X-522-73-135

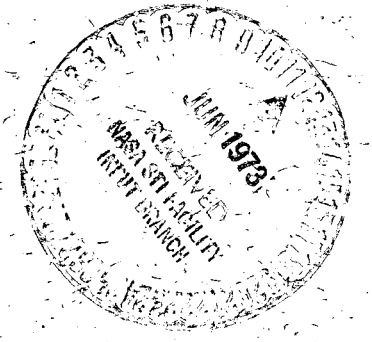
PREPRINT

NASA TM X 66248

A NEW APPROACH TO TELEMETRY DATA PROCESSING

CARLO J. BROGLIO

MAY 1973



GSFC

GODDARD SPACE FLIGHT CENTER

GREENBELT, MARYLAND

A NEW APPROACH TO TELEMETRY DATA PROCESSING

by
Carlo Joseph Broglio

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1973

ACKNOWLEDGEMENT

The author acknowledges a special debt of gratitude to his advisor, Dr. James Pugsley, for his numerous helpful suggestions and especially for his patience in the critical review of this thesis.

ABSTRACT

Title of Thesis: A New Approach to Telemetry Data Processing

Carlo Joseph Broglio, Doctor of Philosophy, 1973

Thesis directed by: Dr. James Pugsley, Associate Professor

A new approach for a preprocessing system for telemetry data processing has been developed. The philosophy of this approach is the development of a preprocessing system to interface with the main processor and relieve it of the burden of stripping information from a telemetry data stream. To accomplish this task, a telemetry preprocessing language has been developed. This higher level language contains statements designed using the jargon of telemetry data engineers and a set of simple but powerful operators for manipulating telemetry data. Also, a hardware device for implementing the operation of this language was designed using a cellular logic module concept.

In the development of the hardware device and the cellular logic module, a distributed form of control has been implemented. This is accomplished by a technique of one-to-one intermodule communications and a set of privileged communication operations. By creating a special state (called the control state), each module can direct the activities of the system. By transferring this control state from module to module, the control function is dispersed through the system.

A compiler for translating the preprocessing language statements into an operations table for the hardware device was also developed. This compiler uses a simple left to right single pass compilation algorithm. It can do so because the language is simple and has no operator precedence.

Finally, to complete the system design and verify it, a simulator for the cellular logic module was written using the APL/360 system. This simulator

contains data sets which are images of the programs that are loaded into the various modules of the system. It then emulates the operations of the modules and produces timing data. The simulator was used to prove that the concepts and microcode loaded into the modules worked. The timing data gathered by it was used to form comparisons with a medium speed machine of the operations of a preprocessing program on the modular device with those on the medium speed machine. The results of this comparison show that the device compares very well, being a fraction of two to six slower on arithmetic operations, but two orders of magnitude better on the bit manipulation operations.

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER I

INTRODUCTION

The action of telemetering data from spacecraft sensors to ground based processing equipment introduces a number of unique data manipulation problems. The basic cause of these problems is the need to combat noise in the space-to-earth communications channel. Another cause of these problems is the use of spacecraft tape recorders. Since typically a tape cannot be changed while in flight, a method of recording in one direction and reading in the reverse direction is used. This, however, also causes the data to be transmitted backwards compared to non-recorded data.

The data under consideration in this thesis is strictly digital data. By this is meant, a sensor measurement value is coded into a set of ones and zeroes called binary digits. These binary digits (bits) are then telemetered to a ground based receiving station where they are recorded on an analog tape. This analog tape is transported to a processing facility. However, during the telemetering process the binary bits were encoded into one of several position-time sequences. These sequences are designed to combat a particular kind of noise which may be known or suspected to be present (reference 1). During the telemetering and recording process, the timing information necessary to reconstruct the sets of data bits has been lost. Hence, to reconstruct this information, special purpose equipment is required and various special techniques are used (reference 2).

First, the data bits must be reconstructed as accurately as possible. For this purpose a device known as a bit synchronizer is used. This device produces a "best estimate" of what the original bits were. It typically employs a maximum likelihood decision model. At this point in the processing, a stream of data bits is present. This data stream contains errors and must be regrouped into the

original sets of data values transmitted. These values are now called data words, each word being a known number of bits in length. But during the bit synchronization process, the starting bit position of the first word is lost and, thus, it is unknown where any data word begins or ends. Thus the technique of creating a special grouping of data bits into sets called data frames is used. These sets contain (usually as a prefix) a special known bit pattern called a frame synchronization pattern (FSP). A special device known as a frame synchronizer is used to "search" the data stream for this pattern. This is done typically by placing the desired FSP in a data register and shifting the data stream through another data register. A comparator is placed between the two registers and contains a preselected error tolerance. When a bit by bit match is obtained between the two registers that falls within the error tolerance, the pattern is considered found. At this point it is possible to separate the data bits into the specified data words and the telemetered data is considered to be recovered. Typically, the data is then transferred to a general purpose computer and the data processing phase begins.

At this point in the operation, several observations should be made. No spacecraft currently being flown contains only one sensor. In fact some contain hundreds of sensors. In the data frame scheme described, not all sensor outputs need to be in a single data frame because, such a scheme implies a fixed sampling rate. Furthermore, a standard governing the size (in bits) of data words and data frame (reference 3) exists. Hence, not all sensors can be placed in one data frame and, quite often, sensor values cannot be placed in single data words nor even in consecutive data words. Thus, some sensor data values may be distributed in words throughout the data frame. Other words of the data frame may contain a sequence of sensor values on consecutive data frames (a process called subcommutation). When subcommutation is used, a method for determining the start of the sequence must be present. Usually a data word is used for

this purpose. For example, a word may contain a binary counter which represents the sequence number of this data frame and identifies what sensor values are present.

In addition, many spacecraft use a method of parity generation to insure error detection capabilities. When this is present, the parity for the received data must be computed and compared with the received parity to determine if an error has occurred. As a further assurance of how well the system is operating, the bits contained in the FSP are compared with those expected and a count of the errors is maintained. This error count is used as a measure of the error level of the bit stream. Finally, to assist the bit synchronization process in cases where it is suspected that data values may not change for many bit times, certain bits of the data stream are complemented.

All of these observations noted here require a set of data processing functions to be implemented. These functions are needed to transform the data frames into data values that the computer can work with. However, these functions do not contribute directly to the data processing operation. Further, these functions are awkwardly handled in a large general purpose processor since most of these machines are designed for data computation and have limited bit manipulation capabilities. The problem is further compounded by the fact that most higher level programming languages are also designed to do computations and many are very inefficient at bit manipulation. Efficiency becomes important for two major factors. First, spacecraft generate a large volume of data; greatly reducing total run times can be accomplished by saving instructions in highly repetitive operations. Second, a need often exists to handle the data as it is received in real time and hence not much processing time is available. A final observation is that the data is transferred into the computer over one of its input-output channels. This means that the data words are stored consecutively in the

computer's internal data words. The size of these two different words is rarely identical and thus represents an unnatural data set to the computer because, data words are not on computer word boundaries.

Under the current method of telemetry data processing, these problems are handled by programs coded at the machine language level. Hence, if more than one type of computer is involved at the frame level of processing, as is typical, a costly duplication of programming effort is required. Further, any event which causes a change in the data format (e.g., a failure on the spacecraft while in orbit or, a design change in a family of spacecraft) requires extensive reprogramming to accommodate. Several attempts have been made in the past to generalize some of these functions (references 4, 5, and 6). These approaches, however, were either too specialized to a specific machine configuration or too cumbersome and complex to be used effectively.

The work of this thesis is directed toward the solution of these problems while overcoming the difficulties of the past approaches. In the context of this solution, it is assumed that a special-purpose device will be placed between the frame synchronizer and the host computer's input-output channel. This device will have the ability to pass the necessary parameters to the frame synchronizer subsystem to enable it to run. The device will then accept data from the frame synchronizer, and reformat this data into sensor values which will appear on the host computer's word boundaries. Finally, the functions of parity checking, word reversal, bit complementing, data counter continuity checking and FSP error measurements will also be done in this device, thereby allowing the host computer to concentrate on data processing.

To overcome the difficulties and costs encountered in programming the required bit manipulation functions, a special-purpose higher level telemetry preprocessing language has been designed. This language concentrates on bit

manipulation methods and has only a minimal set of computation instructions. The statements of this language are derived from telemetry data handling engineer's jargon and hence, programs in this language represent a concise description of the telemetry data frame. Perhaps the biggest advantage of this language is the ease with which the programmer can accommodate changes in the data format's structure.

Since the device which implements the language must be able to interface with a wide variety of host computers on one end and a number of different frame synchronizers on the other end, a microprogrammed (reference 7) approach was taken. Another factor influencing this decision was the wide variety of internal computer formats which must be accounted for, since this device must appear to be a standard device to the host machine. By appearing as a standard device, the host machine's operating system can be used with minor changes and hence the system integration costs are minimized.

Having decided to use a microprogrammed approach, the operations specified in the preprocessing language were examined to determine what microcoded functions were required. It was observed that: a) telemetry words vary in size from 6 to 32 bits,[1] b) bit for bit word reversal is a nontrivial function, and c) selective bit complementation is a special operation compared to all the other functions required to implement the preprocessor. Upon examining "off the shelf" microcoded machinery, these operations are not part of the standard functions offered. Further, many of the word sizes required are not compatible with the machines' internal data structures; hence, using one of these machines

---

1. The Standards (reference 3) claim larger word sizes, but in practice they are never used. In fact, the hardware currently in use will accommodate a maximum word size of 32 bits.

would represent a transferal of the problem from the conventional general purpose machine to the microcoded one and, many of the past difficulties would still be present. Thus the operations of the preprocessing system were examined to find the most uniform approach to the total system design. To accomplish this task, a design of an integrated circuit chip, implementing a concept known as a functional memory (reference 8), was completed. This module design offers the power of implementing all the required logical functions with a single chip structure. The interconnection between these chips is accomplished with another chip structure thereby yielding a system with only two basic parts.

The functional memory module is discussed in detail in Appendix A. The term functional memory denotes a device used to generate Boolean functionals in a memory device. Basically, it is a method of arranging a cellular memory array such that, each cell of the array can be either an associative memory cell or a conventional memory cell. Additional gating is provided at the array boundaries so that Boolean functions can be generated by using the above two memory types in combination. The associative nature of the memory is used to "search" for a set of preprogrammed Boolean expressions in the input data. Then the results of this search are used as a conventional address to "read" the function output from specified cells of the array.

In order to implement a system of this type the concept of distributed control was used. This concept treats each module as an independent processing station and represents a means of networking these stations.

The concept of distributed control was hinted at in an article by L. J. Koczela (reference 9). This is basically the replacement of the conventional single control unit by a transferable abstraction of the control function. This implies that each functional memory array of the system contains a flip-flop indicating whether or not it has system control. If it has system control, then it

is allowed to carry out certain privileged global operations. These operations primarily deal with the intermodule data communication system. Only a module in control is allowed to transfer data over the communication bus to other modules and initiate cycles within those modules. Further, control can be both transferred and retained by the issuing module, thus allowing independent control sequences to be simultaneously initiated. By the use of this concept, the data flow paths through the various functional elements are directed with the net result being that the data flow path through the system determines the total processing function to be performed.

A conventional control approach with one or more modules making up the control function could have been implemented, but this approach would represent a more complex design. With distributed control, no subsystem needs to know what functions are being executed at all parts of the machine. It merely needs to know what the data destination is and where to pass control. This simplifies the design by necessitating fewer system wide control lines and allowing independent functions to complete their tasks at the rate which the tasks require.

In summary, this system offers two advantages capable of overcoming the difficulties of past approaches. These advantages are a higher level telemetry preprocessing language and a peripheral device to implement that language in microcode. The language offers to the user the capability of specifying in nearly English terms, the transformations that he desires to perform on the telemetry data stream. These terms were derived from the jargon used by the handlers of telemetered data and hence, should be readily understood by workers in this area.

The microprogrammed peripheral device has the advantage of separating the operations to be performed from the main computer. This helps the problem solution in two ways. First, it simplifies the processing program in the main

computer by preparing a data set for it and hence, relieving this processor of many bookkeeping and manipulation functions which do not contribute to the processing operation directly. Further, many of these functions are not implemented in the standard instruction set and thus are inefficient to implement. Secondly, by being a microcoded device, it is adaptable to most commercially available computers since the microcode can be changed to suit the host computer and thus the device becomes installation independent.

The remainder of this thesis will describe the above problem solution in detail and develop in depth the concepts and techniques used. Chapter II presents the telemetry preprocessing language. The elements of this language are defined and explained by use of examples. A compiler for this language which was written in APL (references 10 and 11) is also described in this chapter.

Chapter III discusses the design and microcode sequences of the peripheral device developed to implement this language. Chapter IV describes an APL simulation of the system and presents a discussion of it. Chapter V presents the results, conclusions, and recommendations of this work. Appendix A is a detailed presentation of the functional memory module chip design. Appendix B is a meta-language description of the telemetry processing language. Appendix C is a description and listing of the APL compiler. Finally, Appendix D is a description and listing of the APL simulation programs.

CHAPTER II

THE TELEMETRY PREPROCESSING LANGUAGE

A. The Problem Environment

In Chapter I, the various parts of a telemetry data acquisition system were introduced and the interfaces to this research were discussed. These concepts are shown again here in Figure 2-1. In this figure, the telemetry data is input to the system on what is called an analog tape. The name "analog" is somewhat misleading because the recorded signals are digital pulses (i.e., on-off type). It is called analog because the timing information needed to recover the data is not recorded on the tape and the techniques used to recover that information are analog in nature (e.g., a phase-locked loop).

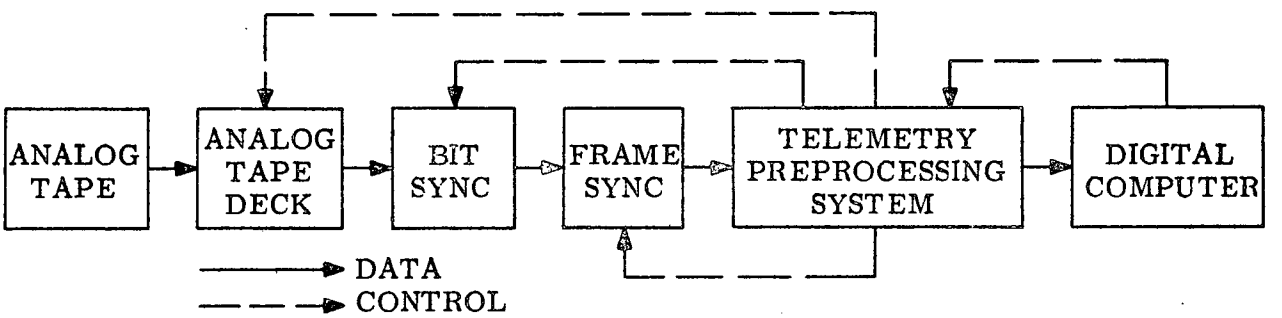


Figure 2-1. PROPOSED TELEMETRY DATA ACQUISITION SYSTEM

As is shown in Figure 2-1, the proposed telemetry preprocessing system has two functions. First, its main task is to reformat the frame synchronized data into sensor values capable of being processed directly by the digital computer. Second, it is designed to distribute the necessary control parameters to the acquisition system to allow it to operate. While this capability is designed into the system, it is not necessary for its operation. Hence, the preprocessing system can operate in cases where other forms of control are desirable or in cases where other configurations are necessary (e.g., a digital tape could input directly to the telemetry preprocessing system in the figure).

Upon closer examination of the system of Figure 2-1, the parameters necessary for this systems operation are easily determined. The analog tape deck needs to know what speed is required to run the analog tape since these units are typically designed to run at several speeds. The bit synchronizers are designed to run over a broad range of data rates (called bit rates) and also can handle several forms of modulation codes (called code types). Hence the desired bit rate and code type are necessary parameters. Finally, the frame sysnchronizers are typically generalized to handle all the standard types of formats (reference 13). The necessary parameters required to define these formats are the number of bits in a telemetry data word, the number of telemetry data words in a telemetry data frame, and the frame synchronization pattern.

The last three parameters (bits/word, words/frame and FSP) form part of the input constraints to the telemetry preprocessing system and are used by that system to locate data values and to perform some of the error measurement computations briefly discussed in Chapter I. These computations will be described in greater detail later.

The other input constraints were also briefly mentioned in Chapter I and consist primarily of data formatting and translation problems. The data formatting problems are: 1) reversed data where the least significant bit of the data value is transmitted last; this requires a bit for bit reversal of the data value; 2) dispersed data where parts of the data value are found in different telemetry data words; this requires a bit-by-bit assembly of the data value from several telemetry data words; and 3) complemented data bits; this requires selected bit complementation.

The data translation constraints consist primarily of error checking computations. Data parity checks can be generated by either the spacecraft telemetering system or by the experiment measuring system of the spacecraft. These

checks can be either even or odd parity. If the telemetering system imposes the parity, then the parity check must be done on the bits in the order which they were transmitted. If the experiment system imposes the parity, then the parity check must be done on the bits after the data value has been assembled.

Counters appearing in the data should be checked for the continuity of the count. These counters may be either forward counting or backward counting. The telemetry preprocessing system should be capable of establishing the correct count value, flagging counts that are received in error, and keeping a record of the number of counts that are received in error.

Finally, since the frame synchronization pattern is the primary measure of the received error rate and is used for data quality assurance purposes, the telemetry preprocessing system should be capable of performing all the necessary error computations on this pattern. These statistics are collected on a frame-by-frame basis, as well as in an overall cumulative form. The statistics to be collected are: 1) a bit-by-bit error distribution, 2) a one-to-zero error distribution, and 3) the total number of errors.

In addition to the input constraints on the preprocessing system, the host digital computer which receives the data imposes output constraints. Since the main purpose of the preprocessor system is to free the host computer from those functions that are not directly related to the processing of sensor data, care must be taken to avoid additional non-related processing tasks which would be required to accommodate the output of the preprocessor. Thus the preprocessor output must be compatible with the host computer's internal data format. If the preprocessor were to appear to the host computer as a standard peripheral device, the programming impact on its operating system to accommodate the preprocessor could be minimized. Further, the data storage scheme used by the telemetry preprocessing language should be compatible with the data retrieval scheme

used by the host computer's data processing program; i.e., an operation similar to a COMMON statement for FORTRAN programs should be implemented. While many of these functions are hardware related and are handled by the preprocessor hardware design (described in the next Chapter), the data storage allocations and contents are described in the telemetry preprocessing language.

Finally, as in any language, certain instructions are provided for programming convenience. These are computed branch instructions provided to alter the execution sequence of instructions, conditional branch instructions provided to alter the execution sequence dependent upon conditions in the data, and looping instructions provided to allow repetitive operations to be concisely stated.

All of the above constraints and conditions described in this introduction form the context within which the telemetry preprocessing language is intended to operate. To facilitate the coding of statements in this language a compiler was designed and coded in the APL language (references 10 and 11). For simplicity, this compiler assumes that the language statements are to be input on 80 column cards. Hence, general field delimiters are not used; instead, column positions and blank columns are used to delimit statement fields. The language is context dependent; hence, not all statements have the same number of fields. However, all cases are unique; thus no ambiguities can arise. The output of this compiler is an operations table which is a coded set of the operational steps that are required to implement the statement of the language. This table will be described later in this chapter.

The compiler contains an extensive set of error messages to assist the programmer in detecting and correcting syntax errors. It also allows the programmer to use symbolic addressing and symbolic address computation. Hence, by the use of descriptive names for the sensor value addresses, it is possible to

have the telemetry preprocessing language program appear to a knowledgeable reader as a shorthand English description of the telemetry data frame.

B. The Language

This section describes the syntax of the telemetry preprocessing language along with those compiler functions necessary to implement that syntax. The elements of the telemetry language consist of all the capital letters, the digits zero to nine, and the special characters: + - ÷ ( ) / , . = and blank. These elements are then grouped to form variables which are used to either index data values, to form data destination addresses, or to specify the operations to be performed on the telemetry data. The telemetry language is logically divided into two segments: the set-up descriptors and the frame descriptors.

The set-up descriptors form a table which contains the parameters required to configure the frame synchronization data acquisition system discussed in the introduction. This segment of the language contains statements. Such statements, being the specifications of operations to be performed, consist of three fields: a location field, an instruction field, and a parameter field. The names given to these statements derive from the content of their instruction fields. The segment begins with a FORMAT statement and ends with an END statement. The location field contains the identifier of the telemetry data frame being described. The location fields of all other statements in this segment are ignored. The location field is considered to be nine elements long and is blank-filled to that size with the information being left-justified.

The identifier field is 10 elements long and begins in column 10 of the input card. Table 2-1 contains a list of the parameters identified in this segment.

FSP denotes the right-justified frame synchronization pattern. The pattern which appears in the parameter field of the statement may be an octal number which is prefixed by the letter "o", a binary number which is surrounded by

parentheses, or a combination of the two. In any case, the binary equivalent of this parameter must represent the exact pattern right-justified. In this manner the compiler can determine the number of bits to use in setting the preprocessing system for the required error calculations and also what pattern to send to the frame synchronizer.

Table 2-1. SET-UP SEGMENT RESERVE WORD LIST

INSTRUCTION
FORMAT
FSP
CODE
BIT RATE
BITS/WORD
WORDS/FRAM
TAPE SPD
END

CODE specifies the telemetry modulation type to be used by the bit synchronizer and is an optional parameter. The legal modulation types are: SPPH, BIPH, RZ, NRZ, NRZM, NRZL, and NRZC. These codes are described in the standards (reference 3).

BIT RATE is the telemetry transfer rate at which the bit synchronizer is to run, and is an optional parameter. The value in the parameter field may be a decimal, octal, or binary number.

BITS/WORD specifies the number of bits in a telemetered data word. This parameter is required since the preprocessing system uses it to determine how to store the telemetry data in its internal structure.

WORDS/FRAM specifies the number of telemetered data words in a telemetry frame. This parameter is required so that the preprocessing system can allocate its internal resources.

TAPE SPD is the speed to be used in reading an analog tape and is an optional parameter. The legal speeds are: 120, 60, 30, 15, 7-1/2, 3-3/4, and 1-7/8 inches per second.

All of the necessary parameters must be present when specifying the set-up description segment; otherwise, an improper format syntax error is encountered.

The frame description segment contains the instructions which form the main working portion of the language. These instructions form three functional groups: a control group, a storage allocation group, and a data handling group. This segment begins with a FRAME statement and ends with an END statement. The location field of the FRAME statement identifies what data frame is being described. The name located here must match the name in the location field of the FORMAT statement defining the set-up table for this telemetry data frame.

All the statements of this segment except the END statement contain a nine element location field which may be blank. Further, with the exception of the FRAME statement, these location field variables may be subscripted. When subscripting is used, the assigned location of the variable is used as a base address to which the computed value of the subscript is added to yield a final storage address.

The storage allocation group of instructions contains three fields: a location field, a type field, and a parameter field. Since it is being assumed that there are two independent types of storage in the preprocessing system (a program store and a data store), and that the contents of these storages may not be mixed between program and data, two types of storage allocation instructions are required.

The storage allocation instruction that specifies addresses in the program memory is the CONTINUE statement. The location field of this statement contains a symbolic name of an address in the program memory. When this name is referred to by other statements in the language, its value will be the address of the statement following the CONTINUE statement in the program.

The storage allocation instruction that addresses only the data memory is the DIMENSION statement. This statement uses all three fields of this group. All variable names which reference the data memory must be dimensioned before they are used in a statement. The parameter field of this statement denotes how many consecutive data storage locations are to be assigned to this name.

