered manually. Although the previous formula entry process is frustrating and error-prone, a feature formula editor could be designed to eliminate this problem.
2. Currently in Logic Calc, the formula is entered in a form such that it must be executed as a single calculation. A feature for recalculation of the formula could be added to Logic Calc. This feature could be designed to store the formula in a cell and then calculate the result of the formula.

AD-A185 289

LOGIC CALC: A DESIGN TOOL FOR DIGITAL SYSTEMS(U) AIR
FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
G D ROSENBERGER AUG 87 AFIT/CI/NR-87-55T

242

UNCLASSIFIED

F/G 12/5

NL

END
11/1
2016



need the ability to reconstruct the original cell formula for editing purposes.

3. Certain errors in cell formulas, such as adding nil to an integer, are identified by the Explorer debugger. Although the Explorer debugger prints out detailed error messages and is well-documented, a better error-handler could be developed from within Logic Calc. This error-handler could simply point out errors by displaying "Error" in cells that contain formula errors. This method, patterned after error-handling in Lotus 1-2-3, would allow the user the capability to rapidly detect the source of errors without leaving Logic Calc and without learning the operation of the Explorer debugger.
4. Although the file operations in Logic Calc are adequate for most applications, it would be useful to add more extensive file operations similar to those offered by Lotus 1-2-3. Additional file operations to save, retrieve, and combine portions of spreadsheets would allow merging of spreadsheets. This merging of spreadsheets can be used to test alternate hardware or software within the same design. Similarly, these additional file operations offer an ability to design a new digital system by assembling parts of digital systems previously designed on Logic Calc. Assembling previously designed and tested parts in this manner offers a fast and flexible technique to the Logic Calc design process.

Logic Calc, patterned after Lotus 1-2-3, retained many general purpose features of Lotus 1-2-3. Another version of Logic Calc, further specialized for design of digital systems, could be designed. Some of the features suggested for this second version are:

1. A simulation of the digital system clock from within Logic Calc. By including global variables and functions describing the state of a clock simulated within Logic Calc, digital components could be constructed on the spreadsheet without first constructing a clock and the functions used to test

its state. The display should be modified to show the state of the clock. Including this internally-simulated clock would also provide a significantly faster spreadsheet recalculation due to the use of global constructs to simulate the clock.

2. Special purpose cell types. In addition to the cell types already included in Logic Calc (null, constant, text, and formula), it would be valuable to include special purpose cell types that simulate specific digital components such as registers, flip-flops, multiplexors... The user would then need to describe only the input into the component. Further simplification of cell construction could be achieved by "prewiring" those special purpose cell types that require a clock to the internally-simulated clock described above. This method of cell construction would be superior to use of functions such as "Dual-Rank-Register" and its bulky list of parameters.
3. Special purpose grouping of cells. Primarily to simulate a memory system, it would be valuable to group cells as one component. Currently in Logic Calc, it is necessary to have a formula for each cell functioning as a single address location within a memory system. This greatly adds to the number of cells that must be recalculated during spreadsheet recalculation. Many cell formulas could be eliminated if a grouping of cells could operate under a single formula. A special purpose grouping of cells could provide this feature and thereby speed up spreadsheet recalculation.

Another area for future research in this area would be to design a digital system more complex than the microprocessor presented in Section 5. It would be valuable to test Logic Calc's capabilities to design a digital system that uses parallel or pipelined architectural features.

Logic Calc also offers the capability to develop new software on existing digital systems. Microstore could be quickly changed by replacing those cells that function as microstore with

new cells representing the new microstore. Assemblers and loaders could be developed as Logic Calc driving programs to automatically generate binary code from assembly language mnemonics and load the binary code in memory. A debugger could also be written as a Logic Calc driving program to run the simulations. Logic Calc could be used to keep track of machine cycles, memory bandwidth, frequency of branching, and other such parameters that are measurements of performance when developing new software. It would be valuable to evaluate Logic Calc's ability to perform as such a software development tool.

A final area for future research would be the development of an expert graphics editor that could interface to the front end of Logic Calc. With such a device, digital systems could be designed by developing a graph such as shown in Figure 5 on an expert machine. The graphics editor could then make Logic Calc cell entries automatically. This graphics approach to digital design would be quite interesting.

Interested readers can contact Professor Harvey Cragon at the Electrical Engineering Department, University of Texas at Austin, for further information and a copy of Logic Calc.

APPENDIX A: LOGIC CALC SOURCE CODE

; LOGIC-CALC

; A SPREADSHEET PROGRAM DEVELOPED BY:

; GLENN D. ROSENBERGER, CAPTAIN USAF
; 112 HABICHT STREET, JOHNSTOWN PA 15906
; 814-536-1089

; DEVELOPED AT THE UNIVERSITY OF TEXAS, AUSTIN
; IN THE SPRING SEMESTER, 1987

; LOGIC CALC WAS WRITTEN TO BE USED AS A INTERACTIVE DESIGN TOOL
; FOR DIGITAL SYSTEMS. THE PROGRAM CONSTRUCTS AN ARRAY OF CELL
; OBJECTS AND THEN ALLOWS THE USER TO MANIPULATE PROPERTIES OF
; THE OBJECTS VIA AN INTERACTIVE MOUSE AND KEYBOARD INTERFACE
; SIMILAR TO MOST SPREADSHEETS. BY CONSTRUCTION OF FORMULAS THAT
; SIMULATE DIGITAL LOGIC UNITS, THE SPREADSHEET CAN SIMULATE MOST
; DIGITAL SYSTEMS. DURING EACH SPREADSHEET RECALCULATION, THE
; FORMULAS ARE REEVALUATED AND EACH CELL IS SET TO ITS NEW VALUE.
; THIS METHOD OF RECALCULATION SIMULATES THE FUNCTION OF A CLOCK
; IN A DIGITAL SYSTEM.

; THE PROGRAM IS CONSTRUCTED NEARLY ENTIRELY OF LISP METHODS AND
; FORMULAS, EACH OF WHICH IS DOCUMENTED IN SUFFICIENT DETAIL.
; MOST FUNCTIONS ARE USED BY THE SPREADSHEET PROGRAM ITSELF, BUT
; A FEW ARE DESIGNED TO BE ENTERED INTO A CELL AS A FORMULA CALL,
; AND A FEW ARE DESIGNED TO BE USED IN A USER-DEFINED DRIVER
; PROGRAM. THIS DRIVER PROGRAM CAN BE USED TO RECALCULATE THE
; SPREADSHEET, AS WELL AS TEST AND LOAD CELL CONTENTS AND THUS
; SIMULATE A CONTROLLING PROCESSOR.

; THE MAIN-PROGRAM LOOP RESIDES IN THE FORMULA "RESTART". THE
; PROGRAM CALLS OTHER FUNCTIONS FROM THIS LOOP BASED ON THE
; USER'S INPUT. TO EXIT THE EDITING MODE THAT THIS MAIN PROGRAM
; PROVIDES, THE USER SHOULD CLICK ON THE "EXIT" ITEM ON THE
; PROGRAM'S MAIN MENU. SUBSEQUENT REENTRIES CAN BE ACCOMPLISHED
; BY CALLING THE FUNCTION "RESTART".

; SPREADSHEET FILES CAN BE STORED AND READ FROM THE LOCAL
; MACHINE'S LOGIC-CALC DIRECTORY USING THE "FILES" MENU ITEM.

```

; The cell object is the primary data object of the spreadsheet.
; Property Definitions:
;   Type : empty, constant, string, or formula.
;   Value: the value of this cell, this property may be either
;           an integer or a string.
;   Formula: the lisp formula used to obtain the value (or nil
;             if not a formula cell.)
;   Bits: the maximum number of bits that are used to hold the
;          cell's value. If the value exceeds this amount, the
;          higher order bits are stripped. This property allows
;          the spreadsheet to closely simulate a digital system
;          by providing a specification on the size of a
;          digital component (in bits). The maximum value for
;          bits should be 127 without modifying the program.
;   Recalc: this is a boolean variable that is set in formula
;           cells during recalculation. It is necessary to
;           permit access to other cells in an arbitrary manner,
;           and allow circular references.
;   Changed-value: this boolean property is maintained in
;                  order to speed up the display process. During recal-
;                  culation, only cells whose value has changed are
;                  redisplayed.
;   Output-display: X hexadecimal, B binary.
(defflavor cell ((value 0) (formula nil) (bits 32)(recalc nil)
  (type 'empty)(changed-value t) (output-display 'X)) ()
  :gettable-instance-variables :settable-instance-variables :
  inittable-instance-variables)

```

```

;Global variables follow with their definition and use

```

```

(defvar max-number-of-columns 300
  "the maximum number of columns in any spreadsheet")
(defvar max-number-of-rows 400
  "the maximum number of rows in any spreadsheet")
(defvar number-of-rows 10
  "the current number of rows in the working spreadsheet")
(defvar number-of-columns 20
  "the current number of columns in the working spreadsheet")
(defvar first-display-row 1
  "the first row printed when the spreadsheet is in view")
(defvar first-display-column 1
  "the first column printed when the spreadsheet is in view")
(defvar last-display-row :unbound
  "the last row displayed. Bound in print-spreadsheet")
(defvar last-display-column :unbound
  "the last column displayed. Bound in print-spreadsheet")
(defvar user-input :unbound

