AN ABSTRACT OF THE THESIS OF

JOHN EDWARD SMATHERS for the DOCTOR OF PHILOSOPHY
(Name) (Degree)

in Electrical and Electronics Engineering presented on Oct. 16, 1968
(Major) (Date)

Title: DISTRIBUTED LOGIC MEMORY COMPUTER FOR PROCESS CONTROL

Abstract approved: Redacted for Privacy
John F. Engle

An instruction set and programming examples are described for a Distributed Logic Memory computer organization. The computer is designed to take advantage of the economies of very large-scale circuit integration. In addition, the computer can grow in an orderly way. As it grows there is increased parallelism possible so that the amount of spare real time in a control application is not greatly reduced. Finally, such an organization should permit stored program control in relatively small applications where up to now control by a conventionally organized computer has been prohibitively expensive.

The computer consists of a linear array of identical, small, sequential machines, or cells. The structure is similar to that of the Distributed Logic Memory originally proposed by C. Y. Lee. It was demonstrated by J. N. Sturman that the addition of sequential logic to each cell permits the memory to become a self-contained computing system.

It is the purpose of this thesis to produce an application-oriented process control computer design based on the concepts of Lee and Sturman. It was found necessary to increase the length of the memory word in each cell. The ability to store instructions and data in cells is retained. Increasing the memory word length of each cell permits an expanded instruction repertoire. The low-ordered three bits of every memory word are arranged to identify a cell as one of eight possible types. A program instruction includes modifier bits which specify the types of cells on which the instruction is to operate. This facility enhances the efficiency of programs.

The logic design of the cell is complete enough to permit estimating gate count per cell. An analysis of the sensitivity of gate count to changes in the instruction set is included. A program simulation of the Distributed Logic Memory computer assisted in its development and later permitted verification of programs written for the computer. The existence of a compiler permitted such programs to be written in a convenient, symbolic form.

A data multiplexer is developed as a practical application for the Distributed Logic Memory computer structure. The necessary data multiplexer program, which consists of about 100 instructions, is shown.

Distributed Logic Memory Computer for Process Control

by

John Edward Smathers

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

June 1969

APPROVED:

Redacted for Privacy

Associate Professor of Electrical and Electronics Engineering

in charge of major

Redacted for Privacy

Head of Department of Electrical and Electronics Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented Oct. 16 1968

Typed by Clover Redfern for John Edward Smathers

ACKNOWLEDGMENT

The author is indebted to Professor John F. Engle for his guidance and encouragement during the course of this work. In addition, the author would like to express his sincere appreciation to Mr. Terrell N. Lowry of Bell Telephone Laboratories, who discussed the project with me on several occasions. The research was carried out under the sponsorship of Bell Telephone Laboratories.

TABLE OF CONTENTS

LIST OF FIGURES

DISTRIBUTED LOGIC MEMORY COMPUTER FOR PROCESS CONTROL

I. INTRODUCTION

The natural evolution in semiconductor technology to large-scale circuit integration is currently taking place. A drastic reduction in cost, size and power consumption of logic circuit elements is predicted. The ability to batch fabricate 100 to 1000 gates on a silicon chip is having its impact on computer architecture.

Attempts are being made to functionally partition conventionally organized computer systems into realizable arrays of gates (Levy, _et al._, 1967; Boysel, 1968). One physical restriction is the number of terminals or pins possible on an array package; perhaps a figure of 100 terminals is a practical upper limit. The resulting system may require 10 to 20 separate integrated circuit designs.

This study is motivated by the new economics wrought by batch fabrication technology. A conventional computer stores instructions and data in a binary store, performs arithmetic and logical operations in an arithmetic unit, stores data temporarily in hardware registers and performs input-output operations in peripheral units. The separate units are optimized to their specific functions; thus, the computer is an ensemble of distinct functional blocks. The economy of batch fabrication permits combining the memory and logic functions into one modular unit. Only one integrated circuit design is used repetitively to implement the Distributed Logic Memory (DLM) computer. The result is a one-dimensional array computer. There is economy in standardizing on one integrated circuit design.

Distributed Logic Memory was originally proposed as a memory by C. Y. Lee (1962) especially for applications in information retrieval, and later improved by Lee and others (Lee and Paull, 1963; Gaines and Lee, 1965; and Crane and Githens, 1965). It consists of a content-addressable memory driven by a conventionally organized, stored program computer. It has the usual associative memory property of being able to search, store or read over any or all fields of all memory locations simultaneously. However, in addition, it has the ability to cause communication between adjacent memory locations. Much use is made of such communication in the computer organization to be described. The memory consists of building blocks called "cells", interconnected in a regular pattern. Cells are identical and consist of a set of flip-flops for storing the memory word and additional gating logic. The primary communication between cells is carried on a common group of leads interconnecting all cells.

It is the addition of logic to each cell which permits the Distributed Logic Memory to become a self-contained computing system. Indeed it was first demonstrated by J. N. Sturman (1968b) that the placing of sequential logic in each cell eliminates the need for a driving computer; that is, an entire computing system can be constructed out of a linear array of interconnected cells. No external memories, control units or arithmetic units are needed. Instructions as well as data are stored in the cells. Especially in small applications, the system cost should be less when a driving computer is not required. When a program instruction is initiated by one cell, the instruction is carried simultaneously to all cells via the common bus that interconnects the cells. The inherent parallelism possible becomes clear. A number of selected cells can be operated on simultaneously by each instruction, rather than one cell at a time.

The cell logic in Sturman's machine uses three bits for the memory word, four bits for "activity" which is related to the content addressing structure and four bits for sequential control. Each cell therefore contains 11 flip-flops, which reflects some measure of its complexity. The three bits of the memory word permit coding eight distinct instructions.

The current study concerns a DLM computer designed specifically for control applications, especially small applications consisting of perhaps 100 to 1000 cells. The instruction set to be described is tailored accordingly. It consists primarily of bit manipulating instructions. Another part of the study concerns expanding the cell's memory word to perhaps 15 or 25 bits, mainly for data storage efficiency. However, at the same time this permits expanding the instruction set. A balance is sought between the complexity of each cell and the total number of cells required for a particular application since the system cost is related to the product of these two factors. The low-ordered three bits of the cell's memory word are set aside to act as a key to the cell's use. There are therefore eight distinct types of cell uses. Modifier bits are part of every program instruction and specify which types of cells are to respond to the instruction, according to their stored use key. This facility permits efficient programming.

The content addressable structure permits the DLM computer to start small and grow in an orderly way. In addition, there is inherent parallelism so that the effective computing rate can actually increase with growth. Content addressing appears to be naturally suited to a modular structure. By adding cells, the DLM computer grows in terms of control as well as memory. The ability to start small and yet grow in performance by small steps is a desirable feature for control computers.

The economics of batch fabrication dictate that consideration be given toward reducing the number of external connections required to be brought off an integrated circuit chip. Besides the obvious physical restrictions, the cost of connectorization and test may eventually outweigh the cost of silicon chip processing. Therefore, in this study a primary effort has been made to reduce the number of terminals required per cell. The use of a single-rail interconnecting bus (one lead per bit) rather than a double-rail bus is a departure from previous DLM work. This leads to the use of a two-step timed sequence of signals to represent a 0 signal, a 1 signal or a "don't care" signal. Furthermore, the common bus leads are time-shared in order to carry both control information and data to the various cells. However, input-output connections to the outside world are available on each cell rather than time-sharing the common bus for this function also.

The DLM computer is structurally fault tolerant. The loss of one cell does not necessarily prevent the remainder of the machine from continuing to operate. Repair may consist of strapping the faulty cell out of service and adding a new cell to an end of the array. Content addressing removes certain of the physical ordering requirements.

An instruction set is proposed and programming examples are given for the DLM computer. The logic of the cell is developed sufficiently to be able to estimate gate count. A practical application for the DLM structure in the data transmission field is presented. A data multiplexer is designed which time multiplexes the signals from 80 teletypewriter lines onto a single high-speed data channel. The necessary data multiplexer program, which consists of about 100 DLM instructions, is included.

II. DLM COMPUTER STRUCTURE

Consider now the DLM computer structure. Figure 1 shows the interconnection pattern for the linear array of cells. The cells are truly identical and are interconnected in a regular way. The machine structure is therefore well suited to large-scale integration. One cell per chip should be achievable now; multiple cells per chip may be worth considering because of the large number of shared terminals in the common bus. The need for relatively few interconnecting leads (perhaps less than 50) is consistent with integrated circuit requirements. The DLM computer could be both compact and relatively inexpensive if realized with integrated circuit cells.

Some preliminary estimates of the number of logic circuits per cell and the number of interconnecting leads can be given. Thirty-two logical states, coded onto five state flip-flops, have been found ample for sequential control within each cell. One Match flip-flop and one Control flip-flop are provided for "activity" in the content addressing structure. The size of the memory word may depend somewhat on the application. It is analogous to the word length of a conventionally organized machine; certainly many sizes have been proposed. Many times, however, the programmer finds himself short of bits per memory word but is able to program around the restriction at some additional cost in program. Programming experience with the proposed DLM computer suggests that perhaps 15 to 25 bits are sufficient. At the present time, a 16-bit word is provided for in the simulator. The resulting number of flip-flops together with the necessary gating in each cell will amount to about 200 to 300 gates.

The common bus presently consists of eighteen leads and there are three intercell leads. The input-output leads will number two or more. Actually, input-output leads would only be attached to cells associated with input-output; however, it is proposed that such terminals be provided on each cell in order that all cells may be truly identical.

The number of leads on the common bus is related to the length of the cell's memory word. Therefore, the memory word cannot be greatly extended without causing the cell to be terminal limited. That is to say that probably a 100-bit word is not feasible whereas a 15-bit or 25-bit word is feasible.

A synchronous or "clocked" computer is assumed. However, J. N. Sturman (1968a) has demonstrated an asynchronous version, achieved by adding several leads to the common bus and some additional logic to each cell. The asynchronous machine runs more rapidly because usually an instruction does not have to propagate to the ends of the machine before a succeeding instruction can be issued. The increased speed in small applications may or may not be worthwhile; but this question will not be treated here.

III. INSTRUCTION SET

The DLM computer has a rich structure; that is, there are many, many alternatives to the design of the logic within the cell and to the number of leads interconnecting cells. The selection of an instruction set is made empirically, just as it is with a conventionally organized machine. The test of the power of an instruction set comes when actual programs are written and compared.

The instruction set for the improved cell memory (Gaines and Lee, 1965) is taken as the starting point. In particular, MATCH, SET and STORE are bit manipulating instructions. The pattern B refers to the bit pattern of the memory word. Any or all bits of the memory word may be specified; any bits that are not specified are treated as "don't cares". An active cell is one with its M (Match) flip-flop set, and C is the cell's so-called Control flip-flop. The following describes the instructions of the improved cell memory:

MATCH B	- Set the Match flip-flop in each cell in which the pattern B and the condition on C, if specified, are stored.
SET B	- Store the pattern B and the conditions on C or M, if specified, in every cell.
STORE B	- Store the pattern B and the condition on C, if specified, in each active cell.

READ	- Read out the pattern bits of any active cell onto the common bus.
MARK	- Simultaneously activate all cells to the right of each active cell, up to an including the first cell whose C flip-flop is set. (In order to avoid a timing problem, an active cell with its C flip-flop set would not activate its right neighbor.)
LEFT	- Activate the left neighbor of each cell whose C flip-flop is set.
RIGHT	- Activate the right neighbor of each cell whose C flip-flop is set.

For example, STORE ($C=1, X_8=0$) would cause active cells only to have their C flip-flops set to 1 and the eighth bit of their memory words set to 0. Other instructions concerned with program sequencing and with input-output must be added to the above instructions since the DLM computer will no longer have a driving computer.

Cells may be used to store instruction or data words. The two must be distinguished in some way because an instruction normally operates on data words and not on other instructions. In fact, it can be shown that other instructions can be inadvertently destroyed if instructions are permitted to operate indiscriminately on all cells. Two approaches were tried. In the first case a unique logical state was initially assigned to a cell depending upon the cell's use. However, this leads to a complicated logical state structure. In the second case a portion of the memory word, say the low-order bits, was assigned as a key to the cell's use. Not just two but five distinct uses for cells have been found helpful, and these are shown in Figure 2. Notice that the assignment of USEKEY's is arbitrary and may be altered in an actual realization. The coding of five USEKEY's onto three bits leaves three combinations available for the use of the programmer.

Program labels are a convenient means to deal with the "addresslessness" of a content-addressed memory. Thus, at a conditional or unconditional branch which interrupts an instruction sequence, the program label of the next series of instructions is identified and instruction sequencing carries on from that point. Program labels are also used to provide a rapid means of locating a particular point within a particular instruction string to facilitate instruction modification. Other cell uses will be explained later.

Cells must go into intermediate logic states as instructions are carried out. At the present time, twenty-two logical states are used, coded onto five logical state flip-flops. The number of logical states required, of course, depends on the instruction set proposed.

The instruction set is shown in Figure 3. The present contents of a cell are represented by a double (USEKEY, Symbol). For example, (6, 18)(5, I) is shown for the short form of MATCH. This is a two-cell instruction. The first cell contains a USEKEY of 6 because it stores the op-code, which is 18. The second cell is the operand, which carries a USEKEY of 5; "I" symbolizes the stored bit pattern of the symbol. In a typical three-cell instruction, such as (6,5)(5,I)(5,J), "I" and "J" symbolize stored bit patterns. The assignment of operation code values is arbitrary.

Some instructions take several cells to arrange. An attempt has been made to reduce the number of cells required for an instruction at the expense of additional operation codes. It is felt worthwhile to increase the complexity of each cell somewhat if the total number of cells required for a job can be significantly reduced.

The proposed instruction set has been derived in part from previous works (Gaines and Lee, 1965; Crane and Githens, 1965; Sturman, 1968b). (Among other important results, Crane and Githens (1965) give efficient algorithms for numerical operations using a similar instruction set.) Both long and short forms of certain instructions are provided. This is analogous to the use of variable length instructions in current, conventionally organized machines. Significant storage and execution time can be saved if the shorter instructions can be used extensively.

The detailed structure of a typical instruction is shown in Figure 4. For STORE 0 ($C=0, X_8=1$) three cells are required. The first cell contains the op-code and has a USEKEY of 6; the second and third cells contain operand fields and have USEKEY's of 5. Using separate

USEKEY's for program op-code and program operand cells is an alternative which simplifies the logical state diagram. With but one USEKEY, eight additional logical states would be needed. The op-code is stored in bits X_3 through X_7. The remaining eight bits of the first cell (X_8 through X_{15}) act as modifiers for the instruction and specify on which types of cells the instruction is to operate. One bit is associated with each of the eight USEKEY's and these are identified in Figure 4. For example, bit X_8 is associated with a USEKEY of 0, and bit X_{15} is associated with a USEKEY of 7. Thus, a 1 only in bit X_8 specifies that the instruction is to operate only on DATA-WORD's, which have a USEKEY of 0.

The operand fields of the second and third cells specify which bits are involved and which bits are don't cares. Bit positions X_3 through X_{13} can be specified. Bit positions X_{14} and X_{15} are used to specify conditions for the C and M flip-flops, respectively. Therefore, bit positions X_{14} and X_{15} of data words are normally unused. Ones are placed in the operand field of the second cell in those bit positions specified as ones; other bit positions are filled with zeros. Similarly, zeros are placed in the operand field of the third cell in those bit positions specified as zeros; ones are filled into the other bit positions. In this way a zero in the second cell and a one in the same bit position of the third cell specify a don't care. In the example of Figure 4, ones are placed in both X_8 bits to accomplish $X_8=1$

and zeros are placed in both X_{14} bits for C=0.

DLM computer instructions are specified in a three-element symbolic format: op-code, modifiers and information for the operand or operands. Recall the current example:

STORE 0 (C=0, X_8=1)

"STORE" is a mnemonic for the op-code. "0" specifies a modifier field which includes only cells whose USEKEY's are 0, i. e., DATA-WORD cells. "(C=0, X_8=1)" specifies the bit pattern required in the operand cells. Consider one further example:

RESETM 0, 1, 2, 3, 4, 5, 6, 7

RESETM is a one-cell instruction so there is no information for operands. The modifier field specifies all USEKEY's. Therefore, the instruction is to operate on all cells.

The three intercell leads A, M and Z may now be described. Both A and M are carried over directly from the cell memory (Gaines and Lee, 1965). Lead A is associated with MARK-LEFT or MARK-RIGHT and propagates a signal either to the left or right, respectively. Lead M is associated with LEFT or RIGHT and also propagates a signal either to the left or right, respectively. A signal on lead A and a signal on lead M will never occur simultaneously so that a single intercell lead would be sufficient at the cost of encoding and decoding circuitry in each cell. Lead Z propagates a signal for instruction sequencing only to the right. Z signals apply only to cells used as PROGRAM-OPCODE's, PROGRAM-OPERAND's and PROGRAM-LABEL's; other cells disregard Z signals. To begin execution of a program, a Z signal is supplied as the left input to the entry cell of the program. This cell responds by placing its contents onto the common bus. At the next clock time it issues a Z signal to its right neighbor. Instruction sequencing continues in this way until either a branch instruction or the end of the program is reached. The branch instruction is arranged to automatically stop the current chain of Z signals and to begin a new one at the next instruction to be executed.

The common bus transmits a cell's contents to all cells. It consists of one lead per bit for bits X_3 through X_{15}; the USEKEY in bits X_0 through X_2 need not be transmitted. Only if the bit transmitted is a logical one will there be a signal on that bus lead. In addition, there is a $\emptyset$ lead which runs to all cells. The $\emptyset$ lead is active whenever the first cell of an instruction is being transmitted on the common bus, i. e., the op-code of the instruction. In this way cells can keep in step with the instruction stream that is coming down the common bus. There is a possible tradeoff here between supplying the $\emptyset$ lead and using a separate USEKEY for PROGRAM-OPCODE and PROGRAM-OPERAND, or else adding eight additional logical states which act essentially as timing states to keep the cells in step.

The following sections describe the individual instructions in detail. Each instruction has modifiers that specify which cells are to be operated on, according to USEKEY. Cells that are not specified simply ignore the instruction entirely.

LEFT, RIGHT, MARK-LEFT and MARK-RIGHT

These are all one-cell instructions which cause communication between cells. LEFT or RIGHT causes the left or right neighbor, respectively, of each cell whose C flip-flop is set to be made active. This is done by a signal on the M intercell lead.

MARK-LEFT or MARK-RIGHT causes simultaneous activation of all cells to the left or right, respectively, of each active cell up to and including the first cell whose C flip-flop is set. This is done by a signal on the A intercell lead. To avoid a timing problem, an active cell whose C flip-flop is set does not begin activation. Notice that MARK-LEFT or MARK-RIGHT can cause many cells to be activated simultaneously, i. e., within one machine cycle. A primary use of the C flip-flop, therefore, is to control intercell communication.

One condition can cause stoppage of the A-signal propagation before a cell is encountered whose C flip-flop is set: If a cell in the chain is found whose USEKEY was not specified in the modifiers of the marking instruction, then not only is the A-signal propagation stopped, but also that particular cell is not activated.

MATCH, MATCH-LEFT, MATCH-RIGHT, SET and STORE

Members of this family of instructions have similar formats. Both long and short forms are available except for MATCH-LEFT, which is always a three-cell instruction. These instructions are the most often used and are the primary data manipulating instructions. In all of these instructions, only bits X_3 through X_{13} can be specified; bits X_{14} and X_{15} of a cell are ignored. Also, bits X_0 through X_2 (USEKEY) are specified as a modifier that goes with the instruction's op-code. Bits X_{14} and X_{15} of the instruction operand specify conditions on M and C flip-flops.

Consider MATCH first. The short form (6, 18)(5, I) applies to the case where the symbol to be matched against can be completely specified, i. e. , no bit positions are don't cares. The op-code 18 tells all cells that the instruction is a short-form MATCH, and that during the next clock time, the symbol I to be matched against will appear on the common bus. Cells that match set their M flip-flops and other do not.

The long form (6, 5)(5, I)(5, J) applies to the usual case where the symbol to be matched against contains don't cares in certain bit positions. The symbol I contains ones in those bit positions that are to be matched as ones, and zeros in the don't care positions. Similarly the symbol J contains zeros in those bit positions that are to be matched as zeros, and ones in the don't care positions. By supplying symbols I and J properly, don't care bit positions are easily accommodated.

MATCH-LEFT and MATCH-RIGHT are similar except that in case of a match condition, it is the left or right neighbor, respectively, that is activated via the M intercell lead.

SET and STORE both cause symbols to be written into cells. STORE has the restriction that it operates only upon active cells whereas SET operates on all cells whose USEKEY's were specified. Because SET and STORE are so similar they can share the same logical states.

The short forms of these instructions are a convenience and save one cell whenever they can be used. However, the long forms are sufficient for all cases.

RESETM and RESETC

Both of these are one-cell instructions. RESETM is equivalent to the three-cell instruction SET (M=0) and similarly RESETC is equivalent to SET (C=0). These operations often come up in programs so that it is thought worthwhile to include short-form equivalent instructions. Since each instruction has modifiers, one can selectively reset cells, according to USEKEY.

STORE-SPECIAL and SET-SPECIAL

These instructions are included to permit changing a cell's USEKEY and its bits X_{14} and X_{15}. This is useful in instruction modification and also in loading the machine initially.

In the long form of STORE-SPECIAL, no don't cares are permitted; yet it is a three-cell instruction. STORE-SPECIAL operates only on active cells. The first operand contains the desired USEKEY in bit positions X_3 through X_5 and the desired values for bits X_{14} and X_{15} in bit positions X_6 and X_7, respectively; bit positions X_8 through X_{15} are unused. The second operand contains the desired bit pattern in bit positions X_3 through X_{13} and the condition on C in bit position X_{14}; bit position X_{15} is unused.