The control group of instructions contains three fields: a location field, an instruction field, and a parameter field. This group of instructions deals primarily with bookkeeping, decision making, and order of execution types of operations. These instructions are used to compute and assign values to index registers to perform both conditional and unconditional branching operations, and to form program loops. Inclusion of these types of operations enables programs to be written in compact form and also allows conditions within the data to alter the program execution sequence. Table 2-2 lists the instructions included in this group.

Table 2-2. CONTROL INSTRUCTION GROUP

LOCATION	INSTRUCTION	PARAMETER
	⟨Name⟩ = ⟨arithmetic expression⟩	
	GO TO	⟨location⟩
	REPEAT	F, V = I, E
	IF (⟨logical expression⟩)	TRUE
N	EQUATE	0

The first expression in the table is an index definition statement as delimited by the presence of the equal sign. The name to the left of the equal sign is the symbolic name of the index being defined. The arithmetic expression on the right may have any number of levels of parentheses, and any legal (to be defined when the compiler is discussed) combination of adds, subtracts, multiplies, and divides. The variables appearing on the right may be decimal numbers or the symbolic names of previously defined variables. This statement may have a

location symbol; if it does, the value of the index as computed from the arithmetic expression will be placed in the data memory location specified by the location field.

The GO TO, REPEAT, and IF instructions must have null location fields. The names in the GO TO and IF instructions must appear as a location expression of a CONTINUE statement. The GO TO instruction is an unconditioned branch to the specified location. The IF instruction is a conditional branch instruction. If the logical expression (to be defined) is true, then, the branch to the specified location is executed; otherwise, execution continues with the next instruction.

The REPEAT instruction is used to perform a looping operation. The loop index (V) is specified by name. The initial value of the index is the value of the first simple arithmetic expression (I). The final value which will cause the loop to be exited is the value of the second simple arithmetic expression (E). The REPEAT instruction is used to repeat the next F instruction in the program. The way in which the loop operates is as follows: the loop index is incremented by one and compared to the final loop value for the equality condition every time the end of the loop is reached. When equality occurs, the loop is exited at the next instruction beyond the end of the loop.

Two important restrictions must be remembered. First, if the variable is not a decimal number, it must be the name of a previously defined index. Second, the loop index may be altered within the loop, but care must be taken to insure that equality will result at the end of loop test.

The EQUATE statement must have a name in the location field. This name must match the name of a frame identifier. This statement is used to define a frame which is simply the reverse of another defined frame; a condition commonly prevalent when spacecraft recorders are used.

The index definition instruction may have a location symbol; if it does, the value of the index as computed by the definition of that index will be placed in the specified data memory location.

A note on the card formats for these instructions, the GO TO and REPEAT instructions have arguments which must begin in column 20. All others begin in column 10 and continue until a blank is encountered.

Logical expressions are formed by a set of arithmetic expressions whose values are either zero or one, and variables whose values are either zero or one connected by the logical relators: EQUAL, NOT EQUAL, LESS THAN, LESS THAN OR EQUAL TO, GREATER THAN, GREATER THAN OR EQUAL TO, AND, OR, and NOT. One important rule of operation must be remembered: the order of evaluation is left to right with no precedence among the operators. However, note that in both arithmetic and logical expressions, any level of parenthesis is allowed and here the evaluation sequence may be altered. Also, in logical operations, the unary operator NOT means complement the operand to its right. If that operand is a variable, the value of that variable will be complemented prior to comparison.

The data handling instructions form the main working section of the language. They are used to extract sensor data values from the telemetry data words and to perform all of the data manipulation functions that are required. These instructions contain two fields: a location field and an operation field. The location field contains a pointer to the data memory address where the assembled data value is to be stored. The instructions of this group are listed in Table 2-3.

Table 2-3. DATA HANDLING INSTRUCTIONS

LOCATION	INSTRUCTION
	SYNC, M (A, B, C) ⟨location name⟩, M (A, B, C) WORD, M (A, B, C) SUB, M, K, O (A, B, C)

In this table, M is a modifier operation which may be L (to indicate that the word has its least significant bit first and, an end for end bit reversal must be done) and/or one of the following: PET, PEA, POT, POA, or blank. The modifiers PET, PEA, POT, and POA specify that the associated data value is to be checked for a parity error. The parity bit to be used for comparison is the bit specified by the word control section of the instruction containing one of these modifiers. The type of check to be performed is specified by E for even parity or O for odd parity. The time that the check is to be performed is specified by either T for before manipulation or A for after manipulation. The result of the parity check is to set the sign bit of the specified data memory word to 1, if a parity error is detected and, to 0, if not.

The data value to be checked is specified by those instructions that are linked to the one containing the parity check modifier. Instructions may be linked by either plus or minus signs. The linkage symbols are fully distinguished from their arithmetic counterparts by their location in the instruction stream.

The plus linkage implies that the bits specified by the instruction immediately following will be appended to the right of the bits already extracted. The minus linkage implies that the complement of those bits will be appended. In this manner a new data value is formed from the input bit stream.

The bits to be manipulated in the above manner are specified by the parameter set (A, B, C). A is the telemetry word number to be processed. This parameter can be a number, a variable, or a simple two variable arithmetic operation. B is the number of the starting bit within the telemetry word. C is the number of bits to take from the telemetry word. Taken together these parameters define a new information word derived from the original stream.

SYNC is the instruction used to indicate the location of the frame sync pattern (FSP) in the telemetry data frame. The SYNC instruction doesn't place the

frame sync code in the data memory; instead, it retrieves the frame sync code from the format table word and compares it with the data value bit by bit. It then replaces this data value in the data memory by a set of words representing: 1) the total number of errors in the frame sync pattern, 2) the number of pattern ones in error, and 3) the "exclusive or" of the pattern and the received frame synchronization code. In addition to this, it keeps a cumulative set of registers containing the statistics of items 1 and 2 above.

In the location name operation, the name must be a defined data memory address. Hence the parameter set will operate on the data memory locations specified by name, whereas in the other data handling, the operations are performed on the input data set.

The WORD instruction is used to extract sensor data values from the telemetry data words.

SUB is the instruction used to indicate the presence of a subcommutation counter and initiates the accumulating of statistics and the smoothing of this counter; i.e., error flagging and a continuity check.

The SUB instruction and the SYNC instruction are the only two instructions which involve more than one data frame in their execution. The SUB instruction is designed to determine whether or not the sequence of values received for a data counter is correct. It also gathers error statistics concerning the condition of the data counter. In order to perform these functions, this instruction must determine what the expected count value should be. This is done by searching for three consecutive received counts. The count mode (K) of the instruction specifies whether to look for forward (F) or backward (B) counting. Further, the counter modulus (O) in the instruction specifies the range of the data counter.

This instruction forms a value for entry into the data memory in the same way as the word instruction, but with two exceptions. While searching for three

consecutive count values to establish the counting sequence, the sign bit of the stored data word is set. Then, once a sequence has been established, the expected count value will replace any count value received in error, with the sign bit being set to denote the replacement of the received data. A count sequence, once established, may be broken by the reception of three consecutive counts received in error.

This instruction maintains a set of error counters throughout the processing operation. These error counters are read out to the host machine upon request and represent: 1) the number of times a new sequence had to be established, 2) the number of counts in the search state, 3) the number of erroneous counts received while in a sequence, 4) the total number of counts received while in a sequence, and 5) the number of attempts made to establish a sequence.

Comments in the language, when punched on an 80 column card, can be entered either with an asterisk (*) in column 1 or after a blank at the end of a language statement. Comments may appear anywhere within the telemetry language. A final note on card formats is that, if the linked data handling instructions or an index definition statement forms a character string too long to fit on one card, an asterisk (*) in column 80 signifies that the entire next card is a continuation card. As many continuation cards as needed may be used. However, on a continuation card an * in column 1 does not signify a comment card. In this case, the * is interpreted as an operator.

As an example of the use of this language, consider the direct digital frame format of the OAO-A2 spacecraft (reference 14).

As is shown in Table 2-4, the set-up table indicates that the modulation code is NRZC. The 64 words of the telemetry frame are each 32 bits long and and are transmitted to the ground at a rate of 50,000 bits per second. Notice that in the FSP specification the "o" preceding the numbers indicates that the

octal number system is to be used. However, the "( )" near the end indicates a switch to the binary system. The resulting bit string is the FSP right-justified; i.e., 11100010010000111011010001110110.

Table 2-4. OAO-A2 DIRECT DIGITAL FRAME FORMAT

LOCATION	INSTRUCTION	PARAMETER
DD	FORMAT	
	FSP	O7044166435(10)
	CODE	NRZC
	BIT RATE	50000
	BITS/WORD	32
	WORDS/FRAM	64
	END	

Table 2-5 indicates a further breakdown of the bit stream into 8-bit elements, where the first 4 elements are the FSP, the 5th element is the TV line number, and the 6th to 256th are the TV intensity elements. Further, notice that the data is least significant bit first and, in the case of the intensity elements, every odd number bit is complemented. Preparing this information for use in a general purpose computer requires a lengthy and complex program. However, as Table 2-6 shows, the telemetry preprocessing language makes it rather simple to describe the reconstruction of this data.

Table 2-5. DIRECT DIGITAL DATA ELEMENTS

ELEMENT	BIT 1 2 3 4 5 6 7 8
1	1 1 1 0 0 0 1 0
2	0 1 0 0 0 0 1 1
3	1 0 1 1 0 1 0 0
4	0 1 1 1 0 1 1 0
5	$y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7$
6-256	$i_0 n_1 i_2 n_3 i_4 n_5 i_6 p$
where:	$y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0$ is a binary TV line number.
	$n_x = i_x$ complemented.
	$i_6 i_5 i_4 i_3 i_2 i_1 i_0$ is a binary intensity value.
	p is the even parity as transmitted bit.

Table 2-6. DIRECT DIGITAL DATA PROCESSING PROGRAM

LOCATION	INSTRUCTION	PARAMETER
DD	FRAME	
STAT	DIMENSION	3
LINE	DIMENSION	1
ELMT	DIMENSION	251
STAT	SYNC (1)	
LINE	WORD, L(2,1,8)	
ELMT(1)	WORD, PET(2,16,1) + WORD(2,15,1) - WORD(2,14,1) + WORD(2,13,1) - WORD(2,12,1) + WORD(2,11,1) - WORD(2,10,1) + WORD(2,9,1)	
ELMT(2)	WORD, PET(2,24,1) + WORD(2,23,1) - WORD(2,22,1) + WORD(2,21,1) - WORD(2,20,1) + WORD(2,19,1) - WORD(2,18,1) + WORD(2,17,1)	
ELMT(3)	WORD, PET(2,32,1) + WORD(2,31,1) - WORD(2,30,1) + WORD(2,29,1) - WORD(2,28,1) + WORD(2,27,1) - WORD(2,26,1) + WORD(2,25,1)	
	I = 4	
	REPEAT	5,X=3,64
ELMT(I)	WORD, PET(X,8,1) +WORD(X,7,1) - WORD(X,6,1) + WORD(X,5,1) - WORD(X,4,1) + WORD(X,3,1) - WORD(X,2,1) + WORD(X,1,1)	
ELMT(I+1)	WORD, PET(X,16,1) +WORD(X,15,1) - WORD(X,14,1) + WORD(X,13,1) - WORD(X,12,1) + WORD(X,11,1) - WORD(X,10,1) + WORD(X,9,1)	
ELMT(I+2)	WORD, PET(X,24,1) +WORD(X,23,1) - WORD(X,22,1) + WORD(X,21,1) - WORD(X,20,1) + WORD(X,19,1) - WORD(X,18,1) + WORD(X,17,1)	
ELMT(I+3)	WORD, PET(X,32,1) + WORD(X,31,1) - WORD(X,30,1) + WORD(X,29,1) - WORD(X,28,1) + WORD(X,27,1) - WORD(X,26,1) + WORD(X,25,1)	
	I = I + 4	
	HALT	
	END	

Notice that the SYNC instruction refers to word 1. This causes the system to refer to the DD format BITS/WORD statement and take the first 32 bits of the bit stream. The system then compares these bits with those of the FSP statement and gathers its statistics.

Notice the statement LINE. This statement causes bits 1 to 8 of telemetry word 2 to be reversed and placed in a location (line) reserved for the TV line number.

Next, the intensity elements are dealt with. The first word instruction indicates that this is the even parity bit as transmitted and causes a parity check on this element. The rest of the WORD instructions in the statement cause the uncomplemented intensity element to be found in the appropriate element array location in most significant bit first integer format.

Note that the repeat loop and the index counter are used to save the programmer from the burden of specifying every element. This program would cause an array to be formed which, when output, would have the data storage allocations shown in Table 2-7.

Table 2-7. OUTPUT DATA FORMAT

WORD NUMBER	CONTENT
1	Frame Sync Error pattern
2	Number of FSP errors in this frame
3	Number of FSP ones in error
4	TV line number in integer form
5-256	Parity error bit and 7-bit intensity values 1-251

The first 3 words generated result from the SYNC instruction. The rest are a result of the WORD instructions.

In summary, the telemetry preprocessing language described and illustrated above is proposed as a more natural means of manipulating telemetry

data streams. This is because the information content of these data streams is described by using the telemetry format specification to extract the data. A further benefit is that the code generated in this manner inherently contains the internal bookkeeping required to extract the data and, thus, is a much simpler code. Hence, it is easier to debug, maintain, and modify as opposed to either machine language coding or FORTRAN coding.

C. The Compiler

The telemetry preprocessing language compiler is a set of subroutines written in the APL language which converts a program input data stream into an operations table to be described.

The APL language was chosen for its availability, interpretive implementation, and its powerful set of operators. These features considerably shortened the development and debug times in designing and implementing the compiler and allowed compiler concepts to be explored without excessive concern for internal bookkeeping chores.

Because of the simple nature of the telemetry language, a simple left to right single pass compilation algorithm was able to be used. This algorithm is described in reference 17.

The main body of the compiler is driven by a reserve word list (reference 16) consisting of the instruction set and operators. This list drives the compiler to the appropriate subroutine, which parses the particular instruction being examined. Most of these routines are context sensitive (reference 15) since most of the instructions have a rigid format.

The most interesting of these routines is the arithmetic and logical parser. This routine is driven by two tables and is used to form a reverse Polish string (reference 12) of these expressions. The first of these tables (Table 2-8) is the Input String Precedence Table. This table represents an exhaustive listing of the

allowed order of elements in the input string. If an error is detected here, an error message is generated and the entire expression is discarded.

Table 2-8. INPUT STRING PRECEDENCE

			HEAD OF INPUT						
	SYMBOL	CLASS							
SYMBOL			relation	logical	.NOT.	arithmetic	variable	(	)
CLASS			1	2	3	4	5	6	7
last input	relation	1	x	x	x	+/-			x
	logical	2	x	x		x			x
	.NOT.	3	x	x		x			x
	arith-metic	4	x	x	x	x			x
	variable	5			x		x	x	
	(	6	x	x		+/-			
	)	7					x	x	

x = not allowed
+/- = plus or minus only
relation = .NE., .EQ., .LE., .GE., .LT., .GT.
logical = .AND., .OR.
arithmetic = +, -, *, ÷

Table 2-9, the Parser Decision Table, is used by this routine to assign weight to the elements at both the top of the storage (TOS) string and the head of the input string (HIS).

Table 2-9. PARSER DECISION TABLE

SYMBOL	TOS	HIS
logical/relator	2	1
.NOT.	4	3
arithmetic	6	5
variable	8	7
(	0	9
)		0

Elements are then transferred to the output reverse Polish string according to the simple algorithm:

HIS>TOS means HIS moved to TOS and drop HIS.
HIS = TOS means drop both HIS and TOS.
HIS<TOS means TOS moved to output and dropped.

This relatively simple system adequately parses all of the logical and arithmetic expressions found in this language.

The compiler is structured into two major divisions: the format parser and the frame parser. Both of these divisions accept card images as input and produce the above mentioned outputs. They scan the input character stream for illegal conditions and produce error messages when such conditions are found.

The format parser is entered upon recognition of a FORMAT statement. At this time the location field is placed in the format identifier table. During this process, it is checked for multiple entries. If a case of multiple identifiers is encountered, an error message is generated and the generated format table is discarded. Also in the error case, the input data stream is read and ignored until the END statement signifying the end of the format definition is found. The format identifier table is used to set up a linkage between the format table and the frame operations table.

The format parser reads and decodes the statements of the format segment. The output generated by the parser is the format table. This table is checked for multiple entries. If a multiple entry occurs, an error message is generated and the latest parameter replaces the old one. When an END statement is found, the output format table is checked for the presence of all necessary parameters. The necessary parameters are: BITS/WORD, WORDS/FRAM, and FSP. If any one of these is missing, an improper format specification message is generated and the format table is discarded. If all the parameters are present, then the error messages (if any) are output and the format table as

shown in Table 2-10 is output. The word number column of the table refers to the nine words which comprise a format table.

Table 2-10. FORMAT OUTPUT TABLE

WORD NUMBER	CONTENT
1	Identifier
2	Number of bits in the FSP
3	Number of bits in a telemetry word
4	Number of telemetry words in a frame
5	Right most 16 bits of the FSP in octal
6	Left most 16 bits of the FSP in octal
7	Encoded telemetry modulation code
8	Telemetry bit rate
9	Encoded tape playback speed

The codes for the two encoded words are listed in tables 2-11 and 2-12. The last 3 words of the format output table are optional parameters. If they are absent, a zero is inserted or, in the case of the codes, a seven.

Table 2-11. ENCODED MODULATION CODE VALUES

INPUT	ENCODED VALUE
SPPH	0
BIPH	1
RZ	2
NRZM	3
NRZL	4
NRZ	5
NRZC	6

Table 2-12. ENCODED TAPE PLAYBACK SPEEDS

INPUT	ENCODED VALUE
120	1
60	2
30	3
15	4
71/2	5
33/4	6
17/8	7

The frame parser scans the input program statements and produces an operations table of the format shown in Table 2-13. The parser is entered upon recognition of a FRAME statement. During this process, the location field is

checked for multiple entries. If such a case is found, an error message is generated and this entire segment of the program is disregarded.

The operation table being produced as output from the frame parser and the format table produced as output from the format parser represent the total output of the compiler. This output is a coded set of operations representing the input statements. The codes used are explained in Table 2-13. The form of this output is suitable for execution by a telemetry preprocessing system. It may be produced as a magnetic tape, a card deck, or directly inserted into the preprocessor depending on the configuration of the installation.

Table 2-13. OPERATIONS TABLE

LOC	V1	OP	V2	I	L	A	V1	OP	V2	R	V1	OP	V2	V1	OP	V2
XXX	BXXX	O	BXXX	X	X	XXX	BXXX	OI	BXXX	XXXX	BXXX	O	BXXX	BXXX	O	BXXX

XXX is the numerical value defined as:

- A symbol address if no prefix is present
- A value as specified by the prefix

B is the prefix encoded as:

- 1 means the value is a number
- 2 means the value is a data address
- 3 means the value is a program address

O is a type 1 operation code meaning:

- 1 means add
- 2 means subtract
- 3 means multiply
- 4 means divide

OI is a type 2 operation code meaning:

- 1 means add
- 2 means subtract
- 3 means multiply

Table 2-13. OPERATIONS TABLE (Cont)

- 4 means divide
- 5 means equal
- 6 means not equal
- 7 means less than or equal to
- 8 means greater than or equal to
- 9 means less than
- 10 means greater than
- 11 means or
- 12 means and
- 13 means not

I is the instruction operation code:

- 0 means HALT
- 1 means GO TO
- 2 means REPEAT
- 3 means IF
- 4 means equation
- 5 means location expression
- 6 means SYNC
- 7 means SUB
- 8 means WORD

L is the linkage flag defined by:

- 0 means no linkage
- 1 means + linkage to the next instruction
- 2 means - linkage to the next instruction

Table 2-13. OPERATIONS TABLE (Cont)

A is an address field used as:

instruction	meaning
GO TO	branch address
REPEAT	loop index address
IF	branch address
equation	address of index being defined
location expression	address of old location symbol

R is either the result of an arithmetic expression or a modifier code as:

Code	Modifier
0	end of modifiers
1	PET
2	POT
3	PEA
4	POA
5	L
6	F
7	B

CHAPTER III

THE TELEMETRY PREPROCESSING SYSTEM

This chapter contains a functional description of a machine designed to process the statements of the telemetry preprocessing language described in Chapter II. The design of the machine is a natural development based upon the format of the language, as will be demonstrated when the system block diagram is derived in a later section. However, the method used to implement this design is a new and powerful approach to system implementation. This method, known as a functional memory, will be described in this chapter. A hardware design of a functional memory device is presented in Appendix A.

The present chapter has nine sections. The first section describes the functional memory device. Next the telemetry preprocessing system block diagram is derived. Sections three to eight describe the six subsystems derived in the block diagram. Finally, section nine presents a summary of this approach, discussing the advantages and disadvantages of the functional memory module and the telemetry preprocessing system.

A. The Functional Memory Module

The telemetry preprocessing language described in Chapter II will be used to define a hardware system. In using the language to guide the hardware design, many approaches could be taken. The main consideration in choosing an approach is that the system should have a minimum impact on its operating environment. As was pointed out in Chapter I, this environment is highly variable in that it depends on what host computer is chosen and on what telemetry data acquisition system is being used. To accommodate this variability with minimum impact, a microprogrammed approach shows the most promise. Further, when considering the various microprogrammed approaches available and matching them

with the requirements of the preprocessing system, the most promising method is a cellular logic array. This method offers the required flexibility to accommodate the various environments envisioned for the preprocessing system, while simultaneously taking advantage of the lower implementation costs that large scale integration offers.

The hardware technique chosen to implement this cellular logic array method consists of a basic building block called a functional memory module. The term functional memory denotes a device used to generate Boolean functions in a memory system. In this device, a cellular memory array is arranged so that each cell of the array can be either an associative memory cell or a conventional memory cell. In order to generate Boolean functions, the value of the input variables is gated into the memory array. The cells of the array are preprogrammed with the value of the variables required for each term of the function. During this phase of the operation (called a search), the value of the input variables is "associated" with the value of the terms and, for every match that is found, a register latch is set. Then the contents of this register are used as a driving signal to read all the addresses of the memory that were selected. The data at these locations is preprogrammed to represent the bit pattern of the function's output when the corresponding input variables are present. In this way a Boolean function can be created by programming a memory device rather than by wiring a combination of logic devices as is normally done.

The module that was designed in this work has 16 associative functional words of the type described above, each of which is 20 bits wide. Hence, functions with 16 or less terms and with a total number of input and output bits less than or equal to 20 can be implemented in a single module. As is explained in detail in Appendix A, the terms selected during a search phase can be logically combined prior to setting the register latch. Further by appropriate programming,

use can be made of "don't care terms", and carry terms (from one word to the next) can be generated to reduce the Boolean functions. Thus by using a combination of the above operations, functions with more than 16 terms can be readily reduced to fit in the memory module.

The module is 27 bits wide and contains 64 conventional addresses. The upper 7 bits are not used for function generation, but are operation codes for the 45 microinstructions implemented in the module. These microinstructions are used to clear the various registers of the module, shift data, selectively move data from the module to several other modules, connect other modules to this one, transfer control to another module, and other housekeeping functions.

This module is designed to be suitable for construction using large scale integrated circuit technology. In order to implement this design on a single chip, a constraint must be imposed on the number of input-output pins available. Hence, the number of signals used for communications is restricted. To overcome this restriction during the generation of the various required functions, two powerful techniques were employed.

The first is the method of connecting functional memory modules to the communications bus for the purpose of cooperating in function execution. Since it is the intent of this method of design that most functions be implemented on a small number of modules, an instruction is provided which is capable of forming a connection between 16 modules and the one sending the command at one time. However, this method only leaves three bits available for addressing purposes because of the pin constraint on the modules. Thus, a maximum of 128 modules are available for function generation.