```

```

"The user's input to the spreadsheet during editing")
(defvar formula-list nil
  "a list cell indices for all cells that contain formulas")

;the primary data array is a global variable named spread:
(defvar spread (make-array (list (+ 1 number-of-rows)
                                (+ 1 number-of-columns)))
  "the array of cells that comprise the working spreadsheet")

;the user interface for data display is developed as a
; mouse sensitive window:
(defflavor spreadsheet-window ()(tv:basic-mouse-sensitive-items
  tv:truncating-window tv:stream-mixin))

;four item-type association lists are used with the mouse
; sensitive window
(defvar edit-list '((cell-type cell-view
  "Left: View this cell   Right: Menu of Cell Operations"
  ("MENU OF CELL OPERATIONS" :no-select t)
  ("View Cell" :value cell-view
    :documentation "View this cell")
  ("Edit Cell" :value cell-edit
    :documentation "Edit this cell")
  ("Change Cell Size" :value cell-size
    :documentation "Change Size of size in bits")
  ("Change Output Display" :value cell-output-display
    :documentation "Change the output display of this cell")
  ("Move Cell" :value cell-move
    :documentation "Move Cell to New Location")
  ("Copy Cell" :value cell-copy
    :documentation "Copy Cell to Another Location")
  ("Erase Cell" :value cell-empty :documentation "Erase Cell"))
  (row-type insert-row "Left: Insert a Row
    Right: Menu of Row Operations"
    ("Insert Row" :value insert-row
      :documentation "Insert a Row at this Location")
    ("Delete Row" :value delete-row
      :documentation "Delete this Row")
    ("Move Row" :value move-row
      :documentation "Move all Cells in this Row to a New Row")
    ("Copy Row" :value copy-row :documentation
      "Copy all Cells in this Row to another Row"))
  (column-type width
    "Left: Change Column Width
    Right: Menu of Column Operations"
    ("Change Column Width" :value width
      :documentation "Change Column Width")
    ("Insert Column" :value insert-column

```

```

      :documentation "Insert a Column at this Location")
    ("Delete Column" :value delete-column
     :documentation "Delete this Column")
    ("Move Column" :value move-column
     :documentation
      "Move all Cells in this Column to a New Column")
    ("Copy Column" :value copy-column
     :documentation
      "Copy all Cells in this Column to another Column"))))
  "Edit-list is the primary mouse sensitive item-type
  association list. Used during edit mode" )

(defvar cell-move-copy '((cell-type mark
  "Left: Make this cell target location
  Right: Select or Abort"
  ("MENU" :no-select t)
  ("Select this cell" :value mark
   :documentation "Make this cell target location")
  ("Abort Move/Copy Operation" :value abort
   :documentation "Do not perform Move/Copy Operation"))))
  "Cell-move-copy is a mouse sensitive item type association
  list. Used when moving or copying cells")

  defvar row-move-copy '((row-type mark
    "Left: Make this row target location
    Right: Select or Abort"
    ("MENU" :no-select t)
    ("Select this row" :value mark
     :documentation "Make this row target location")
    ("Abort Move/Copy Operation" :value abort
     :documentation "Do not perform Move/Copy Operation"))))
    "Row-move-copy is a mouse sensitive item type association list.
    Used when moving or copying rows")

  (defvar column-move-copy '((column-type mark
    "Left: Make this column target location
    Right: Select or Abort"
    ("MENU" :no-select t)
    ("Select this column" :value mark
     :documentation "Make this column target location")
    ("Abort Move/Copy Operation" :value abort
     :documentation "Do not perform Move/Copy Operation"))))
    "Column-move-copy is a mouse sensitive item type association
    list. Used when moving/copying columns")

  (defvar program-constraint-window
    (make-instance 'tv:bordered-constraint-frame
      :panes

```

```

'((spreadsheet-pane spreadsheet-window
  :label (:string "LOGIC CALC"
    :font fonts:bigfnt :centered :top)
  :blinker-p nil
  :item-type-alist edit-list)
(main-menu-pane tv:command-menu
  :item-list
    (("File" :value file :documentation
      "Read or Write Spreadsheet Files")
     ("Size" :value size
      :documentation "Change Worksheet Size")
     ("Go To" :value go-to :documentation
      "Show a different region in Spreadsheet")
     ("Calculate" :funcall calc :documentation
      "Make a single Worksheet Recalculation")
     ("Exit" :value exit
      :documentation "Exit Logic Calc"))
  :default-font fonts:courier
  :label (:string "MAIN MENU"
    :font fonts:courier :centered))
(interaction-window-pane tv:window
  :label (:string "INTERACTION WINDOW")))
':constraints
'((main . ((spreadsheet-pane
  interaction-window-pane main-menu-pane)
  ((spreadsheet-pane 530))
  ((main-menu-pane :ask :pane-size))
  ((interaction-window-pane :even))))))
"Program-constraint-window is a constraint window for the
spreadsheet, interaction-window and menu. This variable
describes the size and position of these three windows")

(defvar spreadsheet (send program-constraint-window
  ':get-pane 'spreadsheet-pane)
"spreadsheet gives access to the spreadsheet window in the
program-constraint-window")

(defvar interaction-window (send program-constraint-window
  ':get-pane 'interaction-window-pane)
"interaction-window gives access to the interaction-window in
the program-constraint-window")

(defvar main-menu (send program-constraint-window
  ':get-pane 'main-menu-pane)
"main-menu gives access to the main-menu window in the
program-constraint-window")

```

```

(defvar program-io-buffer (tv:make-io-buffer 500.)
  "the program's io buffer")

;the column object is developed to hold three properties:
; letter: gives unique column identifier to each column
; width: the printed width of the column
; position: the x-coordinate on the screen for the beginning
; of the column. This is needed during redisplay of
; individual items.
(defflavor column-flavor (letter (width 20) position)()
  :gettable-instance-variables :settable-instance-variables
  :inittable-instance-variables)

(defvar column (make-array (+ 1 max-number-of-columns))
  "an one dimensional array of column objects. Initialized
  in the restart function.")

;when invoked the make-constant method sets the cell type to
; 'constant and the value to the provided input-value.
(defmethod (cell :make-constant) (input-value)
  (let ((cell-bits))
    (send self :set-type 'constant)
    ;Right justify cell's value to the specified number of bits.
    ; The complicated shifting for large cell sizes is necessary
    ; because byte operations only work with fixnums as byte
    ; specifiers.
    (send self :set-value
      (cond ((> 64 (setq cell-bits (send self :bits)))
        (ldb (byte cell-bits 0) input-value))
      (t (+ (ash (ldb (byte (- cell-bits 63) 0)
        (ash input-value -63)) 63)
        (ldb (byte 63 0) input-value))))))
    (send self :set-changed-value t)
    (send self :set-formula nil)))

;when invoked the make-text method sets the cell type to 'text
; and the value to the provided input-value
(defmethod (cell :make-text) (input-value)
  (send self :set-type 'text)
  (send self :set-value input-value)
  (send self :set-changed-value t)
  (send self :set-formula nil)
  (send self :set-bits 32))

;when invoked the copy method sets the cell's type, value,
; formula, bits, and recalc properties to the same values as the
; properties of the cell located at i j.
(defmethod (cell :copy-cell) (i j)

```

```

(send self :set-type (send (aref spread i j) :type))
(send self :set-value (send (aref spread i j) :value))
(send self :set-formula (send (aref spread i j) :formula))
(send self :set-bits (send (aref spread i j) :bits))
(send self :set-recalc (send (aref spread i j) :recalc))
(send self :set-changed-value t))

```

;when invoked the make-formula method sets the cell type to
; 'formula and the formula to the provided input-formula.

```

(defmethod (cell :make-formula) (input-formula)
  (send self :set-type 'formula)
  (send self :set-formula input-formula)
  (send self :set-recalc nil)
  (send self :set-changed-value t))

```

```

(defun column-string (j)
  "Returns a string corresponding to the supplied column number"
  (let ((first) (second))
    (cond ((< j 27)(make-string 1 :initial-element (+ 64 j)))
          (t (progn (multiple-value-setq (first second)
                                           (truncate j 26))
                     (cond ((equal second 0)
                           (progn (setq first (- first 1))
                                   (setq second 26))))
                     (concatenate 'string
                                   (make-string 1 :initial-element (+ 64 first))
                                   (make-string 1
                                               :initial-element (+ 64 second)))))))

```

;the next four functions are called to prompt the user for more
; data in the development of different types of cell objects.
; The program's formula list is updated constantly.

```

(defun make-empty-cell (i j)
  "makes the cell at location i j empty by simply creating
  a new cell object"
  (aset (make-instance 'cell) spread i j)
  (setq formula-list
        (remove (list i j) formula-list :test 'equal)))