SET-SPECIAL and the short form of STORE-SPECIAL only permit changing a cell's USEKEY and are two-cell instructions. The operand contains the desired USEKEY in bit positions X_3 through X_5 and other bit positions are unused. These two instructions are similar and share the same logical states. STORE-SPECIAL operates only on active cells whereas SET-SPECIAL operates on all cells.

OUTPUT

The OUTPUT instruction (6, 14)(5, 0) alerts all active cells to place their contents on the common bus during the next clock time and to reset their M flip-flops. If there is more than one cell active there may be an ambiguous result on the common bus because the bus OR's the symbols presented to it. If so, then an isolating algorithm which makes just one particular cell active should precede the OUTPUT instruction. On the other hand, certain algorithms rely on the OR function of the bus. The null operand (5, 0) does not perturb the common bus and, therefore, the bus is free to carry the released symbol or symbols. However, the null operand does supply continuity in the Z-signal propagation for proper instruction sequencing.

CONDITIONAL-BRANCH and UNCONDITIONAL-BRANCH

Branching, i. e., interrupting normal instruction sequencing, is a relatively complicated operation. The current Z-signal propagation must be stopped, the next instruction string to be executed must be located and the new Z-signal propagation must begin at this point. Nevertheless, conditional branching is basic, especially to a control application. Therefore, an attempt has been made to provide efficient branching instructions.

UNCONDITIONAL-BRANCH occurs at the end of one instruction string, linking to the next instruction string. To identify the start of an instruction string it is convenient to use a unique PROGRAM-LABEL (4, I). The cell acts as an address--a place to branch to. UNCONDITIONAL-BRANCH (6, 6)(5, I) alerts all program labels to match the symbol on the bus at the next clock time. Symbol I is the next symbol on the bus. The program label with symbol I matches this symbol and issues a Z signal to its right at the next clock time.

Conditional branching is further complicated by having to branch or not, depending upon the result of a previous computation. Branch on Match is provided. That is, if one or more cells are active, then a branch is to take place; otherwise the present string of instructions is to be continued. In fact the type of cells that must be active for branching is specified as a modifier for the CONDITIONAL-BRANCH1 instruction.

CONDITIONAL-BRANCH1 assumes that specified active cells, if any, will contain the branch address to which control should be transferred. This unusual requirement permits the simplest arrangement found thus far for accomplishing conditional branching. CONDITIONAL-BRANCH1 (6, 16)(5, 0) alerts all program labels to match the next symbol on the bus and at the same time acts like the OUTPUT instruction to cause all selected active cells to place their symbols on the bus during the next clock time and reset their M flip-flops. The (5, 0) null operand frees the bus during its clock time but also interrupts ordinary Z-signal propagation temporarily by going into an intermediate logical state because it has seen the op-code 16 on the bus and received a Z signal at the same time.

Consider first the case where there are no matches and hence no need to branch. The symbol on the bus during the null operand will be all zeros and therefore the program label (4, 0) matches its symbol and issues a Z signal to its right. The doublet (4, 0)(6, 15) need appear just once in the program whenever any number of CONDITIONAL-BRANCH1 instructions are present. CONDITIONAL-BRANCH2 (6,15) is the next instruction that goes out on the bus but further Z-signal propagation is automatically stopped at this point. The null operand is monitoring this clock time for the op-code 15. In this case the op-code 15 is seen and the null operand issues a Z signal to the right. Ordinary processing continues from this point. If the op-code were not 15, the null operand would simply reset and not issue a Z signal.

Consider now the case where there is at least one selected active cell and hence the need to branch. The symbol on the bus during the null operand will automatically be the preset program label of the next instruction sequence. Therefore, that PROGRAM-LABEL will match and issue a Z signal to its right. The null operand which was monitoring the bus for the 15 will simply reset and thereby stop the old Z-signal propagation.

TRANSFER

Transferring data from one part of memory to another is a relatively inefficient process. TRANSFER requires selecting the receiving cell, selecting the supplying cell and finally causing the receiving cell to pick the information off the common bus during a null operand on the bus. Therefore, where possible, data will be used wherever it exists and transfer of data will be avoided.

TRANSFER provides for moving one symbol of data in memory. Multiple transfers would require multiple TRANSFER instructions.

The first step before TRANSFER is to match the receiving cell (or cells) using ordinary instructions. Then TRANSFER (6, 1) causes selected active cells to change logical state and to reset their match flip-flops. Such cells now monitor the bus for the next OUTPUT sequence (6, 14)(5, 0) and ignore all other instructions.

Because the transfer operation can be a variable length operation, any number of instructions may now be inserted before the output sequence. The one function that must be completed is to match the supplying cell.

The OUTPUT instruction (6, 14)(5, 0) causes the selected active cell (which is the supplying cell in this case) to place its symbol on the bus. The receiving cell (or cells), which has been monitoring the bus for (6, 14), takes up the symbol during the null operand to complete the transfer. All bit positions X_3 through X_{15} are transferred.

INTERPRETIVE-MATCH

INTERPRETIVE-MATCH is a convenient means of comparing the contents of two isolated cells when the full contents of neither cell are known. It is similar in some ways to the transfer instruction. A receiving cell is selected, a supplying cell is selected and finally the symbol of the supplying cell is put onto the common bus. The receiving cell matches its contents against the bus and sets its Match flip-flop accordingly. In this case only bits X_3 through X_{11} currently enter into the match; this permits the other bits to be used as flags which need not be identical.

The first step before INTERPRETIVE-MATCH is to match the receiving cell. Then INTERPRETIVE-MATCH (6, 4) causes selected active cells to change logical state and to reset their Match flip-flops. Such cells now monitor the common bus for the OUTPUT sequence (6, 14)(5, 0) and ignore all other instructions. The supplying cell is now matched. The OUTPUT instruction (6, 14) causes a selected active cell to place its symbol on the bus during the following null operand. The receiving cell matches its contents against the common bus during the null operand and sets its Match flip-flop only if a match exists on bit positions X_3 through X_{11}. At this time the supplying cell is not active and the result of the interpretive match is at the receiving cell.

PROGRAM-LABEL and DATA-LABEL

PROGRAM-LABEL (4, I) is a unique label that can be placed within instruction strings not only in connection with branching instructions but also to facilitate instruction modification. It propagates Z signals immediately without the usual one clock time delay and does not perturb the common bus. It responds to instructions when its modifier has been specified just as all cells do. PROGRAM-LABEL provides a rapid means of locating a particular point within a particular instruction string.

DATA-LABEL (1, I) is another unique label that can be placed within data areas to facilitate rapid isolation of selected data areas for further processing.

Since PROGRAM-LABEL's and DATA-LABEL's are sometimes accessed by MATCH, MATCH-RIGHT or MATCH-LEFT instructions, which check bit positions X_3 through X_{13} only, there are really only 2^{11} or 2048 useful combinations available for each type of label. For small applications of perhaps 1000 instructions, this number of labels should be adequate. However, for larger applications, the following two design alternatives are available:

1. Expand the memory word to more bits. Each added bit doubles the number of useful combinations for labels. At the same time this may permit more efficient programs to be written because of the additional bits per cell.
2. Modify the cell logic to permit labels to extend over more than one cell. For example, there could be one-cell labels, two-cell labels, three-cell labels and so forth. Potentially any number of labels are permitted but there is an overhead associated with isolating the particular label of interest.

IV. Z-SIGNAL PROPAGATION

This section will complete the description of Z-signal propagation. Z signals control instruction sequencing; there is no instruction counter. In a typical instruction chain, a Z signal propagates to the right from cell to cell at each cycle time. Receipt of a Z signal instructs a cell to place its contents onto the common bus and to send a Z signal to its right neighbor during the next clock time. There normally will be only one active Z signal at a time in the entire computer. A cell's Z flip-flop is automatically reset. There are five special situations:

a) A cell in a data area (i. e. , a cell with a USEKEY of 0, 1, 2, 3 or 7) does not set its Z flip-flop upon receipt of a Z signal. No properly written program will try to execute data as instructions; control is transferred around data areas by unconditional branching. Thus the contents of a cell's memory word condition the setting of its Z flip-flop.

b) A cell being used as a PROGRAM-LABEL propagates a Z signal immediately without the usual one-cycle delay and does not put its contents onto the common bus. The cell provides a direct path through itself for Z signals. Thus, its Z flip-flop is not used.

c) Recall that CONDITIONAL-BRANCH1 is a two-cell instruction of the form (6, 16)(5, 0). The null operand is the cell which temporarily interrupts ordinary Z-signal propagation. It responds to the presence of an op-code of 16 on the bus and the receipt of a Z signal by going into an intermediate logical state.

d) Recall that UNCONDITIONAL-BRANCH is a two-cell instruction of the form (6, 6)(5, I). The operand responds to the presence of an op-code of 6 on the bus and the receipt of a Z signal by inhibiting the setting of its Z flip-flop. Thus, this chain of Z-signal propagation is stopped.

e) Recall that CONDITIONAL-BRANCH2 is a one-cell instruction of the form (6, 15). Upon receipt of a Z signal, the instruction inhibits the setting of its Z flip-flop which stops this chain of Z-signal propagation.

V. STATE TRANSITION DIAGRAM

The logical state of a cell is an encoding of the S_0 through S_4 state flip-flops of Sequence Control. Twenty-two logical states have been found sufficient to implement the proposed instruction set of Figure 3. Consider now the detailed state changes for the typical instruction MATCH - Long Form (6, 5)(5, I)(5, J), shown in Figure 5. State 0 is designated as the normal idle state for all cells. Therefore, the cells storing the MATCH instruction as well as the cells on which the MATCH instruction is to operate all begin in State 0.

Assume that the instruction MATCH 0 ($C=0, X_8=1$) is to be executed. This is a long-form MATCH with MODIFIER 0, which therefore is to operate on cells with USEKEY=0, i. e. , DATA-WORD's. Thus, all DATA-WORD cells with C flip-flop reset and eighth memory word bit set are to set their Match flip-flops; all other cells do not. The three cells containing the MATCH instruction propagate Z signals and place their contents on the common bus during successive clock times while they remain in logical State 0; there is no need for them to go into intermediate states. However, all DATA-WORD's must go into an intermediate state, which in this case is State 4.

All cells are constantly monitoring the $\emptyset$ lead of the bus because when the $\emptyset$ lead is 1 there is an instruction op-code on the bus. Assume that the contents of the first cell of the MATCH instruction have now been placed on the bus; the $\emptyset$ lead is 1 because an op-code is being transmitted. All cells now compare the modifiers being transmitted with their own USEKEY's. If a particular cell finds that its USEKEY has not been specified, then it simply remains in State 0 and waits until the $\emptyset$ lead becomes 1 again during some succeeding clock time. On the other hand, all DATA-WORD's in this case recognize that they are being specified, decode the op-code bits and go into the appropriate intermediate state (State 4). This completes the action during the first clock time.

During the second clock time the contents of the second cell of MATCH are placed on the bus; the $\emptyset$ lead is 0 because an operand is being transmitted. Only cells in State 4 take any action; all other cells wait during this clock time. Cells in State 4 compare their individual contents with the current signals on the common bus. Specifically such cells monitor bus leads X_3 through X_{13} and bus lead X_{14}; bus lead X_{15} is ignored. Bus lead X_{14} carries the specification for the Control flip-flop in this case. Cells try to form a tentative match involving just those bit positions for which ones are being transmitted on the bus. In this example only bus lead X_8 is transmitting a one. Thus, all cells in State 4 whose eighth bit is a one achieve a tentative match and go to State 5; all other cells in State 4 return to State 0 and wait for the next instruction to come down the bus. This completes the action during the second clock time.

During the third clock time the contents of the third cell of MATCH are placed on the bus; the ∅ lead remains at 0 because an operand is being transmitted. Only cells in State 5 take any action; all other cells wait during this clock time. Cells in State 5 try to form a match involving just those bit positions for which zeros are being transmitted on the bus. In this example only bus lead X_{14} is transmitting a zero. Thus all cells in State 5 whose Control flip-flop is reset achieve a match and set their Match flip-flops; all other cells do not. In either case such cells return to State 0. This completes the action for the MATCH instruction.

The complete state transition diagram is shown in Figure 5. Generally three-cell instructions require two intermediate states, as for example the State 0, State 4, State 5 complex used for MATCH (Long Form); two-cell instructions require one intermediate state and one-cell instructions do not require intermediate states. SET and STORE are the only instructions found that can share corresponding intermediate states. Other instructions are independent in terms of the logical states that they use. Thus, at this stage in the design it is a relatively simple matter to add or delete instructions.

There are four stable states: State 0, the normal idle state; State 11, associated with TRANSFER; State 13, associated with INTERPRETIVE-MATCH; and State 17, associated with CONDITIONAL-BRANCH1. Consider again the use of the ∅ lead to keep cells in step with the instruction stream coming down the bus. The alternative is two timing states per stable state for a total of eight additional states as mentioned previously.

States associated with the various other instructions can now be identified. States 1 and 2 are associated with MATCH-LEFT, and States 9 and 10 are associated with the long-form MATCH-RIGHT. These instructions are implemented in a similar way to the long-form MATCH instruction described above; if a cell is in State 1 or State 9 and does not achieve a match on the first operand, the cell returns to State 0.

State 3 is associated with the short-form MATCH. A cell in State 3 returns to State 0 whether or not its contents match the operand; of course, the cell's Match flip-flop is set accordingly in the process. State 6 is associated with the short-form MATCH-RIGHT and operates in a similar way.

States 7 and 8 are shared by the long forms of SET and STORE. A cell can reach State 7 by having its USEKEY specified and an op-code for SET (Long Form) or an op-code for STORE (Long Form) if the cell's Match flip-flop is set. In a similar way State 20 is shared by the short forms of SET and STORE; State 21 is shared by SET-SPECIAL and the short-form STORE-SPECIAL. States 18 and 19 are associated with the long-form STORE-SPECIAL.

States 11 and 12 are associated with TRANSFER. A cell in

State 11 remains there and ignores all instructions on the bus except for OUTPUT; when OUTPUT is received, the cell advances to State 12. In a similar way States 13 and 14 are associated with INTERPRETIVE-MATCH. A cell in State 13 remains there until OUTPUT is received, ignoring all other instructions on the bus.

State 15 is used only by cells which are PROGRAM-LABEL's (USEKEY=4). When a PROGRAM-LABEL cell receives an op-code for UNCONDITIONAL-BRANCH or CONDITIONAL-BRANCH1, the cell goes to State 15. At the next clock time each PROGRAM-LABEL matches its contents with the operand on the bus. That PROGRAM-LABEL which achieves a match issues a Z signal to its right neighbor and returns to State 0; all other PROGRAM-LABEL cells simply return to State 0.

States 16 and 17 are used only by null PROGRAM-OPERAND's (USEKEY=5, Symbol=0) and are associated with conditional branching. A null PROGRAM-OPERAND in State 0 which receives an op-code for CONDITIONAL-BRANCH1 proceeds to State 16. The null PROGRAM-OPERAND which then receives a Z signal proceeds to State 17 at the next clock time; all others return to State 0. A cell in State 17 waits for the next instruction on the bus before returning to State 0. If the next instruction on the bus is CONDITIONAL-BRANCH2, the null PROGRAM-OPERAND in State 17 issues a Z signal to its right neighbor as it returns to State 0; otherwise no Z signal is issued.

VI. PROGRAMMING STRATEGY

Cells are identical but may be put to a variety of uses. Each cell has its own Match and Control flip-flops which are needed for data manipulations or instruction modification. Modifiers on instructions specify just which types of cells are to respond. In this way the machine can pass between various active states without having to continually save and restore M and C flip-flop contents. The concept of USEKEY also permits simple clearing and loading of the machine initially as well as instruction modification.

Physical location within the machine is important even though there is a content addressable structure. Certain instructions communicate with an immediate left or right neighbor of a cell of interest while other cells communicate with a number of cells in either the left or right direction. In addition, labels act as addresses and permit rapid location of a point or points within the machine.

For storage efficiency certain cells may have identical contents while representing entirely different information. There must be a distinguishing feature. It may be proximity to a unique label or simply the leftmost active cell. In the DLM computer, the programmer is continually isolating cells of interest from all others. Cells of interest can be flagged for further processing by setting the higher order bits of the memory word to a particular pattern. Thus typically the lower order bits store the information of a data word and the higher order bits act as flags. In addition, the Control flip-flop can act as a flag when it is not needed in connection with intercell communication.

In the following sections algorithms are presented in a symbolic machine language. A compiler program which interprets the symbolic statements and generates the required machine language code has been written by J. L. Cottrill (1968). For example,

```
MARK-RIGHT 0,1,3
```

would generate a one-cell MARK-RIGHT instruction with modifier bits specifying cells of USEKEY equal to 0, 1 and 3. Similarly,

STORE 0 (X_8=0)

would generate a three-cell STORE instruction with modifier bits specifying cells of USEKEY equal to 0 and operands arranged to reset bit X_8 and leave other memory word bits untouched. Finally,

```
SET 2 (3AE8)
```

would generate a two-cell SET instruction with modifier bits specifying cells of USEKEY equal to 2. In this short-form instruction the bit pattern of the operand is specified by the four-digit hexadecimal number 3AE8. This number is expanded below to show its correspondence with the desired bit pattern:

```
Hexadecimal
Number
3AE8   |0  0  1  1 |  1   0   1   0 | 1   1   1   0 | 1   0   0   0 |
        M  C X13 X12  X11 X10 X9  X8  X7  X6  X5  X4  X3
```

This specification is equivalent to: ($M=0, C=0, X_{13}=1, X_{12}=1, X_{11}=1, X_{10}=0, X_9=1, X_8=0, X_7=1, X_6=1, X_5=1, X_4=0, X_3=1$); since this is a two-cell instruction there can be no don't care bit positions. Because this cell is a program operand, its USEKEY must be 5 and, therefore, bits X_0 through X_2 are not specified.

VII. CLEARING AND LOADING THE DLM COMPUTER

The concept of USEKEY, i. e., the placing of part of the sequential control of the cell's logic in the memory word itself, permits efficient means for clearing and loading the computer. Access only to the common bus and to the intercell leads of the leftmost cell in the computer is all that is needed. (Actually the computer can also be loaded from its right-hand end.) The STORE-SPECIAL instruction which is useful in instruction modification is also required for clearing and loading.

A special loader is envisioned which can be attached to the left-hand end of the DLM computer. The loader places signals on the common bus and occasionally must signal the M and Z intercell leads of the leftmost cell. The DLM computer acts as a cell memory during clearing and loading while being driven by the loader.

Consider first the case when it is necessary to clear the entire machine and load initially. This need arises for example when power is first turned on. Actually the memory elements of the cells may come on in any state and the clearing routine will clear the entire machine with one exception: The Z flip-flop per cell associated with Z-signal propagation must be reset initially. This can be done by an additional lead of the common bus, running to each cell. Such resetting prevents the machine from coming on in a state to be executing instructions in a continuous loop. This feature can also be used to stop the machine as a maintenance tool.

The CLEARING routine consists of the following five instructions:

OUTPUT 0, 1, 2, 3, 4, 5, 6, 7

RESETC 0, 1, 2, 3, 4, 5, 6, 7

MARK-RIGHT 0, 1, 2, 3, 4, 5, 6, 7

STORE-SPECIAL 0, 1, 2, 3, 4, 5, 6, 7 (0000, U=0, C=0)

RESETM 0

This routine is sufficient to cause the resetting of every flip-flop of every cell, assuming that every Z flip-flop has already been reset, even though a cell may have been in any state with any memory word contents. OUTPUT causes all cells to go to State 0. All cells respond because all USEKEY's were specified as modifiers. Cells in stable States 11 and 13 are freed by OUTPUT; cells in State 17 go to 0 on any instruction; cells in unused States 22 through 31 automatically go to State 0 after the first clock time. RESETC resets the C flip-flops in all cells. Now at this point, if the leftmost cell would be made active, a succeeding MARK-RIGHT would activate all other cells in the machine. This is what is done. During the RESETC above, the loader activates the leftmost cell by a signal on the M intercell lead. MARK-RIGHT therefore causes all cells to be activated. All cells respond to STORE-SPECIAL therefore because all cells are active. STORE-SPECIAL causes zeros to be written into all bits of the memory word, including the USEKEY. Thus all cells have been transformed into data words. Finally RESETM resets all Match flip-flops. That the machine can be cleared with five instruction executions begins to illustrate the power of the content-addressed structure in which many cells can be operated upon simultaneously.

Loading of the machine can now begin. Loading takes place from the leftmost cell to the rightmost cell, one cell at a time. The loader accepts one cell's contents from an external source and supplies a short routine which causes this information to be stored in the appropriate cell. This routine is repeated as many times as there are cells to be loaded. The strategy is to activate the cell to be loaded and to store its contents while setting its C flip-flop. Its Match flip-flop is reset, its right neighbor activated and then its C flip-flop reset. The next cell to be loaded is now active so the routine can be repeated. The LOADING routine consists of the following four instructions:

```
STORE-SPECIAL 0, 1, 2, 3, 4, 5, 6, 7 (XXXX, U=X, C=1)
OUTPUT 0, 1, 2, 3, 4, 5, 6, 7
RIGHT 0, 1, 2, 3, 4, 5, 6, 7
RESETC 0, 1, 2, 3, 4, 5, 6, 7
```

Note: "X's" in STORE-SPECIAL designate information supplied by the external source for the particular cell being loaded.

The loader begins by activating the leftmost cell with a signal on its M intercell lead. The appropriate STORE-SPECIAL stores the required bit pattern in the leftmost cell, including USEKEY, and at the same time sets its C flip-flop. OUTPUT reads out onto the bus the information that has just been stored, which the loader may check, and resets the M flip-flop of the leftmost cell. RIGHT activates the right neighbor of the leftmost cell and RESETC resets its C flip-flop. The next cell to be loaded is now the only one active and the routine may be repeated.