The second technique is the distribution of the control function. A module whose control flip-flop is set is in the control state. In this state, the module is able to control the five communication buses. Hence, it may start instruction

sequences in other modules, initiate functional cycles, transfer data, etc. The most important thing it can do is transfer this control state to all other modules connected to the bus and simultaneously retain control or relinquish it. In this manner, many independent control sequences may be established.

These independent control sequences group the modules under their direction into disjoint sets. When the results of several independent operations have to be merged into a common sequence, the designer who is coding the control sequences must carefully avoid conflicting operations (e.g., separate data transfers to a common destination). Further, when several control sequences are being reduced to one control sequence, only one of the original control states is allowed to be transferred to the new sequence. The others must be terminated by transferring their control state to an empty bus.

B. The Preprocessor Block Diagram

It will be shown how the device described in the previous section can be used to implement a telemetry preprocessing system. This example will demonstrate a very effective usage of LSI technology, since a single chip design is used to implement all the functions of the system.

The language described in Chapter II, particularly as shown in Table 2-13, will now be used as a guide to derive the major subsystems which make up the telemetry preprocessor.

The heading of the operation table (Table 2-13 of Chapter II) is repeated here for convenience.

LOC V1 OP V2 I L A V1 OP V2 R V1 OP V2 V1 OP V2

This heading identifies the various operation fields that an instruction of the telemetry preprocessing language contains. Upon examination of these fields, desirable simultaneous operations are noted. Obviously, the instruction must be decoded. Further, the four fields under the headings V1 OP V2 can be

evaluated separately at the same time. While this requirement is not strictly necessary, it does serve to reduce the instruction execution time and further it represents a natural environment of the language. Consequently, it will be shown that the combination of direct and indirect implications of this table will lead to a block diagram of the desired preprocessing system.

The direct implications considered thus far show the need for an instruction decoder and for an arithmetic unit consisting of four arithmetic and logical units (ALU's). Another direct implication of the table is contained in the R field. As was described in Chapter II, this field contains several parameters which require that specialized functions be performed (e.g., parity checking, error statistics, etc.). Hence, a special functions unit to implement all those functions which are not inherently handled by an ALU is a characteristic of the language.

The design concepts that are indirectly implied by the table are the concepts of communications and storage. Obviously, an instruction decoder implies the existence of an instruction stream. This in turn implies that the system contains a program memory. Further, since the instructions are designed to operate upon telemetry data, a data storage is required. Fundamentally, these storage systems could be combined. However, there is no need for instructions to be intermixed with data. Furthermore, the flow paths of these two types of information are considerably different in that, instructions must direct the system's activities, while data undergoes transformations initiated by the system. Hence, it becomes a simpler concept to treat them as independent functions. Further, when the data becomes transformed as a result of the processing program, the storage requirements tend to increase. To keep system costs in line while satisfying this increased requirement, a core storage to hold the processed data is indicated.

Finally, in the area of communications, a method of getting information into and out of the system must be provided. This is accomplished by including a set of input/output routines to interface to the outside world. These six subsystems that have been identified lead to the block diagram of Figure 3-1.

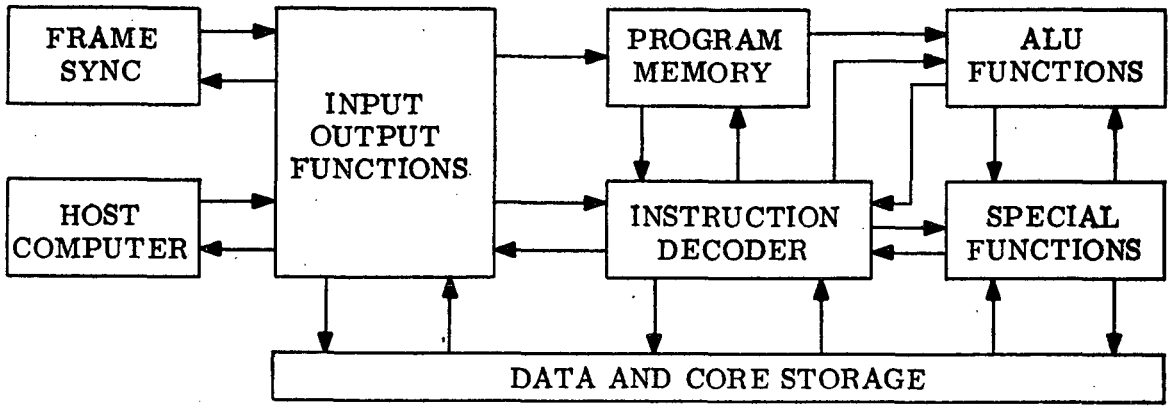


Figure 3-1. PREPROCESSOR BLOCK DIAGRAM

The arrows in this figure represent the various ways that information flows within the system. The input-output functions are responsible for performing the necessary "handshaking" functions with the external equipment, tranforming data word sizes into the internal word size structure, and storing data in the appropriate locations in storage. Data and commands must flow between the host computer and the preprocessor system. These commands refer to the various control functions that are necessary to: load the program memory, start a preprocessor program, read error statistics, or initiate a frame synchronizer set-up function. Thus, two-way flow is needed between the preprocessor and the external equipment.

The instruction decoder is responsible for decoding the program instructions; initiating data transfers between the ALU functions and the special functions; starting the required control sequences of those special functions needed

in the instruction execution sequence; receiving the external commands from the host computer and initiating their execution; and storing and retrieving the error statistics in the core storage.

During the execution of a repeat loop, this unit maintains the loop index, counter, keeps track of the number of instructions in the loop, handles the branch to the head of the loop, and tests for loop exit conditions. During SYNC and SUB instructions this unit directs the gathering of statistics and all other required tests. Finally, upon completion of the processing of a data frame, the unit transfers control to the output routines so that the data may be transferred to the host computer.

The program memory is used to hold the data formatting programs. It is a 1,000 location memory system where each location is 7 words of 20 bits each. These 7 words hold the information described in Table 2-13 of Chapter II. The fields are packed according to the format of Table 3-1.

Table 3-1. PROGRAM MEMORY LAYOUT

PROGRAM WORD	CONTENT
1	LOC, V1
2	V2, OP, 1, L
3	A, R
4	V1, V2
5	V1, V2
6	V1, V2
7	OP, OP, OP

The operators of word 7 go with the variables of words 4, 5, and 6, respectively; these three sets represent the three parameter fields of Table 2-13.

The data storage and core memory subsystems are used to store the telemetered data. The data storage subsystem is a two dimensional array which

holds the incoming data stream. This data is stored, one telemetry data word per location. It is from here that the system extracts telemetry words and parts thereof for formatting purposes. This is accomplished by the addressing algorithm. In this algorithm, if x is the telemetry word number desired and if n is the number of bits to take from the word, then the system simply takes n bits from location x of data storage and right justifies them.

The core memory is used to store the formatted data array with every word assumed to be in integer format and the most significant bit being to the left. Further, in parity checking decisions, the sign bit is on (negative) if a parity error was detected. This array becomes an image of the desired data array in the main computer's memory.

The ALU functions each form an implementation of a central processing unit. The operations implemented in these elements are: +, -, *, ÷, .NE., .EQ., .LE., .LT., .GE., .GT., .AND., .OR., and .NOT.. These elements are primarily used for address computations to locate the telemetry word, the number of bits to extract from that word, and to locate the core memory address into which the data is to be stored. In the case of an IF instruction, ALU 2 performs the programmed operations on the data and makes a true-false decision.

The special functions unit handles all of the required special functions. These are: word reversal, parity checks, forward or backward counter checks, and counter modulus determination. This unit consists primarily of the logic functions required to do these tasks. It is activated by the control unit and places the results of its calculations into the assembly register for later transferal to the core memory.

C. The Input-output Routines

The input-output routines are designed to perform the necessary "handshaking" functions with the host computer and handle the translation processes

necessary to convert the data from the computer's word size structure to the device's and vice-versa. The input routines have the responsibility to determine whether the incoming data is to be a program memory load, a data storage load, or a start system command and take the necessary action. During a program memory load, the input routines translate the data from the computer's internal word size to that of the program memory. They also determine what program memory address the data are to be loaded into and keep track of ensuing addresses.

During a data storage load, the input routines translate the data from the host computer's internal format to a serial bit stream and keep track of where the telemetry data word is to be placed in data storage. During the operation, each telemetry data word is placed in a separate data storage word. During a start command, the input routines determine the starting address of the data formatting program and initiate the instruction decoder unit to start execution from that address.

The output routines are responsible for interrupting the computer upon completion of a data formatting program, identifying the data type, translating the data from the core memory word size to the computer word size and shipping it out, and translating the program memory into the computer's format and transferring it upon request. During a readout of the program memory, the output routines receive a request, a starting address, and an ending address from the input routines. These specified addresses are then translated and read out. The parameters required for this translation (i.e., from the preprocessor word size to the computer's word size) are placed into the microcode of the interface between the preprocessor and the host machine when the device is installed.

D. The Program Memory

The program memory consists of 128 functional memory modules which are structured into 1,000 locations of program memory. Each location of program memory is 7 words wide. Table 3-2 lists the content and destination of the program locations resident in the program memory.

Table 3-2. PROGRAM MEMORY WORD STRUCTURE

WORD	CONTENTS	BIT WIDTH	BIT LOCATION	DESTINATION
1	Base Add. V1	10 10	1-10 11-20	ALU1 ALU1
2	V2 OP I L	10 4 4 2	1-10 11-14 15-18 19-20	ALU1 ALU1 Decoder Decoder
3	A R	10 10	1-10 11-20	Decoder Decoder
4	V1 V2	10 10	1-10 11-20	ALU2 ALU2
5	V1 V2	10 10	1-10 11-20	ALU3 ALU3
6	V1 V2	10 10	1-10 11-20	ALU4 ALU4
7	OP OP OP	4 4 4	1-4 5-8 9-12	ALU2 ALU3 ALU4

As can be seen from the table, the data for the four ALU subsystems is packed into modules 1, 2, 4, 5, 6, and 7. This data is automatically sent to these subsystems every time an instruction is read from the memory.

This device operates on a basic read-execute cycle. After a typical instruction is read from the program memory, it is decoded. Further, the four fields of type V1-OP-V2 are simultaneously acted upon by the four ALU functions. ALU1 is computing an address for data storage in the core memory. The other three ALU's are computing where the required data bits in data

storage are. Once this is done, those bits are extracted from the data storage and the special functions unit performs the required functions on these bits as specified by the instruction. Then the newly formed data word is sent to the core storage and control returns to the program memory for the reading of the next instruction.

Control is passed from the program memory to the instruction decoder after every instruction read. From there, control goes to the control sequence required to execute the instruction. There are two such sequences which require access to the program memory. The first is a temporary storage request. In this request, an address is sent to the memory and control is passed to either the read or write data routine.

The second type of request is a read next instruction cycle. In this cycle, as address is passed to the read program routine and control is passed to it. This address has three sections to it: a 6-bit data address for the interior module address, a 4-bit module number to pick 1 of the 16 modules addressable at any level, and a level number. The address may be one that is loaded into the program memory subsystem as the result of a branching operation, or it may be the next successive address present as a result of the previous operation. However, in the case of a temporary storage request, the address is always directly supplied.

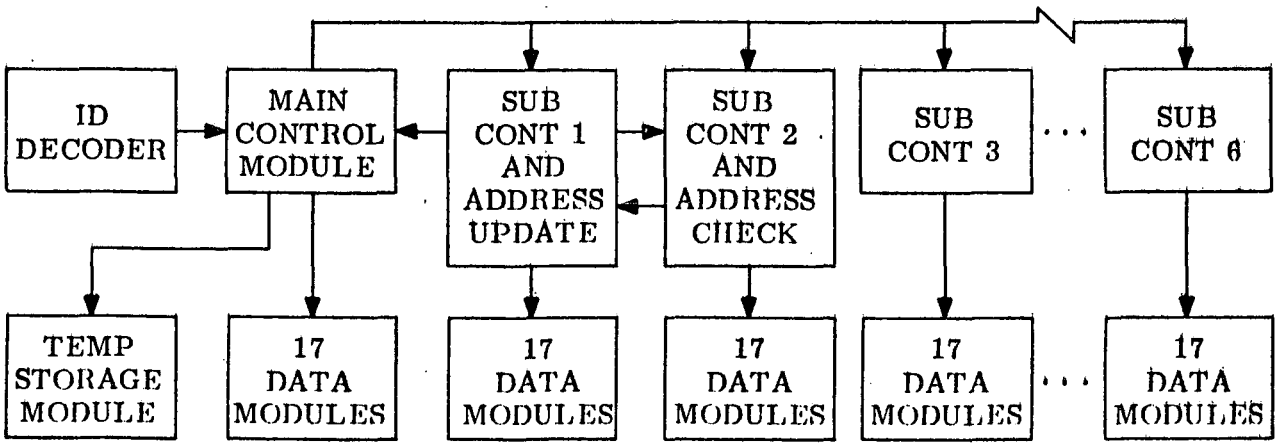


Figure 3-2. PROGRAM MEMORY BLOCK DIAGRAM

The block diagram of the program memory subsystem is shown in Figure 3-2. The address derivation shown there is basically a 10-bit counter whose functional program resides in subcontrol module 1. The address is maintained and updated in subcontrollers 1 and 2. It is sent to the ID decoder where it is translated into a connect command which is loaded into all the subcontroller memories. This command, along with the send address command which the main control module has loaded into the subcontroller memories, specifies what instruction is to be read from the program memory.

Control is then passed to all subcontrollers which, in turn, send the address and a conventional read cycle to the proper data modules. The data modules then receive control to transfer the program word back to the controllers. Finally, the controllers send the data fields to the proper destination and pass control to the instruction decoder. In this manner, the read next instruction cycle ends with the address of the next instruction stored in the address register.

In a temporary storage request, the control module receives an address which is sent to the temporary storage module. Control is then sent to either the read or write routine. In a write request, the data is sent to the storage module and a conventional write is issued. In a read request, control is sent to the storage module after a conventional read cycle has been executed by that module. Then this module returns the data to the requesting module. Finally, control is passed back to the requesting module.

E. The ALU Functions

The ALU functions are implemented by using 25 functional memory modules. The functions contained in these modules are found in Table 3-3.

Each function that this subsystem performs is done through a control sequence. In this sequence, the control function is passed from module to module in a programmed order. The variables and operators are sent to modules 9—12

and control is initially passed to module 10. Module 9 receives variable 1, except in the case of a divide where it receives variable 2. Module 11 normally receives variable 2. In a divide, modules 11 and 12 receive the 32-bit dividend. All other operations are assumed to contain 16-bit operands. The number representation is assumed to be signed binary with 16 bits of significance and bit 17 being the sign bit.

Table 3-3. ALU MODULE FUNCTIONS

MODULE	FUNCTION
1-4	4-bit multiplier table
5-8	Adder table whose input is 3 (4-bit) numbers and 2 carries and whose output is a 4-bit result with 2 carries
9	Communications module which receives variable 1 and holds the result of the computations
10	Communications module which receives the operator and holds the remainder in a divide and the lower 16 bits of the product result
11-12	Communications modules which receive variable 2
13	Counter for the divide algorithm and logical functions
14-18	Comparator modules
10-22	Subtractor modules
23-24	Carry propagator modules
25	Instruction decoder module
Note all of these modules contain portions of the control sequence microcode.	

Module 9 normally returns the result of the operation except in the multiply and divide cases. In the multiply, the product is held in modules 9 and 10. In the divide, module 9 holds the quotient and the remainder is discarded. This is because fractional addresses have no meaning and extensive data processing is not intended to be performed in this unit.

There are 7 control sequences in the ALU subsystem: the input sequence, the add sequence, the subtract sequence, the compare sequence, the logical sequence, the multiply sequence, and the divide sequence. These sequences define all the various functions required by the ALU subsystem.

The input sequence does all the necessary preparatory steps involved in function execution. It begins in module 10 at the point to which initial control is transferred. First the operator is transferred to the decoder; then all the data registers of the subsystem are initiated.

Next, control is transferred to modules 9 and 11 where the input variables are sent to all the data registers that may require them. Control is now passed to the decoder which in turn passes it to the appropriate control sequence. This is done by the decoder module acting as an associative memory. Each 4-bit operator is an address of the start of its control sequence. That address is loaded into the memory address register and the transfer control instruction located there is executed.

The add sequence uses four 4-bit adders to add three 16-bit variables together. The adder adds three variables rather than two because the multiplier to be described later uses this feature. The algorithm is shown in Table 3-4.

This process takes four cycles. First, the 4 adders are cycled. Then, the indicated data transfers take place and the 4 adders are cycled again. The data is now transferred to the carry propagators where the final 2 cycles are performed and the data is sent to module 9 for output.

Table 3-4. ADDER ALGORITHM

CYCLE	ADDER				CARRY PROP.		OUTPUT
	1	2	3	4	1	2	
1	H_{14} H_{24} H_{34} $C_{14}H_4$	H_{13} H_{23} H_{33} $C_{13}H_3$	H_{12} H_{22} H_{32} $C_{12}H_2$	H_{11} H_{21} H_{31} $C_{11}H_1$			
2	C_{14} 0 H_{41}	C_{13} H_4 $C_{23}H_{31}$	C_{12} H_3 $C_{22}H_{21}$	C_{11} H_2 $C_{21}H_{10}$			H_1
3				$H_{41}C_{23}$ H_{41}	$H_{31}C_{22}H_{21}C_{21}$ $C_{41}H_{51}H_{52}$		H_{10}
4				H_{42} C_{41} H_{43}			
Answer is $H_{43}H_{51}H_{52}H_{10}H_1$							

Each of the 4 adders accepts as input 2 carry bits and three 4-bit hexadecimal digits. The output is 2 carry bits and a hexadecimal sum digit. In Appendix A, the search controls of the functional memory module are explained. There it is explained how the interword carries are formed and propagated. Hence, the output equations for the 6 sum bits are

$$H_1 = A_1 \oplus B_1 \oplus C_1 \oplus CI_1$$

$$H_2 = A_2 \oplus B_2 \oplus C_2 \oplus CI_2 \oplus C_{12} \oplus C_{14}$$

$$H_3 = A_3 \oplus B_3 \oplus C_3 \oplus C_{22} \oplus C_{22} \oplus C_{26}$$

$$H_4 = A_4 \oplus B_4 \oplus C_4 \oplus C_{32} \oplus C_{32} \oplus C_{36}$$

$$CO_1 = C_{42} \oplus C_{44} \oplus C_{46}$$

$$CO_2 = C_{52}$$

Where:

$\oplus$ means "exclusive or"

CI_x means input carry x

C_{j2} means the "any 2" carry from word j

C_{j4} means the "any 4" carry from word j

C_{j6} means the "any 6" carry from word j

A_i, B_i, C_i means the ith bit of input A, B, and C, respectively

CO_i means output carry i

H_i means bit i of the output sum

The carry propagator accepts as input a hexadecimal digit followed by a 2-bit carry followed by another hexadecimal digit followed by a single bit carry. This unit outputs a carry bit and 2 hexadecimal digits. The programmed equations are

$$H_{11} = A_1 \oplus CI_{11}$$
$$H_{12} = A_2 \oplus C_{12}$$
$$H_{13} = A_3 \oplus C_{22}$$
$$H_{14} = A_4 \oplus C_{32}$$
$$H_{21} = B_1 \oplus CI_{21} \oplus C_{42}$$
$$H_{22} = B_2 \oplus CI_{22} \oplus C_{52}$$
$$H_{23} = B_3 \oplus C_{62}$$
$$H_{24} = B_4 \oplus C_{72}$$
$$CO = C_{82}$$

The subtractor sequence uses four 4-bit subtractor modules to perform the subtraction between two 16-bit variables. The algorithm used is shown in Table 3-5. The process takes three cycles to complete. First, the 4 subtractors are cycled. Then, the indicated transfers take place and the carry propagators are used in the final 2 cycles to obtain the result.

Table 3-5. SUBTRACTOR ALGORITHM

CYCLE	SUBTRACTOR	CARRY PROPAGATOR	
	1 2 3 4	1	2
1	H_{14} H_{13} H_{12} H_{11} H_{24} H_{22} H_{22} H_{21} C_1H_1 C_2H_2 C_3H_3 C_4H_4		
2		$C_1H_1C_2$ $C_{10}H_{10}$	$H_2C_3H_3C_4$ $C_{20}H_{20}H_{30}$
3		$C_{10}H_{10}C_{20}$ C_0H_{11}	
Answer is $C_0H_{11}H_{20}H_{30}H_4$			

Each of the 4 subtractors accepts as input 2 hexadecimal digits and a carry bit. They in turn output a hexadecimal result, an output carry bit, and a sign bit. The equations that are programmed into the functional memory are

$$H_1 = A_1 \oplus \bar{B}_1 \oplus CI$$

$$H_2 = A_2 \oplus \bar{B}_2 \oplus C_{12}$$

$$H_3 = A_3 \oplus \bar{B}_3 \oplus C_{22}$$

$$H_4 = A_4 \oplus \bar{B}_4 \oplus C_{32}$$

$$C_0 = C_{42}$$

$$S = \bar{C}_{42}$$

Where:

S means the sign bit

$\bar{B}_i$ means the complement of bit B_i

The compare sequence uses 5 modules to perform the required comparison operations of EQ, NE, LT, LE, GT, and GE. Each of the first 4 modules does a comparison of four 4-bit variables. The results of these comparisons are then combined in the 5th module to form the result of two 16-bit variable comparisons.

All of the above mentioned comparisons are formed and the desired one is sent to the output.

To do this, the decoder transfers control to module 12 which loads the data into the comparator modules and cycles them. The appropriate address of the output routine in module 18 is loaded and the corresponding input mask in module 9 is loaded. Control is then passed to module 14 where the results of the compare are sent to module 18. Similarly, control is then passed to modules 15—17 for the other results to be sent to module 18. Module 17 cycles module 18 and then passes control to it. Module 18 will then transfer the appropriate result to module 9 and pass control to the output routine.

The equations loaded into modules 14—17 are

$$E = A_4A_3A_2A_1B_4B_3B_2B_1$$

$$G = (A_4\ \overline{B}_4+A_3\ \overline{B}_3)\ \neg\ (\ \overline{A}_2B_4\ + (A_2\ \overline{B}_2+A_1\ \overline{B}_1)\ \neg$$
$$(\ \overline{A}_4B_4+\ \overline{A}_3B_3+\ \overline{A}_2B_2)$$

$$L = (\overline{A}_4B_4+\ \overline{A}_3B_3)\ \neg\ (A_4\ \overline{B}_4) + (\ \overline{A}_2B_2+\ \overline{A}_1B_1)\ \neg$$
$$(A_4\ \overline{B}_4+A_3\ \overline{B}_3+A_2\ \overline{B}_2)$$

Where $\neg$ means "and not" and the other symbols are as before.

The equations loaded into module 18 are

$$GT = G_1+E_1G_2+E_1E_2G_3+E_1E_2E_3G_4$$

$$LT = L_1+E_1L_2+E_1E_2L_3+E_1E_2E_3L_4$$

$$EQ = E_1E_2E_3E_4$$

$$GE = GT+EQ$$

$$LE = LT+EQ$$

$$NE = GT+LT$$

The logical sequence performs the binary logical functions of AND, OR, and NOT. This is done by decoding the operator and setting up the input mask of module 9. Then the data is sent to module 13 and a functional cycle is

performed. Control is then passed to module 13 for data transferral to module 9. Control then goes to the output routine.

The program loaded in module 13 is

AND = AB

OR = A+B

NOT = $\overline{A}$

The multiply sequence algorithm will now be described. The decoder passes control to module 11. This module sends the first hexadecimal digit of the multiplier to the multiplier modules and cycles them. Then control is passed to modules 1—4 where the results of the first cycle are passed to the adders and they are cycled. Control then returns to module 11 which sends the next hexadecimal digit to the multipliers and cycles them. Control now goes to module 9 where the most significant hex result digit of the previous cycle is sent to the adders. Control then passes through the adder modules where the data is transferred to the appropriate position for the next add, and the result digits are sent to the output module.