```

```

(defun make-constant-cell (i j)
  "prompts the user for an input and makes the cell at location
  (i j) a constant cell if the user supplied a valid value"
  (let ((input-value))
    (cell-view i j)
    (write-string "Enter decimal integer for this cell
                  (Prefix: #X: hex, #O: octal, #B: binary): "
                  interaction-window)
    (tv:turn-on-sheet-blinkers interaction-window)

```

```

(setq input-value (read-from-string
                    (read-line interaction-window) nil 0))
(cond ((integerp input-value)
      (progn
        (setq formula-list
              (remove (list i j) formula-list :test 'equal))
        (send (aref spread i j) :make-constant input-value))
      (t (send interaction-window :beep))))

(defun make-string-cell (i j)
  "prompts the user for an input and makes the cell at location
  (i j) a text cell if the user supplied a valid value"
  (let ((input-value))
    (cell-view i j)
    (write-string "Enter text entry for this cell: "
                  interaction-window)
    (tv:turn-on-sheet-blinkers interaction-window)
    (setq input-value (read-line interaction-window))
    (setq formula-list
          (delete (list i j) formula-list :test 'equal))
    (send (aref spread i j) :make-text input-value)))

(defun make-formula-cell (i j)
  "prompts the user for an Lisp formula input and makes a formula
  cell at location (i j)"
  (let ((input-value))
    (cell-view i j)
    (write-line "Enter lisp formula for this cell: "
                interaction-window)
    (tv:turn-on-sheet-blinkers interaction-window)
    (setq input-value
          (read-from-string (read-line interaction-window) nil nil))
    (cond ((listp input-value)
          (progn
            (setq formula-list
                  (delete (aref spread i j) formula-list))
            (send (aref spread i j) :make-formula input-value)
            (evaluate-cell i j)
            (setq formula-list (cons (list i j) formula-list))))
          (t (send interaction-window :beep)))))

(defun format-cell (i j)
  "returns a string that represents the cell's value. If the
  value is a text string, it is right-trimmed to fit in the
  current column width. If the value is an integer, but too
  large to be displayed properly, a string of astericks is
  returned."
  (let ((printed-width)

```



```

        (list j) (send (aref column j) :letter))
      (send spreadsheet :increment-cursorpos
        (- (send (aref column j) :width) 1)
        0 :character)
      (setq last-display-column j))
      (progn (setq last-display-column (- j 1))
        (return-from column-letters))))))
;print each cell as a mouse sensitive item
(loop for i from first-display-row to last-display-row do
  (send spreadsheet :set-cursorpos 3 display-row :character)
  (send spreadsheet :item 'row-type (list i)
    (format nil "~A" i))
  (loop for j from first-display-column to
    last-display-column do
    (send spreadsheet :set-cursorpos
      (send (aref column j) :position) display-row :character)
    (send spreadsheet :item 'cell-type (list i j)
      (format-cell i j))
    (send (aref spread i j) :set-changed-value nil))
    (setq display-row (+ 1 display-row)))
  (send interaction-window :select)
  (tv:turn-off-sheet-blinkers interaction-window)))

(defun print-spreadsheet-changed-items ()
  "A faster print function. This can reprint the updated cells
  without reprinting column letters and row numbers. Because
  values are bound for last-display-row and last-display column
  are bound in print-spreadsheet, no computation of these values
  is necessary."
  (let ((display-row 2)(output-string))
    (send spreadsheet :select)
    ;print each cell that has changed-value set to true and
    ;update changed value flags
    (loop for i from first-display-row to last-display-row do
      (loop for j from first-display-column to
        last-display-column do
          (cond ((send (aref spread i j) :changed-value)
            (progn
              (send (aref spread i j) :set-changed-value nil)
              (setq output-string (format-cell i j))
              (send spreadsheet :set-cursorpos
                (send (aref column j) :position)
                display-row :character)
              (send spreadsheet :clear-string output-string)
              (send spreadsheet :item 'cell-type
                (list i j) output-string))))))

```

```

    (setq display-row (+ 1 display-row)))
  (send interaction-window :select)
  (tv:turn-off-sheet-blinkers interaction-window)))

(defun row&col (cell)
  "returns two values corresponding to the indices of the cell
  name entered as an argument. If the cell name does not
  correspond to a valid cell, nil is returned. This function is
  called often and designed for speed."
  (let* ((cell-name (format nil "~A" cell))
        (column-number (- (char-code (char cell-name 0)) 64))
        (possible-column-number)
        (start-integer 1)(row-number))
    (cond ((or (< (length cell-name) 2)
              (> 1 column-number) (< 26 column-number))
      (progn (send interaction-window :beep)
              (return-from row&col nil))))
    ;test for second character to be a letter
    (setq possible-column-number
      (- (char-code (char cell-name 1)) 64))
    (cond ((and (< 0 possible-column-number)
                (> 27 possible-column-number))
      (progn (setq column-number (+ possible-column-number
                                     (* 26 column-number)))
              (setq start-integer 2))))
    (setq row-number
      (parse-integer cell-name :start start-integer
                     :junk-allowed t))
    (cond ((and (integerp row-number) (plusp row-number)
                (<= row-number number-of-rows)
                (plusp column-number)
                (<= column-number number-of-columns)
                (<= column-number number-of-columns)
                (<= row-number number-of-rows))
      (values row-number column-number))
      (t (progn (send interaction-window :beep)
                  (return-from row&col nil))))))

(defun evaluate-cell (i j)
  "When called to evaluate a cell, this function evaluates the
  form stored in cell (i j) if the cell type is a formula. It
  must first test and set the recalc flag to true, so that
  circular references will be handled correctly. The specified
  byte is removed from the computed value and the cell's value
  is updated to match this byte."
  (let ((previous-value) (computed-value) (new-value)(bits))
    (cond ((and (equal 'formula (send (aref spread i j) :type))
                (not (send (aref spread i j) :recalc)))

```

```

(progn
  (send (aref spread i j) :set-recalc t)
  (setq previous-value
    (send (aref spread i j) :value))
  (setq computed-value
    (eval (send (aref spread i j) :formula)))
  (cond ((integerp computed-value)
    ;remove byte specified by digits.
    (progn
      (cond ((> 64 (setq bits
        (send (aref spread i j) :bits)))
        (setq new-value (ldb (byte bits 0)
          computed-value)))
        (t (setq new-value
          (+ (ash (ldb (byte (- bits 63) 0)
            (ash computed-value -63)) 63)
            (ldb (byte 63 0)
              computed-value))))))
      (send (aref spread i j) :set-value new-value)
      (cond ((not (equal previous-value new-value))
        (send (aref spread i j)
          :set-changed-value t))))))
  (t (progn (send (aref spread i j)
    :set-value computed-value)
    (send (aref spread i j)
      :set-changed-value t))))))

(defun cell-view (i j)
  "Allows the user to view the cell that was clicked upon"
  (let ((type (send (aref spread i j) :type))
    (output-value (send (aref spread i j) :value)))
    (send interaction-window :select)
    (send interaction-window :refresh)
    (send interaction-window :clear-screen)
    (write-string "Cell " interaction-window)
    (write-string (column-string j) interaction-window)
    (princ i interaction-window)
    (write-string ": Type: " interaction-window)
    (princ type interaction-window)
    (fresh-line interaction-window)
    (write-string "Value: " interaction-window)
    (princ output-value interaction-window)
    (cond ((not (stringp output-value))
      (progn
        (write-string " (Decimal), " interaction-window)
        (format interaction-window "~X (Hexidecimal)"
          (send (aref spread i j) :value))
        (write-string " Bits: " interaction-window)

```

```

      (prnl (send (aref spread i j) :bits)
            interaction-window)
      (write-string " Output display: " interaction-window)
      (case (send (aref spread i j) :output-display)
        ('X (write-string "Hexidecimal" interaction-window))
        ('B (write-string "Binary" interaction-window))))))
    (cond ((equal 'formula type)
      (progn (fresh-line interaction-window)
        (write-string "Formula: " interaction-window)
        (print (send (aref spread i j) :formula)
              interaction-window))))
    (fresh-line interaction-window)
    (tv:turn-off-sheet-blinkers interaction-window)))

(defun cell-edit (i j)
  "Allows the user to edit the cell clicked upon or exit if
  no menu choice is made"
  (case (tv:menu-choose
    '(("Empty Cell" :value empty :documentation
      "Make this cell empty")
      ("Constant Cell" :value constant :documentation
      "Make this cell a constant")
      ("Text Cell" :value text :documentation
      "Make this cell a Text Cell")
      ("Formula Cell" :value formula :documentation
      "Make this cell a Formula cell"))
    (empty (make-empty-cell i j))
    (constant (make-constant-cell i j))
    (text (make-string-cell i j))
    (formula (make-formula-cell i j)))
    (send interaction-window :clear-screen)
    (tv:turn-off-sheet-blinkers interaction-window)
    (print-spreadsheet-changed-items))

(defun cell-size (i j)
  "Allows the user to change the number of bits for this cell.
  This is the size of the byte that is removed from the cell
  during each computation."
  (let ((input-value))
    (cell-view i j)
    (write-string
      "Enter the size in bits for this cell (decimal value): "
      interaction-window)
    (tv:turn-on-sheet-blinkers interaction-window)
    (setq input-value
      (read-from-string (read-line interaction-window) nil nil))
    (cond ((equal 'text (send (aref spread i j) :type))
      (send interaction-window :beep))