When loading is complete, program execution can begin. One alternative to starting execution is to have the loader supply a branch instruction to the entry point of the program; then after execution has begun, the loader is disconnected. However, it is probably safer to do the following: (a) Use the leftmost two cells to store a branch instruction to the entry point of the program. (b) Disconnect the loader immediately after loading is complete. (c) Supply a signal to the leftmost cell on its Z intercell lead. Thus processing will begin by a branch to the entry point.

VIII. PROGRAMMING EXAMPLES

In this section several examples will be described to illustrate how the proposed instruction set can be used. At the present time, approximately 40 short routines have been written for a variety of tasks.

Isolating the Leftmost Active DATA-WORD Cell

Algorithms for the cell memory (Gaines and Lee, 1965; Crane and Githens, 1965) can be rewritten for the DLM computer. Such is the case with this example. Assume that a number of DATA-WORD cells are active and one wishes to deal with the leftmost first. Use is made of MARK-RIGHT which can cause many cells to be activated during one clock time.

Consider the following:

```
MARK-RIGHT 0
STORE 0 (C=1)
RESETM 0
RIGHT 0
STORE 0 (C=0)
RESETM 0, 1, 2, 3, 4, 5, 6, 7
MATCH 0 (C=1)
```

The assumption is made that between active DATA-WORD cells there are no cells which are not DATA-WORD's, that all DATA-WORD's have their C flip-flops reset and that the contents of M flip-flops need not be retained. If these conditions are not met, a more complicated algorithm is needed.

MARK-RIGHT causes all DATA-WORD's to the right of the left-most active DATA-WORD to be activated, up to but not including the first cell encountered with differing USEKEY. STORE (C=1) sets the C flip-flops of all active DATA-WORD's and RESETM resets their M flip-flops; RIGHT activates the right neighbors of such cells. Notice that what had been the leftmost active DATA-WORD is not activated and therefore has been isolated. The remaining steps reset active DATA-WORD's and then activate what had been the leftmost active DATA-WORD.

Isolating a Flagged Cell Among Many

This procedure permits efficient use of flags and illustrates how data labels can be helpful. Consider a portion of memory laid out with groups of data words separated by data labels. Assume previous processing has flagged one data word in some groups by setting bit X_{11}. Hence many data words may be so flagged and isolation of a particular one relies on its proximity to the data label of its group. One can quickly determine if there is a flagged cell in a particular group and activate it by the following:

```
MATCH 0 ($X_{11}$=1)
STORE 0 (C=1)
RESETM 0
MATCH-RIGHT 1 (03F0)
MARK-RIGHT 0
STORE 0 ($X_{13}$=1)
RESETM 0
MATCH 0 (C=1, $X_{13}$=1)
RESETC 0
SET 0 ($X_{13}$=0)
```

MATCH (X_{11}=1) and STORE (C=1) set C flip-flops in flagged data words; RESETM clears the M flip-flop in the same cells. Assume that the group of data words preceded by DATA-LABEL (03F0) is the particular group of interest; MATCH-RIGHT (03F0) activates the right neighbor of this data label. MARK-RIGHT begins at this point to activate cells in a sequence to the right until a data word cell is found with its C flip-flop set or until the next data label is found. In the latter case no flagged cell was found in this group and the remaining instructions simply restore conditions. Consider the former case. The flagged cell is the last cell in the sequence to be activated. It has been isolated because it is the only active data word with its C flip-flop set. The remaining instructions restore conditions and leave the isolated, flagged cell active. STORE (X_{13}=1) sets a temporary flag in all active data words and RESETM clears the M flip-flop in the same cells. MATCH (C=1, X_{13}=1) activates the isolated, flagged data word. Finally RESETC resets C flip-flops and SET (X_{13}=0) resets the temporary flags.

MARK-LEFT and MARK-RIGHT have been found useful to signal from one point in memory to another in one clock time. The flagged cell isolates itself in this case by stopping A-signal propagation. Notice that if there were several flagged data words in one group of interest, this algorithm would isolate the leftmost.

A feature of various DLM instructions should be noted. An instruction, such as MATCH or STORE, may or may not change the contents of any cells depending upon the state of the cells at execution time. Thus, in the above algorithm the latter half can remain the same whether or not a flagged cell is found; there is no need to set up conditional branching machinery to be able to handle the two cases. Nothing would be saved by conditional branching in this example. Of course, there are other cases where the contrary is true and conditional branching instructions are available. However, one would expect to find less branching when compared to a conventionally organized machine not only because of the instruction set but also because there is less need for looping since operations are carried out in parallel.

Subroutine Linkage

Ordinary subroutine linkage provides an example of instruction modification (actually "address" modification). Before a subroutine is entered, the proper return address, i. e., program label is stored in the subroutine by the main program. In fact, the return address is stored as the operand of the UNCONDITIONAL-BRANCH instruction at the logical end of the subroutine. Consider the following:

```
Main Program

------

MATCH-RIGHT 4(3E58)

STORE 5 (0F68)

RESETM 5

UNCONDITIONAL-BRANCH 4 (3F78)

PROGRAM-LABEL (0F68)

------

Subroutine

PROGRAM-LABEL (3F78)

------

------

UNCONDITIONAL-BRANCH 4---┐
                         |
PROGRAM-LABEL (3E58)     |
                         |
(XXXX)-------------------┘
```

In the subroutine notice that PROGRAM-LABEL (3E58) has been inserted between the op-code and operand of UNCONDITIONAL-BRANCH 4(XXXX). The program label does not affect execution of the branch but does permit rapid location of the operand into which the proper return address is to be stored. Refer now to the main program. MATCH-RIGHT (3E58) activates the right neighbor (i. e., the return address operand) of the subroutine's PROGRAM-LABEL (3E58). STORE (0F68) stores the return address and RESETM clears the M flip-flop. Now that the return address has been preset the subroutine can be entered by a branch to its entry point. Notice that PROGRAM-LABEL (3E58) serves as a convenient, unique pointer to a cell that will be changed without interfering with normal instruction execution.

There is normally never a need to physically pass parameters to a subroutine. If the parameters are uniquely flagged, the subroutine can identify them and operate on them where they exist. However, it will often be the case that there is a shortage of unique flags. Therefore, a second technique which employs USEKEY's that are available to the programmer is efficient. For example, assume that three data words are parameters, which are located perhaps in widely separated cells, and are not uniquely flagged in memory. Before branching to the subroutine, the main program can temporarily set the USEKEY's of the parameters to 7, for example, using the short form of STORE-SPECIAL. As long as the parameters are flagged so as to be distinguishable from each other, the subroutine can identify them because of their unique USEKEY. Upon return from the subroutine, the USEKEY's of the parameters can be restored to 0 by using SET-SPECIAL.

Increment Selected Cells by One

This procedure illustrates the parallelism possible in the DLM computer. Assume that certain selected cells each contain a code for one of the decimal digits 0 through 9 and it is desired to increment each digit by 1. The algorithm will accomplish this simultaneously in as many cells as desired. Assume that memory word bits X_6 through X_3 store the decimal digit according to the following table:

Decimal Digit	Memory Word Bits $X_6 X_5 X_4 X_3$
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
10	1 0 1 0

The strategy is to recognize certain patterns and then to supply the appropriate pattern for the incremented digit. Assume that the selected cells which may be anywhere in memory are data words and are temporarily flagged by $X_{13}=1$; as a cell's digit is incremented the flag is reset. Consider the following:

```
MATCH 0 ($X_{13}=1, X_3=0$)
STORE 0 ($X_{13}=0, X_3=1$)
RESETM 0
MATCH 0 ($X_{13}=1, X_4=0, X_3=1$)
STORE 0 ($X_{13}=0, X_4=1, X_3=0$)
RESETM 0
MATCH 0 ($X_{13}=1, X_5=0, X_4=1, X_3=1$)
STORE 0 ($X_{13}=0, X_5=1, X_4=0, X_3=0$)
RESETM 0
MATCH 0 ($X_{13}=1, X_6=0, X_5=1, X_4=1, X_3=1$)
STORE 0 ($X_{13}=0, X_6=1, X_5=0, X_4=0, X_3=0$)
RESETM 0
```

There are four pairs of MATCH and STORE instructions. The first identifies all selected cells which contain an even-numbered decimal digit (i. e., 0, 2, 4, 6 or 8). This digit is incremented simply by setting bit X_3. Similarly, the second pair deals with selected cells whose digits contain the pattern $X_4=0$ and $X_3=1$ (i. e., 1, 5 or 9). In this way all possible cases are treated.

This procedure may appear lengthy just to accomplish addition by one. However, recall that the DLM computer has no adders as such, and that addition by one is being carried out in many cells simultaneously.

Branch on No-Match

This procedure is a convenient means of achieving branching when there are no cells that became active during the previous set of instructions. Note that this is branching in the opposite sense compared to the conditional branching instruction, which provides "Branch on Match". Consider the following:

```
------    } Instructions which may or may
------    } not generate active cells
MARK-RIGHT 0, 1, 2, 3, 4, 5, 6, 7
STORE 0 (C=0)
RESETM 0, 1, 2, 3, 4, 5, 6, 7
RIGHT 0
CONDITIONAL-BRANCH1 0, 4, 5
------    }
          } Work Area
------    }
DATA-WORD (C=1)
DATA-WORD (3AF0)                  /* BRANCH ADDRESS */
------
```

```
------
PROGRAM-LABEL (3AF0)
------
------
PROGRAM-LABEL (0000)
CONDITIONAL-BRANCH2 5
```

Two DATA-WORD cells are placed to the right of the current work area. The first is established as the only cell having its C flip-flop set; its right neighbor contains the branch address (3AF0) which is to be used if a branch is to take place. When this procedure begins it is assumed that if any cells are active, then no branch is desired.

Consider first the case where there are no active cells. The instructions MARK-RIGHT, STORE and RESETM have no effect. RIGHT causes the right neighbor of the cell whose C flip-flop is set to be activated; thus the cell containing the branch address is the only active cell. Then CONDITIONAL-BRANCH1 causes this active cell to release its contents on the bus and instruction sequencing continues from PROGRAM-LABEL (3AF0).

Consider now the case where there are active cells. MARK-RIGHT causes all cells to be activated from the leftmost active cell up to and including the DATA-WORD cell whose C flip-flop is set; this is a means of signaling that at least one active cell is present. STORE resets the C flip-flop of all DATA-WORD's and RESETM resets all M flip-flops. RIGHT has no effect because no cell has its C flip-flop set now. Thus, there are no active cells as CONDITIONAL-BRANCH1 is executed. The net result is that no branch takes place.

A Flexible Multiway Branch

This procedure is similar to the "Computed GO TO" statement in the FORTRAN programming language. It is one way to satisfy the need for a conditional branch, based on a previous computation, to more than two alternative paths.

Consider first a typical "Computed GO TO" statement:

```
GO TO (1513, 21, 783, 2), LINK
```

LINK is an integer which can be given the values 1, 2, 3 or 4. The statement is interpreted to mean that if LINK has the value 1 at this time, then program control is to be transferred to the statement labeled 1513, if 2, then statement 21, if 3, then statement 783 and if 4, then statement 2. Hence the program is directed to one of several paths depending upon the current value of LINK.

The procedure to be described here is somewhat more flexible in that LINK need not be restricted to consecutive values of 1, 2, 3 and 4. Instead LINK can have any value that may be specified by pattern bits X_3 through X_{11}. The following structure can be provided for example:

```
GO TO ((0AF8, 13A0)(0BF0, 0318)(0A18, 0CF8)(0CF8, 3A50)), LINK
```

This statement is interpreted to mean that if LINK contains the symbol (0AF8) then a branch is to occur to PROGRAM-LABEL (13A0); if symbol (0BF0), then branch to (0318); if symbol (0A18), then branch to (0CF8); and finally if symbol (0CF8), then branch to (3A50).

The procedure makes use of a branch table in which each symbol for LINK has the corresponding branch address as its right neighbor. Both TRANSFER and INTERPRETIVE-MATCH instructions are used. Consider the following:

```
MATCH 1 (3E18)
STORE 1 (C=1, 3E18)
MATCH 1 (3E10)
MARK-RIGHT 0, 3
INTERPRETIVE-MATCH 0
RESETM 1, 3
MATCH-RIGHT 1 (03F8)
OUTPUT 0
STORE 0 (C=1)
RIGHT 0
MATCH-RIGHT 4 (30E8)
TRANSFER 5
OUTPUT 3
```

```
RESETM 0

RESETC 0, 1

UNCONDITIONAL-BRANCH 4---,
                          |
PROGRAM-LABEL (30E8)      |
                          |
(XXXX)--------------------'

------

DATA-LABEL (3E10)            /* BRANCH TABLE */

DATA-WORD (0AF8)             /*              */

SPECIAL-3 (13A0)             /*              */

DATA-WORD (0BF0)             /*              */

SPECIAL-3 (0318)             /*              */

DATA-WORD (0A18)             /*              */

SPECIAL-3 (0CF8)             /*              */

DATA-WORD (0CF8)             /*              */

SPECIAL-3 (3A50)             /*              */

DATA-LABEL (3E18)            /* BRANCH TABLE */

------

DATA-LABEL (03F8)

DATA-WORD (0BF0)             /* LINK = (0BF0)   */
```

Let LINK be the DATA-WORD which is the right neighbor of DATA-LABEL (03F8); in this example LINK has the value (0BF0). SPECIAL-3 cells are cells specified by the programmer as having a USEKEY of 3; they are used here to hold branch addresses and are right neighbors to DATA-WORD's which contain the corresponding symbols that LINK may obtain.

The strategy is to isolate the branch table, to match interpretively symbol entries with the current value of LINK, to pick up the corresponding branch address and to place it as the operand of an UNCONDITIONAL-BRANCH instruction. The method thus includes a form of table lookup for which a content addressed memory is ideally suited.

MATCH and STORE activate and set the C flip-flop of the data label marking the right-hand end of the branch table. The second MATCH activates the data label marking the left-hand end of the branch table. MARK-RIGHT activates the cells in between. Note that the use of DATA-LABEL's and MARK-RIGHT permits rapid isolation of a group of cells of interest. INTERPRETIVE-MATCH now causes active DATA-WORD's to go to an intermediate logical state and to reset their M flip-flops in preparation for a match. RESETM resets the M flip-flops of DATA-LABEL and SPECIAL-3 cells. The system is now ready to perform the table lookup. MATCH-RIGHT activates the cell containing LINK and OUTPUT puts LINK onto the bus, which at the same time completes the interpretive match. DATA-WORD's which had been in intermediate states now return to State 0 and the one which contains (0BF0) sets its M flip-flop; the interpretive match has been accomplished. STORE (C=1) sets the C flip-flop of this cell. RIGHT activates the right neighbor. MATCH-RIGHT (30E8) activates the operand of the UNCONDITIONAL-BRANCH and TRANSFER causes this cell to go to an intermediate state and reset its M flip-flop in preparation for picking up the branch address. OUTPUT causes the active cell which contains the branch address to put it onto the bus; the operand of the unconditional branch stores this branch address and returns to State 0. RESETM and RESETC reset M and C flip-flops. UNCONDITIONAL-BRANCH now causes instruction execution to continue from PROGRAM-LABEL (0318).

The above six examples illustrate how the DLM computer can be programmed. There are certainly variations possible to these procedures according to the needs of the particular application.

IX. LOGICAL DESIGN OF CELL

This section will explore further the internal details of the cell. The logical design is not complete; however, enough is known to be able to accurately estimate the total number of gates required for a cell.

The NAND gate will be used as the basic building block to establish gate count. The NAND gate is widely used in current computers, including the central control of No. 1 ESS (Cagle et al., 1964) and the IBM 360 models (Davies et al., 1964). A possible circuit implementation is shown in Figure 6. The diode AND gate is followed by a transistor stage for inversion and amplification. Notice that all inputs must have sufficiently high, positive levels in order that the transistor may be driven to conduct and saturate, and thereby develop a low output level.

A NAND gate developed for large-scale integration might well be different from the circuit shown; however, it should have the same logical properties. That is, it should be permissible to tie outputs together and thereby derive an additional logic function. In the example of Figure 6, a six-gate circuit is reduced to a four-gate circuit by appropriate tying of outputs together. Besides there being fewer gates, there is also less switching delay.

A block diagram of the cell is shown in Figure 7. Sixteen memory word flip-flops (X_0 through X_{15}) are indicated together with M, C, Z and five state memory flip-flops (S_0 through S_4) of Sequence Control. As mentioned previously, the number of bits in the memory word can be adjusted to suit the application.

The Decoder performs checks and translation for Sequence Control. The Decoder remains inactive until the $\emptyset$ lead signals the presence of an op-code on the common bus. At this time the modifier field from the bus is compared with the stored USEKEY. If there is a match, indicating that the cell is to respond to the current instruction, the op-code on the bus is translated and passed to Sequence Control.

Sequence Control represents the 22-state sequential circuit controlling the cell. Sequence Control contains leads to control gating into and out of memory word bits, to control intercell signals and to control the M and C flip-flops.

Logic to perform gating into and out of typical memory word bits is shown in Figure 8. Eight gates per bit are required for this. Leads labeled Set1, Set2, Output, Match1 and Match2 come from Sequence Control and Mismatch is a return lead. Consider memory word bit X_4 for example. The flip-flop itself is realized in the usual way by connecting two NAND gates back-to-back. To gate information from the bus into bit X_4, the leads Set1 and Set2 are used. By convention a 1 signal on the common bus appears as a ground (low) and a 0 is a positive level (high). Separate leads Set1 and Set2 are needed to implement the long-form SET or STORE instructions with their "don't care" conditions. Set1 is active during transmission of the first operand and Set2 is active during the second operand. When the short-form SET or STORE instruction is used, Set1 and Set2 are activated simultaneously. Therefore, when Set1 or Set2 becomes positive, the X_4 bus lead is sampled and the X_4 flip-flop is set or reset appropriately. Conversely, if it is desired to gate information onto the common bus, Output is made high; the memory word flip-flop is sampled and the result passed to the common bus lead.

Separate leads Match1 and Match2 are needed to implement the long-form MATCH instruction with its "don't care" conditions. Match1 is active during transmission of the first operand and Match2 is active during the second operand. When the short-form MATCH instruction is used, Match1 and Match2 are activated simultaneously. Mismatch is the reply signal to Sequence Control indicating, when active, that at least one of the memory word bits in question has a mismatch.

Figure 9 shows the intercell lead arrangements. Consider the A lead first. An A signal propagates to the right or to the left but not in both directions simultaneously. The one-way transmission elements (two NAND gate circuits), inserted in the A lead, are conditioned by Sequence Control. Depending upon their contents, certain cells initiate A-signal propagation, other cells stop propagation and intermediate cells pass A signals with little delay. Indeed the transmission elements must be designed for minimum delay since it must be possible to propagate an A signal down the entire machine during one cycle time.

The M signal is passed to the cell on the left or the cell on the right but not in both directions simultaneously. The M lead is simply an extension of the Match lead that would ordinarily set the cell's own Match flip-flop and is under the control of Sequence Control.

The Z signal is passed only to the right for instruction sequencing. When a Z signal is received, a cell's Z flip-flop is set and Sequence Control is alerted. At the next clock cycle, the Z flip-flop is reset.

This completes the description of the cell's logic except for input-output interfaces which are covered in a later section. Sequence Control is to be a straightforward design of a 22-state, clocked sequential circuit although the details have not yet been completely worked out. A simple clock with two non-overlapping phases, carried on separate bus leads, is thought sufficient. Due to timing requirements in certain parts of the cell's logic, it is probably not convenient to limit each cell to but one clock lead. Furthermore, the need for another clock phase appears in the preferred scheme for bus amplification as described in the following section.

X. GROWTH AND BUS AMPLIFIERS

When cells are added to increase the size of the DLM computer, the common bus must necessarily be extended. The propagation delay for instructions on the common bus to reach all cells is therefore increased. The clock frequency must be correspondingly reduced. However, there is increased parallelism possible with more cells so that the effective computing rate may increase rather than decrease.

In the present design, the machine cycle is limited by the MARK-LEFT and MARK-RIGHT instructions. Recall that MARK-LEFT or MARK-RIGHT can cause the propagation of A signals from cell to cell throughout the entire machine during one machine cycle; the propagation of A-signals will take somewhat longer than the propagation of the instruction on the common bus. Consider the following worst case: All cells have their C flip-flops reset, the rightmost cell has its M flip-flop set and the leftmost cell in the machine issues a MARK-LEFT instruction. The instruction propagates to the right the full length of the machine; the rightmost cell responds by initiating A-signal propagation to the left, which continues the full length of the machine. This underlines the need to minimize delay in the A-signal gating in each cell.

There are several alternatives for preventing MARK instructions from restricting the machine speed.

1. Restrict the programmer to an A-signal propagation of at most 100 cells.
2. Dedicate two or three clock cycles to the MARK-LEFT and MARK-RIGHT instructions by the addition of states to state memory.
3. Permit the clock to monitor instructions on the common bus and automatically provide additional timing whenever a MARK instruction is seen.

As the DLM computer grows, provision must be made for bus amplifiers in the common bus leads. For example, at 100 cells the fanout capability of gates in the cell that drive common bus leads may be exceeded. One method which uses ordinary one-way amplifiers is shown in block diagram form in Figure 10 and in some detail in Figure 11. A common bus lead now becomes a pair of leads, one running in each direction. The design of the cell must be changed to accommodate this. Two terminals on the cell are used where one was required before. In addition, notice the increase from 8 to 10 gates per memory word bit for the gating previously described. When a cell drives a common bus pair, the same signal is sent in each direction. However, when a cell is reading from a common bus pair, an active signal (a low level by convention) on either lead of the pair is sufficient. Requiring additional terminals per cell is more significant than adding to the cell's logic. Reducing the number of terminals required per cell is a very practical goal for an integrated circuit implementation.