Control now goes to the multiplier for data transferral to the adders and the adders are cycled. Control is then returned to module 11. This sequence is then repeated three more times with a zero being entered in the multipliers on the last time through them. Control then goes through the adders as if a normal add sequence were taking place and the result is transferred to the output modules. The multiply units input two 4-bit operands and through a set of Boolean functions produce an 8-bit product. In this way, this unit can multiply two 16-bit variables by performing hexadecimal arithmetic operations. The total process has 7 steps. Each number is considered as comprised of 4 hexadecimal digits. During the first 4 steps each hex digit of the multiplier is multiplied by a hex digit of the multiplicand.

E

During steps 2 to 5, the partial products are formed by the adders. These adders can add 3 hex digits simultaneously. Finally during steps 6 and 7 the final carry ripple is performed. Table 3-6 summarizes this discussion.

Table 3-6. MULTIPLY ALGORITHM

STEP	MULTIPLY OPERATION	ADD OPERATION	CARRY OPERATION
1	$H_{11}H_{12}H_{13}H_{14}$ H_{24} $C_{41}C_{42}C_{43}C_{44}$ $R_{41}R_{42}R_{43}R_{44}$		
2	$H_{11}H_{12}H_{13}H_{14}$ H_{23} $C_{31}C_{32}C_{33}C_{34}$ $R_{31}R_{32}R_{33}R_{34}$	$C_{42}\ C_{43}\ C_{44}$ $R_{41}\ R_{42}\ R_{43}$ $P_{41}\ P_{42}\ P_{43}$	
3	$H_{11}H_{12}H_{13}H_{14}$ H_{22} $C_{21}C_{22}C_{23}C_{24}$ $R_{21}R_{22}R_{23}R_{24}$	$C_{41}\ P_{41}\ P_{42}$ $C_{32}\ C_{33}\ C_{34}\ C_{43}$ $R_{31}\ R_{32}\ R_{33}\ R_{34}$ $P_{31}\ P_{32}\ P_{33}\ P_{34}$	
4	$H_{11}H_{12}H_{13}H_{14}$ H_{21} $C_{11}C_{12}C_{13}C_{14}$ $R_{11}R_{12}R_{13}R_{14}$	$C_{31}\ P_{31}\ P_{32}$ $C_{22}\ C_{23}\ C_{24}\ P_{33}$ $R_{21}\ R_{22}\ R_{23}\ R_{24}$ $P_{21}\ P_{22}\ P_{23}\ P_{24}$	
5		$C_{21}\ P_{21}\ P_{22}$ $C_{12}\ C_{13}\ C_{14}\ P_{23}$ $R_{11}\ R_{12}\ R_{13}\ R_{14}$ $P_{11}\ P_{12}\ P_{13}\ P_{14}$	
6 and 7			$C_{11}P_{11}P_{12}P_{13}$ are carry adj.

Table 3-6. MULTIPLY ALGORITHM (Cont)

STEP	MULTIPLY OPERATION	ADD OPERATION	CARRY OPERATION
	Answer = $C_{11}P_{11}P_{12}P_{13}P_{14}P_{24}P_{34}R_{44}$		
	Multiplier = $H_{11}H_{12}H_{13}H_{14}$		
	Multiplicand = $H_{21}H_{22}H_{23}H_{24}$		
	Where		
	H_{ij} is a 4-bit hex digit		
	$H_{ij}*H_{kj}=C_{kj}R_{kj}$, C_{kj} and R_{kj} are hex digits		
	C_{kj} is the kjth hex carry digit		
	R_{kj} is the kjth hex result digit		
	P_{ij} is the ijth partial product		

The equations loaded into the multiplier modules are

$$R_1 = A_1B_1$$

$$R_2 = A_2B_1 \oplus A_1B_2$$

$$R_3 = A_3B_1 \oplus A_2B_2 \oplus A_1B_3 \oplus C_{22}$$

$$R_4 = A_4B_1 \oplus A_3B_2 \oplus A_2B_3 \oplus A_1B_4 \oplus C_{32} \oplus C_{34}$$

$$C_1 = A_4B_2 \oplus A_3B_3 \oplus A_2B_4 \oplus C_{42} \oplus C_{44} \oplus C_{46}$$

$$C_2 = A_4B_3 \oplus A_3B_4 \oplus C_{52} \oplus C_{54} \oplus C_{56}$$

$$C_3 = A_4B_4 \oplus C_{62} \oplus C_{64}$$

$$C_4 = C_{72}$$

The divide sequence implements a shift and subtract algorithm (Figure 3-1). The decoder passes control to module 10. This module sets up the control sequence addresses and transfers control to module 11. Module 11 performs a shift of the lower half of the numerator one position left and passes control to module 12 to complete the shift. Control then goes to module 13 where the divide shift count is incremented and set to module 18. Control then goes to the comparator modules which transfer their results to module 12. Module 13 then

initiates a subtract if the compare is greater than; clears the remainder if equal; or checks the shift count for an exit condition. If no exit condition exists, control goes to module 9 for a quotient shift and back to module 10 to start the sequence over. If an exit condition is present, control goes to module 10. There the exit routine transfers the data to the output routines and these routines take control.

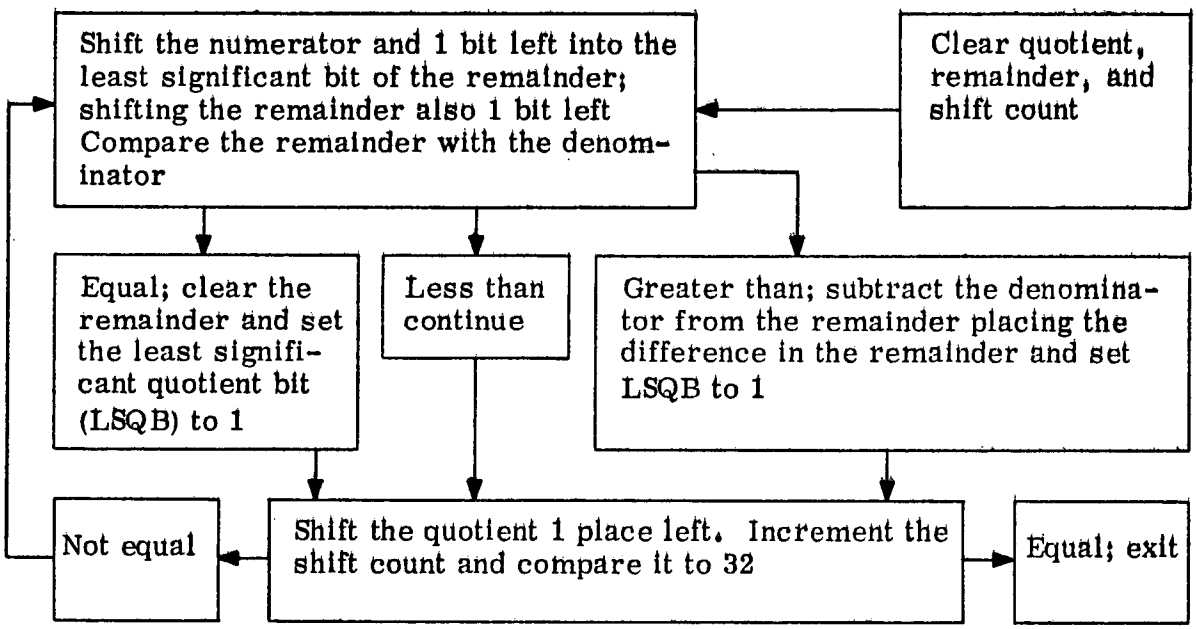


Figure 3-3. DIVIDE ALGORITHM

Thus it is seen that through the use of these control sequences, which are programmed on the module's elementary operation set and through the concept of distributed control, 25 functional memory modules are formed into a conventional ALU. The device being described here contains 4 of these ALU's.

F. The Data Storage and Core Memory System

The data storage and core memory subsystem holds the data while it is being processed. The core memory holds the output data and the data storage holds the input data. This input data is stored such that there is one telemetry

word per module address. A series of loader routines in the data storage system accomplishes this unpacking operation. The data is sent here from the input routines. The loader routines use the bits-per-word count to control a shifting operation which forms a telemetry data word. This word is then sent to the storage system where its storage address is computed and it is stored.

During program execution, the data storage system is used to form new words from parts of other words. To do this, the system is supplied with the telemetry word number, the starting bit within that word, and the number of bits to take. The telemetry word number is converted to an address from which the data is fetched. The starting bit is used to load the input decoders of the holding register so that the word being formed will be right justified in that register. The number of bits to take is converted into an input mask for the holding register. The data is transferred to this register where the new unit of information is held.

The above conversions and load routines are accomplished by an associative table look-up and takes 25 functional memory modules to implement. The data storage system is similar to the program memory system and takes 17 function memory modules to implement.

The core memory is a 2,000-word array of 32-bit words. The length of the memory (2,000 words) is derived from a telemetry standard (reference 13) which limits the size of telemetry data frames to 8192 bits.

Since telemetry words average about 8 bits in length and allowing for status data where selected bits have meaning by themselves, a memory of 2,000 words will be able to hold all the output data. The word length of 32 bits is also chosen because of the same standard, since this is the maximum size any word may take. This memory is supplied with data, an address and a read or write command. The address is computed by the ALU subsystem and transferred to the address

register. The read or write command is sent to this subsystem by the requesting subsystem.

Data is transferred in and out of the memory by an assembly register subsystem. This subsystem assembles the data words from the data holding register into 32-bit words and transfers them into the memory. Conversely, this register receives data from the memory for transferal to the output routines. In addition, a second holding register is provided to transfer data from the memory to the assembly register to form new information words from those already stored in memory.

The data flow path is from the holding register to the special functions unit. From there it is sent to the assembly register subsystem. This subsystem receives the number of bits to be inserted in the data word and those data bits. It must keep track of where these bits are to be placed within the core memory word. That is done by an associative table look-up which loads the input decoders of the assembly register so that the data will be entered into the proper bit positions.

There are 2 cases to consider in this assembly operation; the case where the number of bits to insert is 20 bits or less and the case of 21 to 32 bits. The 6-bit quantity representing the number of bits to insert is sent to a decoder where it is decoded into 2 numbers, such that the first number is 20 or less, and the second is the remainder. The first number is then used as an address to a table of input masks where each mask contains as many ones as the value of the address.

Control is then passed to the module which holds the current start bit value for the assembly register. This value is used as an address to a table of input decoder masks. These masks provide the assembly register module and the input mask shifter modules with input paths for the next 20 bit locations.

The main control sequence then passes to the module holding the input mask value. This module then transfers the input mask to the input mask shifter modules, thereby, properly aligning the input mask bits with the data positions to be filled. Control then passes to the shifter modules which load the input mask of the assembly register.

Meanwhile, the current start bit value and the first number of the decoded number of bits to take value are sent to an adder, and control is sent there also. The adder sends the sum to a decoder identical to that described above and passes control to it. This decoder determines whether the remainder is zero or not. If the remainder is zero, this means that no module change in the assembly register has occurred and hence, the ID commands of the subsystem do not have to be updated. If this number is not zero, then the ID commands are updated to the next module of the assembly register.

At this point, the appropriate updated current start bit value is sent to its module and control passes to data holder number one for data transferal. Control then returns to the initial decoder where the remainder of the decoded number of bits to take is checked for a value of zero. If its value is zero, the operation is done and control passes back to the control subsystem. If the value is non-zero, the above process is repeated again except that control passes to data holder number 2 at the end of the sequence. Then, data holder module 2 transfers the data and passes control to the control subsystem.

G. The Special Functions Unit

The special functions unit implements the function of: parity check (odd or even), word reversal, sync statistics, and sub statistics. These functions are performed by routing the data set through a series of modules. The selection of what modules to use is controlled by which functions are asked for in the instruction. The parity check function has data presented to it from two data holding

modules. The data from data holder one is sent to the parity generator module and this module is cycled. Control then passes to the parity controller which places the result of the first cycle in bit 20 of the parity generator. The data from holder 2 is then transferred there, the generator is cycled, and the result is placed in bit 2 of the module.

Control then passes to main control which determines if an odd or even parity check is to be performed. If an even parity check is required, the parity generator is loaded with a 1 in bit 1 and another cycle is performed. If an odd parity check is desired, no further action is needed. Now the parity bit from the data and the one generated are sent to a 2-bit comparator function and the result of this function is placed in bit 32 of the assembly register. In this manner, it is assured that the result of the parity check will be placed in the sign bit of the host computer by the output routines. Control now passes to main control for the next operation.

The word reversal function receives data from the 2 data holders and also receives the number of bits to take as a parameter. The data is first placed in 4 reversal modules. From here the word is reversed and placed as if it were a 32-bit word left justified in 2 temporary holding registers. The number of bits to take is then used as an address to 12 modules. These modules load the 5 input decoder words and the mask register of the data holding registers in such a way that, when the data is transferred to these modules, it will be left justified. Control is then sent to the temporary holding modules for data transferal to the holding registers. Control now passes to main control for the next task.

The frame sync statistics function stores in memory the result of a comparison of the data with the pattern, computes a total number of errors count and maintains a set of overall error counters for each bit position and the total overall error count.

The frame sync pattern is sent along with the data to eight 4-bit comparators where the error positions are set. These results are then sent to the bit-by-bit counter controls. Here, the first bit is sent to the counter control, where the counter is advanced if the bit is set, or not advanced if the bit is reset. The first bit is also sent to the data memory where the address from ALU 1 has been sent. This address is also sent to a counter where it is advanced and sent back to the memory address register. The bit counter is then advanced and compared to the number of bits in the pattern. If not equal, the whole process is repeated for the next bit of the pattern.

When equality occurs, the total number of errors is sent to memory and to an adder where a running total is kept. Control is then passed to main control for the next task. When the host machine requests the statistics, the 32 error counters and the total error counter are transmitted to it and reset.

The sub statistics function keeps track of counters in the data. These counters are assumed to be binary, changing by only 1 count per cycle and overflowing to a value of 0 at the end of the sequence. This function keeps track of dropouts, errors in the count both on a bit-by-bit basis and as a total, and also handles the modulus rollover to 0. In order to keep track of dropouts, this function must first establish the count. This is done by looking for 3 successive counts in a row. Once this occurs, the system maintains a counter for the count value. To determine a dropout, 3 successive miscomparisons must occur. The system then begins to look for a new count sequence.

The statistics gathered are: the number of dropouts, the number of counts in the search state, the number of counts in the lock state, the number of unsuccessful searches, and the total number of errors. An unsuccessful search is one in which 5 counts have passed without a count reference being established.

To accomplish this function, control is passed to the state module of the subsystem. This module determines whether a count reference has been established or not. In order to establish a count reference, the first 3 counts received are sent to a set of counters with their associated comparators. There the counts are updated and compared with the next count received. If 3 successive counts are found then the reference is established. However, if after 5 counts have been received, no reference is found, then an unsuccessful attempt is tabulated and the process is repeated. Once a count reference has been established, this reference is sent to the main count module. This module sends data to the comparator with the reference count.

If an error is detected, it is counted and the system enters a new state to test for a loss of count reference. If 3 errors are detected in a row, a dropout counter is incremented and the system is set to establish a new counter reference.

The statistics gathered by this function are maintained in their respective counters until the host machine requests them. Then they are shipped out and cleared.

H. The Instruction Decoder Unit

The instruction decoder and main control unit direct the activities of the entire telemetry preprocessor system. This functional subsystem is composed of 2 basic parts: the read next instruction sequence and the instruction execution sequence.

During the read next instruction sequence, the continue indicator is checked first to determine if the previous instruction set is complete. This indicator is set by the linkage flag described in Chapter II. If the flag is set, the next instruction is read from the program memory. If the flag is not set, the repeat indicator is checked next. This indicator is set by a repeat instruction. If the indicator is not set, the next instruction is read from the program memory.

If the repeat indicator is set, the instruction counter is incremented and compared with the number of instructions to repeat. If the comparison is equal, the number of times through the loop counter is incremented and compared with the final loop count and, the number of instructions to repeat counter is reset to zero. If the counter is not equal to the number of instructions to repeat, the next instruction is read from the program memory.

If the loop count is not equal to its final value, the next instruction is read from the loop base address which was set by the repeat instruction. If it is equal, the repeat indicator is reset and the next instruction in sequence is read.

The instruction execution sequences perform all the necessary data manipulation that the instructions require. The halt instruction stops all modules, clears all the data registers, and transfers control to the output routines. The GO TO instruction sets the program memory address register to the address in its address field. It also stores this address in the data memory if an address is specified in the location field.

The REPEAT instruction sets the repeat indicator, places the address of the next instruction in the base loop address register, stores the loop index address, the number of instructions to repeat and the final loop count in their respective registers, and initiates the instruction counter.

The IF instruction will perform the indicated operations until a link code of 0 is found. Then if the result is 0, it will read the next instruction. Otherwise, the branch address will be stored in the program memory address register. If the location field points to a data memory address, the result of the arithmetic expression will be placed there.

In an index expression, the indicated operations are performed until a link code of 0 is found. Then the result will be placed in the indicated index location and in the data memory if required.

The SYNC instruction places the frame sync pattern in the sync function register, initiates a data retrieval operation, and then transfers control to the sync function.

The WORD instruction initiates a data retrieval operation and passes control to whatever special functions are required.

The SUB instruction initiates a data retrieval operation, performs whatever operations are required, and initiates the sub function.

Finally, a location expression will generate the necessary data memory addresses, perform the required operations, and place the new data set in the indicated data memory address.

The control functions also handle the special communications required between the host machine and the preprocessor. These functions are: load the program memory, load the data into the data buffer, read the program memory, read the sync statistics, read the sub statistics, and begin execution. The 6 commands are received and decoded by the control unit and the required action is initiated by this unit. Finally, upon completion of the processing of a telemetry data frame as indicated by a HALT instruction, the control generates an interrupt to the host machine.

I. Summary

This chapter has described in detail the internal organization of the telemetry preprocessing system. It has been shown how a number of functional memory modules could be programmed to implement such a system. This system has advantages and disadvantages which will be described next.

The telemetry preprocessing system described in this report has several advantages over the current method of operation. The language described here offers a natural method of describing telemetry preprocessing operations. It is natural in the sense that the statements of this language represent a description

of the telemetry data frame in nearly the same terms one would use to explain the operations to another person. This makes the language easy to learn and allows programs written in this language to serve as a description of the required operation. Because of this, preprocessing programs are easier to design and debug since improper operations would result in an improper description of the data frame, and this can be readily recognized. Further, it is often the case that the programmer who produced the processing program is not available at a later time when a modification is needed. Because of the nature of this language, such a modification task is easier for another programmer to do since the operations are not obscured by bookkeeping functions.

The telemetry preprocessing system separates those functions which are necessary but do not directly contribute to the data processing phase from the main computer. This allows the main computer to concentrate on the computation phase of the processing operation, a task for which it is well suited. Further, this implies that the computer resources can be applied directly to result-producing tasks and hence, a more cost effective use of these resources is attained.

The telemetry preprocessing system provides a computation-ready data set to the main computer, thereby simplifying the computation program. Since all the problems associated with the telemetry aspects of the data have been corrected outside the main computer, the processing programs can be designed to operate on the sensor data transformations without the need for complex data location and bit manipulation subroutines. This will simplify the processing program and thereby make it less costly to produce and easier to modify.

Because the telemetry preprocessing language is used to specify the data manipulation programs and is implemented on a device outside the main computer, the need for machine level coding to solve these problems is removed.

Thus, the duplication of effort in this phase of the processing, where several computing systems are often involved, is also removed (i.e., the telemetry preprocessing language solution is transferable among computer systems). Further, since the input to the computers is a computation-ready data set, the processing program can be written in a higher level language such as FORTRAN. This allows a speedier solution to the processing problem.

The preprocessing system is more efficient in preparing the data for processing than a general purpose computer because the special functions unit is designed specifically to handle this type of problem. Hence, the same job can be done in a more timely fashion. Thus, the computer time required to do the preprocessing is freed for other uses.

The preprocessing system is designed to interface an existing configuration as though it were a standard device. This allows it to be integrated into a computing system with minimum changes to the system in both software and hardware. This eliminates the need for costly hardware interfacing and costly software modification.

While the system presented here has many advantages, it is not a "cure all" approach. Some problems still remain. The development of a new and specialized language creates the problem of a lack of general familiarity with that language. Thus a training program would be required to educate the programming staff in its usage. There is the additional expense of placing another computing system in the data processing configuration. This expense goes beyond the initial costs since the preprocessing system represents more components in the system which may fail.

The system would not be ready for immediate usage without a large program modification cost since the functions it performs are deeply rooted in the current processing programs. It would have to be gradually phased in as new

projects arise. Further, in a family of spacecraft where the main data manipulation programs are fairly well fixed, it would not be cost effective to use the preprocessing system. These problems indicate that its short range benefits would be small.

Since the preprocessing system is not a multiprocessing one, it cannot handle multiple data streams in parallel. This would require several preprocessor systems (one for each data stream). This adds a good deal of complexity to the processing system and further increases its cost.

The addition of another system to the data processing configuration makes the operation of the system more complex. This will necessitate either the presence of another operations person or special training for the current operations staff. Further, since the preprocessor has no ability to compile its own code, the compilation must be on another computing system and the compiled code then tried on the preprocessor. This makes the debugging operation cumbersome and lengthy.

The functional memory module developed in this research has several disadvantages which require further study to overcome. When using the module as a conventional memory device, the read, write, and address functions are slow and cumbersome to use. This is because the data must be entered through the data register; this requires an input data mask and proper input and output decoder values. This could be remedied by providing a direct memory load path. The input and output decoders are clumsily loaded, in that it takes 5 instructions to perform a full load of either register. Perhaps some form of an encoded load for these registers could be employed.

The number of bits available for functional operations make it difficult to implement large functions. This could be remedied by increasing the number of bits in a memory word. However, a trade-off between the number of pins on the

F

chip and the reliability and cost factors must be carefully considered. Arbitrary shifting of data is a difficult process since the input and output decoders must be loaded to attain the desired shift pattern. An improvement here might be to directly implement a shifting function in the module. Reloading of the module cannot be done by another module. Thus, some form of external load is required. This could be improved by including a move instruction between modules. Hence, pieces of code could be transferred between modules and entire functional sequences could be altered on the fly. Conditional branching and testing operations in the microcode are difficult and slow. Inclusion of these types of instructions in the microcode set would remedy this.

Even with the disadvantages described above, systems composed of functional memory modules offer several major advantages over current methods. First, these modules can be used to implement any Boolean function; hence, the hardware can be tailored to fit the problem being solved by including as many special functions as needed. In this manner, higher level programs can be directly implemented as has been shown. Further, such a system would have only 2 basic building blocks and could be fabricated with no knowledge of the intended use. A system of this nature is completely changeable after the intended application becomes obsolete and thus, such a system is always useful for new applications. This feature is present because the functions and microcoded control sequences are stored in an alterable control storage rather than a read only memory. Hence, the useful lifetime of these modules is greatly extended.

This type of system can interface to any other system in a natural way. Thus system integration costs are minimized. The structure of a machine designed with this type of system is changeable during a problem solution. Hence the system is adaptable to a changing problem environment. Also, many parallel

operations can be activated; hence, the system can employ whatever computing power is required.

The mechanism that makes parallel operations feasible in this kind of system is the concept of distributed control. In this way independent functions can can be carried out independently. The data flow path through the system is what determines the processing operations that the data undergoes. This eliminates the need for massive control functions usually needed to direct the entire operation. Further, by localizing control to specific functional operations and only controlling the direction of data flow, a great deal of freedom to carry out many different parallel operations is gained.