```

```

((not (integerp input-value))
 (send interaction-window :beep))
((not (plusp input-value))
 (send interaction-window :beep))
;127 is max value for bits due to program design
(> 127 input-value)
(progn
 (send (aref spread 1 j) :set-bits input-value)
 (send (aref spread 1 j) :set-value
  (cond ((> 64 input-value)
    (ldb (byte input-value 0)
      (send (aref spread 1 j) :value)))
    (t (+ (ash (ldb (byte (- input-value 63) 0)
      (ash (send (aref spread 1 j) :value)
        -63)) 63) (ldb (byte 63 0)
      (send (aref spread 1 j) :value))))))
 (send (aref spread 1 j) :set-changed-value t)
 (print-spreadsheet-changed-items))
 (t (send interaction-window :beep)))
(send interaction-window :clear-screen)
(tv:turn-off-sheet-blinkers interaction-window)))

(defun cell-output-display (i j)
  "The user can set the cell display to hexadecimal or binary"
  (cell-view i j)
  (case (tv:menu-choose
    '(("CELL OUTPUT DISPLAY" :no-select t)
      ("Hexideximal" :value hex :documentation
        "Set output display to hexadecimal")
      ("Binary" :value binary :documentation
        "Set output display to binary"))))
    (hex (send (aref spread 1 j) :set-output-display 'X))
    (binary (send (aref spread 1 j) :set-output-display 'B)))
  (send (aref spread 1 j) :set-changed-value t)
  (send interaction-window :clear-screen)
  (tv:turn-off-sheet-blinkers interaction-window)
  (print-spreadsheet-changed-items))

(defun get-cell-location-or-abort ()
  "Used during move and copy (of cells). Returns a true if the
  user clicks on a cell. Also allows the Go-To function to be
  performed to allow user access to the entire spreadsheet.
  Returns a nil if a character is typed or the Abort selection
  is chosen."
  (send spreadsheet :set-item-type-alist cell-move-copy)
  (loop (setq user-input (send spreadsheet :any-ty1))
    (cond ((not (listp user-input))
      (return-from get-cell-location-or-abort)))

```

```

((equal :menu (car user-input))
 (cond ((equal 'go-to (caddr(cadr user-input)))
        (funcall 'go-to))))
((equal :typeout-execute (car user-input))
 (cond ((equal 'mark (cadr user-input))
        (return-from get-cell-location-or-abort t))
       ((equal 'abort (cadr user-input))
        (return-from get-cell-location-or-abort))))))

(defun cell-move (i j)
  "Moves cell (i j) to the location specified by the user."
  (cell-view i j)
  (write-string
   "Click Left on the New Location for this Cell
   or Type any Character to Abort Move" interaction-window)
  (cond ((get-cell-location-or-abort)
        (progn
         (aset (aref spread i j) spread
               (car (caddr user-input))(cadr (caddr user-input)))
         (send (aref spread (car (caddr user-input))
                        (cadr (caddr user-input))) :set-changed-value t)
         (cond ((equal 'formula (send (aref spread i j) :type))
                (nsubstitute (caddr user-input) (list i j)
                             formula-list :test 'equal)))
         (aset (make-instance 'cell) spread i j)
         (print-spreadsheet-changed-items)))
        (t (send interaction-window :beep)))
  (send spreadsheet :set-item-type-alist edit-list)
  (send interaction-window :clear-screen)
  (tv:turn-off-sheet-blinkers interaction-window))

(defun cell-copy (i j)
  "Copies cell (i j) to the location specified by the user."
  (cell-view i j)
  (write-string "Click Left on Location for a Copy of this Cell
  or Type any Character to Abort Move" interaction-window)
  (cond ((get-cell-location-or-abort)
        (progn
         (send (aref spread (car (caddr user-input))
                        (cadr (caddr user-input))) :copy-cell i j)
         (cond ((equal 'formula (send (aref spread i j) :type))
                (setq formula-list (cons (caddr user-input)
                                         formula-list))))
         (print-spreadsheet-changed-items)))
        (t (send interaction-window :beep)))
  (send spreadsheet :set-item-type-alist edit-list)
  (send interaction-window :clear-screen)
  (tv:turn-off-sheet-blinkers interaction-window))

```

```

(defun cell-empty (i j)
  "Erases cell (i j) and replaces it with an empty one."
  (make-empty-cell i j)
  (print-spreadsheet-changed-items)
  (send interaction-window :clear-screen)
  (tv:turn-off-sheet-blinkers interaction-window))

(defun insert-row (inserted-i notused)
  "A empty row is inserted at the inserted-i location and the
  rest are moved down. The bottom-most row is removed. The
  formula list must be updated."
  (cond ((tv:menu-choose
    '(("CONFIRM YOU WISH TO INSERT A ROW" :no-select t)
      ("Yes" :value t :documentation
        "Confirm desire to insert row")
      ("No" :value nil :documentation
        "Abort row insertion"))))
    (progn
      (loop for j from 1 to number-of-columns do
        (setq formula-list (remove (list number-of-rows j)
          formula-list :test 'equal)))
      (loop for i from number-of-rows
        downto (+ 1 inserted-i) do
        (loop for j from 1 to number-of-columns do
          (aset (aref spread (- i 1) j) spread i j)
          (send (aref spread i j) :set-changed-value t)
          (cond ((equal 'formula
            (send (aref spread i j) :type))
            (nsubstitute (list i j) (list (- i 1) j)
              formula-list :test 'equal))))))
      (loop for j from 1 to number-of-columns do
        (aset (make-instance 'cell) spread inserted-i j))
      (print-spreadsheet-changed-items))))))

(defun delete-row (deleted-i notused)
  "The row at deleted-i is deleted and the rest are moved up.
  An empty row is inserted at the bottom. The formula list
  is updated."
  (cond ((tv:menu-choose
    '(("CONFIRM YOU WISH TO DELETE A ROW" :no-select t)
      ("Yes" :value t :documentation
        "Confirm desire to delete row")
      ("No" :value nil :documentation
        "Abort row deletion"))))
    (progn
      (loop for j from 1 to number-of-columns do
        (setq formula-list (remove (list deleted-i j)
          formula-list :test 'equal)))
      (loop for i from deleted-i
        upto (number-of-rows - 1) do
        (loop for j from 1 to number-of-columns do
          (aset (aref spread i j) spread (i + 1) j)
          (send (aref spread i j) :set-changed-value t)
          (cond ((equal 'formula
            (send (aref spread i j) :type))
            (nsubstitute (list i j) (list (i + 1) j)
              formula-list :test 'equal))))))
      (loop for j from 1 to number-of-columns do
        (aset (make-instance 'cell) spread number-of-rows j))
      (print-spreadsheet-changed-items))))))

```

```

(progn
  (loop for j from 1 to number-of-columns do
    (setq formula-list (remove (list deleted-i j)
                               formula-list :test 'equal)))
  (loop for i from deleted-i to (- number-of-rows 1) do
    (loop for j from 1 to number-of-columns do
      (aset (aref spread (+ i 1) j) spread i j)
      (send (aref spread i j) :set-changed-value t)
      (cond ((equal 'formula
                    (send (aref spread i j) :type))
              (nsubstitute (list i j) (list (+ 1 i) j)
                           formula-list :test 'equal))))))
  (loop for j from 1 to number-of-columns do
    (aset (make-instance 'cell) spread number-of-rows j))
  (print-spreadsheet-changed-items))))

(defun get-row-location-or-abort ()
  "Used during move-row and copy-row. Returns a true if the user
  clicks on a row. Allows Go-To function to be performed."
  (send spreadsheet :set-item-type-alist row-move-copy)
  (loop (setq user-input (send spreadsheet :any-ty1))
    (cond ((not (listp user-input))
            (return-from get-row-location-or-abort))
          ((equal :menu (car user-input))
            (cond ((equal 'go-to (caddr (cadr user-input)))
                    (funcall 'go-to))))
          ((equal :typeout-execute (car user-input))
            (cond ((equal 'mark (cadr user-input))
                    (return-from get-row-location-or-abort t))
                  ((equal 'abort (cadr user-input))
                    (return-from get-row-location-or-abort))))))

(defun move-row (moved-i notused)
  "Moves row moved-i to a new location specified by the user.
  Updates formula list."
  (let ((moved-to-i))
    (cond ((tv:menu-choose
            '(("CONFIRM YOU WISH TO MOVE A ROW" :no-select t)
              ("Yes" :value t :documentation
                    "Confirm desire to move row")
              ("No" :value nil :documentation
                    "Abort row move"))
            (progn
              (send interaction-window :select)
              (send interaction-window :clear-screen)
              (write-line "Click Left on New Row Location or Type
any Character to Abort Move" interaction-window)
              (cond ((get-row-location-or-abort)