A solution is sought that uses one lead rather than a common bus pair. Consider the method using two-way amplifiers shown in Figures 12 and 13. The design of the cell and the common bus leads are unchanged; the two-way amplifiers are simply inserted into the bus leads as needed.

The requirement on the two-way amplifier is that it regenerate active signals coming from either direction. By convention, the active 1-signal on the bus is a low level and the 0-signal is a high. The amplifier is in a sense inactive when it sees high levels on both its left side and right side. Notice that the amplifier is indeed stable in this case, even regardless of the clock signal.

Assume that a low signal is now initiated from a cell on the left. If the clock signal is high, the amplifier transmits a low signal to its right as required. The low signal is also fed back to the left side of the amplifier; the feedback loop is designed to permit sensing when amplification is to take place from right to left. However, in the present case the effect is to bring the amplifier to a stable, active state in which the amplifier continues to transmit a low signal to the right even though the originating signal on the left disappears. The clock signal therefore is needed to periodically restore the amplifiers. If the clock signal goes low momentarily, the amplifiers are reset.

Gate-1 may be viewed as a detector which senses when amplification is to take place in either direction; Gate-2 and Gate-3 are amplifying gates. If a low signal is initiated by a cell on the right, the bus amplifier transmits a low signal to the left in similar fashion. Notice that the amplifier again achieves a stable, active state and must be reset by the clock signal later.

The use of clocked bus amplifiers must necessarily reduce the effective computing speed of the machine; the amplifiers must be reset during a dedicated phase of the basic machine cycle. However, the resulting savings in terminals required per cell is thought to be a good trade-off.

XI. INPUT AND OUTPUT ARRANGEMENTS

In keeping with the desire for a single cell design, all cells are provided with input-output lead terminals whether or not the terminals are ever to be connected to actual inputs or outputs. A design is sought which minimizes the number of such terminals. Solutions requiring only one lead per equipped memory word bit will be described.

Figure 14 shows three simple interface arrangements. Three types of applique circuits are available for location adjacent to the cell. A single lead from the 0-side of the memory word bit is extended to the applique circuit. The applique circuit is to provide buffering of logic levels and noise protection for the cell. The NAND gates shown are expected to be designed especially to these requirements.

The single lead is sufficient for the output applique. The X_3 output lead tracks the state of the X_3 flip-flop as it is changed during program execution. When X_3 is storing a 1, the X_3 output lead is high.

The provision of a single lead is somewhat restrictive in the case of input. There are cases where it is desirable for the state of the flip-flop to track the input lead. In the present circuit, the flip-flop can be set by the input lead but program action is required to reset the flip-flop. If the program is designed to deal with input cells at sufficiently frequent intervals, no input information is lost. When the X_3 input lead goes high, a 1 is to be stored in X_3; the 0-side of the X_3 flip-flop is driven low, causing it to be set. An appropriate SET or STORE instruction is then sufficient to reset X_3. Notice that the input cell in fact buffers a momentary input signal for later processing; for some applications such buffering is beneficial.

The combination input-output applique provides for sharing the single lead for both input and output. There must again be coordination between the program and the input signals.

Two-way buffer circuits, similar to the bus amplifiers previously described, can be used as interfaces. These should provide better noise protection as compared with extending leads directly from the memory word flip-flops. Figure 15 shows a typical arrangement. Filters on the output leads are needed to eliminate spurious signals caused by the two-way buffer circuits being periodically reset. Input buffers match logic levels for input signals; similarly, the last stage of the filters match logic levels for output signals.

On input there must still be coordination between program actions and input signals. Observe how the X_3 input lead going high causes the X_3 flip-flop to be set: The two-way buffer associated with X_3 receives a low on its right side. In turn it transmits a low to its left side, which is attached to the 0-side of X_3 flip-flop. This low signal causes X_3 to be set. The X_3 output lead goes high and remains so as long as X_3 remains set. This return signal can be used as a form of check that the input signal was received.

The stylized waveforms of Figure 15 illustrate how the X_3 output signal is developed when the X_3 flip-flop changes from the reset state to the set state. The change of state may be due either to an input signal, as described above, or to a program action. When X_3 is set, its 0-side goes low. The low in turn is transmitted by the two-way buffer to the filter, together with extraneous, inverted clock pulses. The clock-like pulses are developed at times when the two-way buffer is being periodically reset. The filter is simply a flip-flop and gating. The filter flip-flop retains its previous state during the clock pulse and therefore the extraneous clock pulses are removed from the output signal.

XII. ESTIMATING THE NUMBER OF GATES PER CELL

The number of gates per cell is a measure of the cell's complexity. In this section, estimated gate counts are given for various lengths of the cell's memory word.

There are practical constraints on the physical realization of a cell as an integrated circuit. The number of permissible external connections to the chip is a limit not easily extended. In the present cell design, reducing the number of required terminals has been a primary consideration. The gate count is related to the area required for the chip. Chip area is also required for leads interconnecting the gates. The cell is obviously not a planar circuit so that a certain number of crossovers of interconnecting leads on the chip are essential. Reducing the number of such crossovers is a secondary design consideration not yet attempted.

Consider now the terminal requirements for the cell under study. Figure 16 shows a listing of the 33 terminals required, assuming eight input-output lead terminals; that is, eight bits of information could be simultaneously put into the cell or read out of the cell. Notice that if two cells could be realized on one chip, the terminal requirements would only be increased by another set of eight input-output lead terminals; the three intermediate, intercell leads would be internal to the chip, and the bus leads are common to both cells.

Figure 17 is a set of curves showing the number of terminals per cell (one cell per chip) as a function of the number of memory word bits and the number of input-output leads. The cell with 16 memory word bits and eight input-output leads is shown to require 33 terminals.

In an earlier section an extensive instruction set for the DLM computer was proposed. It is obvious that the effect of certain of the instructions can be realized by a sequence of other instructions present. The inclusion of the additional instructions permitted a reduction in program storage and execution time. It is well to consider if a significant number of gates per cell would be saved if a reduced instruction set were used. Two reduced instruction sets were postulated, as shown in Figure 18. The "medium" instruction set is generated by eliminating INTERPRETIVE-MATCH, SET-SPECIAL and five of the short-form instructions; the equivalent long-form instructions are sufficient. The "small" instruction set represents a near minimal repertoire. Noticeable by their absence are the two types of branching instructions. Their loss is not quite as restrictive in a DLM computer as compared to a conventionally organized machine because, with parallel processing, looping is frequently not required. However the lack of branching instructions does limit the applicability of the machine.

A breakdown of estimated gate counts for the three instruction sets is shown in Figure 19. Reducing the instruction set produces savings in Sequence Control and Decoder. All items except Sequence Control are combinatorial logic so that gate counts are directly estimable. The number of eight gates per memory word bit, derived previously, was used for X_3 through X_{11}.

Sequence Control was essentially designed for each of the three cases with the aid of a set of computer programs developed by P. M. Sherman (1964) and others. The results consist of a listing of interconnected NAND gates so that gate count is established. The computer programs do an extensive amount of searching for near-minimal solutions to the sequential circuit realizations. Certain simplifying assumptions were required in Sequence Control in order to adapt it to the capabilities of the design programs; to compensate for possible omissions, gate counts for Sequence Control were increased by 10%.

A range of 235 gates to 295 gates, depending upon instruction set, is found for a cell with a 16-bit memory word. Estimated gate counts for a range of word sizes are shown in Figure 20. It should be noted that the proportion of logic saved per cell by using a reduced instruction set is not great. Also a greater number of cells would normally be required. Therefore, the full instruction set should probably be used.

XIII. HOMOGENEITY

A computer has been described which is built out of identical cells. There are no external registers or adders; the logic and memory are distributed throughout the system. Cells used to store instructions are identical with cells used to store data or with cells having input-output applique circuits attached.

Consider the following three categories of cells: (a) Program cells, (b) Data cells and (c) Input-Output cells; and consider whether savings could be effected by developing three types of cells. All cells must have a certain logical consistency in order to work together properly. However, program and data cells do not need input-output lead terminals and data cells do not need Z-signal intercell leads and associated internal gating. Some simplification is therefore possible in the Sequence Control of data cells. Notice, however, that once program cells and data cells are made different, data cells can no longer be changed to program cells, and vice versa, by program instructions, as can be done presently. Moreover, the possible savings in logic are probably less than 10% and therefore not significant.

There is more cause to make input-output cells different. Certain applications may require eight or sixteen input-output lead terminals whereas others only require that one memory word bit be equipped. In addition the arrangement of the input-output interface hardware is variable.

For the present, a general-purpose cell is proposed for all three categories of cells. However, it is recognized that special input-output cells may be sometimes justified.

XIV. DATA MULTIPLEXER AS A PRACTICAL APPLICATION

This section describes in some detail a practical application for the DLM computer structure. An inexpensive data multiplexing system is desired. Specifically the information carried on 80 teletypewriter lines is to be multiplexed onto a single, high-speed data channel of about 9000 bit/second capacity. The present proposal covers only the transmitting end which receives information from the various teletypewriter lines and places it in proper time sequence on the high-speed data channel; the equipment at the receiving end is undoubtedly similar but is not discussed here. It will be seen that parallelism inherent in the DLM structure is exploited to simultaneously assemble the various teletypewriter characters.

Current teletypewriters use an 11-element character, transmitted serially in approximately 100 milliseconds. Refer to Figure 21, which shows stylized, typical waveforms. A teletypewriter character contains eight information bits and three bits (START, STOP-1 and STOP-2) for synchronization. A teletypewriter line, when idle, transmits the 1-level. The START bit is always 0 and establishes character timing. There may be any length idle periods between characters; the START bit effects resynchronization. Note from Figure 21 that signals on the various lines are not in synchronism, which complicates the problem. It is essential that the beginning of a START bit be recognized promptly in order that the following bits may be detected properly in view of the permissible signal distortion on a teletypewriter line. Therefore, it is proposed that every teletypewriter line be sampled about every 70 microseconds; in the worst case less than 1% of a START bit would have passed before detection.

The solution to be described is straightforward to explain but perhaps not the most economical to implement. No hardware interrupt capability is assumed to aid in scheduling tasks. A 31-bit cell is used although clearly a shorter memory word is feasible if the number of cells is increased. A hardware shift register is used at the interface to the high-speed channel equipment to eliminate a timing requirement.

The program itself is essentially straight line code which is to be run every 71 microseconds. A basic machine cycle time of about 0. 25 microseconds is required. With this programming strategy, very little work is accomplished during any particular run of the program; however, if work to be done is found, it is completed on time.

Figure 22 shows a layout of the cellular structure. One group of cells is devoted to program storage. Eighty line cells have access to the individual teletypewriter lines via simple input appliques. The input appliques act to buffer logic levels and provide noise protection. The output buffer cell connects with eight leads to the hardware shift register. The shift register is arranged to signal a cell when a new teletypewriter character may be placed in the output buffer cell; this removes yet another timing requirement from the program. It is clear that hardware-software tradeoffs proposed here would have to be re-examined in an actual implementation.

A single clock with countdown arrangements is sufficient. A Z signal is supplied to the entry point of the program every 71 microseconds. The program runs to completion and the cellular structure is idle until the program is reinitiated.

The high-speed data channel is designed to run at a slightly faster rate than the composite signaling rate of the 80 teletypewriter lines. A transmission frame consists of a unique framing character followed by one teletypewriter character from each of the 80 lines, transmitted serially in sequence. If a teletypewriter line has not supplied a complete character at the time that its time slot occurs, a unique idle character is automatically inserted. Briefly, the job of the DLM computer in this application is to sample lines and assemble the teletypewriter characters and to place the characters in proper time sequence in the output buffer cell.

Consider now the layout of the 31-bit line cell as shown in Figure 23. The lead from the input applique is connected to flip-flop X_3 (present look). Therefore, X_3 represents the current state of the teletypewriter line. X_4 (last look) is a record of the state of the line at the last time the program was run. To protect against noise, two successive, identical samples must be seen before a change of state on the line is accepted. X_5 (corrected last look) is updated whenever the present look and last look agree.

Bits X_{25} and X_{26} are used to establish four separate modes:

Mode	X_{26}	X_{25}	
0	0	0	Idle
1	0	1	Receiving START Bit
2	1	0	Assembling Character
3	1	1	Character Complete

Bits X_6 through X_{12} are used as a 7-bit counter. In nonidle line cells, the counter is incremented each time the program is run, i.e., every 71 microseconds, and thereby provides timing for sampling the individual bits of the teletypewriter character, Note that

$$71\ \mu\text{sec} \times 2^7 = 9.09\ \text{msec}$$

which is the nominal length of each bit of the teletypewriter character.

Bits X_{13} through X_{16} are used as a 4-bit counter to establish which bit of the teletypewriter character is currently being sampled. Bits X_{17} through X_{24} are set aside to store the individual bits and thereby assemble the teletypewriter character.

Finally, bits X_{27} and X_{28} are assigned as flags to aid processing. Bits X_{29} and X_{30} are used in program cells to specify conditions on Match and Control flip-flops and therefore are not used in data cells. The way in which bits in the line cell are processed should become clear as the program is described in the following section.

Description of Data Multiplexer Program

The data multiplexer program is flowcharted in Figure 24. The bulk of the program is straight-line code. The three indicated decision points deal with the transfer of characters from line cells to the output buffer cell. The flowchart should aid in understanding the program itself, which is given in its entirety in the Appendix.

The hardward shift register can signal that it is ready for a new character anytime during the last bit of the previous character. It will be shown that the output buffer cell can supply a new character within 71 microseconds after such a signal.

The several sections of the data multiplexer program, listed in the Appendix, will now be described in sequence:

a) Update Corrected Last Look Bit. The instruction SET 0 (X_3=0) is needed to bring the present look bit (X_3) of each line cell up to date. Recall that with the simplified, one lead per bit, input arrangement, memory bits must be reset by program control. The corrected last look bit (X_5) is then made to agree with present look and last look bit in all cells where they have the same value. The corrected last look therefore represents a smoothed value of the input signal. Finally the last look bit (X_4) becomes the current present look.

b) Detect START Bit. When the mode of a line cell is 0 and its corrected last look becomes 0, the beginning of a START bit has been recognized; the mode is set to 1 and the timer, bit counter and character bits are reset.

c) Sample START Bit at Midpoint. While the mode of a line cell is 1 or 2, its timer is incremented by one each time the data multiplexer program is run, i.e., every 71 microseconds. When the count reaches 64 while mode=1, the state of the corrected last look bit is tested. If CLL=0, a valid START bit is assumed; mode is set to 2. On the other hand, if CLL=1, a false start is recognized and mode is reset to 0.

d) Increment Timers in Nonidle Cells. All line cells with mode of 1 or 2 are set with a temporary flag (X_{27}). The instructions that follow implement the incrementing by one of the 7-bit timer in each flagged cell. The method used is the same as that for a previous programming example. Note that when a timer which has reached its maximum count is incremented, the count is reset to 0.

3) Assemble Character Bit. Note that timers continue to advance while a line cell has a mode of 2. Thus when the count reaches 64 a new bit may be sampled and stored. Such cells are temporarily flagged (by C and X_{27}). Their bit counters are incremented by one.

Such cells with a bit count of 9 have previously stored the eight bits of the character and are currently at the STOP-1 bit; their mode is set to 3 to indicate character complete. In the other eight cases, the mode is not changed and the sampled bit is stored according to the current bit counter value.

f) Test if Shift Register Has Signaled Ready. The shift register signal lead is connected to bit X_3 of a sensor cell in the data area; a unique USEKEY of 7 is assigned to this cell for convenience. Recall that with the simple input arrangement, a momentary signal from the shift register is sufficient to set X_3. The conditional branch transfers control to Label-4 if $X_3 = 1$; if $X_3 = 0$, the program ends because Z-signal propagation is stopped by the dummy DATA-LABEL.

g) Advance Floating Flag. Beginning at Label-4, bit X_3 of the sensor cell is reset so that another signal may be received. Bit X_{28} of every line cell is dedicated to the floating flag procedure. The process is analogous to a ring counter; only one cell has $X_{28}=1$ at any time. The flag advances from left to right, one line cell at a time, as the shift register signals that it is ready for a new character. Floating flag has been found to be an efficient means to keep track of the current time slot during each transmission frame. A cell with unique USEKEY of 2 has been placed at the right of the last line cell. This permits determination of whether the floating flag has reached the far right-hand line cell and must next be restarted at the left.

The conditional branch causes program control to branch to Label-3 if the floating flag must be restarted; if not, the program continues.

h) Test Flagged Line Cell for Character Complete. A TRANSFER sequence is used to move information from the flagged cell to the output buffer cell. The output buffer cell is given a unique USE-KEY of 3 for convenience. If the flagged cell does not have a complete character, note that the MATCH instruction will not succeed in matching. Therefore, during the null operand of the OUTPUT instruction, no signal will appear on the bus and the information stored in the output buffer cell will be all 0's. A test is made for the all 0's case, and if found, an idle character of seven 1's and one 0 is placed in the output buffer cell. The program ends again at a dummy DATA-LABEL.

i) Restart Floating Flag at the Left. Beginning at Label-3, the floating flag is stored in a spare DATA-WORD cell at the left of the first line cell. Then a framing character is placed in the output buffer cell. The program ends because Z-signal propagation is stopped by the Label-2 DATA-LABEL.

Observations on Data Multiplexer Program

Recall that only two CONDITIONAL-BRANCH1 instructions appear in the program whereas three decision points are clearly indicated on the flowchart. This illustrates again that decision-making is built into some of the instructions so that conditional branching is not always required. Indeed whether a particular cell responds to an instruction or not is a decision.

Some 116 instructions were required for the data multiplexer program. Figure 25 summarizes some of the statistics for the program. It is clear that the sample is too small to be able to reach many general conclusions, e. g., on typical instruction mix. However, for the particular program, the bulk of the instructions used were of the three-cell type. MATCH, STORE and RESETM were the most often used instructions and this will probably be generally true with such a content-addressed structure. There was little opportunity to use the short-form instructions. The powerful instructions MARK-LEFT and MARK-RIGHT were not used at all.

A 31-bit memory word cell is convenient as a line cell but certainly wasteful as a program cell; the additional bits are not put to use. For comparison, the entire data multiplexer program was rewritten with a 16-bit cell restriction and the results are summarized here. It is clear that the data area will require more cells and the program size will increase somewhat to deal with the new cells. Specifically, two auxiliary cells were associated with each line cell, one placed on its left and the other on its right. This essentially triples the number of cells required in the data area. However, the number of instructions only increases to 141, requiring 324 cells. The total system consists of 570 cells, each having a memory word of 16 bits. The maximum number of machine cycles per program run is only increased to 321. It is clear that more must be known about the cost of manufacturing cells in quantity before one or the other system can be preferred. It is certainly feasible to design programs around the shorter memory word cell however.

The use of the floating flag provides a potential dividend. If technology improves so that more teletypewriter lines can be multiplexed onto a higher speed data channel, additional line cells can be inserted into the structure without reprogramming. The floating flag circulates through the old line cells and through the new line cells to keep track of time slots. Of course, the hardware shift register timing must be increased appropriately.

XV. FUTURE WORK

It is felt that the DLM structure may find application in telephone switching systems as well as data transmission terminals. For example, the task of autonomous line scanning and dialed digit collection is a candidate. The task of path search for the switching network is another. Some work has already been done on the latter (Crane, 1968).

Testing the feasibility of the DLM structure programmed as an autonomous line scanner is yet to be done. However, possible similarities in programming strategies for line scanner and data multiplexer are already evident.

XVI. DLM COMPUTER SIMULATOR

A simulator for the DLM computer was written in FORTRAN IV and Assembler language for an IBM 360/30, which has about 44, 000 bytes of core storage available to the user. The simulator permits a check on the system design by simulating individual DLM computer instructions and furthermore is a debugging aid for such programs. Simulations are presently limited to 2000 cells. The simulator can trace individual cell contents, common bus signals and intercell lead signals after each machine cycle, if desired. Therefore, it is easy to see the response of cells to the individual instructions of a DLM computer program being executed. The example programs described previously (including the data multiplexer program) were run on the simulator, as were approximately 35 other short programs.

XVII. SUMMARY

The advent of large-scale integration has caused renewed interest in new computer organizations. The straightforward partitioning of a conventionally organized, central processing unit into blocks of logic results in a number of different, integrated circuit designs and perhaps also a large number of terminals per circuit. One goal of the present study was a computer organization utilizing only one integrated circuit design, requiring a suitable number of terminals.

Furthermore, a small but growable computer was sought to compete with special-purpose, wired-logic controllers. The usual functions performed by static registers, shift registers and counters are programmable in the DLM computer. It is felt that the DLM computer will prove to be less expensive to manufacture and program than would a number of unique wired-logic controllers, where several control applications are anticipated.

The economy of large-scale integration is certainly required for feasibility. The total number of gates in a DLM computer is large. Practicality must probably wait for a manufacturing cost per gate of a few cents. But such costs are predicted in connection with large demand, large-scale integrated circuits (Petritz, 1967).

Finally, not all computer organizations involving large-scale integration have yet been investigated. The present study covered a particular structure, leading to a family of computers. Certainly quite different structures are possible. However, content addressing seems to be naturally suited to a modular structure.

BIBLIOGRAPHY

Boysel, Lee L. 1968. Adder on a chip: LSI helps reduce cost of small machine. Electronics 41(6):119-124.

Cagle, W. B. et al. 1964. No. 1 ESS logic circuits and their application to the design of the central control. Bell System Technical Journal 43:2055-2095.

Cottrill, J. L. 1968. Compiler for distributed logic memory computer. Columbus, Ohio, Bell Telephone Laboratories. 6 p. (Unpublished Programmer's Notes)

Crane, B. A. 1968. Path finding with associative memory. The Institute of Electrical and Electronics Engineers, Transactions on Computers.C-17:691-693.