The functional memory module approach to the system implementation allows relative freedom from internal architectures and can accommodate a wide variety of internal data structures. By using this approach to systems design, the data processing system can be tuned to the problem's structure and the solution can, in effect, be hardware implemented as is demonstrated by the implementation of the telemetry language described in this chapter. This implementation allows the telemetry preprocessing system to be integrated into a wide variety of data processing system configurations with minimal impact on the total (hardware and software) system design. This makes the telemetry language installation independent and allows telemetry preprocessing programs to be transferred from system to system.

CHAPTER IV

THE TELEMETRY PREPROCESSOR SIMULATION — DESCRIPTION AND RESULTS

The system described in Chapter III is complex enough so that the concepts and algorithms presented there need to be verified. When dealing with a concept of machine design that is almost totally dependent upon program code, the debugging and verification of that code becomes almost as important as the concept. Further, since Chapter III is presenting a new design of a system building block, it is imperative that the feasibility of this design be established and that proof of its correct operation be provided. Finally, some comparative timing data between the new approach and current methods should be provided.

To accomplish this task, a simulation program of a functional memory module was written in APL (references 10 and 11). The APL system was chosen for three reasons: its interactive nature, its availability, and its powerful data structure operations. The interactive nature of the system greatly shortened development time. Also another important factor in shortening development time was the ease in forming the required data structures and the simple but powerful matrix operations that the language provides. However, because of the way in which the language is implemented, the simulation had to be constrained to working on a single functional memory module at a time, since the working memory size available is only 32,000 bytes of 8 bits each. Thus, the data for the preprocessing system was stored on the disk storage provided with the APL system with each record representing an image of a particular module.

Figure 4-1 shows the functional memory module that was simulated. This module is described in detail in Appendix A; thus, only those features necessary to clarify the discussion of the simulation will be presented here. The memory array shown in Figure 4-1 is composed of 64 words which are 27 bits each.

However, 20 of these bits are used to perform functional operations and a functional operation requires 4 words to implement. This array, therefore, is simulated by a three-dimensional data structure whose dimensions are 16 by 27 by 4. In this manner, the 16 possible distinct functional operations which require 4 words each can be easily implemented.

This memory is used to store the microinstructions which control the data flow paths and operations of the system. Thus, the microprograms must be excluded from functional operations. This is done by the inhibit register which is simulated by indexing only those locations of the first dimension of the memory array which are to be included in the functional operation.

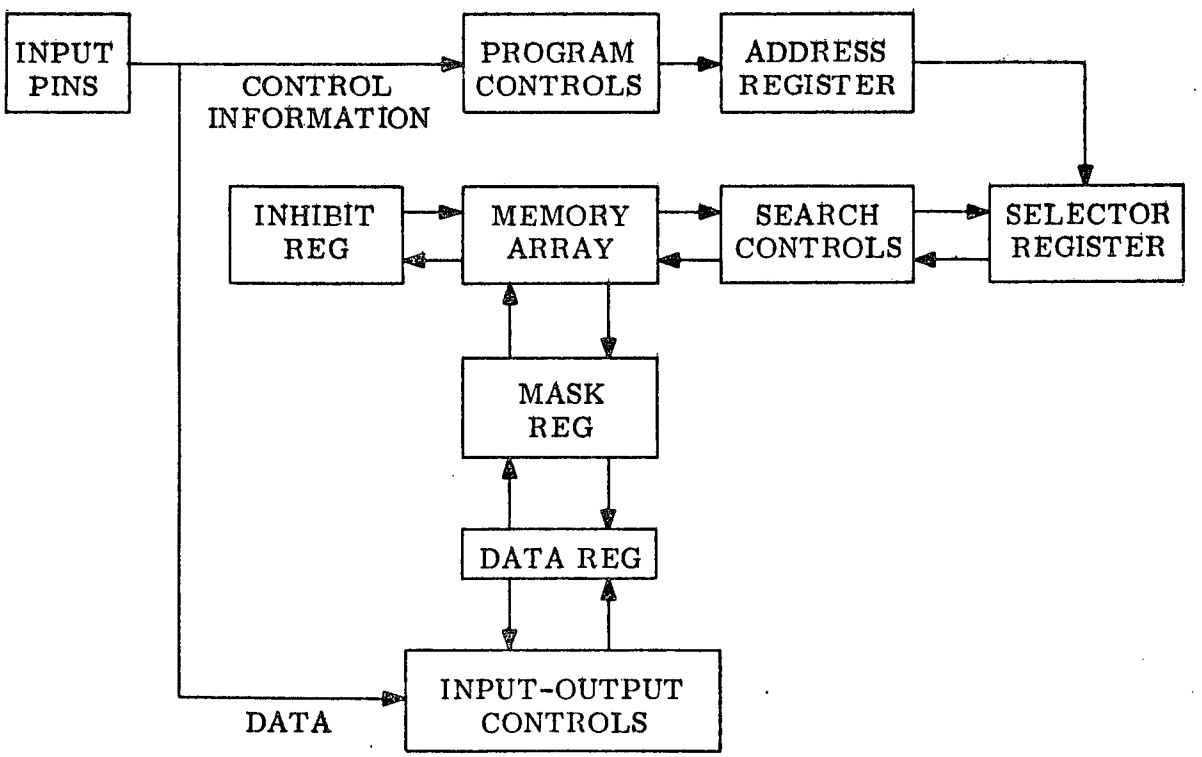


Figure 4-1. INFORMATION FLOW DIAGRAM OF A FUNCTIONAL MEMORY MODULE

The data register contains the inputs to the function being implemented in the memory array, as well as the outputs of that function. To separate these

values, the mask register is used in gating information into and out of the memory array. This is accomplished by the logical product of these two registers for gating information into the memory array, and the logical product of the data register and the complement of the mask register for gating information out of the memory array into the data register.

Once information has been gated into the memory for a functional operation, the logical product between the true information and the true select bit of all the selected locations must be formed. Also, the logical product of the false information and the false select bit must be generated. This is done by forming a matrix logical product between the true information and the 3rd element of dimension 3 and between the false information and the 4th element of dimension 3. These 2 logical products are then "ORed" together to form the functional output of each input data value and its corresponding set of functional terms.

At this point in the operation, the information can be gated onto 1 of 4 term lines. This is done by using the first 2 elements of dimension 3 to form an index to a term matrix. The elements of the rows of this matrix are then "ORed" together and complemented to form the required Boolean expressions. As is explained in Appendix A, these terms are now combined in various ways to form carry terms and to set the corresponding cell of the selector register. These operations are simulated by performing logical matrix operations which are part of the APL operator set.

The last step in the simulation of a functional cycle is to use the selector register to compute the indices of the first dimension of the memory array which are to be used in reading the function's output. Then the cycle type controls are decoded to determine if this is to be a "read true" only cycle, a "read false" only cycle, or a read "exclusive or" cycle. The appropriate information bits are

formed and the logical product between these bits and the complement of the mask register is performed to enter the functions output into the data register. In this manner, all of the functional operations that a module can perform are directly simulated.

As was mentioned above, the module contains a microinstruction operation set. There are 43 instructions in this set. These instructions are grouped into 3 groups: the load group, the clear group, and the communications group. The first 2 groups load or clear the various registers of the module. The third group handles all the communication and control functions of the module. The load and clear type instructions are simulated directly on instruction by instruction basis. The other instructions set the various control states in the module or generate communication requests as required.

The instructions are decoded by using this operation code as an index to matrix of program labels pointing to the individual instruction subroutines. These routines perform all the tasks required of the instructions as described in Appendix A. The most interesting task performed is the communication request. In the simulation, 5 types of communication bus requests are defined for the bus simulation routine. The first of these requests is the connect request, where the requesting module is connecting its output to one or more other modules. This is done by generating a list of module numbers which will be read from the disc in sequence whenever a communication is issued.

The second is a drop connect request which occurs when the requesting module is placed in a stop condition as dictated by bit 21 of the microinstruction set. This is implemented by emptying the list of connected modules.[2]

[2] In the APL system, an empty data structure is permissible. This structure is a vector with no elements.

The next request is a normal transfer request. In this request, any instruction (except a connect request, a selector transferal or a control transferal) may be sent to all connected modules. This is done by reading into the work space, each module in the connect list, in sequence, and executing the instruction present in the bus register. After execution, the modules are returned to the disk file. Hence, the disk file contains a current image of the system being simulated.

A selector transferal request is the only two-way communication in the system. This instruction is intended to be used for functions whose total number of inputs and outputs exceeds 20. In this case, the expressions to be generated by the input values may be located in one module and the function's output in another. The operation then will be to do a functional search on the input module. Then the output module will send for and receive the selector register. The output module then can execute a functional read to complete the function. This is simulated by reading the input module from the disk and transferring its selector back to the output module.

The final request is a transfer control request. Since the simulator is not designed for parallel operations, this request will create a job queue if a multiple control transfer is encountered. The form of this queue is a matrix whose number of rows equals the number of modules connected. The columns are comprised of the current state of the system (i.e., the destination module of the request, the requesting module number, and the instruction being transferred). If the requesting module is not in the stop condition, the simulator

will continue to read and execute instructions from that module until either a stop is reached or another control transferal is reached, in which case, it will be placed at the bottom of the queue. Then the task at the top of the queue will be removed from the queue and that control path will be executed until one of the above two conditions is met. In this manner, the job queue will eventually empty when the system returns to a single control path.

At this point, the reasons for a simulation have been presented and the simulator has been described with the exception of the measurements it gathers. The parameters measured by the simulation are: the number of microinstructions read, the number of bus connects issued, and the number of bus transfer requests issued. These parameters were chosen because they represent all of the different timing sequences that are found in the system and hence by making various assumptions about the length of these sequences, representative timing data regarding system operations can be computed.

Not all of the 6 subsystems described in Chapter III were simulated. This was primarily because of the large amount of processing time that is required to run the simulation: The central processor time required for the IBM 360/95 computer varied from 22 seconds for a compare which is the fastest operation to about 30 minutes for a divide. This made it impractical to simulate the entire system. Further, certain subsystems are simple enough so that a simulation is not necessary. The instruction decoder simply addresses a set of microroutines which in turn activate control sequence in the other subsystems. Thus if the control sequences are simulated, there is no need to simulate the decoder. Hence it was not simulated. The program storage

and data storage subsystems are used as a conventional memory. Since the conventional memory features are checked in simulating other functions, there was no need to simulate these subsystems.

The functions that were simulated were the data input routines, the ALU functions, the data assembly registers, and the special functions unit. Because of the similarity between formatting the input data for storage in the data store and formatting the output data for transmission to the host computer, there was no need to simulate the output routines. The results of these simulations will be presented next.

The ALU functions that were simulated represent the programs that would be required in the 25 functional memory modules which make 1 ALU subsystem. These include all the functional tables and microroutines needed to implement the unit as described in Chapter III.

Table 4-1 shows some typical results of a simulation. The 2 columns under the subtract function arise because of the need to complement the answer when the sign of the answer goes negative. This is a requirement because the number representation of this system is signed magnitude in which all magnitudes are positive binary integers. The 2 columns under the divide function represent the minimum and maximum length sequences required to implement the shift, compare, subtract algorithm discussed in Chapter 3. Lastly, the two columns under the compare function arise because of the parameters. When the signs are different, there is no need to generate a full comparison between the numbers.

Table 4-1. SIMULATION RESULTS FOR THE ALU FUNCTION

FUNCTION	ADD	SUB-TRACT	MULTI-PLY	DIVIDE	LOGI-CAL	COM-PARE
Functional cycles	8	9 13	44	129 385	1	1 10
Instruction reads	166	175 235	541	2396 9052	74	102 137
Bus connects	40	40 53	146	648 2151	18	22 29
Bus transfers	72	74 97	261	935 3911	31	43 60

To assign a time value to these operations, assume that emitter coupled logic (reference 19) is used to implement the circuit design described in Appendix A; then a functional cycle typically takes about 100 nanoseconds, a read typically takes about 20 nanoseconds, a bus connect typically takes about 10 nanoseconds, and a bus transfer typically takes about 20 nanoseconds. Then an add of 2 16-bit numbers would take 5.96 microseconds, a subtract would vary between 6.28 and 8.47 microseconds, a multiply would take 21.9 microseconds, a divide would vary between 86 and 319.27 microseconds, a logical operation would take 2.38 microseconds, and a comparison would vary between 2.22 and 4.15 microseconds. These timing values compare with a medium speed conventional machine with a slow divide. Different assumptions about the operation times (particularly the functional cycle) would yield different results.

The input routines were simulated by executing the microcode of the ten modules that make up this subsystem. This subsystem assumes that 12 bits of data are received. It contains as a parameter, the number of bits in a telemetry data word. The received bits are then shifted into a 32-bit shift register, 1 bit at a time. After each shift, a bits per word counter and an input bit counter are updated. These counts are then checked for exit conditions. If the end of word is detected, the contents of the shift register are transferred to the data storage memory. If the end of input is detected, a request for the next 12 bits is issued.

The measurement parameters for these routines are presented for a 6-bit word and a 32-bit word. For the 6-bit word, the values are: 281 instructions executed, 56 connects, 116 transfers, and 14 functional cycles. The values for the 32-bit word are: 1089 instructions executed, 271 connects, 479 transfers, and 67 functional cycles. Thus the data storage time required to format and insert these words into the memory varies from 9.3 microseconds for a 6-bit word to 38.2 microseconds for a 32-bit word. These times translate to a data transfer rate slightly under a megabit.

The data assembly register subsystem is a 22 module subsystem whose parameters are the number of bits per word and the number of bits to take. This latter parameter is sent here from the ALU subsystem. The assembly subsystem contains a 32-bit data holding register and a 32-bit assembly register. The rest of the system is concerned with keeping track of the current bit position for data insertion into the assembly register and with shifting the input data to that bit position. This is done by a table look-up of the proper assembly input mask and the proper data holder output decoder values. The address for the table look-up is the value of the current bit position. This value is kept current by adding the number of bits to take to the old current value to obtain the new current value.

There are two operations in the subsystem: assemble and dump. An assembly has 155 instructions executed, 41 connect, 87 bus transfers, and 2 functional cycles, while a dump consists of 55 instructions, 11 bus connects, and 35 bus transfers. These figures translate to times of 5 microseconds and 1.7 microseconds, respectively.

The special functions subsystem is a complex unit containing 123 modules. It implements the functions of data retrieval, parity checks, data reversal, sync word error statistics accumulation, and subcom counter error statistics accumulation. The data retrieval section contains 16 modules. It receives as parameters from the ALU functions the current word number, the starting bit within the word, and the number of bits to take. It uses the current word number of derive the address in data storage of the data word. The other 2 parameters are used to load the input decoder and mask, so that when the data bits are received from the memory they will be right justified in the 32-bit data holding register.

The parity check section contains 5 modules. It receives the data and a request to check for either even or odd parity. The check is performed with a functional cycle and the error bits are sent to the data assembly register. The reversal section has as a parameter, the number of bits to take. This parameter is used as an address for a look-up table of input masks and output decoder values. These values are loaded into the 32-bit reverse data holding register. These values insure that when the data is sent back to the data holding register, it will be right justified. The subsystem contains 8 modules.

The frame sync word error statistics collection subsystem contains 50 modules. Of these 32 are used to hold the bit by bit error distribution counters. In addition, the subsystem accumulates the total error count and the one-to-zero error count. It also outputs these statistics on a frame by frame basis. This subsystem has as parameters the number of bits in the pattern and the pattern. The data containing the received pattern is sent to this subsystem where it is compared 4 bits at a time with the true pattern and the errors are totalled.

The subframe counter error statistics collection subsystem contains 44 modules. It receives as parameters, the number of bits in the counter, the counter modulus and the counter mode (forward or backward). The system keeps track of the current count and the status of that count. The count status is in a searching mode until 2 consecutive counts are detected. It then enters the count verify state. The system will return to the search state if the next count is not in sequence, or will go to the locked on state if it is. It will remain in the lock state until 3 consecutive non-sequential counts are detected. The statistics gathered by this subsystem are: the number of dropouts (i.e., the number of transitions from lock to search), the number of counts received while in the search state, the number of counts received while in the lock state, the number of non-sequential counts received while in the lock state, and the number of unsuccessful search tries. For this last statistic, a successful search is defined as the ability to attain lock within 5 counts. This statistic gives an indication of cyclical type errors. The subsystem accomplishes this task by maintaining a set of counters for the various states.

Table 4-2 shows the results of the special functions unit simulation. These results show the individual timing for each function on a per word basis. Thus, for example, if the sync pattern had to be parity checked and reversed, the entire operation would take 13.2 microseconds including the time required for data retrieval.

In conclusion, this chapter has presented a description of the program written to simulate the telemetry preprocessing system. This program was useful in verifying the correct operation of the system and in proving the feasibility of using functional memory modules to implement such a system. Further, the timing results gathered by this program were presented without comparison to other implementation methods. This comparison will be the topic of the next chapter, which will present the results and conclusion of this work.

Table 4-2. SPECIAL FUNCTIONS SIMULATION RESULTS

FUNCTION	NUMBER OF INSTRUC-TIONS	NUMBER OF BUS CONNECTS	NUMBER OF BUS TRANSFERS	NUMBER OF FUNC-TIONAL CYCLES	EXE-CUTION TIME (USEC)
Data Retrieval	160	40	90	0	5
Parity	28	6	14	2	1
Reversal	32	16	48	0	2.5
Sync	230	50	60	4	6.7
Sub	63	16	35	4	2.5

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

A. Conclusions

The main effort of this research was to develop a higher level language capable of performing the necessary bit manipulation operations to prepare telemetry data for computer processing. This effort was greatly assisted by the usage of the APL programming system. This system made it possible to easily explore various compiling methods and various language operations. Further, because of its interpretive nature, the development time for this project was greatly reduced. Another factor in reducing this development time is APL's simple but powerful vector operators. They made it possible to perform complex operations without the need for complicated bookkeeping algorithms.

The telemetry preprocessing language developed also has a set of simple but powerful operators. It provides a means for describing complex bit manipulation problems in a concise fashion without the need for lengthly housekeeping algorithms. By using this language, the development times for producing telemetry data processing programs can be shortened because the data is ready for ingestion into a well known processing algorithm (depending on what sensor is involved). Hence the programmer can begin to work directly on the problem solution instead of spending a great deal of time designing data structures and tracing bit paths from the telemetry data to this data structure set.

As was shown in Chapter III, the language described in Chapter II was used as a guide for specifying a telemetry preprocessing system. This system was designed as a peripheral device to a host computer. This method was chosen so that the total system is capable of being installed at the various installations

which process telemetered data. In this manner, the difficulties of past attempts are overcome.

Although the original goal of this investigation was to produce a system which would enhance the art of telemetry data processing; the most significant result was produced along the way. The original goals have been fulfilled with the design and specification of the telemetry preprocessor system and the telemetry preprocessing language. But it is the extension of the concept of a functional memory (reference 8) into the functional memory module that represents the real contribution of this work.

The concept of implementing Boolean functions in a cellular memory array for the purpose of designing flexible control units for microprogramming applications has been extended into a systems building block. A device has been designed which is ideally suited for LSI implementation since it has a high ratio of gates to input pins and a highly repetitive structure. Further, since within this device can be stored not only the functional programs required to implement Boolean functions but also the microcode which directs the flow of information between modules, this device attains the greatest possible flexibility of any LSI chip structure. A set of chips of this nature are capable of implementing any Boolean function.

The work performed in this investigation demonstrated that 435 of these chips could be interconnected to form a telemetry preprocessing system. The major portions of this system were simulated and the results of that simulation effort were presented in Chapter IV. Following is a comparative study of this simulated data with that of an optimized process on the CDC 3200 computer.

In comparing the time to perform various operations of the preprocessing system to that of a medium speed machine such as the CDC 3200, it is found that the preprocessor arithmetic operations are somewhat slower (e.g., an add takes 6 microseconds on the preprocessor while only 2 on the CDC machine). The logical operations also are slower by a factor of 2 to 4 but the special functions are much faster. A reversal of a 32-bit data word for example would take 260 microseconds on the CDC while only 2.5 on the preprocessor. Similarily the other special functions show a factor of 100 speed improvement. To further illustrate the significance of this fact, the processing time for the entire OAO-A2 direct digital data frame presented in Chapter II was computed for both systems. The CDC machine using an optimal algorithm takes 36,640 milliseconds to do these operations while the preprocessor takes only 20,184 milliseconds, a factor of 45% improvement. These functional memory modules, however, are more complex than the 3200 because they are designed for flexibility. One measure of complexity that supports this fact is a physical comparison of the systems. The 3200 computer contains 250,000 gates while the functional memory modules that make up the telemetry preprocessor contain 5,220,000 gates. However, because of the difference in logic technology between these 2 systems, the 3200 contains 9400 logic cards while the telemetry preprocessor system contains only 9 logic boards.

Another aspect of the functional memory concept that contributes to its usefulness as a system building block is the manner in which the control function is handled. Each module contains a state indicator which can make it a control module. As a control module, it can direct the activities of other modules. This makes it possible for many control sequences to be activated in a system constructed with these modules. Thus independent functions can be executed independently.

In summary, the major contribution of this research was the development of an advanced hardware device for use as a system building block. This device is a four-state cellular logic module known as a functional memory module and is suitable for LSI implementation. The fact that this device is a building block was demonstrated by showing how 435 of these modules could be used to implement a telemetry preprocessing system which was designed by using the higher level language discussed above as a guide. This system is more complex, hardware-wise, when compared to a Control Data 3200 computer. However, it compares favorably timewise when a benchmark program is executed. The 3200 computer was chosen for comparison because it is currently being used in this function at the Goddard Space Flight Center.

B. Recommendations For Future Research

During the course of this research, some interesting related problems have been posed. The functional memory module offers a powerful tool for implementing digital machinery. However, several questions in this area need to be considered first. How fast can arithmetic operations be implemented using devices like these? Can large high speed computing systems be implemented using these modules as their only building blocks? What is the feasibility of making these modules a standard building block for digital systems? Several improvements to the module which offer partial answers to these questions are: inclusion of a general shift register, increasing the size of the memory, inclusion of a move instruction to move the microcode from one module to another and the inclusion of conditional test and branch microoperations. The impact of each of these changes should be evaluated from the viewpoint of the above questions to determine if enhanced capability is being provided to the module or not.

The functional memory approach to building systems offers the possibility for implementing dynamically alterable machine structures. Such a structure raises a number of questions. How can such a machine be controlled? How can a data processing problem take advantage of an adaptable machine? How can machine failures and programming errors be diagnosed? One method of controlling such a machine would be to construct part of the machine such that it contains unalterable code required by the operating system. The compilers that would allow the user to make up his own instructions could be designed. Hence the system could be tuned to match the problem being solved. Then as the processing progressed a machine status vector could be periodically updated to provide a history of what is happending in the machine so that errors can be detected. Thus the items that require investigation are what the unalterable code should be, what information belongs in the machine state vector, and what method of description should be presented to a compiler to allow the user to make up his own instructions.

APPENDIX A

INTRODUCTION

The telemetry preprocessing system described in this thesis must be capable of handling word sizes that range from 6- to 32-bits, be able to reverse these words end for end, be able to selectively complement bits in these words, perform error measurements on the words and collect statistics, and interface to a wide variety of computing and data recovery machinery. While a classical hardwired approach could be implemented to accomplish this task, a number of special cases would arise which would lead to a costly and overly complex approach; e.g., a separate hardware interface for each type of computer would be required. Microprogramming techniques (reference 18) offer solutions to these problems; however, implementations of microprogramming systems have a fixed set of hardware registers and a limited number of interconnections capable of being program controlled. Consequently, using these machines still represents a transferal of the coding problems from machine language code to microcode. Many of the inefficiencies of the problem solution still remain.