```



```

                                (list copied-to-i j)
                                formula-list))))))
                                (print-spreadsheet-changed-items))))))
                                (t (send interaction-window :beep)))
                                (send spreadsheet :set-item-type-alist edit-list)
                                (send interaction-window :clear-screen)
                                (tv:turn-off-sheet-blinkers interaction-window))))))

(defun width (j notused)
  "The user can specify a column width (for display purposes) for
  the column he selected"
  (let ((input-value))
    (send interaction-window :select)
    (send interaction-window :clear-screen)
    (write-string "Column" interaction-window)
    (format interaction-window " ~A "
      (send (aref column j) :letter))
    (fresh-line interaction-window)
    (write-string "Enter column width in decimal for this column: "
      interaction-window)
    (tv:turn-on-sheet-blinkers interaction-window)
    (setq input-value (read-from-string
      (read-line interaction-window) nil nil))
    (cond ((integerp input-value)
      (cond ((plusp input-value)
        (send (aref column j) :set-width input-value))
        (t (send interaction-window :beep))))
      (t (send interaction-window :beep)))
    (tv:turn-off-sheet-blinkers interaction-window)
    ;the entire spreadsheet needs to be redisplayed
    (print-spreadsheet)
    (send interaction-window :clear-screen)))

(defun insert-column (inserted-j notused)
  "A blank column is inserted at column inserted-j. The rest are
  moved to the right. The right-most column is removed. Column
  widths are not changed."
  (cond ((tv:menu-choose
    '(("CONFIRM YOU WISH TO INSERT A COLUMN" :no-select t)
      ("Yes" :value t :documentation
        "Confirm desire to insert column")
      ("No" :value nil :documentation
        "Abort column insertion"))))
    (progn
      (loop for i from 1 to number-of-rows do
        (setq formula-list (delete (list i number-of-columns)
          formula-list :test 'equal)))

```

```

(loop for j from number-of-columns
      downto (+ 1 inserted-j) do
  (aset (aref spread i (- j 1)) spread i j)
  (send (aref spread i j) :set-changed-value t)
  (cond ((equal 'formula
                (send (aref spread i j) :type))
        (nsubstitute (list i j) (list i (- j 1))
                      formula-list :test 'equal))))
  (aset (make-instance 'cell) spread i inserted-j))
(print-spreadsheet-changed-items))))

(defun delete-column (deleted-j notused)
  "The column at deleted-j is removed and the rest are moved
  to the left. A blank column is moved in on the right.
  Column widths are not changed."
  (cond ((tv:menu-choose
        '(("CONFIRM YOU WISH TO DELETE A COLUMN" :no-select t)
          ("Yes" :value t :documentation
                "Confirm desire to delete column")
          ("No" :value nil :documentation
                "Abort column deletion"))))
    (progn
      (loop for i from 1 to number-of-rows do
        (setq formula-list (delete (list i deleted-j)
                                   formula-list :test 'equal))
        (loop for j from deleted-j to
              (- number-of-columns 1) do
          (aset (aref spread i (+ 1 j)) spread i j)
          (send (aref spread i j) :set-changed-value t)
          (cond ((equal 'formula
                        (send (aref spread i j) :type))
                (nsubstitute (list i j) (list i (+ 1 j))
                              formula-list :test 'equal))))
          (aset (make-instance 'cell)
                spread i number-of-columns))
        (print-spreadsheet-changed-items))))))

(defun get-column-location-or-abort ()
  "Used during move-column and copy-column. Returns a true if the
  user clicks on a column. Also allows the Go-To function to be
  performed to access the entire spreadsheet"
  (send spreadsheet :set-item-type-alist column-move-copy)
  (loop (setq user-input (send spreadsheet :any-ty))
    (cond ((not (listp user-input))
          (return-from get-column-location-or-abort))
          ((equal :menu (car user-input))
           (cond ((equal 'go-to (caddr (cadr user-input)))
                  (funcall 'go-to))))))

```

```

((equal :timeout-execute (car user-input))
 (cond ((equal 'mark (cadr user-input))
        (return-from get-column-location-or-abort t))
        ((equal 'abort (cadr user-input))
         (return-from get-column-location-or-abort))))))

(defun move-column (moved-j notused)
  "the user can move column moved-j to a new location."
  (let ((moved-to-j))
    (cond ((tv:menu-choose
            '(("CONFIRM YOU WISH TO MOVE A COLUMN" :no-select t)
              ("Yes" :value t :documentation
                    "Confirm desire to move column")
              ("No" :value nil :documentation
                    "Abort column move"))
            (progn
              (send interaction-window :select)
              (send interaction-window :clear-screen)
              (write-line "Click Left on New Column Location
                          or Type any Character to Abort Move"
                          interaction-window)
              (cond ((get-column-location-or-abort)
                     (cond
                      ((not (equal moved-j (setq moved-to-j
                                                    (car (caddr user-input)))))
                       (progn
                        (loop for i from 1 to number-of-rows do
                          (aset (aref spread i moved-j)
                                spread i moved-to-j)
                          (send (aref spread i moved-to-j)
                                :set-changed-value t)
                          (aset (make-instance 'cell) spread
                                i moved-j)
                          (cond ((equal 'formula (send (aref
                                                         spread i moved-to-j) :type))
                                (nsubstitute (list i moved-to-j)
                                              (list i moved-j)
                                              formula-list :test 'equal)))
                          (print-spreadsheet-changed-items))))
                        (t (send interaction-window :beep)))
                      (send spreadsheet :set-item-type-alist edit-list)
                      (send interaction-window :clear-screen)
                      (tv:turn-off-sheet-blinkers interaction-window))))))
            (defun copy-column (copied-j notused)
              "The user can copy column copied-j to a new location."
              (let ((copied-to-j))
                (cond ((tv:menu-choose

```

```

'(("CONFIRM YOU WISH TO COPY A COLUMN" :no-select t)
  ("Yes" :value t :documentation
    "Confirm desire to copy column")
  ("No" :value nil :documentation
    "Abort column copy"))
(progn
  (send interaction-window :select)
  (send interaction-window :clear-screen)
  (write-line
    "Click Left on Location for a Copy of this Column
    or Type any Character to Abort" interaction-window)
  (cond ((get-column-location-or-abort)
    (cond ((not (equal copied-j (setq copied-to-j
      (car (caddr user-input))))))
      (progn
        (loop for i from 1 to
          number-of-rows do
            (send (aref spread i copied-to-j)
              :copy-cell i copied-j)
            (cond ((equal 'formula (send (aref
              spread i copied-j) :type))
              (setq formula-list (cons
                (list i copied-to-j)
                formula-list))))
            (print-spreadsheet-changed-items))))
        (t (send interaction-window :beep)))
    (send spreadsheet :set-item-type-alist edit-list)
    (send interaction-window :clear-screen)
    (tv:turn-off-sheet-blinkers interaction-window))))))

(defun save-file ()
  "The current spreadsheet is save as a user-specified file
  in the local machine's Logic-calc directory"
  (let ((output-pathname)(out-stream)
    (filename-string)(output-value))
    (write-string
      "Enter the file name to store this worksheet in: "
      interaction-window)
    (cond ((equal "" (setq filename-string
      (remove #\sp (read-line interaction-window))))
      (return-from save-file)))
    (setq output-pathname (make-pathname :host "lm"
      :directory "logic-calc"
      :name filename-string
      :type "logic-calc"))
    (setq out-stream (open output-pathname :if-exists :new-version
      :direction :output))
    (format out-stream "-A-~" number-of-rows)

```

```

(format out-stream "~A~%" number-of-columns)
(format out-stream "~A~%" first-display-row)
(format out-stream "~A~%" first-display-column)
(loop for j from 1 to number-of-columns do
  (format out-stream "~A~%" (send (aref column j) :width)))
; save items unique to each cell in ASCII form to allow
; easy file interface
(loop for i from 1 to number-of-rows do
  (loop for j from 1 to number-of-columns do
    (case (send (aref spread i j) :type)
      (empty (format out-stream "E"))
      (constant (progn
        (format out-stream "C")
        (format out-stream "~A"
          (send (aref spread i j) :output-display))
        (format out-stream "~A~%"
          (send (aref spread i j) :bits))
        (format out-stream "~A~%"
          (send (aref spread i j) :value))))
      (text (progn (format out-stream "T")
        (format out-stream "~A~%"
          (send (aref spread i j) :value))))
      (formula (progn
        (format out-stream "F")
        (format out-stream "~A"
          (send (aref spread i j) :output-display))
        (format out-stream "~A"
          (send (aref spread i j) :bits))
        (print (send (aref spread i j) :formula)
          out-stream)
        ; identify string values
        (cond ((stringp (setq output-value
          (send (aref spread i j) :value)))
          (format out-stream "~%S"))
          (t (format out-stream "~%N"))))
        (format out-stream "~A~%" output-value))))))
    (close out-stream)
  (write-string "File written - Press Return to Continue"
    interaction-window)
  (read-line interaction-window)))

(defun read-file ()
  "The user can replace the current spreadsheet with one saved on
  disk. All files are saved in the local machine's logic-calc
  directory"
  (let ((input-pathname)(in-stream)(filename-string))
    (write-string "Enter the file name to read: "
      interaction-window)