Crane, B. A. and J. A. Githens. 1965. Bulk processing in distributed logic memory. The Institute of Electrical and Electronics Engineers, Transactions on Electronic Computers EC-14:186-196.

Davis, E. M. et al. 1964. Solid logic technology: versatile, high-performance microelectronics. IBM Journal of Research and Development 8:102-114.

Gaines, R. S. and C. Y. Lee. 1965. An improved cell memory. The Institute of Electrical and Electronics Engineers, Transactions on Electronic Computers EC-14:72-75.

Lee, C. Y. 1962. Intercommunicating cells, basis for a distributed logic computer. American Federation of Information Processing Societies, Proceedings of the Fall Joint Computer Conference 22:130-136.

Lee, C. Y. and M. C. Paull. 1963. A content addressable distributed logic memory with applications to information retrieval. The Institute of Electrical and Electronics Engineers Proceedings, 51:924-932.

Levy, Saul Y. et al. 1967. System utilization of large-scale integration. The Institute of Electrical and Electronics Engineers, Transactions on Electronic Computers EC-16:562-566.

Petritz, Richard L. 1967. Current status of large-scale integration technology. The Institute of Electrical and Electronics Engineers, Journal of Solid-State Circuits SC-2:130-147.

Sherman, P.M. 1964. A logic-design programming package for the 7090-7094 computers. Murray Hill, New Jersey, Bell Telephone Laboratories. 23 p. (Unpublished Technical Memorandum MM-64-6263-5)

Sturman, Joel N. 1968a. Asynchronous operation of an iteratively structured general-purpose digital computer. The Institute of Electrical and Electronics Engineers, Transactions on Computers C-17:10-17.

__________ 1968b. An iteratively structured general-purpose digital computer. The Institute of Electrical and Electronics Engineering, Transactions on Computers C-17:2-9.

APPENDIX

Data Multiplexer Program Instructions and Data Areas

```
/*  UPDATE CORRECTED LAST LOOK BIT IN ALL LINE CELLS  */
      SET      0  (X3=0)          /*  UPDATE PRESENT LOOK IN ALL LINE CELLS  */
      MATCH    0  (X4=0,X3=0)     /*  LAST LOOK=0, PRESENT LOOK=0  */
      STORE    0  (X5=0)          /*  CORRECTED LAST LOOK → 0  */
      RESETM   0
      MATCH    0  (X4=1,X3=1)     /*  LAST LOOK=1, PRESENT LOOK=1  */
      STORE    0  (X5=1)          /*  CORRECTED LAST LOOK → 1  */
      RESETM   0
      MATCH    0  (X3=1)          /*  PRESENT LOOK=1  */
      SET      0  (X4=0)
      STORE    0  (X4=1)          /*  UPDATE LAST LOOK  */
      RESETM   0

/*  DETECT START BIT  */
      MATCH    0  (X5=0,X26=0,X25=0)                                                  /*  CORRECTED LAST LOOK=0, MODE=0, I.E., START BIT BEGINNING  */
      STORE    0  (X25=1,X12=0,X11=0,X10=0,X9=0,X8=0,X7=0,X6=0,X16=0,X15=0,X14=0,X13=0,
                   X17=0,X18=0,X19=0,X20=0,X21=0,X22=0,X23=0,X24=0)                   /*  MODE → 1  RESET TIMER, BIT COUNTER, CHARACTER BITS  */
      RESETM   0

/*  SAMPLE START BIT  AT MIDPOINT AND RESPOND  */
      MATCH    0  (X5=0,X26=0,X25=1,X12=1,X11=0,X10=0,X9=0,X8=0,X7=0,X6=0)    /*  CLL=0, MODE=1, TIMER=64  */
      STORE    0  (X26=1,X25=0)  /*  MODE → 2, BEGIN TO ASSEMBLE CHARACTER      */
      RESETM   0
      MATCH    0  (X5=1,X26=0,X25=1,X12=1,X11=0,X10=0,X9=0,X8=0,X7=0,X6=0)    /*  CLL=1, MODE=1, TIMER=64  */
      STORE    0  (X26=0,X25=0)  /*  MODE → 0, FALSE START SO RETURN TO IDLE    */
      RESETM   0
/*  INCREMENT TIMERS IN NONIDLE CELLS  */
      MATCH    0  (X26=0,X25=1)
      MATCH    0  (X26=1,X25=0)                     /*  SELECT NONIDLE CELLS, MODE=1,2  */
      STORE    0  (X27=1)                           /*  SET TEMPORARY FLAG DURING INCREMENTING  */
      RESETM   0
      MATCH    0  (X27=1,X6=0)
      STORE    0  (X27=0,X6=1)                      /*  ------0 → ------1  */
      RESETM   0
      MATCH    0  (X27=1,X7=0,X6=1)
      STORE    0  (X27=0,X7=1,X6=0)                 /*  -----01 → -----10  */
      RESETM   0
      MATCH    0  (X27=1,X8=0,X7=1,X6=1)
      STORE    0  (X27=0,X8=1,X7=0,X6=0)            /*  ----011 → ----100  */
      RESETM   0
      MATCH    0  (X27=1,X9=0,X8=1,X7=1,X6=1)
      STORE    0  (X27=0,X9=1,X8=0,X7=0,X6=0)       /*  ---0111 → ---1000  */
      RESETM   0
```

```
        MATCH   0  (X27=1,X10=0,X9=1,X8=1,X7=1,X6=1)
        STORE   0  (X27=0,X10=1,X9=0,X8=0,X7=0,X6=0)                  /*  --01111 → --10000  */
        RESETM  0
        MATCH   0  (X27=1,X11=0,X10=1,X9=1,X8=1,X7=1,X6=1)
        STORE   0  (X27=0,X11=1,X10=0,X9=0,X8=0,X7=0,X6=0)            /*  -011111 → -100000  */
        RESETM  0
        MATCH   0  (X27=1,X12=0,X11=1,X10=1,X9=1,X8=1,X7=1,X6=1)
        STORE   0  (X27=0,X12=1,X11=0,X10=0,X9=0,X8=0,X7=0,X6=0)      /*  0111111 → 1000000  */
        RESETM  0
        MATCH   0  (X27=1,X12=1,X11=1,X10=1,X9=1,X8=1,X7=1,X6=1)
        STORE   0  (X27=0,X12=0,X11=0,X10=0,X9=0,X8=0,X7=0,X6=0)      /*  1111111 → 0000000  */
        RESETM  0

/*  ASSEMBLE CHARACTER BIT IN ALL LINE CELLS DUE TO BE SAMPLED  */
/*  INCREMENT BIT COUNTER AS FIRST STEP  */
        MATCH   0  (X26=1,X25=0,X12=1,X11=0,X10=0,X9=0,X8=0,X7=0,X6=0)  /*  MODE=2, TIMER=64  */
        STORE   0  (C=1,X27=1)                                           /*  SET TEMPORARY FLAGS  */
        RESETM  0
        MATCH   0  (X27=1,X13=0)
        STORE   0  (X27=0,X13=1)                                         /*  ---0 → ---1  */
        RESETM  0
        MATCH   0  (X27=1,X14=0,X13=1)
        STORE   0  (X27=0,X14=1,X13=0)                                   /*  --01 → --10  */
        RESETM  0
        MATCH   0  (X27=1,X15=0,X14=1,X13=1)
        STORE   0  (X27=0,X15=1,X14=0,X13=0)                             /*  -011 → -100  */
        RESETM  0
        MATCH   0  (X27=1,X16=0,X15=1,X14=1,X13=1)
        STORE   0  (X27=0,X16=1,X15=0,X14=0,X13=0)                       /*  0111 → 1000  */
        RESETM  0

/*  STORE INDIVIDUAL CHARACTER BITS  */
        MATCH   0  (C=1,X16=1,X15=0,X14=0,X13=1,X5=0,X17=0,X18=0,
                    X19=0,X20=0,X21=0,X22=0,X23=0,X24=0)                 /*  BIT COUNT=9, CLL=0, ALL SPACE CHARACTER  */
        STORE   0  (C=0,X25=1,X24=1)                                     /*  UNIQUE CODE FOR ALL SPACE CHARACTER, MODE → 3,  CHARACTER COMPLETE  */
        MATCH   0  (C=1,X16=1,X15=0,X14=0,X13=1)                         /*  BIT COUNT=9  */
        STORE   0  (X25=1)                                               /*  MODE → 3,  CHARACTER COMPLETE  */
        RESETM  0
        MATCH   0  (C=1,X16=1,X15=0,X14=0,X13=0,X 5 =1)                  /*  BIT COUNT=8, CLL=1  */
        STORE   0  (X24=1)                                               /*  BIT 8=1  */
        RESETM  0
        MATCH   0  (C=1,X16=0,X15=1,X14=1,X13=1,X 5 =1)                  /*  BIT COUNT=7, CLL=1  */
        STORE   0  (X23=1)                                               /*  BIT 7=1  */
        RESETM  0
        MATCH   0  (C=1,X16=0,X15=1,X14=1,X13=0,X 5 =1)                  /*  BIT COUNT=6, CLL=1  */
        STORE   0  (X22=1)                                               /*  BIT 6=1  */
        RESETM  0
        MATCH   0  (C=1,X16=0,X15=1,X14=0,X13=1,X 5 =1)                  /*  BIT COUNT=5,CLL=1  */
        STORE   0  (X21=1)                                               /*  BIT 5=1  */
        RESETM  0
        MATCH   0  (C=1,X16=0,X15=1,X14=0,X13=0,X 5 =1)                  /*  BIT COUNT=4, CLL=1  */
        STORE   0  (X20=1)                                               /*  BIT 4=1  */
        RESETM  0
```

```
    MATCH    0  (C=1,X16=0,X15=0,X14=1,X13=1,X5=1)       /*  BIT COUNT=3, CLL=1  */
    STORE    0  (X19=1)                                   /*  BIT 3=1  */
    RESETM   0
    MATCH    0  (C=1,X16=0,X15=0,X14=1,X13=0,X5=1)       /*  BIT COUNT=2, CLL=1  */
    STORE    0  (X18=1)                                   /*  BIT 2=1  */
    RESETM   0
    MATCH    0  (C=1,X16=0,X15=0,X14=0,X13=1,X5=1)       /*  BIT COUNT=1, CLL=1  */
    STORE    0  (X17=1)                                   /*  BIT 1=1  */
    RESETM   0
    RESETC   0

/*  TEST IF SHIFT REGISTER HAS SIGNALED THAT IT IS READY FOR A NEW CHARACTER  */
    MATCH              7    (0001 0008)           /*  TEST SHIFT REGISTER SENSOR FOR X3=1  */
    CONDITIONAL-BRANCH1  4,5,7                    /*  IF MATCH, GO TO LABEL-4  */
    DATA-LABEL              (0000 0000)           /*  DUMMY CELL USED TO STOP Z-SIGNAL PROPAGATION  */

    PROGRAM-LABEL           (0000 0000)           /*  LABEL=1  */
    CONDITIONAL-BRANCH2  5                        /*  PROGRAM LEG IF NO MATCH AT CONDITIONAL BRANCH  */

/*  ADVANCE FLOATING FLAG  */
    PROGRAM-LABEL           (0001 0008)           /*  LABEL-4  */
    SET                7    (X3=0)                /*  RESET X3 FLIP-FLOP OF SPECIAL-7  */
    MATCH-RIGHT        0    (X28=1)               /*  SELECT CELL TO RIGHT OF FLAGGED CELL  */
    SET                0,2  (X28=0)               /*  CLEAR FLOATING FLAG  */
    STORE              0,2  (X28=1)               /*  STORE FLOATING FLAG  */
    RESETM             0,2

/*  TEST IF FLOATING FLAG HAS ADVANCED BEYOND RIGHTMOST LINE CELL  */
    MATCH              2    (1000 1000)           /*  X28=1  */
    STORE              2    (0000 1000)
    CONDITIONAL-BRANCH1  2,4,5                    /*  IF MATCH, GO TO LABEL-3  */

/*  TEST FLAGGED LINE CELL FOR CHARACTER COMPLETE  */
    SET                3    (4000 0000)                                    /*  RESET OUTPUT BUFFER CELL BUT SET M FLIP-FLOP  */
    TRANSFER           3
    MATCH              0    (X28=1,X26=1,X25=1)                            /*  SELECT FLAGGED CELL WITH CHARACTER COMPLETE  */
    STORE              0    (X26=0,X25=0)                                  /*  MODE → 0, IDLE  */
    OUTPUT             0,3                                                 /*  COMPLETE TRANSFER SEQUENCE  */
    MATCH              3    (X28=0)                                        /*  TEST FOR NO INFORMATION TRANSFERRED  */
    STORE              3    (X17=1,X18=1,X19=1,X20=1,X21=1,X22=1,X23=1,X24=0)  /*  STORE IDLE CHARACTER  */
    RESETM             3
    DATA-LABEL              (0000 0000)                                    /*  DUMMY CELL USED TO STOP Z-SIGNAL PROPAGATION  */

/*  RESTART FLOATING FLAG AT THE LEFT OF LINE CELLS  */
    PROGRAM-LABEL           (0000 1000)           /*  LABEL-3  */
    MATCH-RIGHT        1    (0000 0100)           /*  SELECT CELL IMMEDIATELY TO LEFT OF LINE CELL 1  */
    STORE              0    (1000 0000)           /*  STORE FLOATING FLAG, X28=1  */
    RESETM             0
    SET                3    (00AA 0000)           /*  STORE FRAMING CHARACTER IN OUTPUT BUFFER CELL, FRAMING CHARACTER=10101010  */
```

```
/*  DATA AREA  */
      DATA-LABEL      (0000 0100)       /*  LABEL-2  */
      DATA-WORD       (1000 0000)       /*  CELL USED WHEN FLOATING FLAG RESTARTED, INITIALLY X28=1  */
      DATA-WORD       (0000 0000)       /*  LINE CELL 1, X3  CONNECTED TO INPUT APPLIQUE  */
      DATA-WORD       (0000 0000)       /*  LINE CELL 2, X3  CONNECTED TO INPUT APPLIQUE  */
          '
          '
          '
      DATA-WORD       (0000 0000)       /*  LINE CELL 80, X3 CONNECTED TO INPUT APPLIQUE  */
      SPECIAL-2       (0000 1000)       /*  USEKEY=2, SENSOR FOR FLOATING FLAG NEEDED TO BE RESTARTED  */
      SPECIAL-7       (0001 0000)       /*  USEKEY=7, SENSOR FOR SHIFT REGISTER READY, X3  CONNECTED TO SHIFT REGISTER  */
      SPECIAL-3       (0000 0000)       /*  USEKEY=3, OUTPUT BUFFER CELL, X17 THROUGH X24 CONNECTED TO SHIFT REGISTER  */
```

Note: An eight-digit hexadecimal number, such as (00AA 0000) is used as an alternate method of specifying the bit pattern for the 31-bit cell. This number is expanded below to show its correspondence with the desired bit pattern:

```
Hexadecimal
Number                                        Desired Bit Pattern

00AA 0000   | 0   0   0   0 | 0   0   0   0 | 1   0   1   0 | 1   0   1   0 | 0   0   0   0 | 0   0   0   0 | 0   0   0   0 | 0   0   0   0 |
                  M   C   X28 X27 X26 X25 X24 X23 X22 X21 X20 X19 X18 X17 X16 X15 X14 X13 X12 X11 X10 X9  X8  X7  X6  X5  X4  X3  X2  X1  X0
```

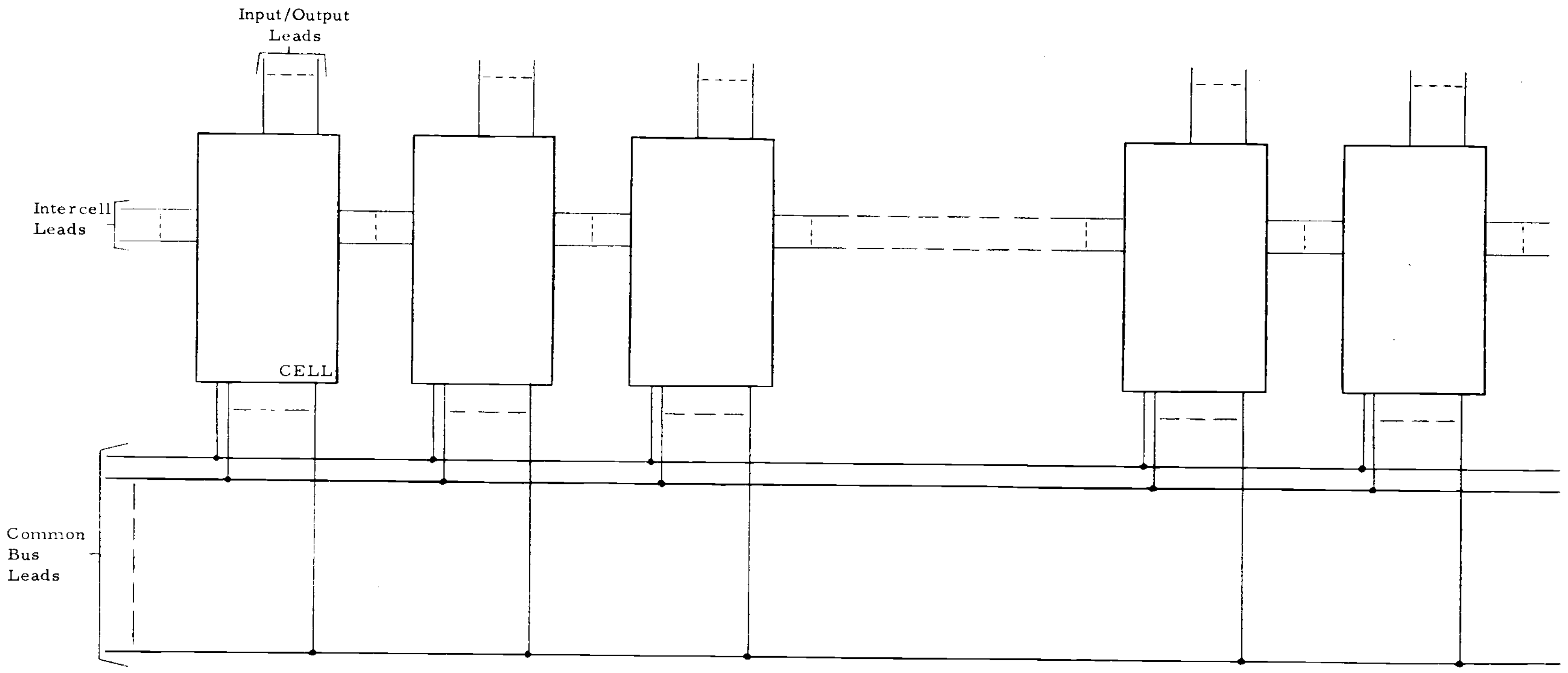


Figure 1. Cell interconnections for DLM computer.

USEKEY	Bits of Memory Word $X_2 X_1 X_0$	Use of Cell
0	0 0 0	Data Word
1	0 0 1	Data Label
2	0 1 0	(Available to Programmer)
3	0 1 1	(Available to Programmer)
4	1 0 0	Program Label
5	1 0 1	Program Operand
6	1 1 0	Program Op-Code
7	1 1 1	(Available to Programmer)

Figure 2. Assignment of USEKEY's.

TRANSFER	(6, 1)
LEFT.	(6, 2)
RIGHT	(6, 3)
INTERPRETIVE-MATCH	(6, 4)
MATCH - Long Form	(6, 5)(5, I)(5, J)
UNCONDITIONAL-BRANCH.	(6, 6)(5, I)
RESETM.	(6, 7)
STORE - Long Form	(6, 8)(5, I)(5, J)
SET - Long Form	(6, 9)(5, I)(5, J)
MATCH-LEFT.	(6, 10)(5, I)(5, J)
MATCH-RIGHT - Long Form.	(6, 11)(5, I)(5, J)
MARK-LEFT	(6, 12)
MARK-RIGHT	(6, 13)
OUTPUT	(6, 14)(5, 0)
CONDITIONAL-BRANCH2.	(6, 15)
CONDITIONAL-BRANCH1	(6, 16)(5, 0)
RESETC	(6, 17)
MATCH - Short Form.	(6, 18)(5, I)
MATCH-RIGHT - Short Form.	(6, 19)(5, I)
STORE-SPECIAL - Long Form.	(6, 20)(5, I)(5, J)
STORE - Short Form	(6, 21)(5, I)
SET - Short Form	(6, 22)(5, I)
STORE-SPECIAL - Short Form.	(6, 23)(5, I)
SET-SPECIAL.	(6, 24)(5, I)

Figure 3. Proposed instruction set for DLM computer.