Thus a decision was made to design a system with a minimal number of building blocks to reduce costs, and with enough variability to cover all aspects of the problem. The first constraint dictated the use of large scale integration (LSI) technology. The second constraint implies a microprogram approach but with an unspecified number of hardware registers so that the machine could be coded to fit the requirements of the problems.

As a result of the decision to use LSI technology, two assumptions were made: (1) no functional process would be hardwired into the machine and (2) the number of lines interfacing the module to the rest of the machine would be forty or less. The first assumption was made to provide the machine with the greatest

possible freedom from architectural constraints. The second assumption concerns the practical aspects of using LSI since (a) backplane wiring problems must be considered and (b) the majority of failures and the largest costs of LSI chips are associated with the number of pins on the chip. The device that fits these characteristics best is known as a functional memory module.

The term functional memory connotes a device used to generate Boolean expressions in a memory device. Basically, it is a method of arranging a cellular memory array such that each cell of the array can be either an associative cell or a conventional memory cell. Additional gating is provided at the array boundaries so that Boolean functions can be generated by using the above two memory types in combination. The associative nature of the memory is used to "search" for a set of preprogrammed Boolean expressions in the input data. Then the results of this search are used as a conventional address to "read" the function output from specified cells of the array.

The format of this presentation of the design features of a functional memory module will start with two Boolean functions. These functions will be implemented with AND-OR logic and then successive partitions of that implementation will be employed to demonstrate the transformations required in going from conventional logic to functional logic. Also, in this manner, the important characteristics of a functional memory concept are best illustrated because the reader can be led from familiar concepts to the newer less familiar ones.

A. Functional Memory — A General Description

The basic concept in constructing a functional memory is De Morgan's law which states that the Boolean expression ABC may be represented as $\overline{(\bar{A} + \bar{B} + \bar{C})}$. The overbar means NOT and + means OR.) To implement this law a cellular memory array is constructed. This array has a two phase cycle consisting of

search and read. Further, this operation is associative in nature, in that only those terms which satisfy the search criteria contribute to the read phase.

Consider, as an example, two Boolean functions of three literals (variables) each:

$$f_1 = A + \overline{B}\,C$$

$$f_2 = \overline{A}\,\overline{B}\,C$$

Classically these functions could be implemented with the following structured set of gating:

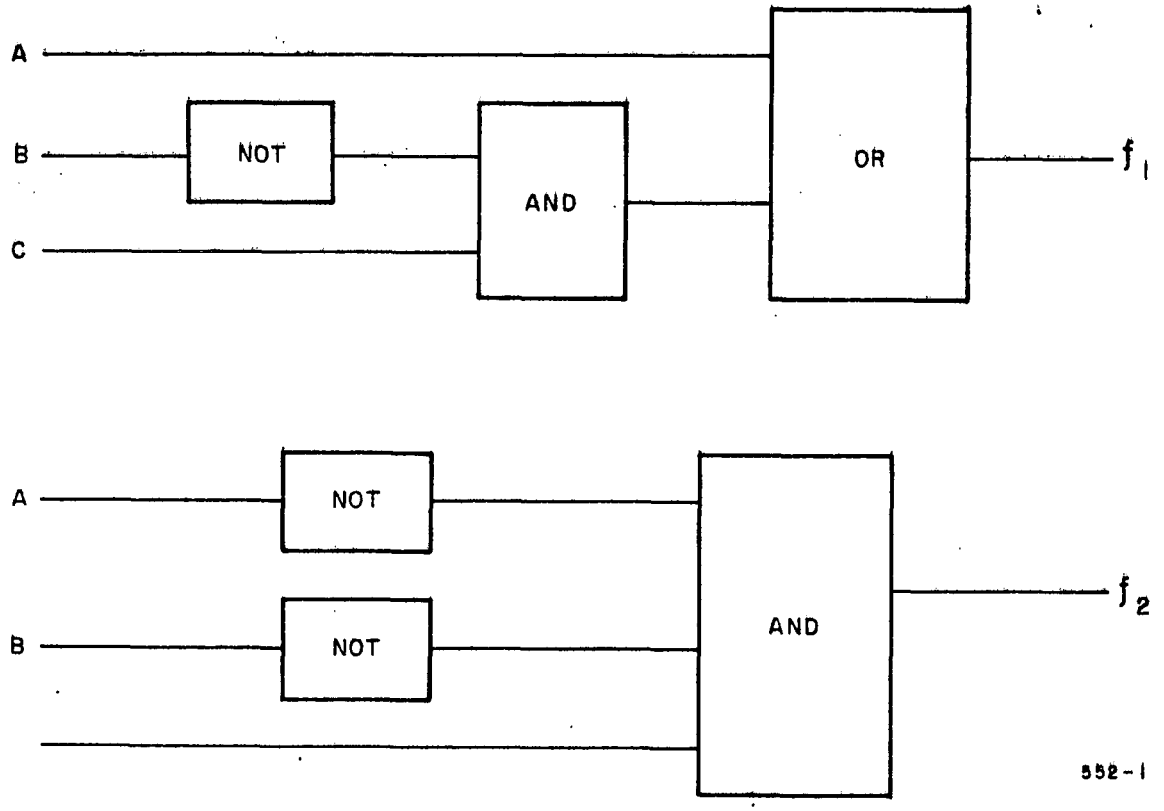


Figure A-1. CLASSIC GATING STRUCTURE

Now applying De Morgan's law on a term-by-term basis, the equivalent functions and implementations could be built.

$$f_1 = \overline{\overline{A}} + (\overline{B + \overline{C}})$$

$$f_2 = \overline{(A + B + \overline{C})}$$

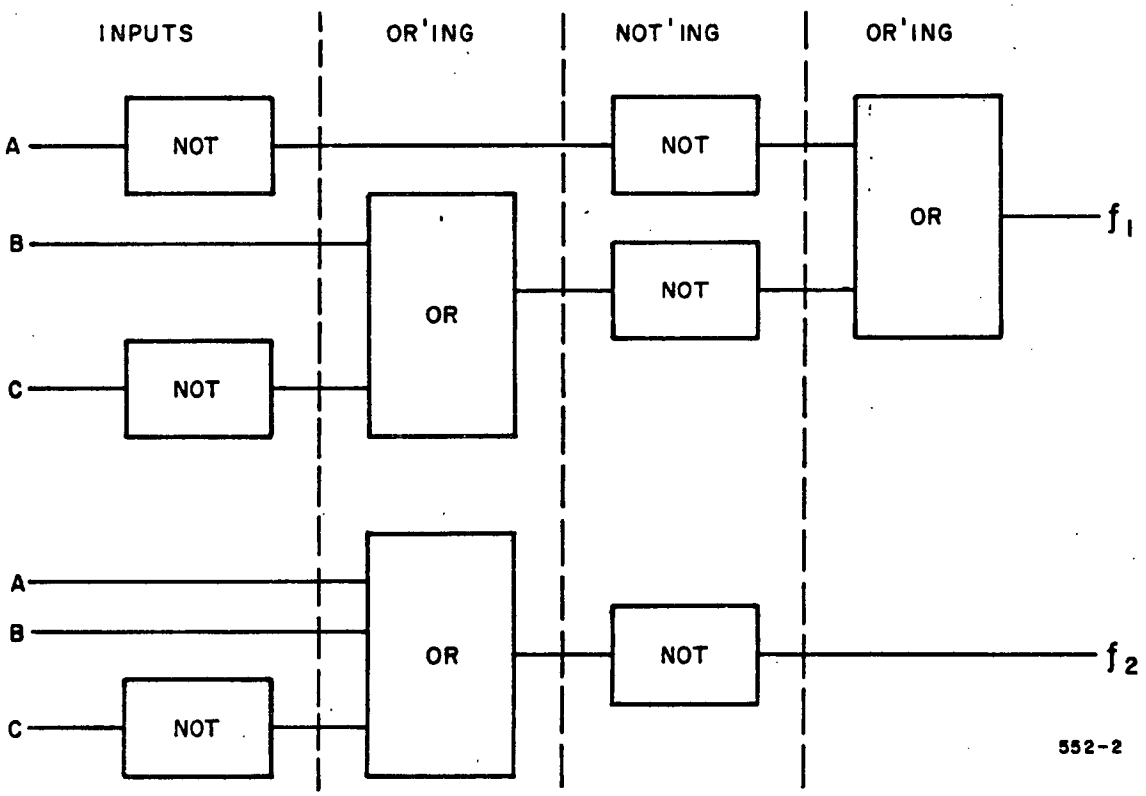


Figure A-2. MODIFIED GATING STRUCTURE

In this implementation, notice the 4 divisions of logic which result. The inputs are selected either in true or false form and then a sequence of OR-NOT-OR is performed to obtain the required function. This is the form of a functional memory implementation. Consider a memory array where each cell is comprised of a true flip-flop and a complement flip-flop. Then the selection of inputs (true or false) can be done by setting the appropriate flip-flop and allowing the true and false forms of the true or false bit line to drive a pair of AND gates as is shown in Figure A-3.

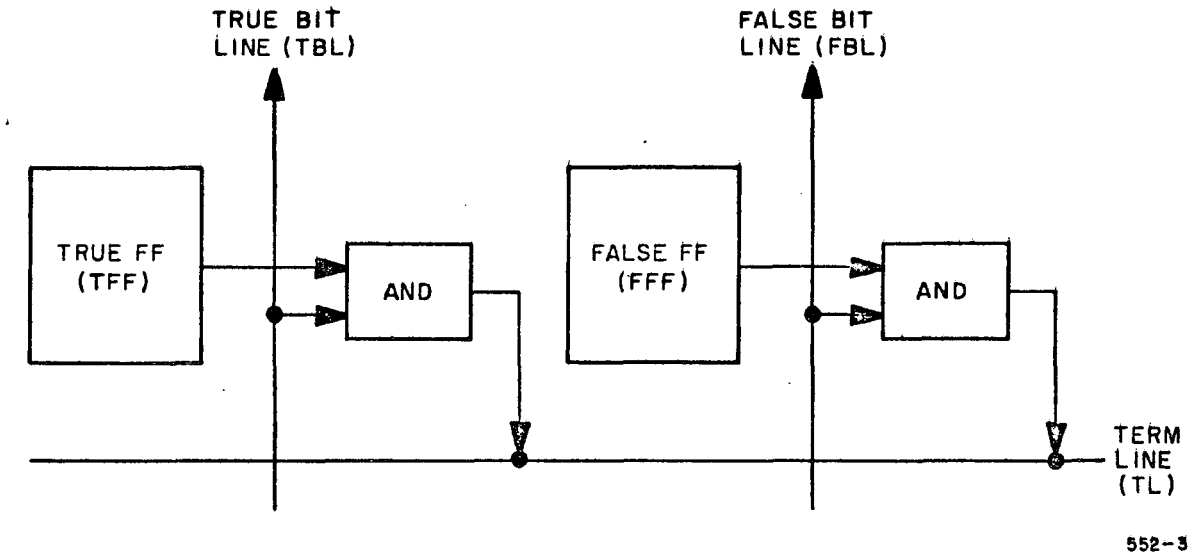


Figure A-3. CONCEPTUAL FUNCTIONAL MEMORY CELL

These AND gates are then tied together on the term line which effectively accomplishes the first ORing operation. By placing an inverter at the end of this term line, the NOTing operation is accomplished. Recalling that we are describing a two phase memory operation; the output of this inverter is the input to a register of latches called selectors. What has been described to this point is the "search phase" of a functional memory which accomplishes the input, first ORing, and NOTing operations. This operation consists of "searching" for which terms are present and "selecting" those terms for the "reading" cycle. Thus,

referring to Figure A-2, the inverters of the NOTing step each represent a term line (or conventional word line) of the functional memory array.

During the reading operation the roles of the term lines and the bit lines are reversed. As shown in Figure A-4 the selected term lines are selectively gated onto the bit lines by placing data related to the desired function output in the true and false flip-flop.

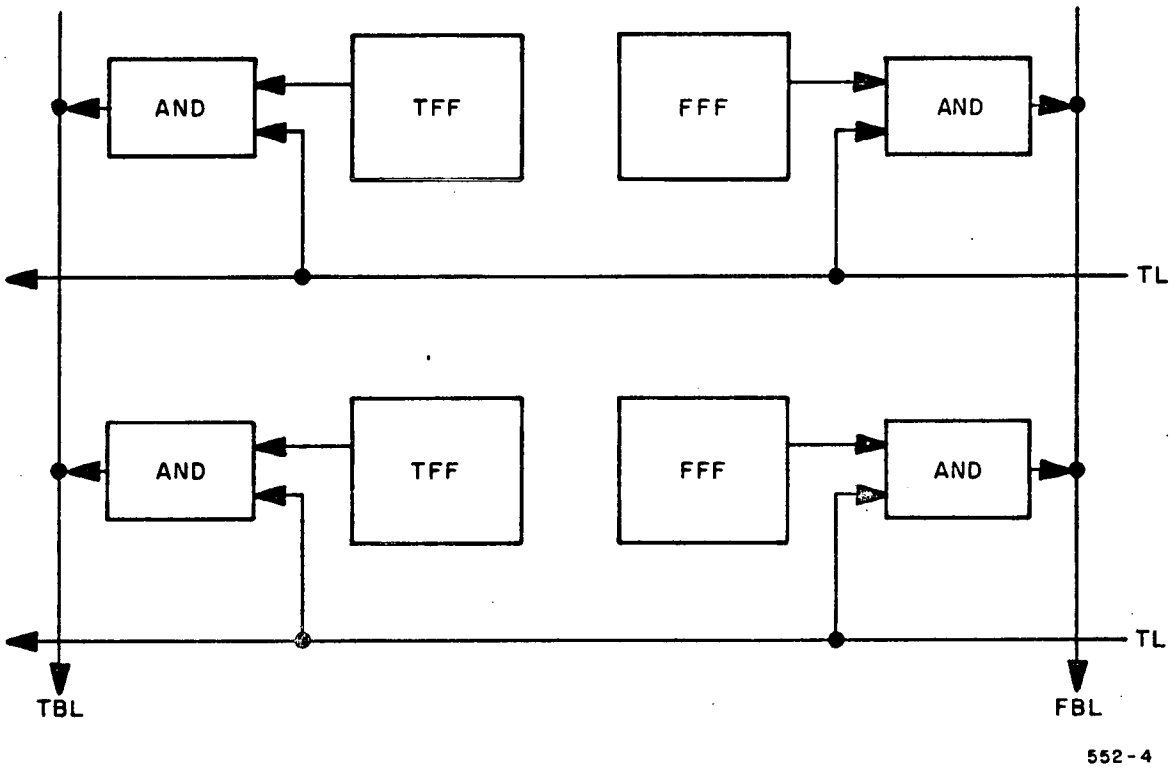


Figure A-4. CELL INTERCONNECTION

Hence, the second ORing operation is accomplished through the connection of the outputs of these AND gates to the true and false bit lines. Consequently, either bit line is capable of carrying the entire function output or a portion of it. At the end of the bit lines, the function may be obtained by gating either the true bit line or the false bit line into a data register.

Several things should be mentioned at this point. First, the cells of the memory involved in the search phase and in the read phase are obviously not the same, although they could be if the cells were reloaded but this would be

inefficient. Hence some means of separating the search variables from the read variables must be included. This is accomplished by means of a data mask register as is shown in Figure A-5. This register is connected between the data register and the memory array. The true side of the mask register will allow data to enter the array for search purposes. The false side of the mask then allows data to be gated out of the array into the data register during the read phase.

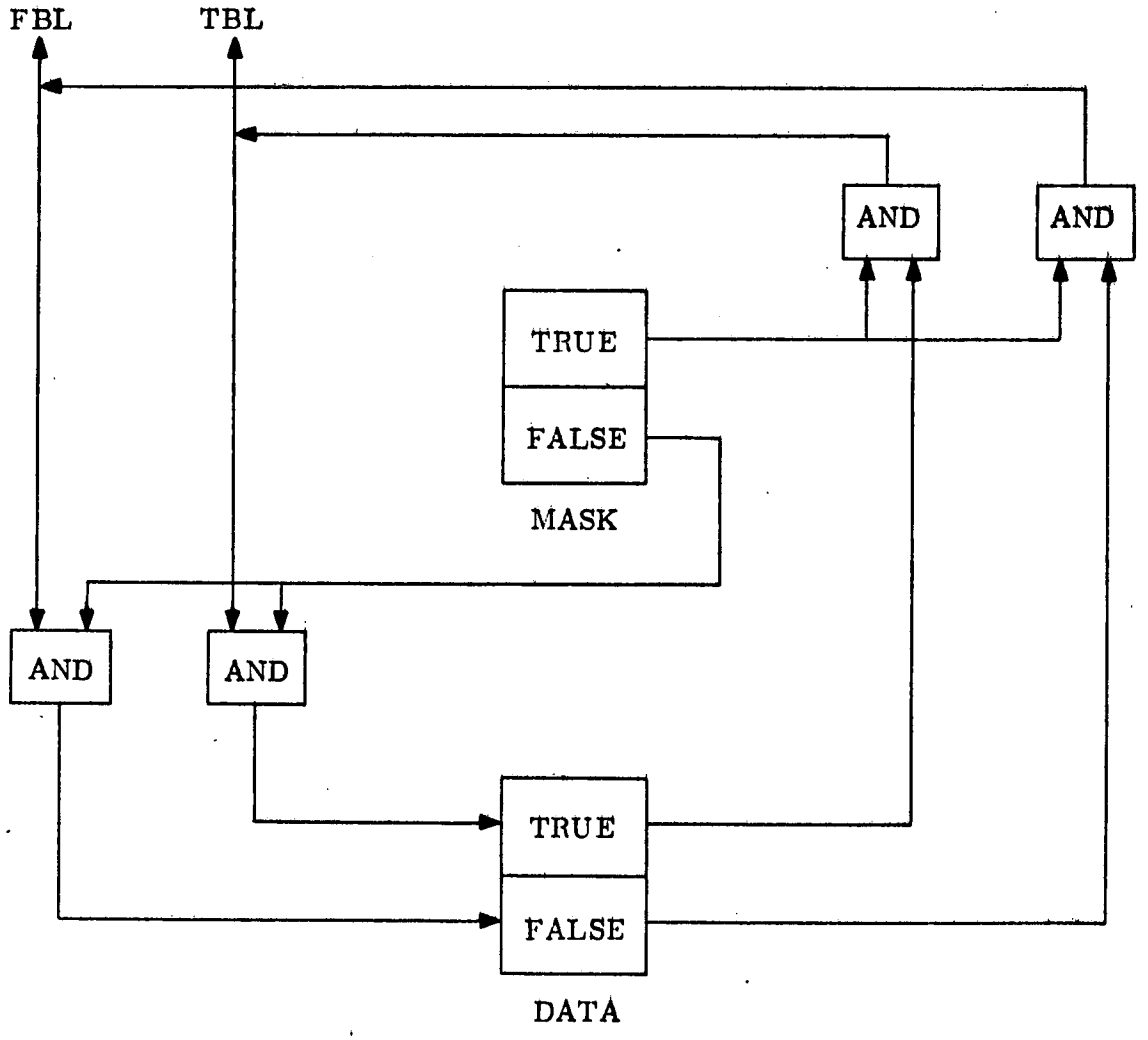


Figure A-5. TYPICAL MASK AND DATA CELL INTERCONNECTION

Second, since each cell of the array contains two flip-flops, there are two other states of these cells which can be used during the search phase. By setting

both the true and false flip-flops to zero, it is seen that this particular variable has no effect on the term line since a true state will be generated no matter what the condition of the input data, hence this is called the "don't care state." Further, by setting both of these flip-flops to one, it is seen that as long as this variable is allowed to partake in the search phase this term line cannot be selected because of the inverter and thus this state is called the "inhibit state."

Finally, it seems that some use could be made of the true and false bit lines during the reading phase. Indeed, by placing an exclusive OR function between these bit lines prior to setting the data register, a very powerful operation in terms of shortening the length of these arrays is obtained.

Now let us examine what our two example functions would look like using a functional memory. Referring to Figure A-6, the data register is at the bottom and the inputs are labeled. This figure represents the function exactly as in Figure A-2. The top word of the memory is the A input to function 1. The next word is the $\overline{B}$ C input. The last word represents function 2. Notice the mask which separates the inputs from the outputs. Also notice the manner in which the ORing operations are accomplished.

This is the basic concept of a functional memory, most of which was described in a paper by P. L. Gardner (reference 8). Next, the concept of the functional memory module will be described. This module contains several extensions of the above described concept.

B. The Functional Memory Module — A Systems Building Block

In designing the concepts of a functional memory into a practical system, the block diagram of Figure A-7 evolved. The memory array, selector register, and mask and data registers represent the essence of the concept as previously described. However, in order to take advantage of the conventional memory

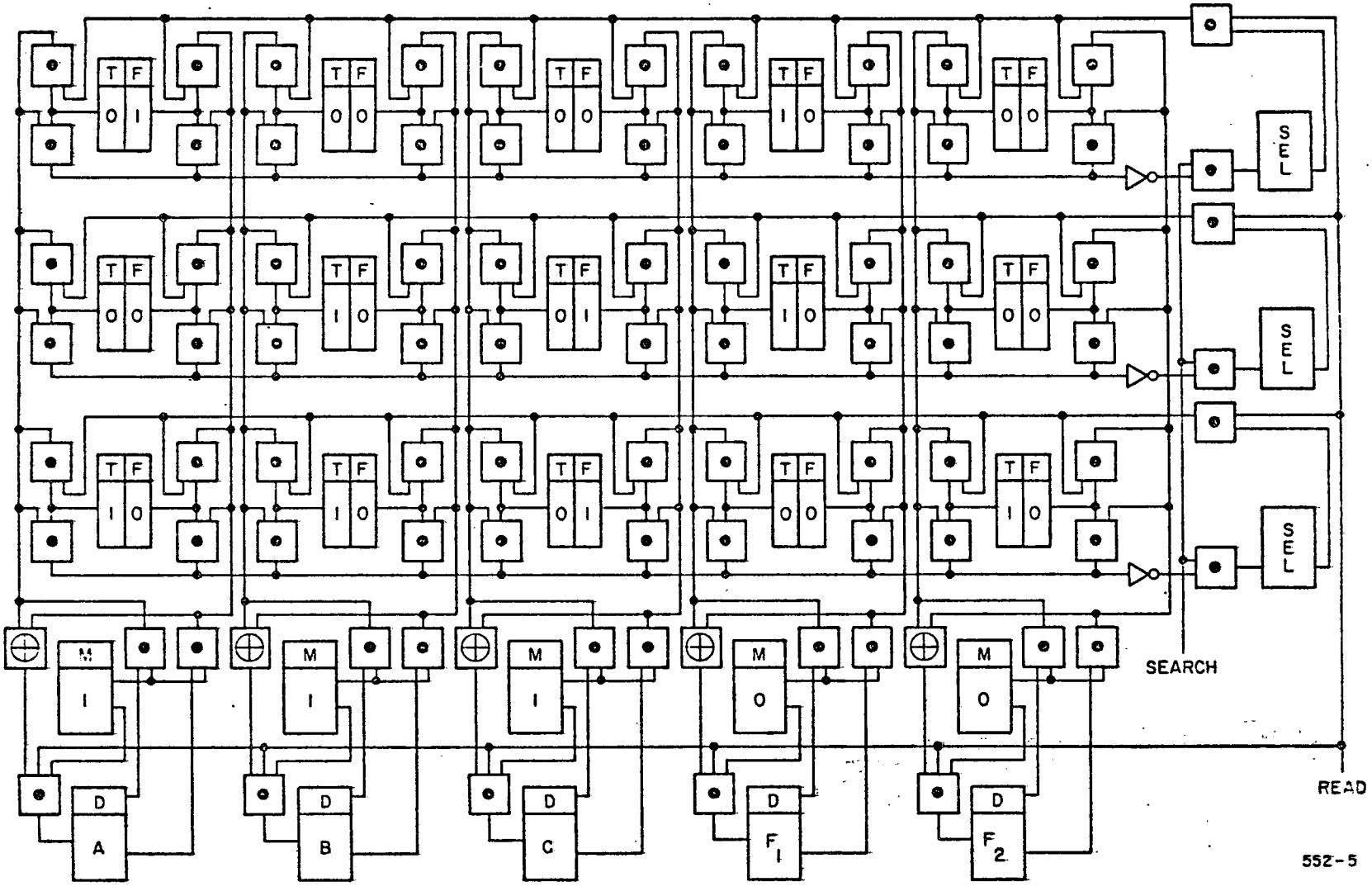


Figure A-6. PROGRAMMED FUNCTIONAL MEMORY UNIT

aspects of this device, the memory control, address register, and read control sections were necessary.