```

```

(cond ((equal "" (setq filename-string
      (remove #\sp (read-line interaction-window ))))
      (return-from read-file)))
(setq input-pathname (make-pathname :host "lm"
      :directory "logic-calc"
      :name filename-string
      :type "logic-calc"))
(send interaction-window :clear-screen)
(cond ((setq in-stream (open input-pathname :direction :input
      :if-does-not-exist nil))
      (progn (write-string "STANDBY - Reading file: "
        interaction-window)
        (write-line filename-string interaction-window)))
      (t (progn (send interaction-window :beep)
        (write-string "ERROR: File Not Found: "
          interaction-window)
        (write-line filename-string interaction-window)
        (write-string "Press Return to Continue"
          interaction-window)
        (read-line interaction-window)
        (return-from read-file))))))
(setq formula-list nil)
(setq number-of-rows (read-from-string (read-line in-stream)))
(setq number-of-columns
  (read-from-string (read-line in-stream)))
(setq first-display-row
  (read-from-string (read-line in-stream)))
(setq first-display-column
  (read-from-string (read-line in-stream)))
(loop for j from 1 to number-of-columns do
  (send (aref column j) :set-width
    (read-from-string (read-line in-stream))))
(loop for j from (+ 1 number-of-columns) to
  max-number-of-columns do
  (send (aref column j) :set-width 20))
(setq spread (make-array (list (+ 1 number-of-rows)
  (+ 1 number-of-columns))))
;reconstruct each cell
(loop for i from 1 to number-of-rows do
  (loop for j from 1 to number-of-columns do
    (case (read-char in-stream)
      (#\E (aset (make-instance 'cell) spread i j))
      (#\C (aset (make-instance 'cell :type 'constant
        :output-display (case (read-char in-stream)
          (#\X 'X)
          (#\B 'B))

```

```

        :bits (read-from-string (read-line in-stream))
        :value (read-from-string (read-line in-stream)))
    spread i j))
  (#\T (aset (make-instance 'cell :type 'text
    :value (read-line in-stream)) spread i j))
  (#\F (progn (aset (make-instance 'cell :type 'formula
    :output-display (case (read-char in-stream)
      (#\X 'X)
      (#\B 'B))
    :bits (read-from-string
      (read-line in-stream))
    :formula (read-from-string
      (read-line in-stream))) spread i j)
    (cond ((equal #\S (read-char in-stream))
      (send (aref spread i j)
        :set-value (read-line in-stream)))
      (t (send (aref spread i j) :set-value
        (read-from-string
          (read-line in-stream))))))
    (setq formula-list
      (cons (list i j) formula-list))))))
  (close in-stream)
  (print-spreadsheet)
  (write-string "File retrieved - Press Return to Continue"
    interaction-window)
  (read-line interaction-window)))

(defun file ()
  "The user can specify read or write operations."
  (send interaction-window :select)
  (send interaction-window :clear-screen)
  (tv:turn-on-sheet-blinkers interaction-window)
  (case (tv:menu-choose
    '(("FILE OPERATIONS" :no-select t)
      ("Save File" :value save :documentation
        "Save this spreadsheet")
      ("Read File" :value read :documentation
        "Read previously saved file"))))
    (save (save-file))
    (read (read-file)))
  (send interaction-window :clear-screen)
  (tv:turn-off-sheet-blinkers interaction-window))

(defun size ()
  "The size of a spreadsheet can be changed with this function."
  (let ((new-rows)(new-columns)(old-spread))
    (send interaction-window :select)
    (send interaction-window :clear-screen)

```

```

(tv:turn-on-sheet-blinkers interaction-window)
(format interaction-window "Current number of rows is : ~d"
  number-of-rows)
(fresh-line interaction-window)
(format interaction-window "Current number of columns is : ~d"
  number-of-columns)
(fresh-line interaction-window)
(write-string "Enter the new number of rows: "
  interaction-window)
(setq new-rows (read-from-string
  (read-line interaction-window) nil nil))
(write-string "Enter the new number of columns: "
  interaction-window)
(setq new-columns (read-from-string
  (read-line interaction-window) nil nil))
(cond ((and (integerp new-rows)(plusp new-rows)
  (plusp new-columns)(<= new-rows max-number-of-rows)
  (integerp new-columns)
  (<= new-columns max-number-of-columns)))
;construct the new spreadsheet
(progn
  (setq old-spread spread)
  (setq spread (make-array (list (+ 1 new-rows)
    (+ 1 new-columns)))))
(loop for i from 1 to new-rows do
  (loop for j from 1 to new-columns do
    (cond
      ((and (<= i number-of-rows)
        (<= j number-of-columns))
        (aset (aref old-spread i j) spread i j))
      (t (aset (make-instance 'cell) spread i j)))))
;remove non-existent cells from formula list
(loop for i from (+ 1 new-rows) to number-of-rows do
  (loop for j from 1 to number-of-columns do
    (setq formula-list (delete (list i j)
      formula-list :test 'equal))))
(loop for j from (+ 1 new-columns) to
  number-of-columns do
  (loop for i from 1 to number-of-rows do
    (setq formula-list (delete (list i j)
      formula-list :test 'equal))))
;reset global variables
(setq number-of-rows new-rows)
(setq number-of-columns new-columns)
(cond ((or (> first-display-row number-of-rows)
  (> first-display-column number-of-columns))
  (progn (setq first-display-row 1)
    (setq first-display-column 1))))

```

```

        (print-spreadsheet)))
      (t (send interaction-window :beep)))
    (send interaction-window :select)
    (send interaction-window :clear-screen)
    (tv:turn-off-sheet-blinkers interaction-window)))

(defun go-to ()
  "A new region can be viewed with this function."
  (let ((input-value)(column-number)(row-number)
        (original-x-position)(original-y-position))
    (send interaction-window :select)
    (fresh-line interaction-window)
    (multiple-value-setq (original-x-position original-y-position)
      (send interaction-window :read-cursorpos))
    (write-string
      "Enter cell name to be placed in upper left corner
      of screen: " interaction-window)
    (tv:turn-on-sheet-blinkers interaction-window)
    (setq input-value (string-upcase (string-left-trim '("#\sp)
      (read-line interaction-window))))
    (multiple-value-setq (row-number column-number)
      (row&col (read-from-string input-value)))
    (cond ((and (integerp row-number) (plusp row-number)
      (<= row-number number-of-rows)(plusp column-number)
      (<= column-number number-of-columns))
      (progn (setq first-display-row row-number)
        (setq first-display-column column-number)
        (print-spreadsheet)))
      (t (send interaction-window :beep)))
    (send interaction-window :select)
    (send interaction-window
      :set-cursorpos original-x-position original-y-position)
    (send interaction-window :clear-eof)
    (tv:turn-off-sheet-blinkers interaction-window)))

(defun reset-recalc-recursive (x)
  "All formula cells' recalculation flag must be reset before a
  spreadsheet calculation."
  (cond (x (progn (send (aref spread (car (car x))(cadr (car x)))
    :set-recalc nil)
    (reset-recalc-recursive (cdr x))))))

(defun calculate-recursive (x)
  "Recalculates all formula cells in the formula list."
  (cond (x (progn (evaluate-cell (car (car x))(cadr (car x)))
    (calculate-recursive (cdr x))))))

(defun calc (&key no-redisplay)

```

"All formula cells within the spreadsheet are recalculated. If the keyword no-redisplay is set to true then the recalculation will not display updated results. This is useful for testing over several recalculations. The program operates faster."

```
(reset-recalc-recursive formula-list)
(calculate-recursive formula-list)
(cond ((not no-redisplay) (print-spreadsheet-changed-items))))
```

```
(defun restart ())
```

"allows user to restart edit mode from outside Logic Calc."

;view upper-left portion of spreadsheet

```
(setq first-display-row 1)
```

```
(setq first-display-column 1)
```

;connect i-o buffers

```
(send spreadsheet ':set-io-buffer program-io-buffer)
```

```
(send interaction-window ':set-io-buffer program-io-buffer)
```

```
(send main-menu ':set-io-buffer program-io-buffer)
```

;set the font for the spreadsheet

```
(send spreadsheet :set-font-map (fillarray (make-array 26.)
      (list fonts:tvfont)))
```

```
(send spreadsheet :set-item-type-alist edit-list)
```

```
(send program-constraint-window :expose)
```

```
(send spreadsheet :expose)
```

```
(send interaction-window :expose)
```

```
(send main-menu :expose)
```

```
(print-spreadsheet)
```

;loop until exit is seen.