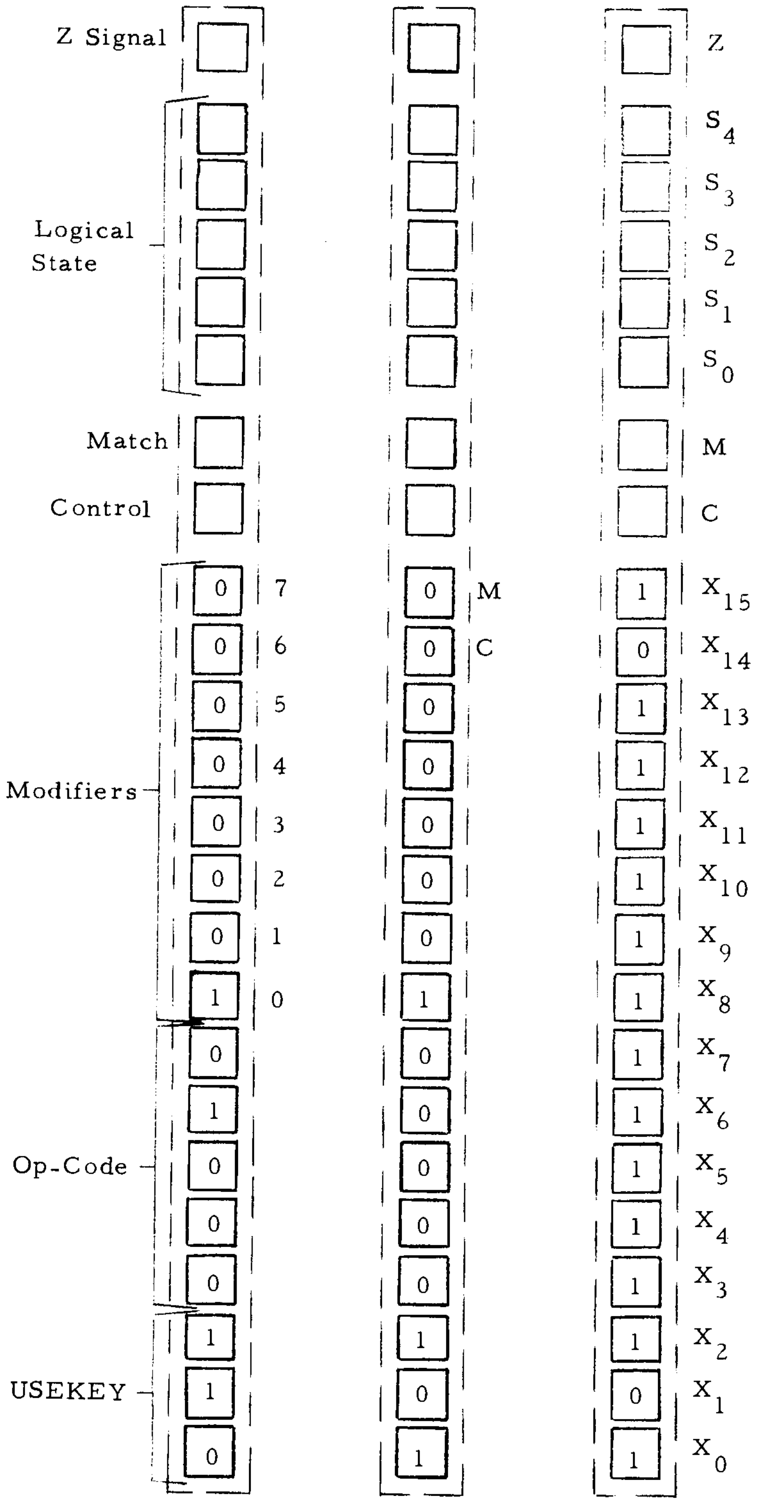
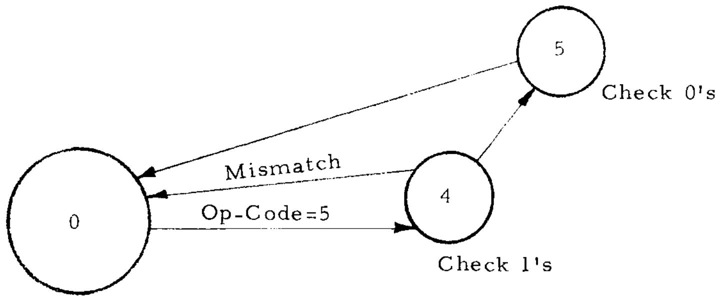


Figure 4. Example of typical instruction: STORE 0 ($C=0, X_8=1$).



That portion of State Transition Diagram for MATCH - Long Form
(6, 5)(5, I)(5, J)

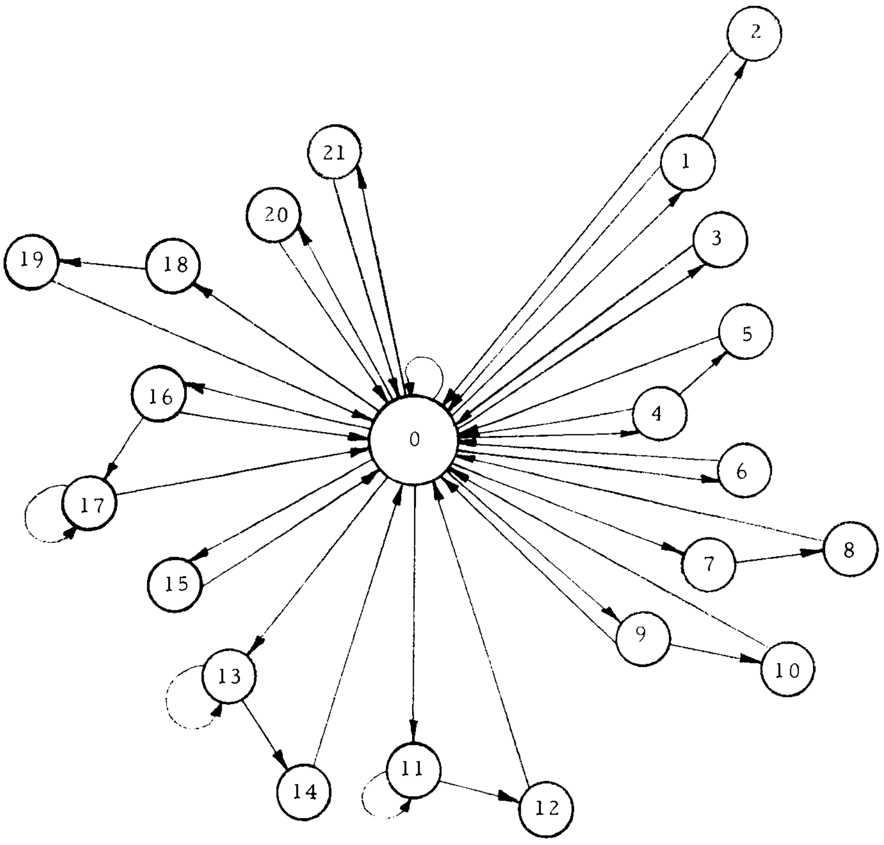
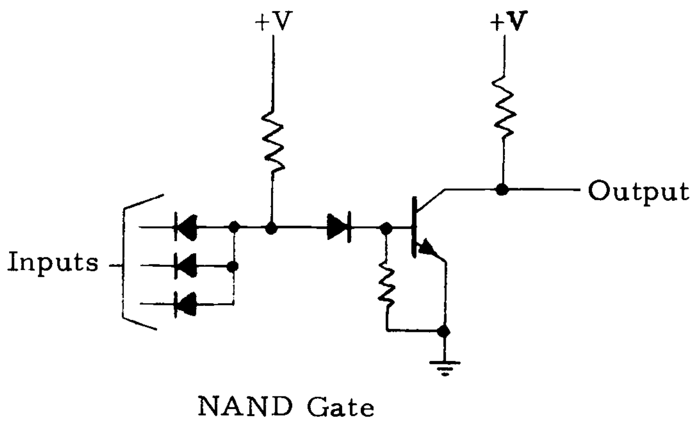
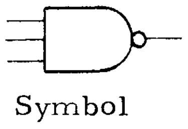


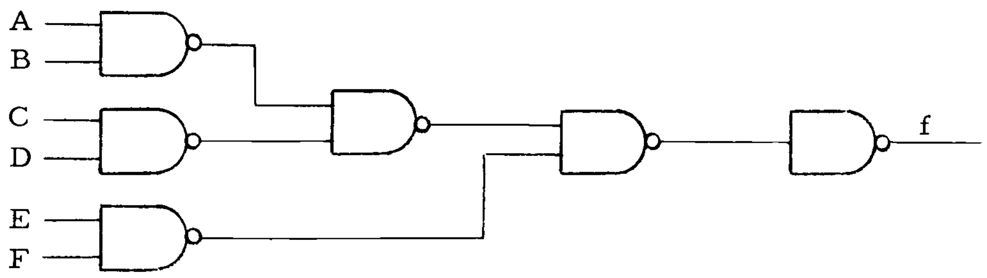
Figure 5. State transition diagram.





Logical Function

$f = (AB+CD)(EF)'$



Equivalent Form

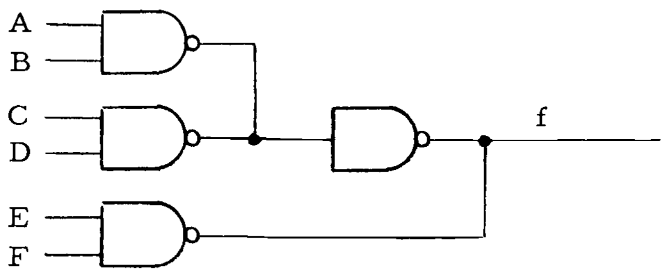


Figure 6. Example of NAND gates whose outputs may be tied together.

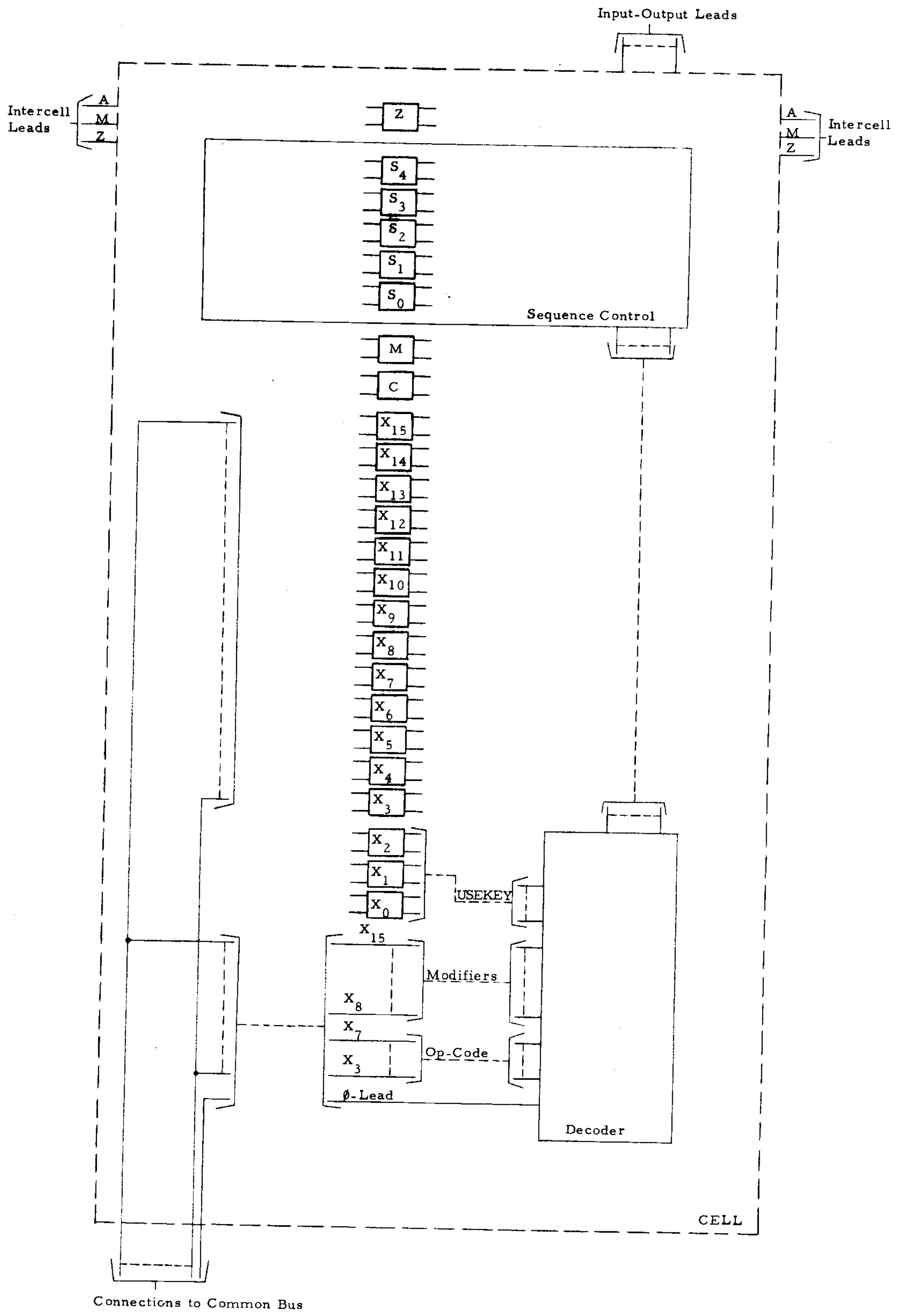


Figure 7. Block diagram of cell..

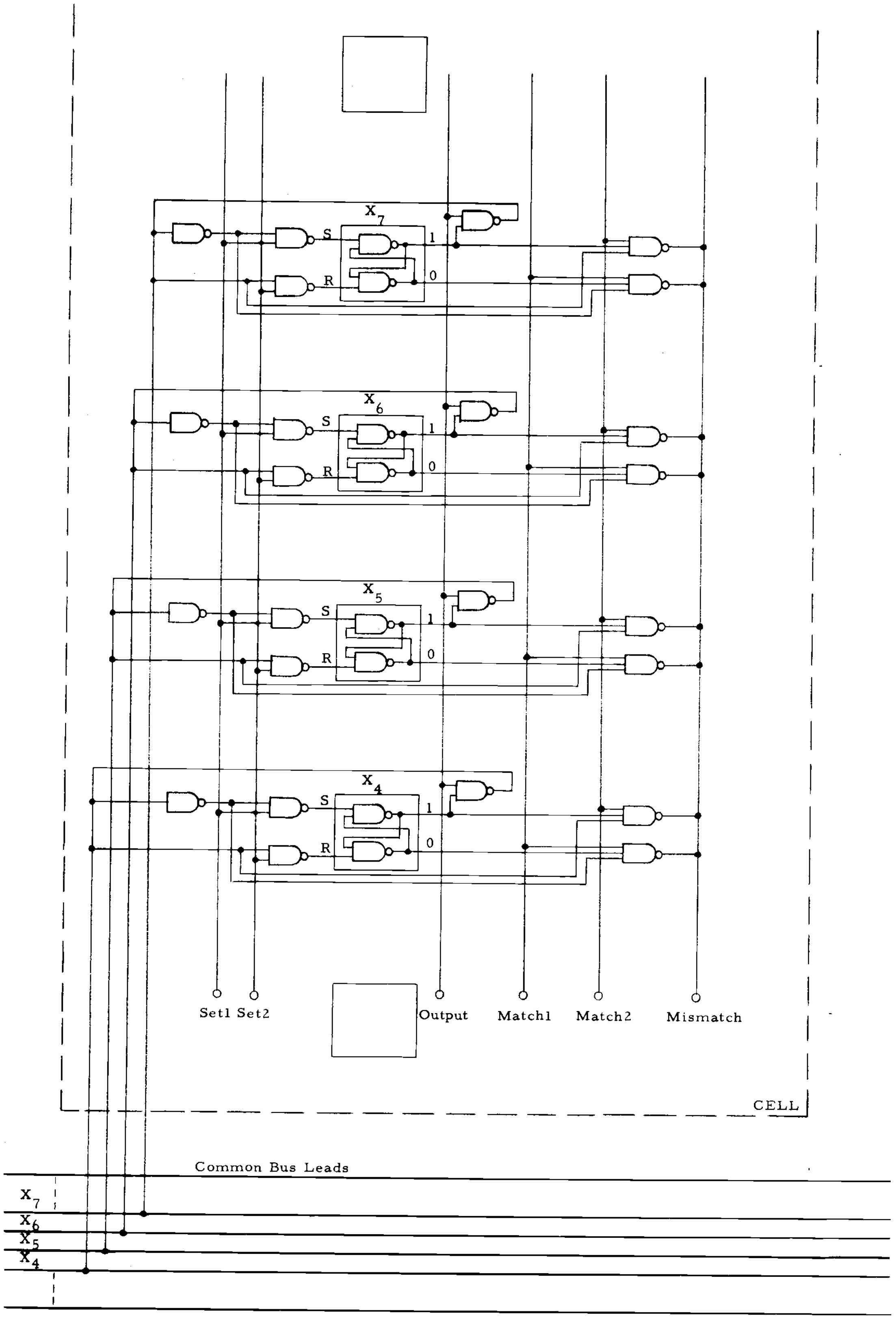


Figure 8. Input-output gating of memory word and common bus interconnections.

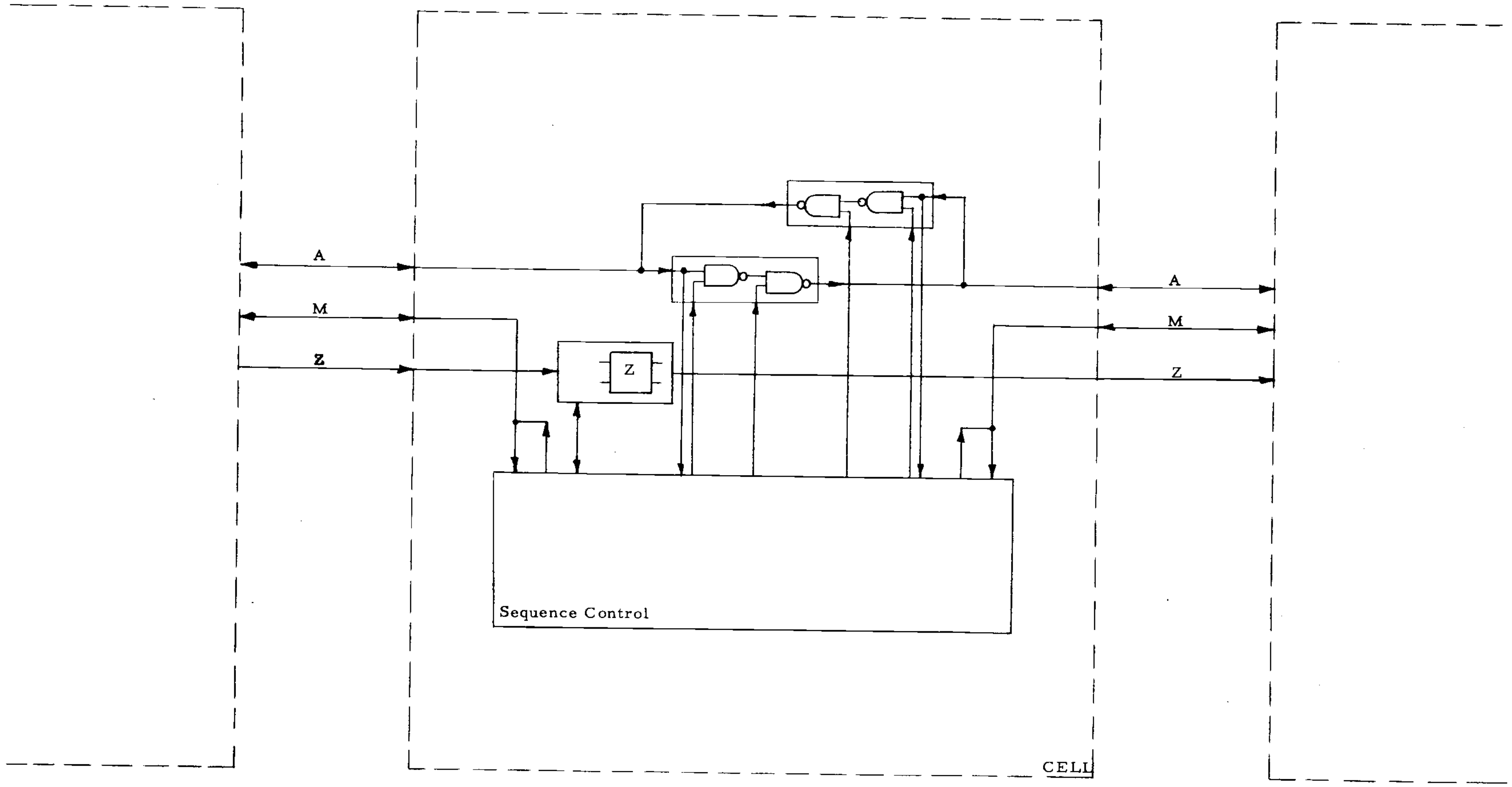


Figure 9. Intercell lead arrangements.

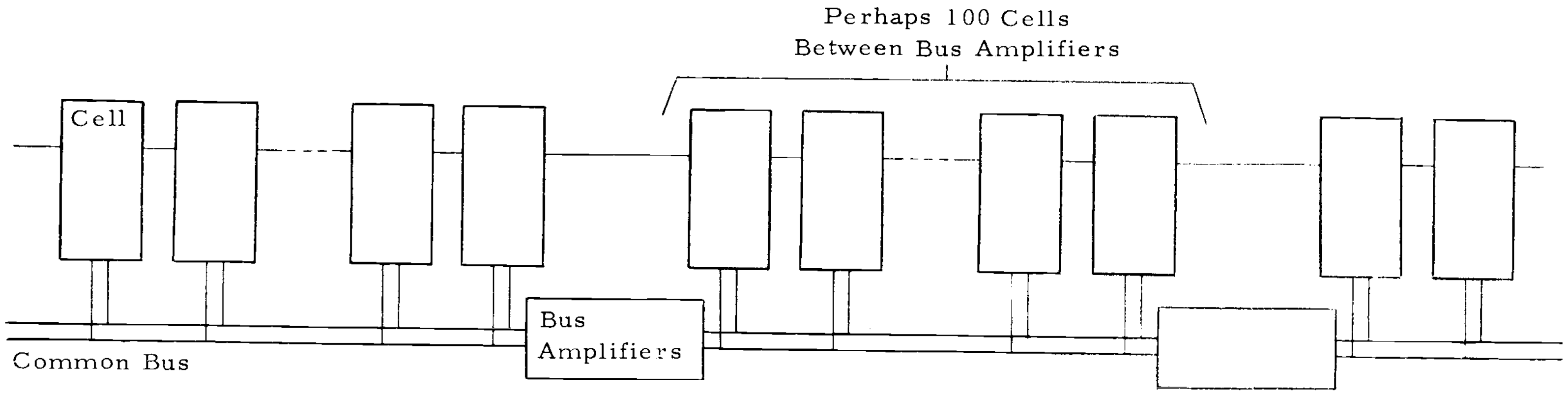


Figure 10. Cell interconnections including one-way bus amplifiers.