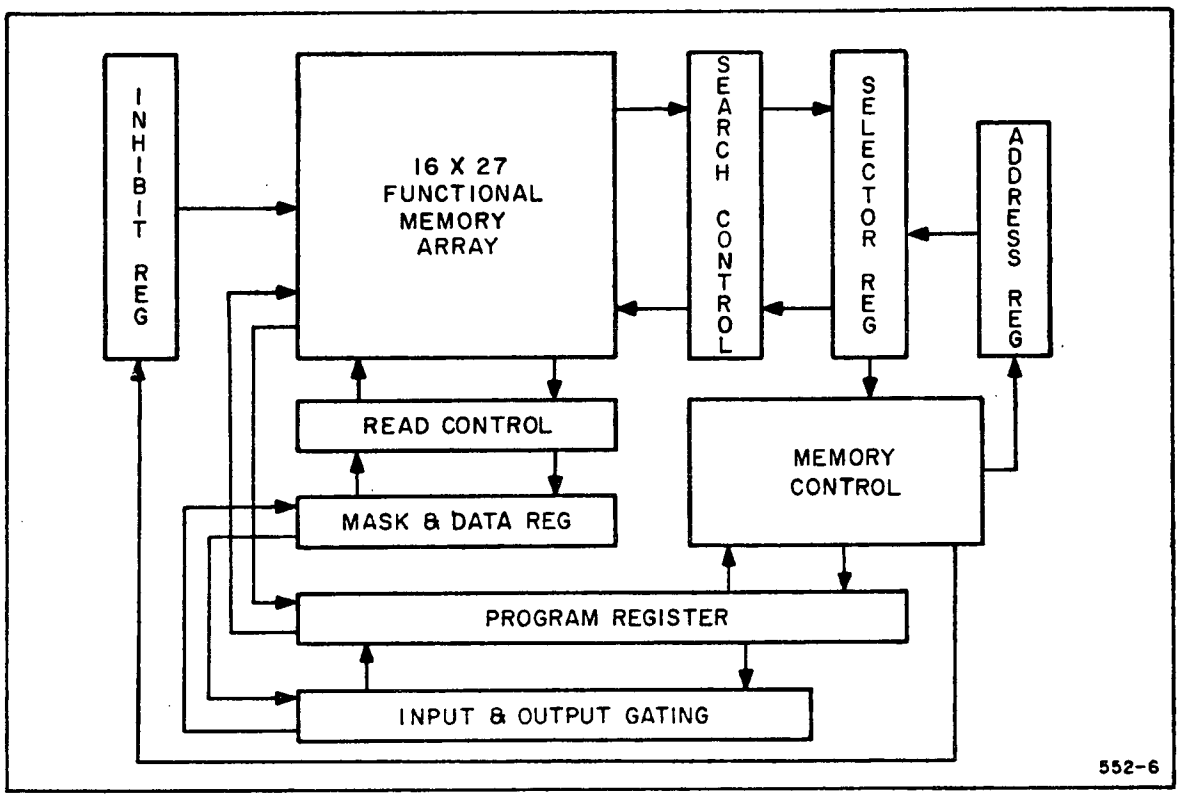


Figure A-7. FUNCTIONAL MEMORY MODULE BLOCK DIAGRAM

Since applications typically involve more than one module and since data outputs from one module are not normally aligned with data inputs to another module, the input and output gating sections are necessary. Further, this process of transferring data between modules requires control so the program register is used to decode and control data flow type instructions. These instructions are typically stored in the same module as the functional table; and, thus, some means of excluding them from the search phase must be provided. This is done in the inhibit register. Finally, it was found that part of the read functions, as described

above, should be carried out during the search phase and so the search controls are provided. These sections will now be described in detail.

Examining various applications of the memory array as pictured in Figure A-6, it was found that inefficient use was being made of memory words. In many cases, several terms could have been formed in one word if only the means were available. Furthermore, iterative processes such as carry propagation could be handled more efficiently if greater flexibility in combining and creating term line expressions were provided.

C. The Memory Cell

Figure A-8 shows a typical cell of the 16 x 27 functional memory array. In addition to the true and false flip-flops (TFF and FFF), two control flip-flops (CFF1 and CFF2) have been added. These flip-flops are used as search control parameters to gate the incoming Boolean variable to one of four term lines TL1—TL4. This is done through gates D1-4 and C-3. As can be seen, either the true bit line (TBL), through gate 01, or the false bit line (FBL), through gate 02, may be placed on any one of the four term lines. This allows one to generate four terms to be combined during the search cycle as will be seen when the search controls are discussed.

During the read phase of the functional memory cycle, the word line becomes active if this word cell is selected. Then through gates R1—R4, one to four terms of the output are gated onto bit lines BL1, BL2, BL3, and BL4 simultaneously. These terms will later be shown as inputs to an exclusive OR network when the read controls are discussed.

The CLOCK, used concurrently with LD1-4 and their complement, represents the load memory controls. This CLOCK line is activated by the upper four bits of the memory address, while the lower two bits generate LD1-4. Hence, it takes four memory accesses to load a word of memory for functional memory

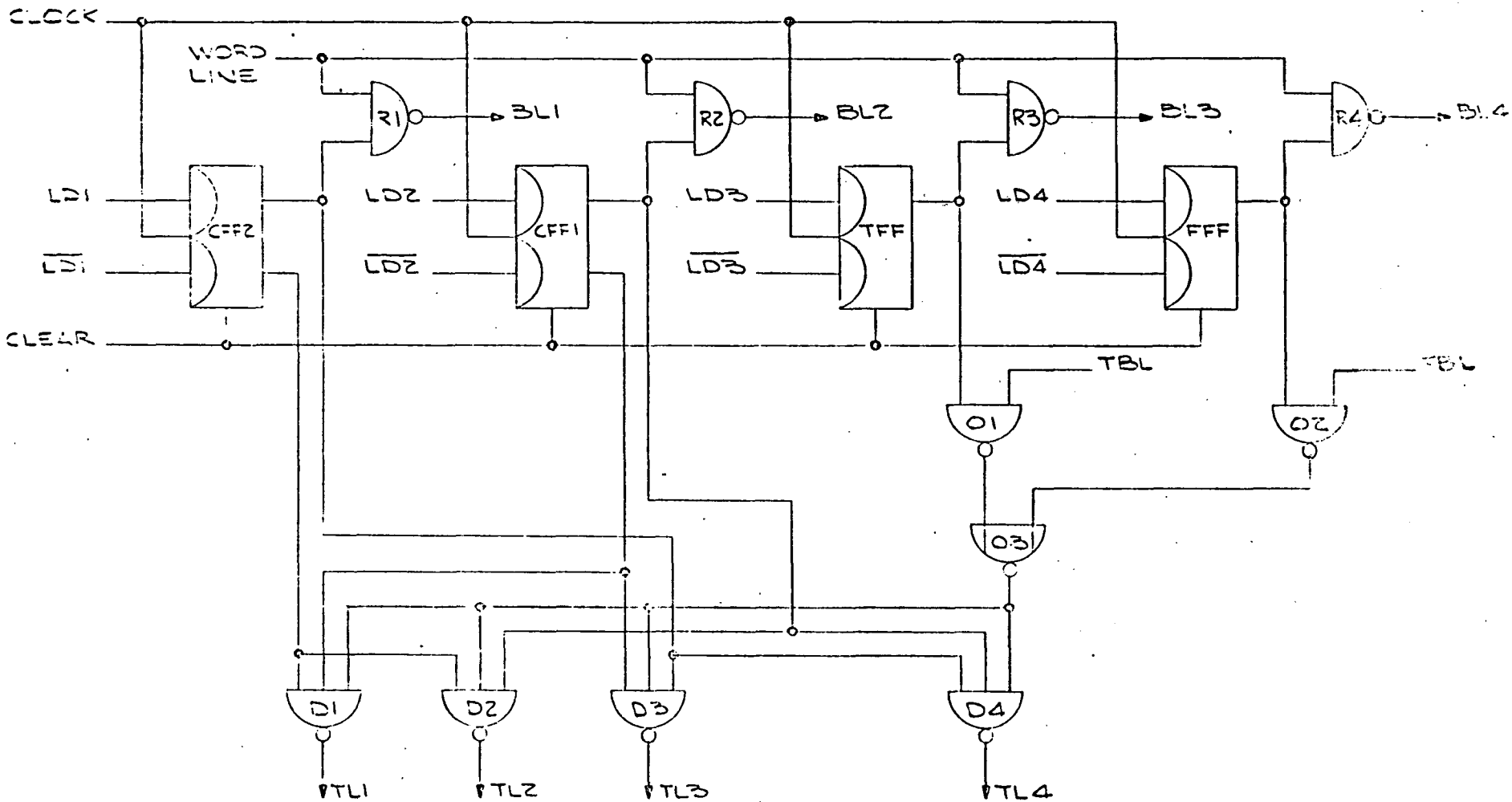


Figure A-8. FUNCTIONAL MEMORY CELL

usage. Conversely, when the module is being used conventionally, each word line accesses four words of memory. The lower two bits of the address are then used to gate the appropriate bit line into the program register.

D. The Inhibit Register

The inhibit register (shown in Figure A-9 is a 16-bit register. For each bit of this register that is set, the corresponding term lines are activated. This effectively inhibits this functional word from taking part in the functional search cycle since the inverter (which will be discussed in the search controls) output is held inactive.

E. The Search Control and Selector Register

In the search control (shown in Figure A-10), the 4 term lines are terminated in inverters. The outputs of these inverters (together with the terms C2I, C4I, and C6I) then form the inputs to a combinatorial network. The purpose of this network is to form the terms C2O (meaning any two inputs active), C4O (any four inputs active), and C6O (any six inputs active). These terms then become the C2I, C4I, and C6I inputs of the next word above this one in the memory array. They are used primarily to generate a carry propagation term during the search phase of the functional memory cycle. The network of Figure A-9 implements the following logical equations.

$$C20 = TL1\ (TL2 + TL3 + TL4) + TL2\ (TL3 + TL4) + TL3TL4 + C2I\ (TL1 + TL2 + TL3) + C4I$$

$$C40 = TL1 \cdot TL2TL3TL4 + C2ITL2TL3TL4 + C4ITL1\ (TL2 + TL3 + TL4) + (C4I + C2ITL1)\left[TL3TL4 + TL2\ (TL3 + TL4)\right] + C6\ \ (TL1 + TL2 + TL3 + TL4)$$

$$C60 = C6ITL1\left[TL3TL4 + TL2\ (TL3 + TL4)\right] + C6ITL2TL3TL4 + C4ITL1TL2TL3TL4$$

The manner in which the 7 inputs (TL1—TL4, C2I, C4I, and C6I) are combined to become inputs to the selector cell is controlled by the flip-flops EOCFF1

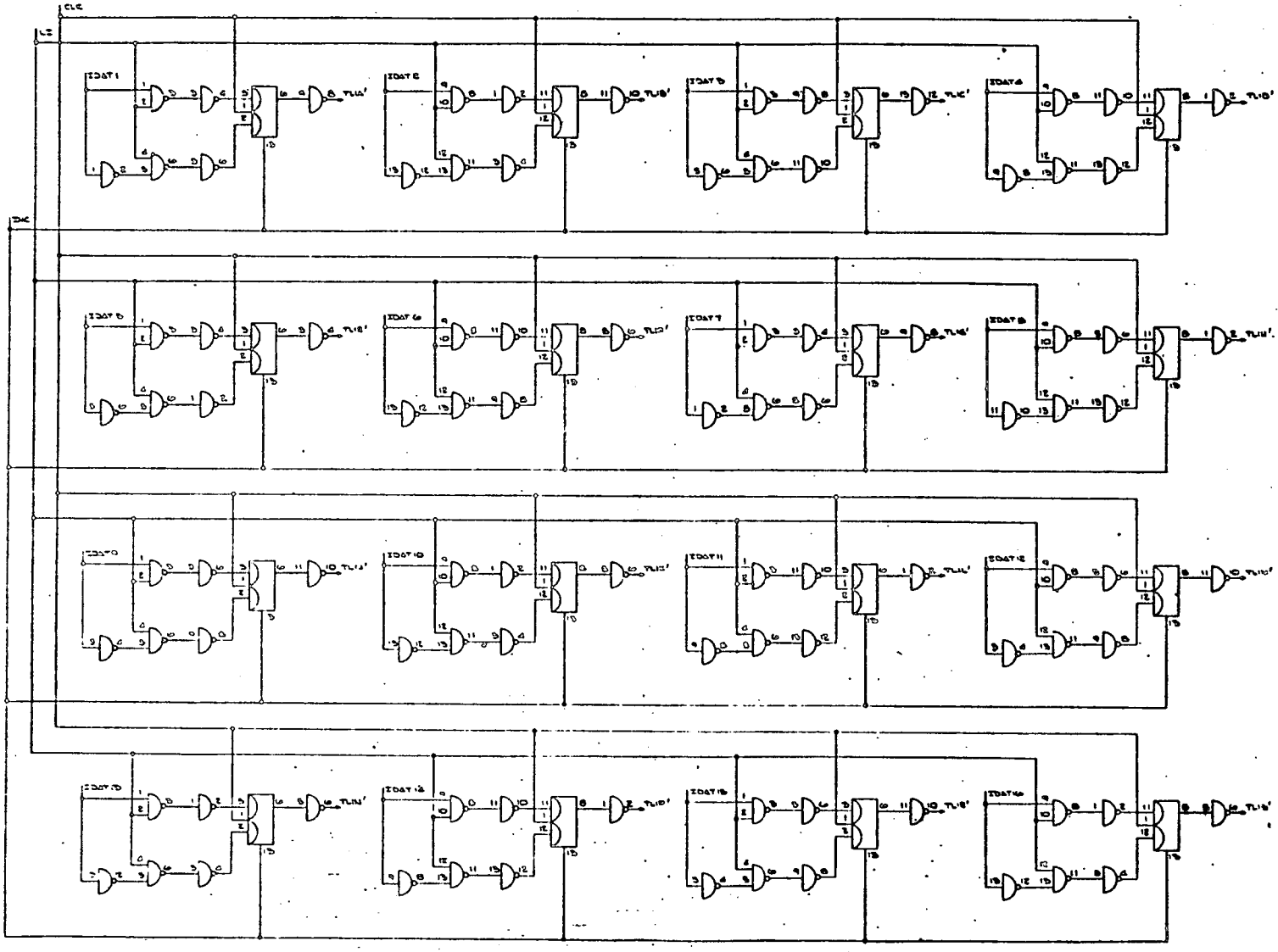


Figure A-9. INHIBIT REGISTER

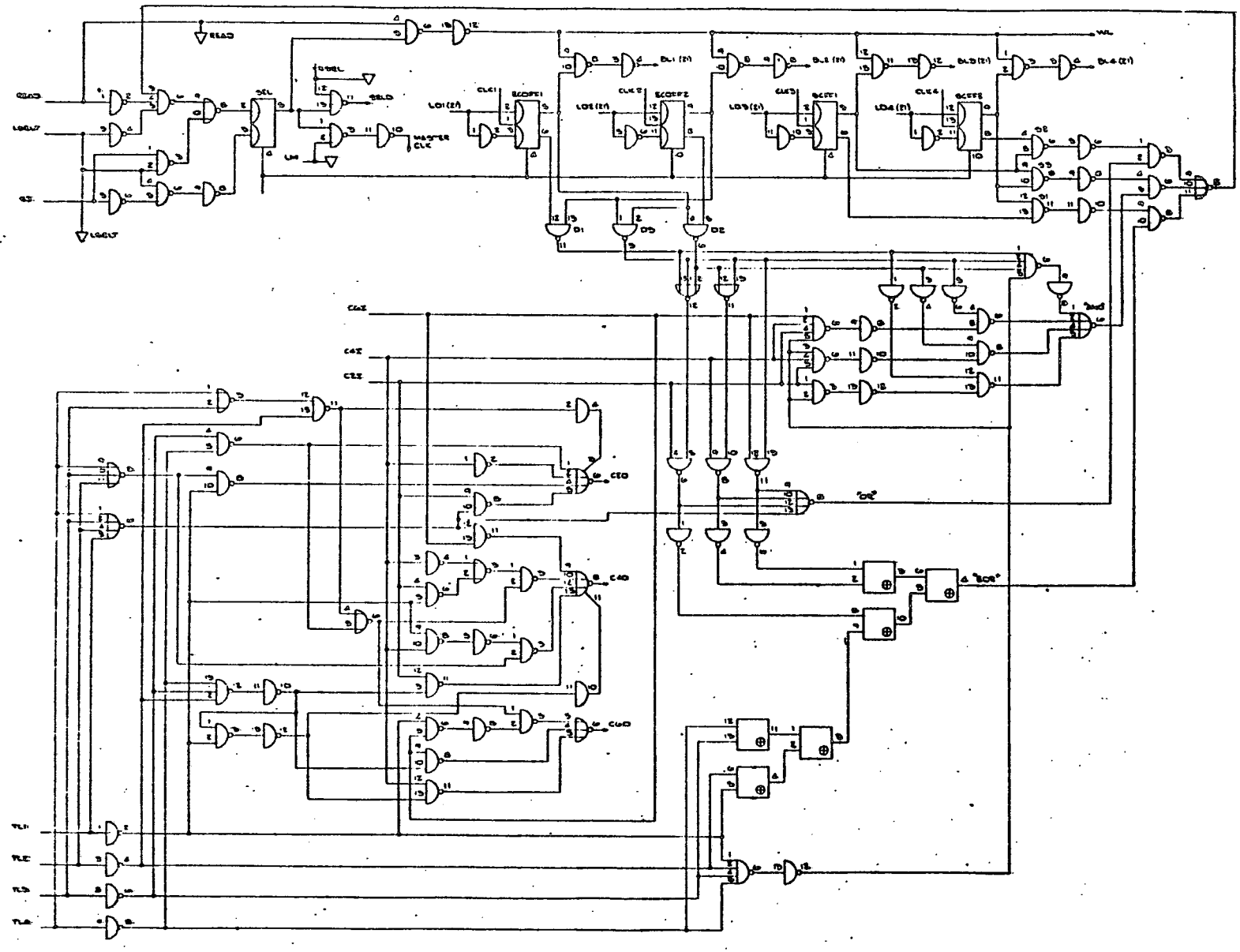


Figure A-10. SEARCH CONTROL AND SELECTOR REGISTER CELL

and EOCFF2. If gate D1 is active, then C2I is gated into the exclusive OR, the OR, and the AND networks. If gate D2 is active then C2I and C4I are both sent into these networks. Finally, if gate D3 is active all 7 inputs are sent into the networks.

The selector cell (SEL) may be set by 1 of 3 gating networks. This is controlled by the flip-flops SCFF1 and SCFF2. When gate S1 is active, then the exclusive OR of term lines 1 through 4 and any combination of C2I, C4I, and C6I as discussed above becomes the input. Similarily, gates S2 and S3 select the OR input or the AND input respectively. In this manner, a good deal of power is added to the functional memory search cycle in that combinations of terms both from this word and prior words are allowed to select the outputs of this word. This type of operation becomes very valuable in implementing iterative type functions.

The 4 control flip-flops which direct the setting of the SEL flip-flop also form the 21st cell of the functional memory array. These flip-flops are loaded by signals LD121, LD221, LD321, and LD421. When this data is presented to the cell and the proper clock signal (CLD1, CLK2, CLK3, or CLK4) is activated, the selected flip-flop is loaded. Hence, with a sequence of four loads, the cell is loaded. The clock signals are controlled by the address system described later. These signals are activated by the LM signal. During a conventional read operation, the data from this cell is read out on lines BL121, BL221, BL321, and BL421. These lines are controlled by the word line WL.

Line WL is part of the addressing scheme of the memory array and is activated by the selector flip-flop (SEL). If SEL is set and the control section activates the READ line, then line WL will read this entire word of the memory. The contents of SEL may be read by activating line RSEL. Flip-flop SEL is set during a functional cycle in the manner described above, provided that a READ is

not in progress and an address load (LSELT) is not being done. The address load is the other way that SEL can be loaded. The decoded value of the address is input on line SD and the load signal LSELT is sent to gate the data into SEL.

F. Read Controls and Mask and Data Cells

The read controls and the mask and data registers (shown in Figure A-11) will be described next. Referring to Figure A-8, the interconnections between the read control and the memory array appear on the left side of Figure A-11. Recalling that LD1-LD4 and $\overline{\text{LD1-LD4}}$ represent data to be loaded into the memory, it is seen that A1—A4 control which flip-flop is to be loaded. A1—A4 are derived by decoding the 2 least significant bits of the address register as will be shown in Figure A-14. The data inputs OR gate (DIO) is present because these data can be loaded either from the data register or from the program register. Input load memory program (LMP) represents data sent from the program register. Input LM is used to load the memory from the data register. However, in this mode, only the lower 20 bits can be loaded.

During a search operation the input SCH will allow the true (TBL) and false bit lines (FBL) to be driven by the respective set and reset sides of the data cell only if the mask cell is set. In this manner, the functional inputs are gated into the memory array.

During a read operation of a functional cycle, the 4 inputs from the memory array (BL1, BL2, BL3, and BL4) are gated into the exclusive OR network. At this time, A1—A4 will not be active and, thus, the reading OR gate (RO) will accept data from 1 of 3 sources depending upon what type of functional read cycle is called for. In a read exclusive OR cycle (RDEO), the exclusive OR output is used; in a read true only cycle (RDTO), the true bit line (TBL) is used; and in a read false only (RFO) cycle, the false bit line (FBL) is used.