;Call functions as appropriate response to user-inputs

```
(block main-program-loop
```

```
  (loop (setq user-input (send spreadsheet :list-tyi))
```

```
    (cond ((equal :menu (car user-input))
```

```
      (cond ((equal 'exit (caddr (cadr user-input)))
```

```
        (return-from main-program-loop))
```

```
        (t (funcall (caddr (cadr user-input))))))
```

```
    ((equal :typeout-execute (car user-input))
```

```
      (funcall (cadr user-input) (car (caddr user-input))
```

```
      (cadr (caddr user-input)))))
```

```
(send spreadsheet :kill)
```

```
(send program-constraint-window :kill))
```

; The next four functions are designed to be incorporated as

; entries into a cell's formula. They provide means of accessing

; other cell's contents.

```
(defun cell (&quote x)
```

"Returns the contents of the cell specified by parameter x. The referenced cell must first be evaluated by evaluate-cell.

Allows indefinite indirection by permitting an entry such as:

```
(Cell (Cell (Cell C12))).
```

All cells in the indirect chain except the last must have a string value corresponding to a cell location"

```
(let ((row)(col))
  (cond ((and (listp x) (equal 'cell (car x)))
    (setq x (read-from-string (eval x)))))
  (cond ((multiple-value-setq (row col) (row&col x))
    (progn (evaluate-cell row col)
      (send (aref spread row col) :value)))
    (t "ERROR"))))
```

(defun cell-offset ("e x j i)
 "Returns the contents of the cell specified by parameter x
 offset by i rows and j columns. The referenced cell must first
 be evaluated by evaluate-cell. No indirection allowed."

```
(let ((row)(col))
  (setq i (eval i))
  (setq j (eval j))
  (cond ((and (integerp i)(integerp j))
    (multiple-value-setq (row col) (row&col x)))
    (progn
      (setq row (+ row i))
      (setq col (+ col j))
      (cond ((and (<= row number-of-rows )(plusp row)
        (<= col number-of-columns)(plusp col))
        (progn (evaluate-cell row col)
          (send (aref spread row col) :value)))
        (t "ERROR"))))
    (t "ERROR"))))
```

(defun cell-indirect ("e x)
 "Returns the contents of the cell specified at cell-location x.
 Provides a single level of indirection."

```
(let ((pointer)(pointer-row)(pointer-col)(row)(col))
  (multiple-value-setq (pointer-row pointer-col)(row&col x))
  (cond ((and pointer-row pointer-col)
    (progn
      (evaluate-cell pointer-row pointer-col)
      (setq pointer
        (send (aref spread pointer-row pointer-col) :value))
      (cond ((stringp pointer)
        (progn (setq pointer (read-from-string
          (remove #\space pointer)))
          (multiple-value-setq (row col)
            (row&col pointer))
          (cond ((and row col)
```

```

                                (progn
                                  (evaluate-cell row col)
                                  (send (aref spread row col)
                                        :value)))
                                (t "ERROR"))))
                                (t "ERROR"))))
                                (t "ERROR"))))
                                (t "ERROR"))))

(defun dual-rank-register ("internal in out inval)
  "the user must supply a UNIQUE name for the internal parameter.
  A cell with this function entered acts as a dual-ranked
  register. It's 'inval' is gated in whenever 'in' is true.
  The output is maintained, however, until 'out' is true."
  (let ((in (eval in))(out (eval out)))
    (cond ((not (boundp internal))(set internal (list 0 0)))
          (cond (in (cond (out (progn
                                (rplaca (eval internal) (eval inval))
                                (rplacd (eval internal)
                                      (list (car (eval internal)))
                                      (cadr (eval internal))))
                            (t (cadr (rplaca (eval internal)
                                              (eval inval))))))
                  (out (cadr (rplacd (eval internal)
                                      (list (car (eval internal))))))
                  (t (cadr (eval internal)))))))

;The next function is designed to be used in a driver routine
; to put a value in a cell without changing its type.
(defun load-cell ("x value)
  "This procedure will load a value into the cell named by
  paramter x, setting its changed value flag, and without
  changing its formula."
  (let ((i)(j))
    (cond ((multiple-value-setq (i j) (row&col x))
      (progn
        (send (aref spread i j) :set-value (eval value))
        (send (aref spread i j) :set-changed-value t))))))

;the main program begins

;create cell objects for each indice in the array "spread"
(loop for i from 1 to number-of-rows do
  (loop for j from 1 to number-of-columns do
    (aset (make-instance 'cell ) spread i j)))

;create column objects for all
; possible columns in the array "column"

```

```
(loop for j from 1 to max-number-of-columns do  
  (aset (make-instance 'column-flavor  
    :letter (column-string j)) column j))
```

```
;call restart to put in edit mode  
(restart)
```

APPENDIX B: LOGIC CALC DRIVING PROGRAM FOR MICROPROCESSOR

;The following functions provide cell names.

```
(Defun PC      () (Cell A9))    ;Program Counter
(Defun IR      () (Cell A12))   ;Instruction Register
(Defun RegA    () (Cell A15))   ;A Register
(Defun RegB    () (Cell A18))   ;B Register
(Defun RegX    () (Cell A21))   ;X Register
(Defun RegY    () (Cell A24))   ;Y Register
(Defun SP      () (Cell A27))   ;Stack Pointer
(Defun RegS    () (Cell A30))   ;Status Register
(Defun IDB     () (Cell B2))    ;Internal Data Bus
(Defun MDR     () (Cell B6))    ;Memory Data Register
(Defun MAR     () (Cell B9))    ;Memory Address Register
(Defun DBI     () (Cell B12))   ;Data Bus Interface Register
(Defun EDB     () (Cell B15))   ;External Data Bus
(Defun EAB     () (Cell B18))   ;Internal Data Bus
(Defun Adder   () (Cell C12))   ;adder/subtractor
```

;The following functions test the state of the clock, returning
; a value of true if the clock is in the corresponding state.

```
(Defun Rising () (Zerop (Cell A2)))
(Defun High   () (Equal (Cell A2) 1))
(Defun Falling () (Equal (Cell A2) 2))
(Defun Low    () (Equal (Cell A2) 3))
```

;The following functions test signals, returning a value of true
; if the corresponding signal is active.

```
(Defun Reset () (Zerop (Cell A6)))
(Defun MemR  () (Zerop (Cell B21)))
(Defun MemW  () (Zerop (Cell B24)))
```

;The following functions test up to 5 bits in the Micro-
; instruction Register, returning a true value if all bits are
; set to 1 for Microbitp or all bits set to zero for Microbitn.

```
(Defun Microbitp (A &Optional B C D E)
  (And (Cond (E (Logbitp E (Cell D2)))(T))
        (Cond (D (Logbitp D (Cell D2)))(T))
        (Cond (C (Logbitp C (Cell D2)))(T))
        (Cond (B (Logbitp B (Cell D2)))(T))
        (Logbitp A (Cell D2))))
```

```

(Defun Microbitn (A &Optional B C D E)
  (Not(Or (Cond (E (Logbitp E (Cell D2))))
    (Cond (D (Logbitp D (Cell D2))))
    (Cond (C (Logbitp C (Cell D2))))
    (Cond (B (Logbitp B (Cell D2))))
    (Logbitp A (Cell D2)))))

(Defun RisingMicrobitp (A &Optional B C D E)
  (And (Cond (E (Logbitp E (Cell D2)))(T))
    (Cond (D (Logbitp D (Cell D2)))(T))
    (Cond (C (Logbitp C (Cell D2)))(T))
    (Cond (B (Logbitp B (Cell D2)))(T))
    (Logbitp A (Cell D2))
    (Zerop (Cell A2))))

```

;The following function allows the user to select between
; three clocking modes: Full Speed, Fixed Number of Cycles,
; or Single Cycles. For each cycle, the spreadsheet is
; recalculated four times. If Full Speed operations is
; selected, interim results are not displayed.
; The variable Cycle-Global is used to keep track of the
; mode and the number of cycles if in the Fixed Number of
; Cycles Mode.

```

(Setup Cycle-Global 0)
(Defun Cycle ()
  (Cond ((Plusp Cycle-Global)
    (Progn (Setup Cycle-Global (- Cycle-Global 1))
      (Calc :No-Redisplay Nil)
      (Calc :No-Redisplay Nil)
      (Calc :No-Redisplay Nil)
      (Calc)))
    ((Minusp Cycle-Global)
      (Progn (Calc :No-Redisplay T)
        (Calc :No-Redisplay T)
        (Calc :No-Redisplay T)
        (Calc)))
    (T (Progn
      (Setup Cycle-Global)
      (Case (Tv:Menu-Choose
        '(("Cycle Menu" :No-Select T)
          ("Full Speed" :Value -1)
          ("One Cycle" :Value Nil)
          ("Set Number Of Cycles" :Value 1))))