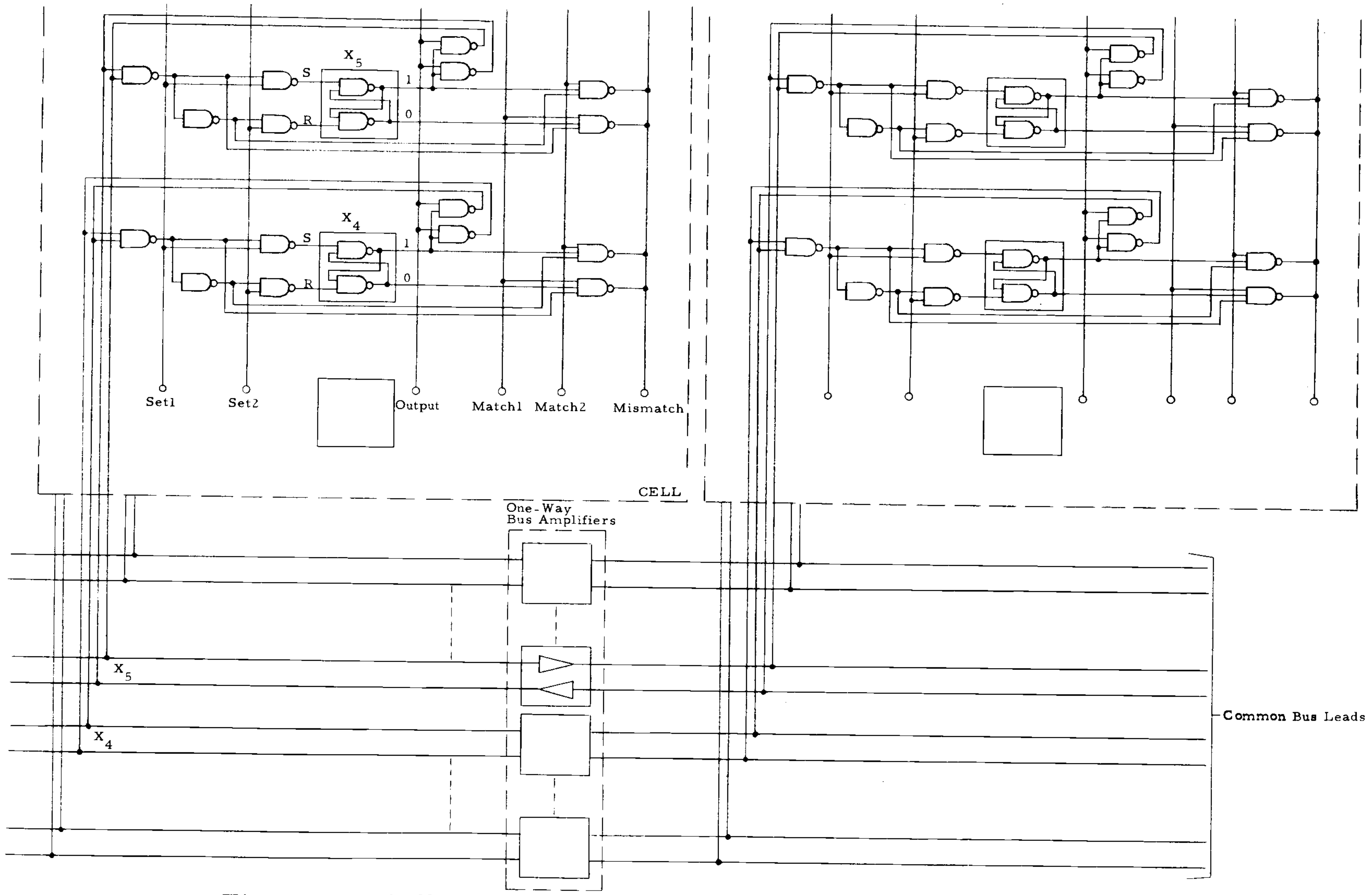


Figure 11. Cell interconnections with one-way bus amplifiers.

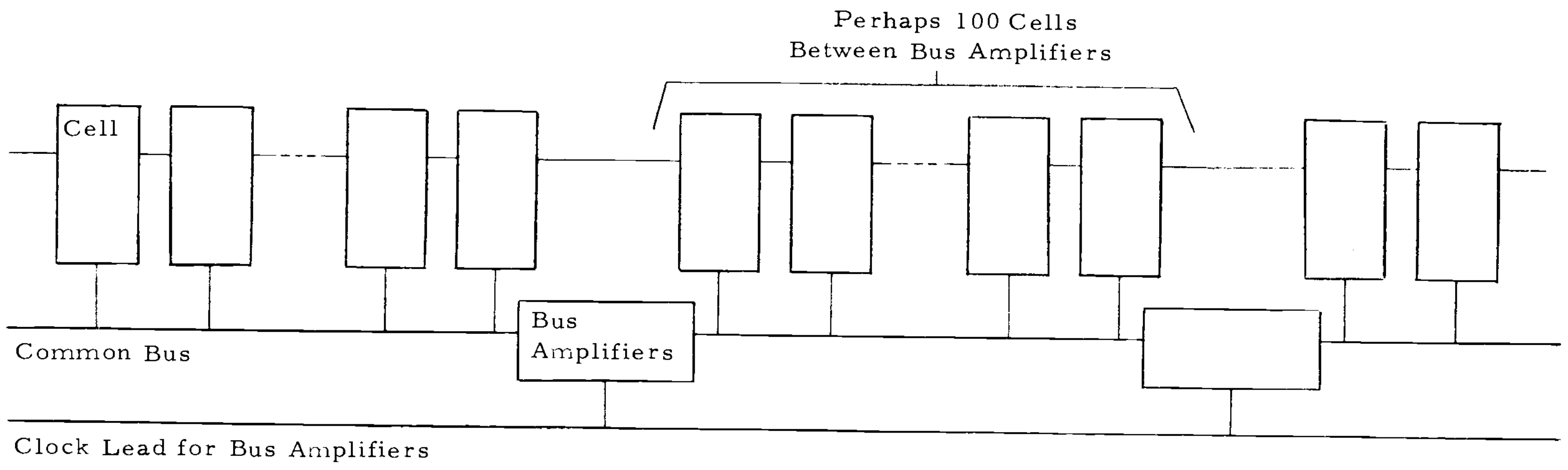


Figure 12. Cell interconnections including clocked two-way bus amplifiers.

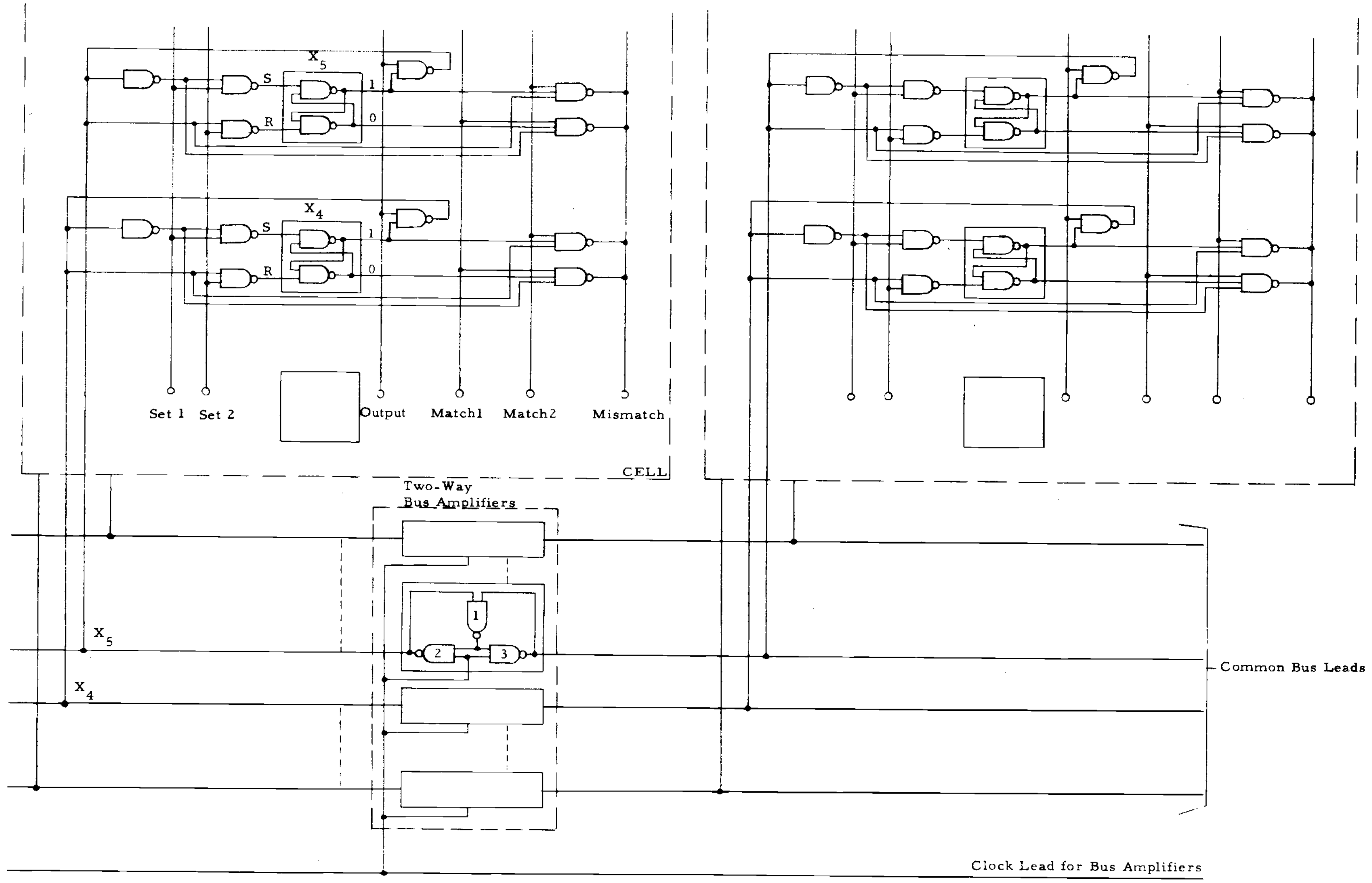


Figure 13. Cell interconnections with two-way bus amplifiers.

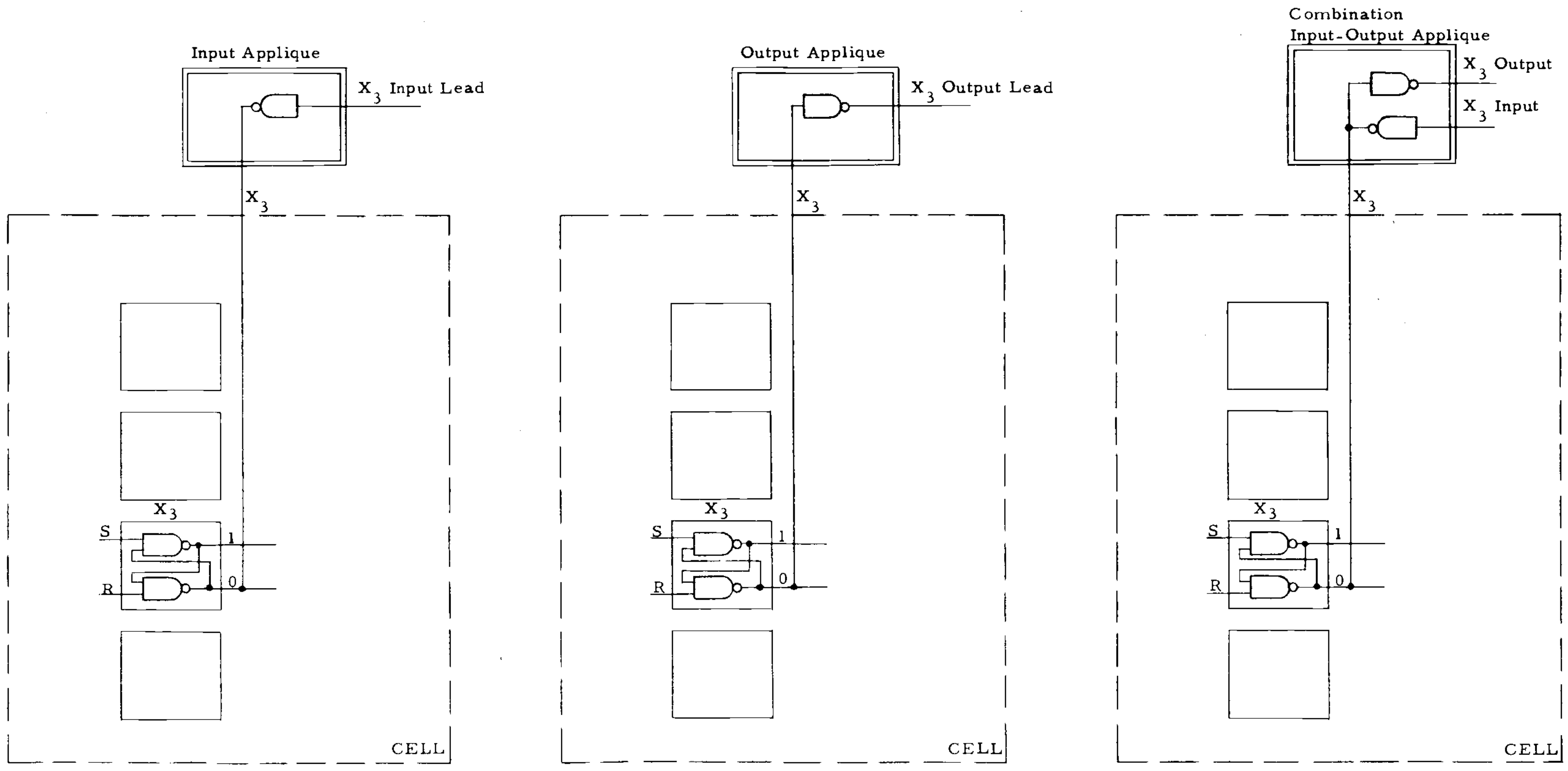


Figure 14. Examples of simple input-output interfaces.

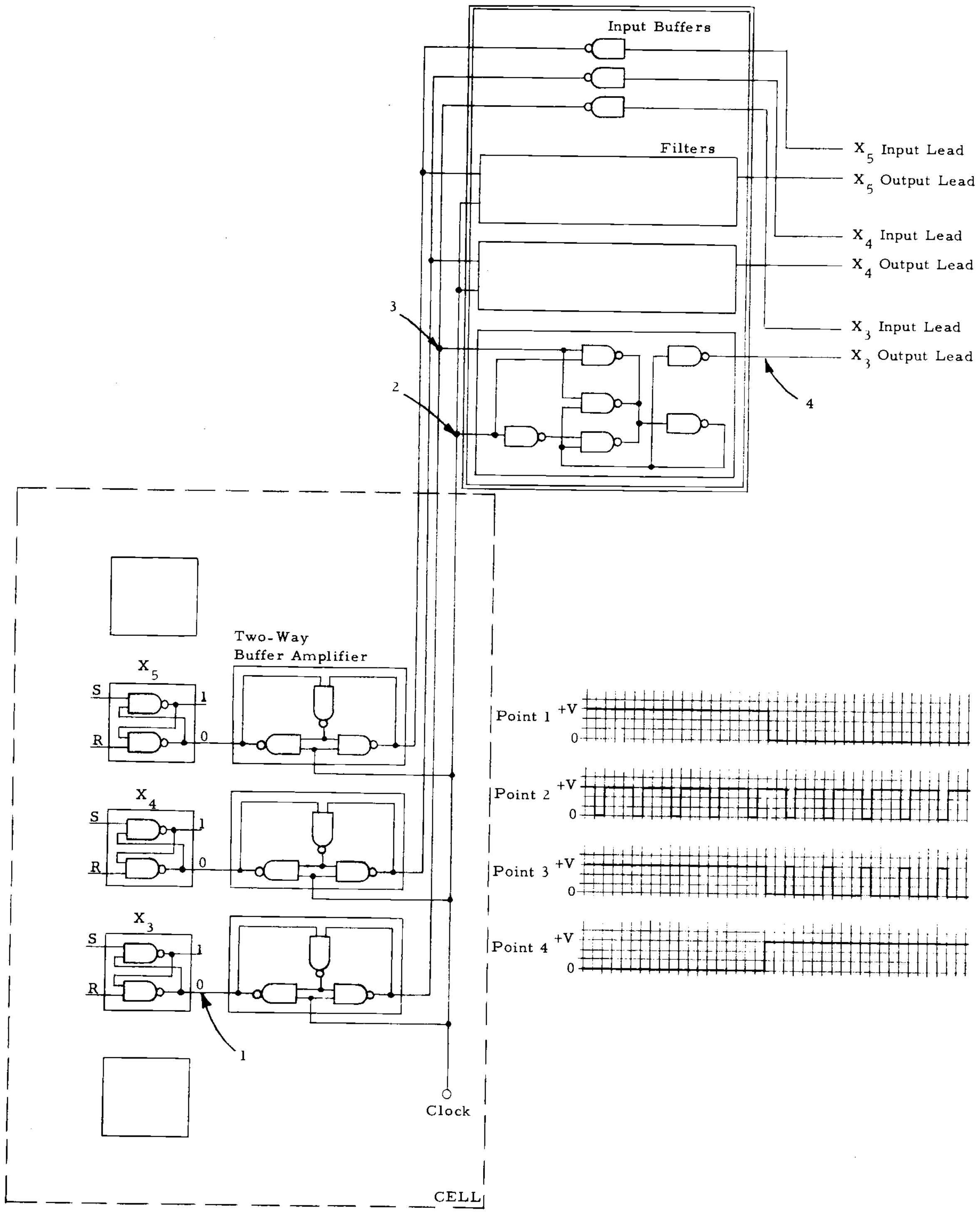


Figure 15. Using two-way buffer amplifiers and filters for input-output interfaces.

Common Bus Leads

Terminal	1.	+V
	2.	Common
	3.	Clock-1
	4.	Clock-2
	5.	Ø Lead
	6.	Z Reset
	7.	X_3
	8.	X_4
	9.	X_5
	10.	X_6
	11.	X_7
	12.	X_8
	13.	X_9
	14.	X_{10}
	15.	X_{11}
	16.	X_{12}
	17.	X_{13}
	18.	X_{14}
	19.	X_{15}

Intercell Leads

Terminal	20.	Left A
	21.	Left M
	22.	Left Z
	23.	Right A
	24.	Right M
	25.	Right Z

Input-Output Leads

Terminal	26.	X_3
	27.	X_4
	28.	X_5
	29.	X_6
	30.	X_7
	31.	X_8
	32.	X_9
	33.	X_{10}

Assumptions: 1) Sixteen-bit Memory Word.

2) Eight bits of Memory Word equipped for input-output.

3) Two clock leads with non-overlapping phases.

Figure 16. Listing of terminal requirements for cell.

Assumptions:

1) One terminal per memory word bit equipped for gating on and off common bus.

2) One terminal per memory word bit equipped with input-output interface.

3) Three-bit USEKEY.

4) Two clock leads with non-overlapping phases.

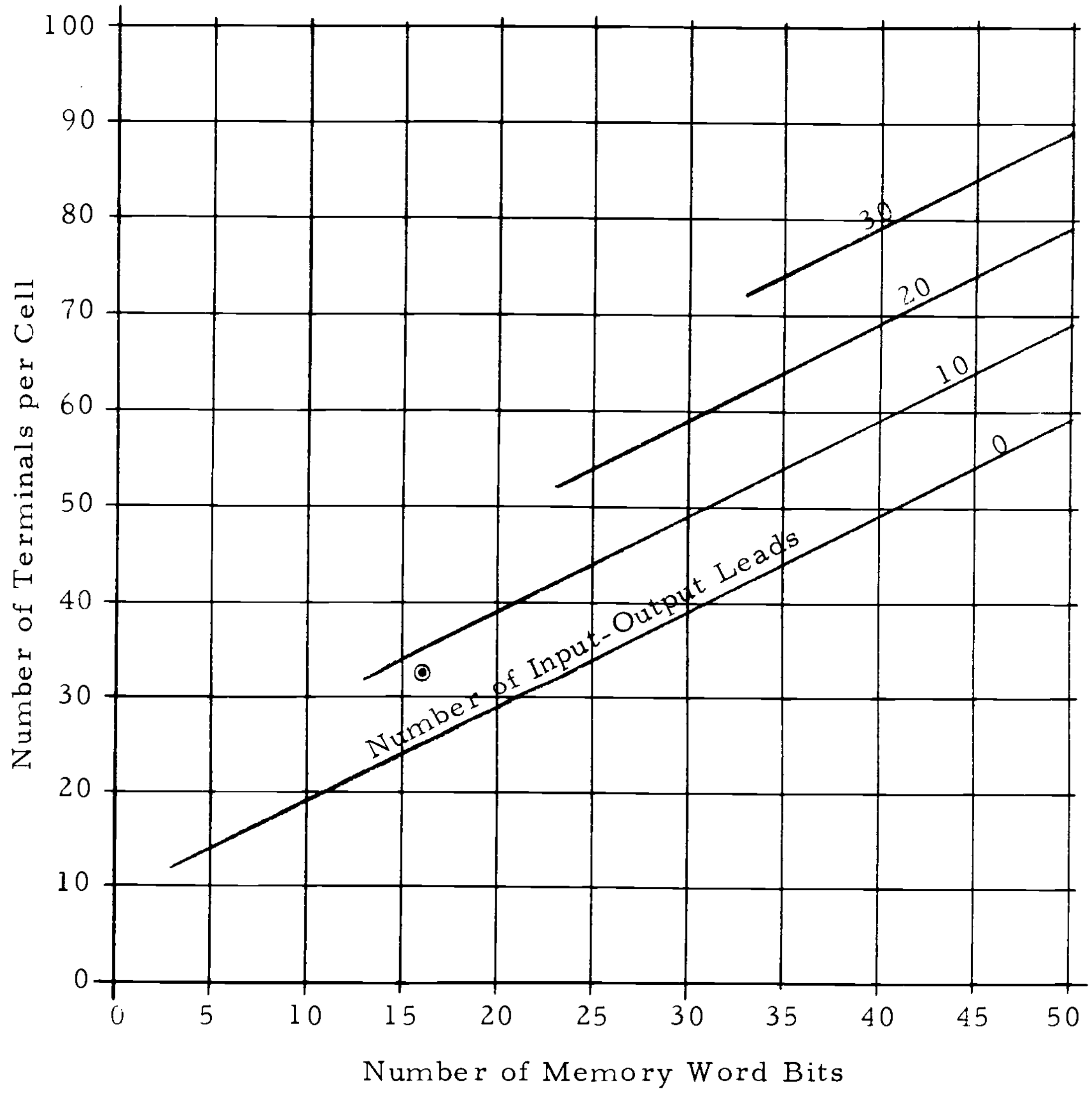


Figure 17. Number of terminals per cell vs. number of memory word bits for various numbers of input-output leads.

Small Instruction Set (9 States)

1. TRANSFER
2. LEFT
3. RIGHT
4. MATCH - Long Form
5. STORE - Long Form
6. SET - Long Form
7. MARK-LEFT
8. MARK-RIGHT
9. OUTPUT
10. STORE-SPECIAL - Long Form

Medium Instruction Set (16 States)

1. TRANSFER
2. LEFT
3. RIGHT
4. MATCH - Long Form
5. UNCONDITIONAL-BRANCH
6. RESETM
7. STORE - Long Form
8. SET - Long Form
9. MATCH-LEFT
10. MATCH-RIGHT - Long Form
11. MARK-LEFT
12. MARK-RIGHT
13. OUTPUT
14. CONDITIONAL-BRANCH2
15. CONDITIONAL-BRANCH1
16. RESETC
17. STORE-SPECIAL - Long Form

Large Instruction Set (22 States)

1. TRANSFER
2. LEFT
3. RIGHT
4. INTERPRETIVE-MATCH
5. MATCH - Long Form
6. UNCONDITIONAL-BRANCH
7. RESETM
8. STORE - Long Form
9. SET - Long Form
10. MATCH-LEFT
11. MATCH-RIGHT - Long Form
12. MARK-LEFT
13. MARK-RIGHT
14. OUTPUT
15. CONDITIONAL-BRANCH2
16. CONDITIONAL-BRANCH1
17. RESETC
18. MATCH - Short Form
19. MATCH-RIGHT - Short Form
20. STORE-SPECIAL - Long Form
21. STORE - Short Form
22. SET - Short Form
23. STORE-SPECIAL - Short Form
24. SET-SPECIAL

FIGURE 18. Three instruction sets for which the gate counts of the corresponding sequence controls were estimated.

Item	Small Instruction Set		Medium Instruction Set		Large Instruction Set	
	Estimated Gate Count					
1. Sequence Control	9 States	47	16 States	72	22 States	85
2. Decoder	10 Instructions	44	17 Instructions	49	24 Instructions	58
3. Memory Word Bits and Gating						
X_0 through X_2		12		12		12
X_3 through X_{11}		72		72		72
X_{12} through X_{15}		32		32		40
4. Intercell Lead Gating		15		15		15
5. Match, Control Flip-Flops and Gating		13		13		13
6. Input-Output		None*		None*		None*
	Totals	235 gates		265 gates		295 gates

* It is assumed that external input-output interfaces are sufficient so no allowance is made for gates as part of the cell.

Figure 19. Breakdown of estimated gate counts for the three instructions sets.

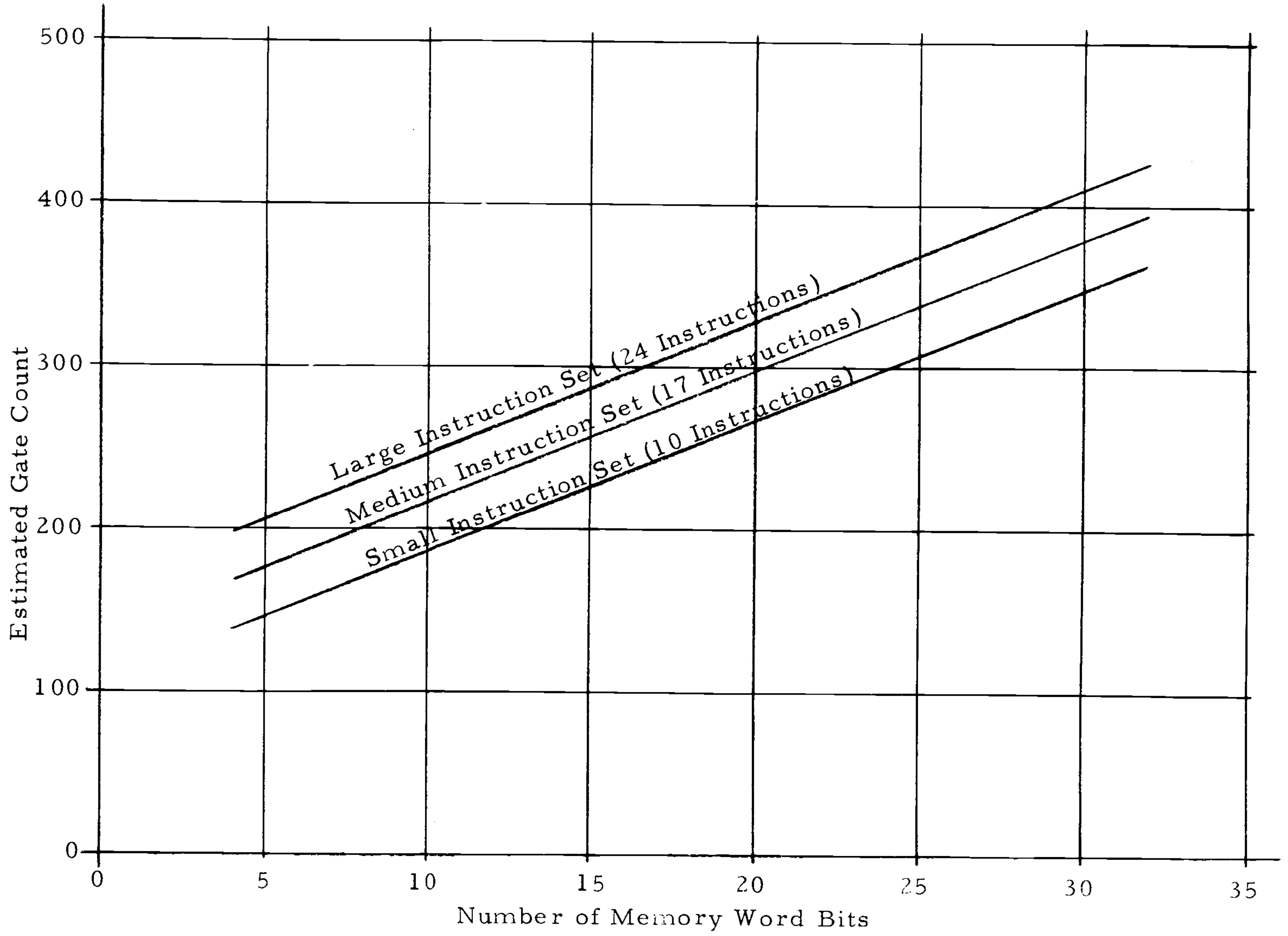


Figure 20. Estimated gate count per cell vs. number of memory word bits for the three instruction sets.

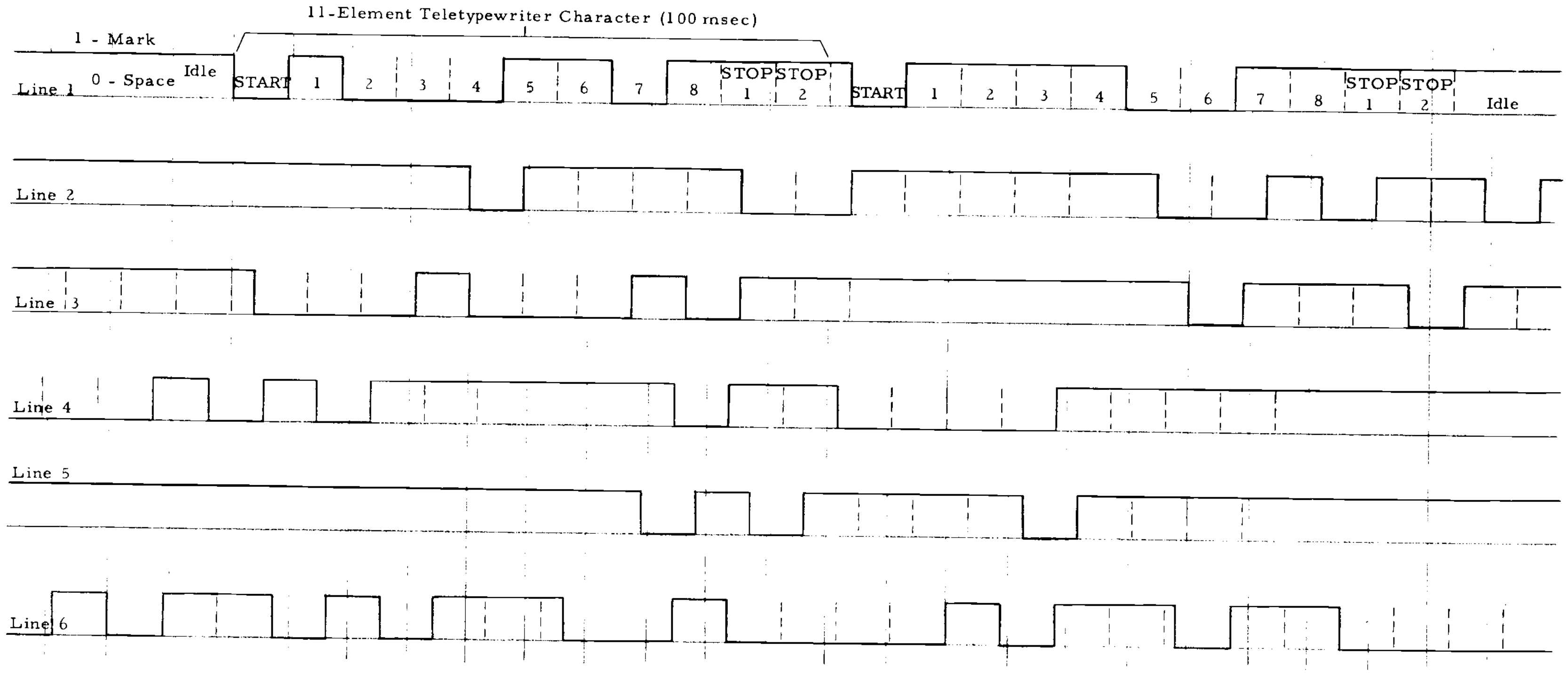


Figure 21. Stylized typical waveforms to be expected at teletypewriter line terminals.

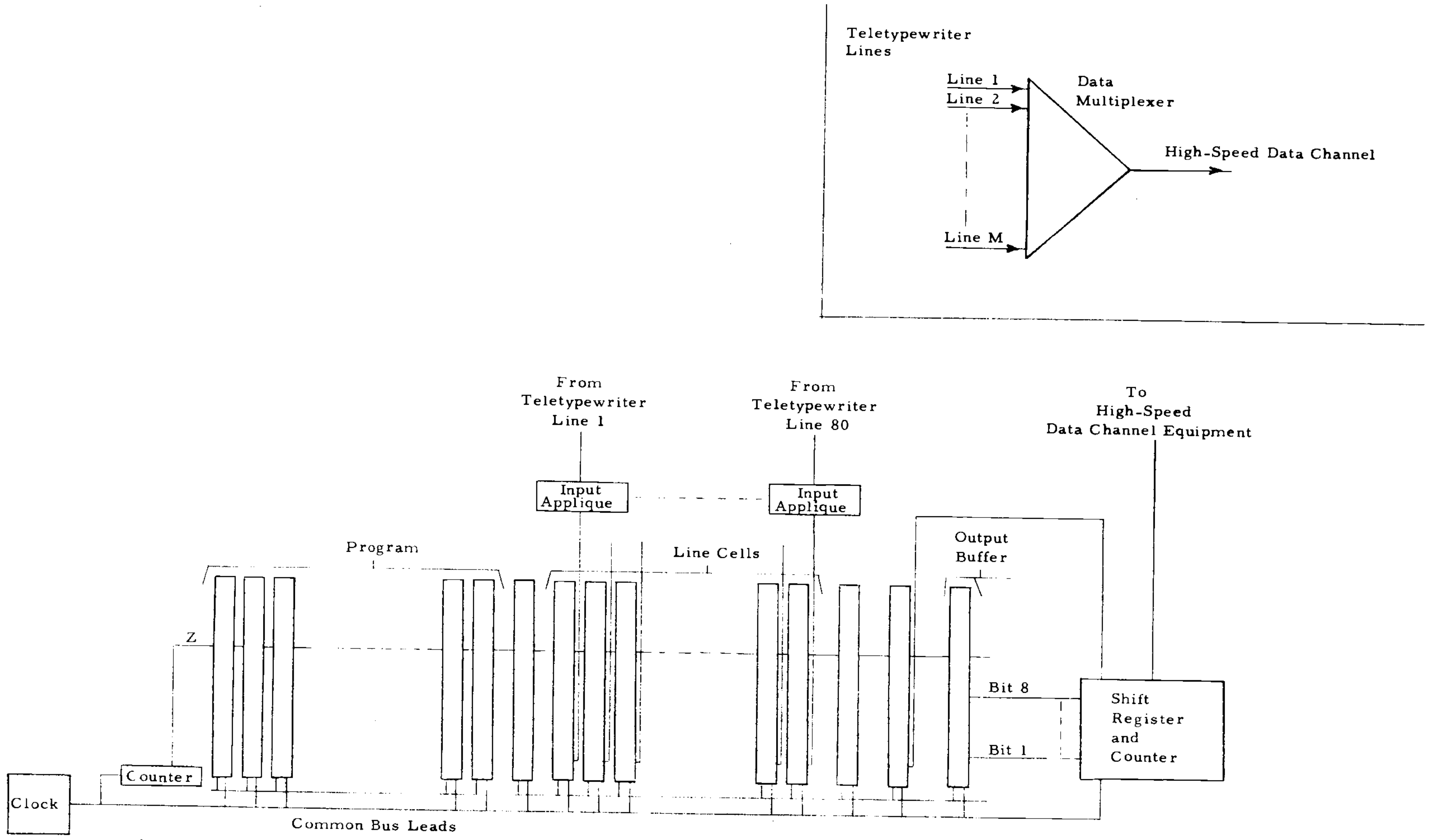


Figure 22. Layout of cells for use as data multiplexer.

X_0, X_1, X_2 - USEKEY

X_3 - Present Look (PL)

X_4 - Last Look (LL)

X_5 - Corrected Last Look (CLL)

$X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}$ - 7-Bit Timer

$X_{13}, X_{14}, X_{15}, X_{16}$ - 4-Bit Bit Counter

$$\left.\begin{array}{l} X_{17} \text{ - Bit 1} \\ X_{18} \text{ - Bit 2} \\ X_{19} \text{ - Bit 3} \\ X_{20} \text{ - Bit 4} \\ X_{21} \text{ - Bit 5} \\ X_{22} \text{ - Bit 6} \\ X_{23} \text{ - Bit 7} \\ X_{24} \text{ - Bit 8} \end{array}\right\} \text{--Teletypewriter Character}$$

X_{25}, X_{26} - Mode bits

X_{27} - Temporary flag used when timer incremented

X_{28} - Floating flag for Transfer

$$\left.\begin{array}{l} X_{29} \text{ - C} \\ X_{30} \text{ - M} \end{array}\right\} \text{--unused for DATA-WORD cells}$$

Figure 23. Memory word bit assignments for 31-bit line cell.

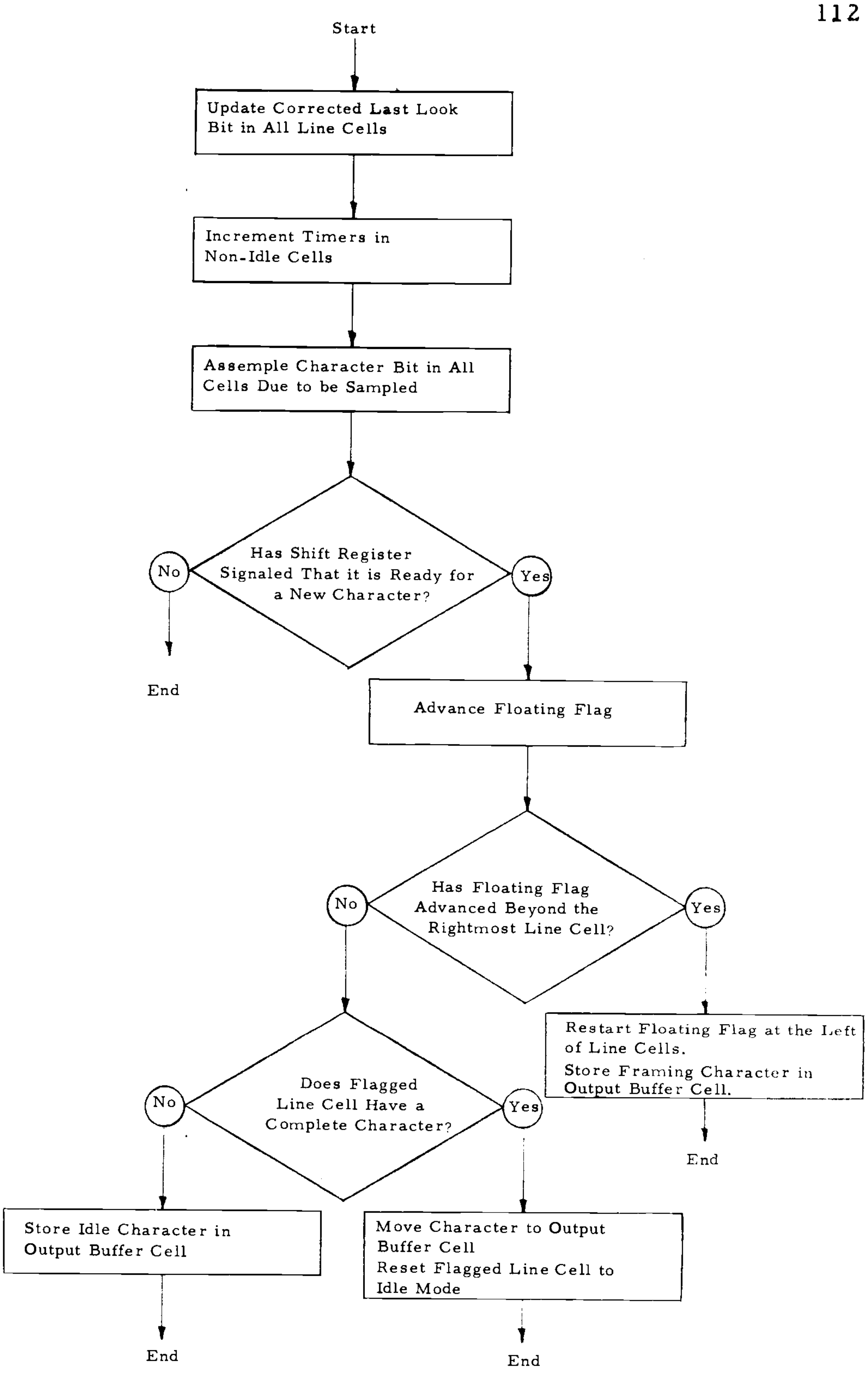

Start
Update Corrected Last Look Bit in All Line Cells
Increment Timers in Non-Idle Cells
Assemple Character Bit in All Cells Due to be Sampled
Has Shift Register Signaled That it is Ready for a New Character?
No
End
Yes
Advance Floating Flag
Has Floating Flag Advanced Beyond the Rightmost Line Cell?
No
Yes
Restart Floating Flag at the Left of Line Cells.
Store Framing Character in Output Buffer Cell.
End
Does Flagged Line Cell Have a Complete Character?
No
Yes
Store Idle Character in Output Buffer Cell
End
Move Character to Output Buffer Cell
Reset Flagged Line Cell to Idle Mode
End

Figure 24. Flowchart for data multiplexer program.

Number of Instructions of Each Type

One-Cell	Two-Cell	Three-Cell	Total
35	10	71	116

Instruction Frequency of Use

TRANSFER	1	MARK-RIGHT	0
LEFT	0	OUTPUT	1
RIGHT	0	CONDITIONAL-BRANCH2	1
INTERPRETIVE-MATCH	0	CONDITIONAL-BRANCH1	2
MATCH - Long Form	33	RESETC	1
UNCONDITIONAL-BRANCH	0	MATCH - Short Form	2
RESETM	32	MATCH-RIGHT - Short Form	1
STORE - Long Form	33	STORE SPECIAL - Long Form	0
SET - Long Form	4	STORE - Short Form	2
MATCH-LEFT	0	SET - Short Form	2
MATCH-RIGHT - Long Form	1	STORE-SPECIAL - Short Form	0
MARK-LEFT	0	SET-SPECIAL	0

Number of Cells Required

Program	Data	Total
273	85	358

Number of Machine Cycles Required per Program Run

Maximum	Minimum
264	223

Figure 25. Data multiplexer program statistics.