```

```

(-1 -1)
(Nil 0)
(1 (Progn
    (Tv:Turn-On-Sheet-Blinkers Interaction-Window)
    (Write-String "Enter Number Of Cycles: "
        Interaction-Window)
    (- (Read-From-String
        (Read-Line Interaction-Window)) 1))))
(Tv:Turn-Off-Sheet-Blinkers Interaction-Window)
(Send Interaction-Window :Clear-Screen)
(Calc)(Calc)(Calc)(Calc))))

```

;The automatic simulation begins here

```

(Print-Spreadsheet) ; display the spreadsheet
(Load-Cell A6 0)     ; activate a reset signal
(Load-Cell A2 0)     ; initialize the clock to a rising edge
(Load-Cell A36 0)    ; initialize cycle counter cell
(Calc)
(Load-Cell A6 1)     ; clear reset signal

```

;begin fetch-execute cycle

```

(Block Main-Loop
  (Loop Do
    (Load-Cell D6 0) ;microcode address for fetch
    (Cycle)
    (Case (IR)       ;microcode addresses for execute
      (00 (Return-From Main-Loop)) ;exit for HALT
      (01 (Progn (Load-Cell D6 01)(Cycle)))
      (02 (Progn (Load-Cell D6 02)(Cycle)))
      (03 (Progn (Load-Cell D6 03)(Cycle)))
      (04 (Progn (Load-Cell D6 04)(Cycle)))
      (05 (Progn (Load-Cell D6 05)(Cycle)))
      (06 (Progn (Load-Cell D6 06)(Cycle)
        (Load-Cell D6 07)(Cycle)))
      (07 (Progn (Load-Cell D6 06)(Cycle)
        (Load-Cell D6 08)(Cycle)))
      (08 (Progn (Load-Cell D6 06)(Cycle)
        (Load-Cell D6 09)(Cycle)))
      (09 (Progn (Load-Cell D6 06)(Cycle)
        (Load-Cell D6 10)(Cycle)))
      (10 (Progn (Load-Cell D6 06)(Cycle)
        (Load-Cell D6 11)(Cycle)))
      (11 (Progn (Load-Cell D6 12)(Cycle)))
      (12 (Progn (Load-Cell D6 13)(Cycle)))
    )
  )
)

```

(13 (Progn (Load-Cell D6 14)(Cycle)))
(14 (Progn (Load-Cell D6 15)(Cycle)))
(15 (Progn (Load-Cell D6 06)(Cycle)
 (Load-Cell D6 16)(Cycle)))
(16 (Progn (Load-Cell D6 06)(Cycle)
 (Load-Cell D6 17)(Cycle)))
(17 (Progn (Load-Cell D6 06)(Cycle)
 (Load-Cell D6 18)(Cycle)))
(18 (Progn (Load-Cell D6 06)(Cycle)
 (Load-Cell D6 19)(Cycle)))
(19 (Progn (Load-Cell D6 06)(Cycle)
 (Load-Cell D6 20)(Cycle)))
(20 (Progn (Load-Cell D6 21)(Cycle)))
(21 (Progn (Load-Cell D6 22)(Cycle)))
(22 (Progn (Load-Cell D6 23)(Cycle)))
(23 (Progn (Load-Cell D6 24)(Cycle)))
(24 (Progn (Load-Cell D6 25)(Cycle)
 (Load-Cell D6 26)(Cycle)))
(25 (Progn (Load-Cell D6 25)(Cycle)
 (Load-Cell D6 27)(Cycle)))
(26 (Progn (Load-Cell D6 25)(Cycle)
 (Load-Cell D6 28)(Cycle)))
(27 (Progn (Load-Cell D6 25)(Cycle)
 (Load-Cell D6 29)(Cycle)))
(28 (Progn (Load-Cell D6 30)(Cycle)))
(29 (Progn (Load-Cell D6 31)(Cycle)))
(30 (Progn (Load-Cell D6 32)(Cycle)))
(31 (Progn (Load-Cell D6 33)(Cycle)))
(32 (Progn (Load-Cell D6 34)(Cycle)
 (Load-Cell D6 35)(Cycle)
 (Load-Cell D6 36)(Cycle)))
(33 (Progn (Load-Cell D6 37)(Cycle)
 (Load-Cell D6 35)(Cycle)
 (Load-Cell D6 36)(Cycle)))
(34 (Progn (Load-Cell D6 34)(Cycle)
 (Load-Cell D6 36)(Cycle)))
(35 (Progn (Load-Cell D6 37)(Cycle)
 (Load-Cell D6 36)(Cycle)))
(36 (Progn (Load-Cell D6 38)(Cycle)
 (Load-Cell D6 39)(Cycle)))
(37 (Progn (Load-Cell D6 40)(Cycle)
 (Load-Cell D6 39)(Cycle)))
(38 (Progn (Load-Cell D6 41)(Cycle)))
(39 (Progn (Load-Cell D6 42)(Cycle)))
(40 (Progn (Load-Cell D6 43)(Cycle)))
(41 (Progn (Load-Cell D6 44)(Cycle)))
(42 (Progn (Load-Cell D6 25)(Cycle)
 (Load-Cell D6 45)(Cycle)))

```

        (Load-Cell D6 46)(Cycle)
        (Load-Cell D6 47)(Cycle)))
(43 (Progn (Load-Cell D6 48)(Cycle)
        (Load-Cell D6 47)(Cycle)))
(44 (Progn (Load-Cell D6 06)(Cycle)
        (Load-Cell D6 47)(Cycle)))
(45 (Progn (Load-Cell D6 06)(Cycle)
        (Cond ((Logbitp 0 (Regs))
                (Progn (Load-Cell D6 47)
                        (Cycle))))))
(46 (Progn (Load-Cell D6 06)(Cycle)
        (Cond ((Logbitp 1 (Regs))
                (Progn (Load-Cell D6 47)
                        (Cycle))))))
(47 (Progn (Load-Cell D6 06)(Cycle)
        (Cond ((Logbitp 2 (Regs))
                (Progn (Load-Cell D6 47)
                        (Cycle)))))))

```

(Restart) ; return to edit mode displaying final results

BIBLIOGRAPHY

- [1] American National Standards Institute. "IEEE Standard for Binary Floating-Point Arithmetic," Std. 754, New York, 1985.
- [2] A. Barna and D. I. Porat, Introduction to Digital Techniques, 2nd ed., New York: John Wiley & Sons, 1987.
- [3] P. M. Chirlian, Analysis and Design of Integrated Circuits, New York: Harper & Row Publishers, 1987.
- [4] H. G. Cragon, "Simulation of Processor Arrays Using Spreadsheet Programming," Unpublished Manuscript, University of Texas at Austin, 1985.
- [5] Explorer Lisp Reference, Austin: Texas Instruments Inc., 1985.
- [6] Explorer Window System Reference, Austin: Texas Instruments Inc., 1985.
- [7] Explorer ZMACS Editor Reference, Austin: Texas Instruments Inc., 1985.
- [8] J. L. Haynes, "Circuit Design with Lotus 1-2-3," in Byte, Vol. 10, pp. 143-156, Fall 1985.
- [9] G. J. Lipovski, Microcomputer Interfacing: Principles and Practice, Lexington, MA: Lexington Books, 1986.
- [10] Lotus Reference Manual Release 2, Cambridge: Lotus Development Corporation, 1985.
- [11] G. M. Robinson, "Technique Exploits Spreadsheet Programs for Solving Complex Engineering Problems," in Design News, Vol. 42, pp. 121-123, October 20, 1986.
- [12] G. L. Steele Jr., Common Lisp: The Language, Burlington MA: Digital Press, 1984.
- [13] I. Unwala, "A Novel Environment for Design and Simulation of Digital Systems Architecture," Unpublished Master's Thesis, University of Texas at Austin, 1986.

- [14] T. J. Wagner and G. J. Lipovski, Fundamentals of Microcomputer Programming, New York: MacMillan Publishing, 1984.
- [15] J. Walden, File Formats, New York: John Wiley & Sons, 1986.
- [16] P. H. Winston and B. K. P. Horn, Lisp, 2nd ed., Reading, MA: Addison-Wesley Publishing, 1984.

VITA

Glenn David Rosenberger was born in Johnstown, Pennsylvania, on January 4, 1959, the son of Walter Francis Rosenberger and Frances Elvira Rosenberger. He completed Ferndale Area High School in 1976. He graduated from the United States Air Force Academy in May, 1980, and was awarded a Bachelor of Science in Electrical Engineering. He completed United States Air Force Undergraduate Pilot Training at Williams Air Force Base, Arizona, in August, 1981. He completed RF-4C training at Shaw Air Force Base in May 1982. During the following years, he served as an aircraft commander, an instructor pilot, and a ground school instructor for the RF-4C Phantom reconnaissance fighter at Bergstrom Air Force Base, Texas. In September, 1985, he entered the Graduate School of the University of Texas. His research interests include Computer Architecture, Computer Aided Design, and Computer Arithmetic. He is a member of Etta Kappa Nu and Tau Beta Pi.

Permanent address: 112 Habicht Street
Johnstown, Pennsylvania 15906

This thesis was typed by Glenn David Rosenberger.

END

11-87

DTIC