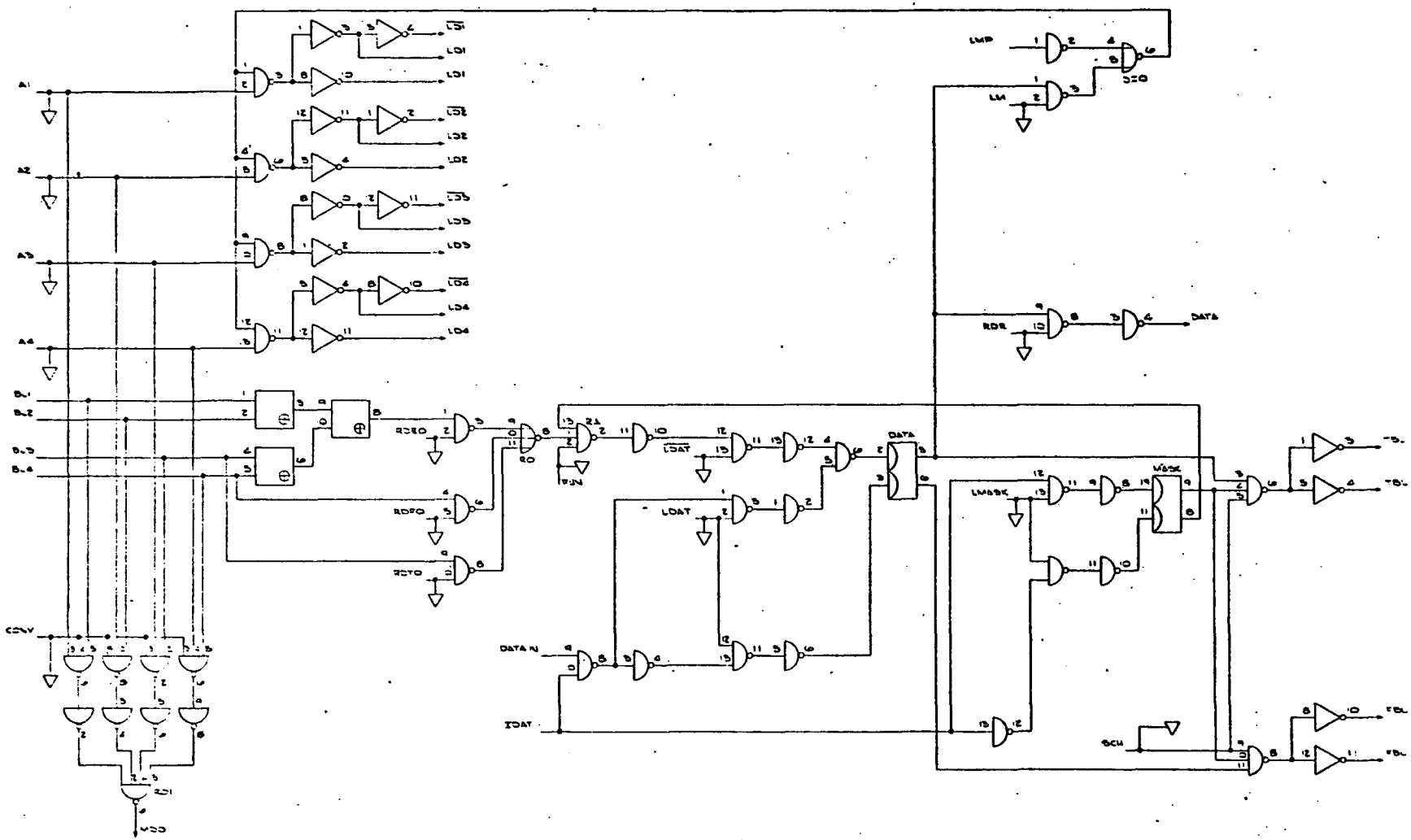


Figure A-11. MASK AND DATA CELL

From the reading OR gate, the data enters the reading AND gate (RA). From there, during a functional cycle (FUN), if the mask bit is reset, the data will be gated into the data register to form an output of the Boolean function programmed into the memory array.

During a conventional read cycle, one of the address lines A1—A4 will be active in conjunction with the conventional signal, CONV. This will cause the appropriate bit line to be sent through the other reading OR gate (RO1) and out through the memory data out (MDO) line to the program register.

The mask and data registers are loaded from the program register by gating this register to the data-in and mask-in lines (IDAT), respectively, and activating either the load data line (LDAT) or the load mask (LMASK) line.

Finally data can be read from the data register by activating the read data line (RDR). This will send data to the output decoders to be transmitted onto the bus line.

G. Input-output Gating Section

A typical cell of input and output gating subsystems is shown in Figures A-12A and A-12B, respectively. The input decoder accepts data from the internal bus system (B1'—B20') and gates it to the selected cells of the data register (DATA-IN), if the input mask register (IMSK) is set. This is done by decoding the outputs of the five decoder flip-flops (ID1—ID5) to form 20 enable terms which select which bit of the input is to enter this data cell.

The input decoder is loaded from the program register by 1 of the 2 load instructions. The interconnections between the decoder cells and the program register cells are shown in the chart on the figure. The input mask is loaded by placing its input on IDAT and enabling signal LIM.

The output decoder is loaded the same way as the input decoder and performs a similar operation with the data. In this case, the data bit leaving the

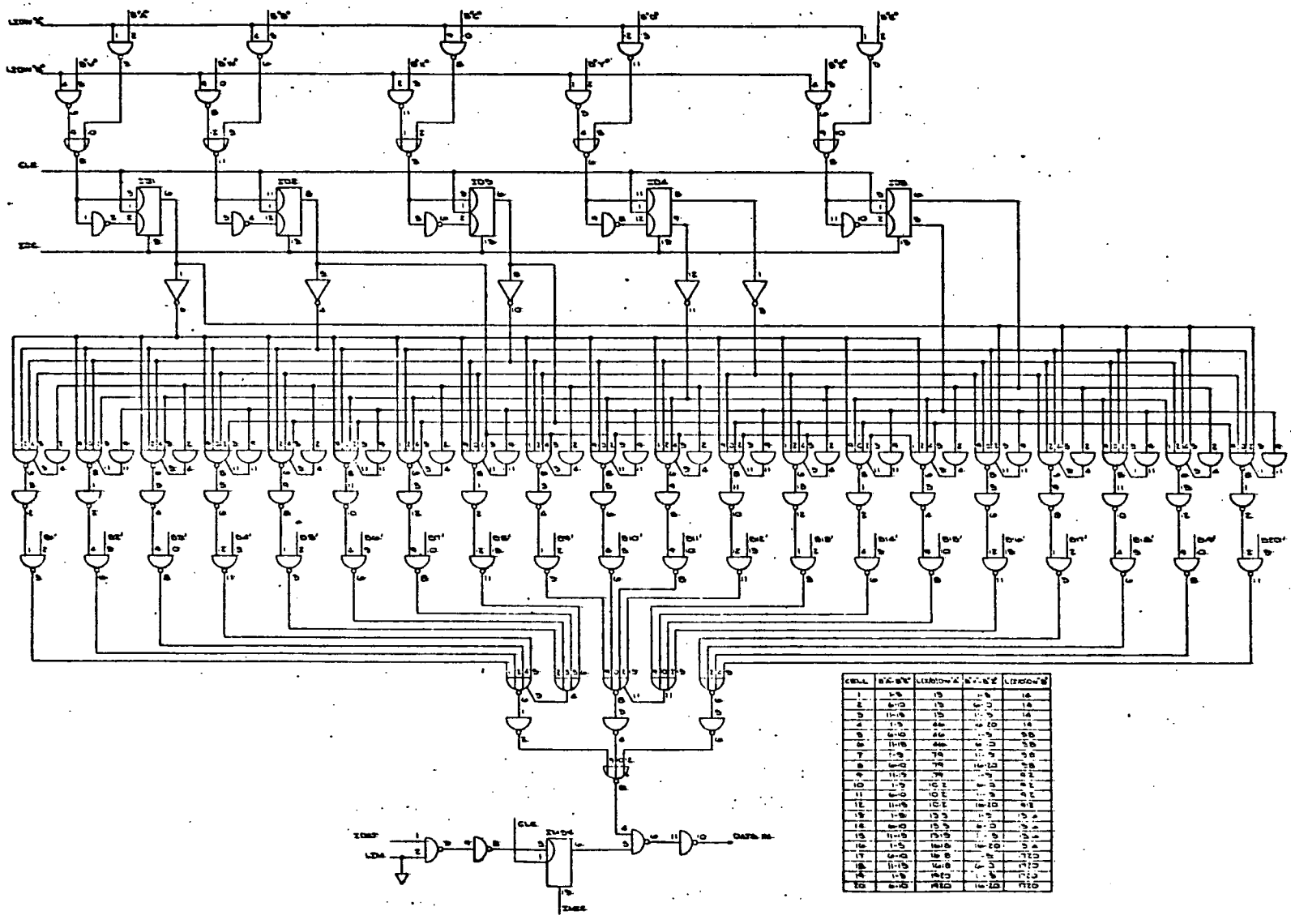


Figure A-12A. INPUT GATING

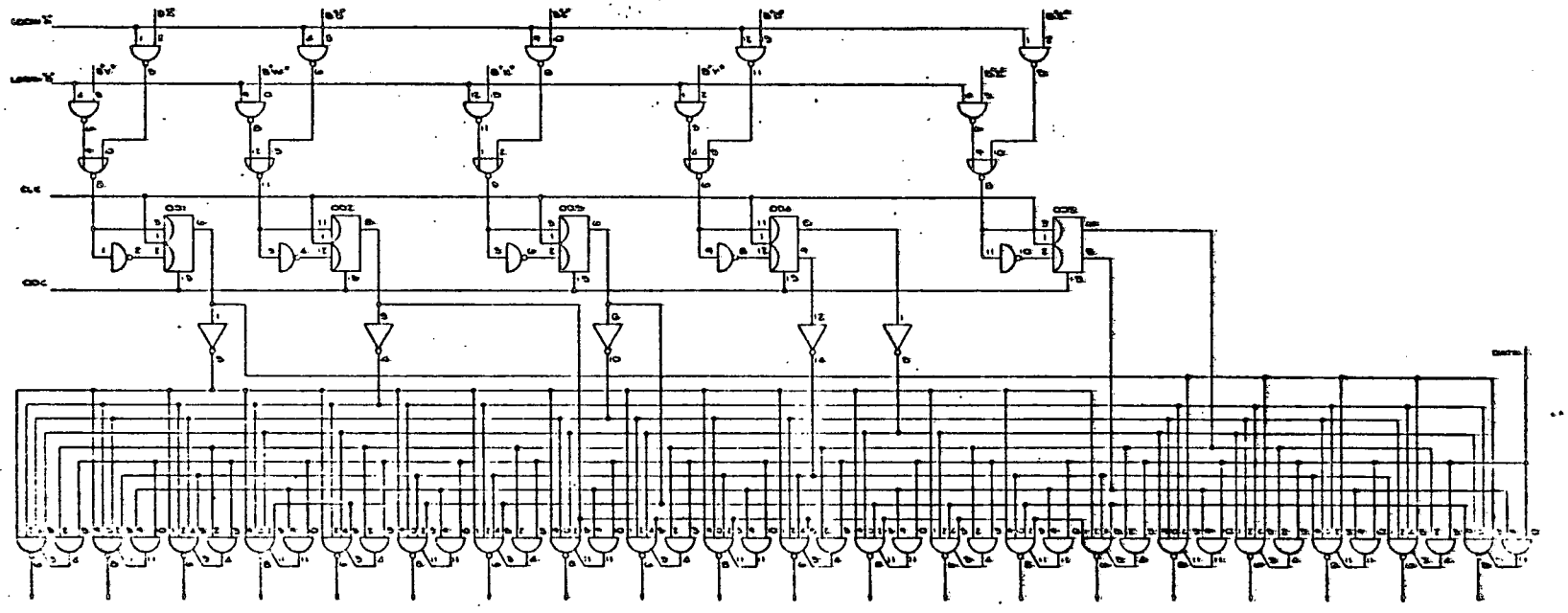


Figure A-12B. OUTPUT GATING

data register is gated to 1 of the 20 internal bus lines by the decoded value of flip-flops OD1—OD5.

By the use of these decoders, the contents of any cell of the data register can be sent out on any line of the external communications bus and, conversely, any line of this bus can enter any cell of the data register. These operations are very important for allowing the outputs of one function to become the inputs of the next one.

These 3 registers can be selectively cleared by activating the ODC, IDC, or IMKC lines. The input and output decoders are loaded four cells at a time in a sequence of five loading operations. What is shown in Figure A-12A is a table indicating how these cells are loaded.

H. The Program Register

The main communications register of the module (shown in Figure A-13) is the program register. The upper 6 bits of this 27-bit register form an encoded instruction set. Bits 22 and 23 are input to decoder number 1 whose output forms the encoded control type selector. There are 3 types of instructions: a decoder output of 1 selects the load group (T1), 2 selects the send-receive group (T2), and 3 selects the clear group (T3).

Bits 24—27 are applied to decoder number 2 whose outputs, when combined with the outputs of decoder 1, form the instruction set listed in Table A-1.

The load group loads the specified register from the program register. The input and output decoder are 5-bit quantities which are loaded 4 cells at a time from the lower 20 bits of the program register; in 5 words all 20 cells are loaded. The address register is a 6-bit quantity loaded from bits 15—20 of the program register. The 20-bit data register is loaded from the lower 20 bits of the program register as are the D and I masks. The 16-bit inhibit register is loaded from the lower 16 bits of the program register.

B

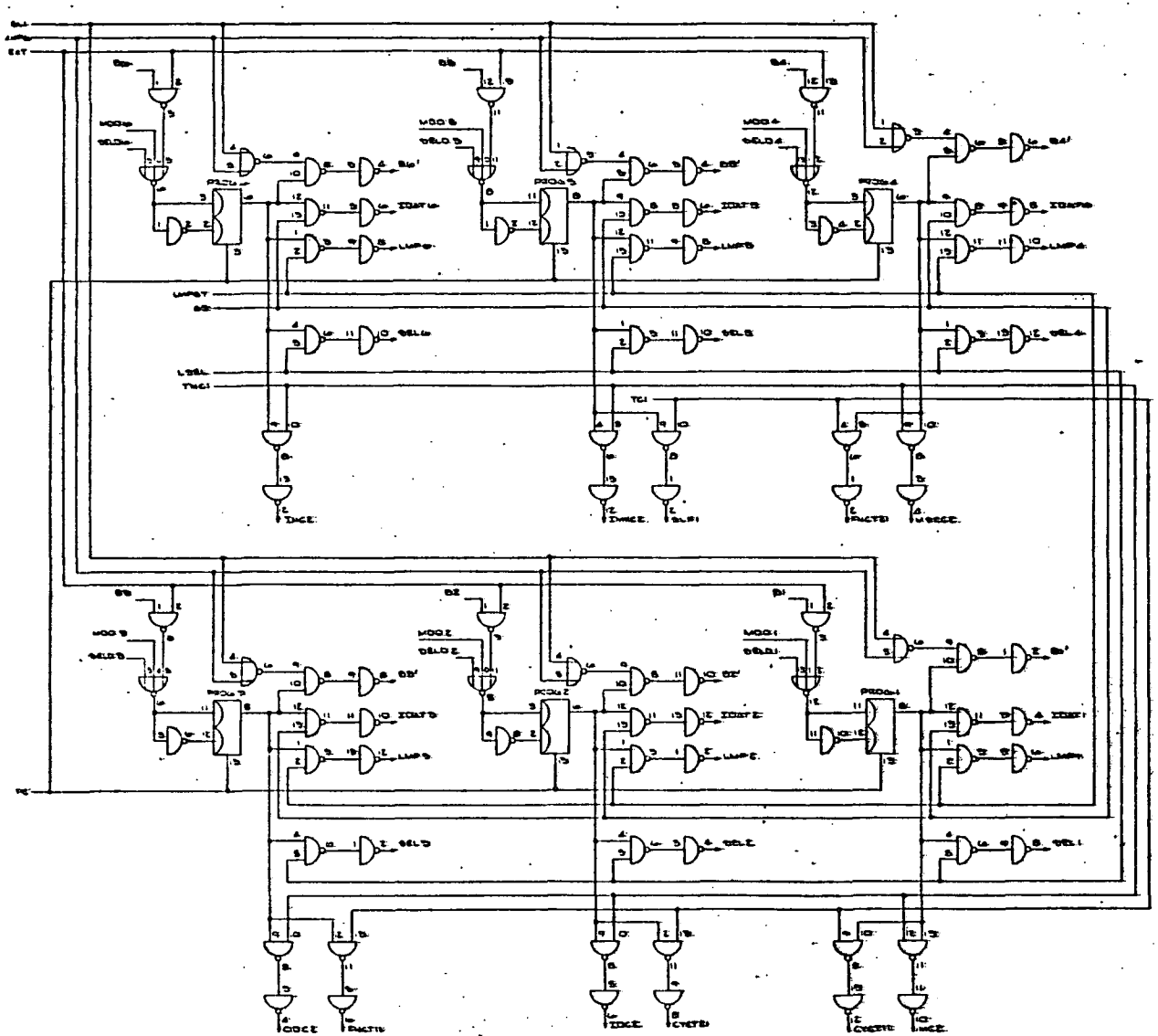


Figure A-13. PROGRAM REGISTER (Sheet 1 of 4)

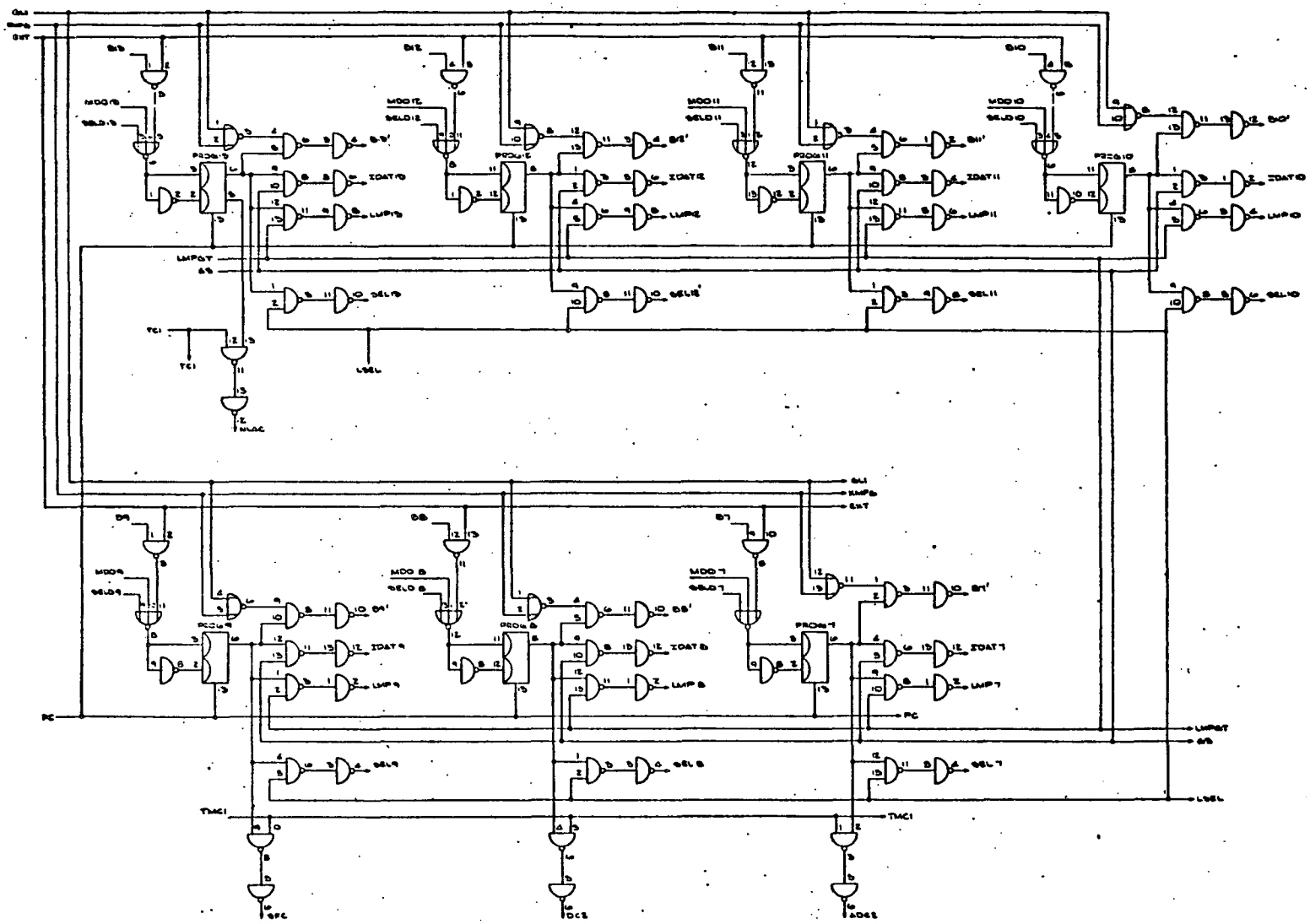


Figure A-13. PROGRAM REGISTER (Sheet 2 of 4)

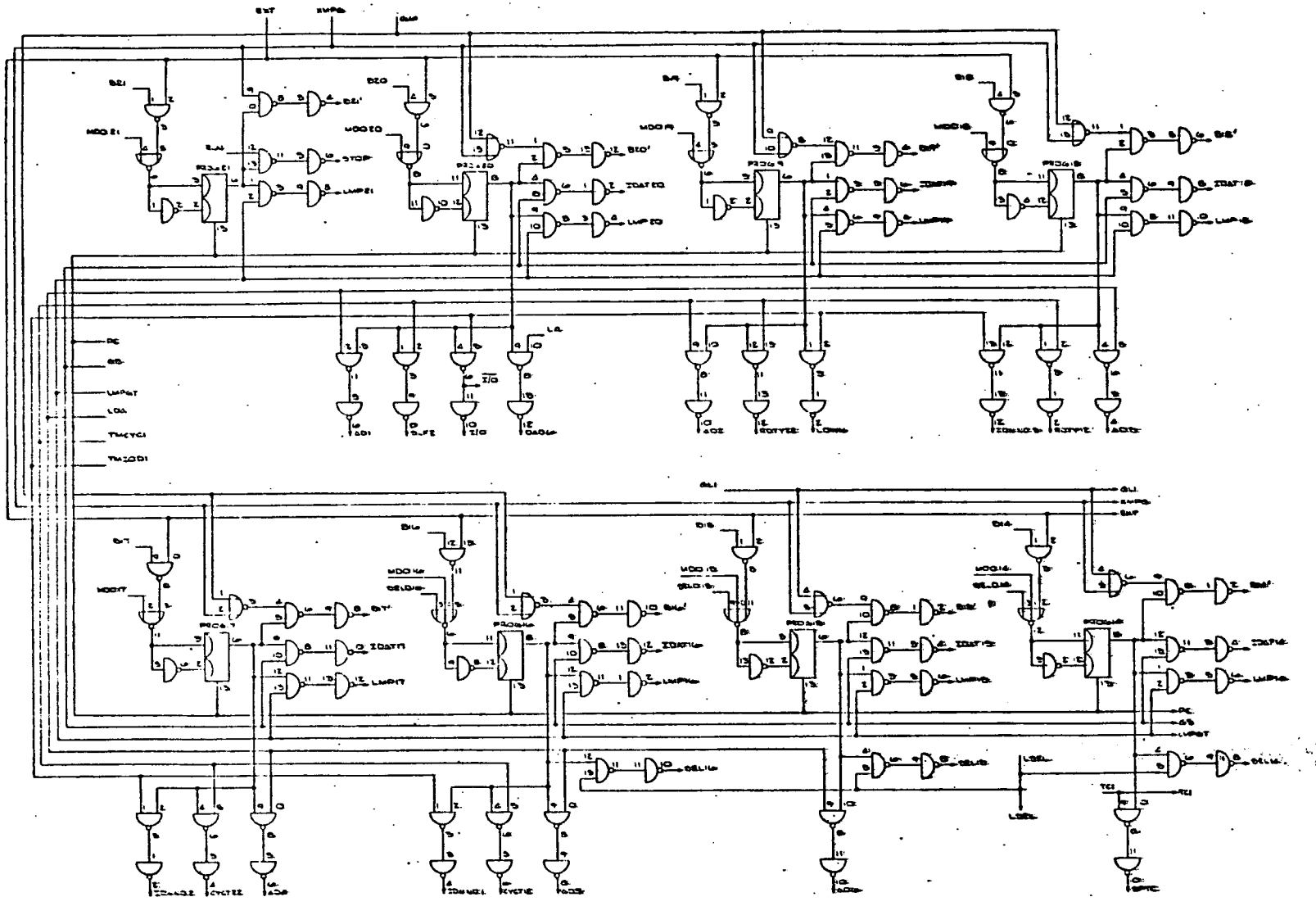


Figure A-13. PROGRAM REGISTER (Sheet 3 of 4)

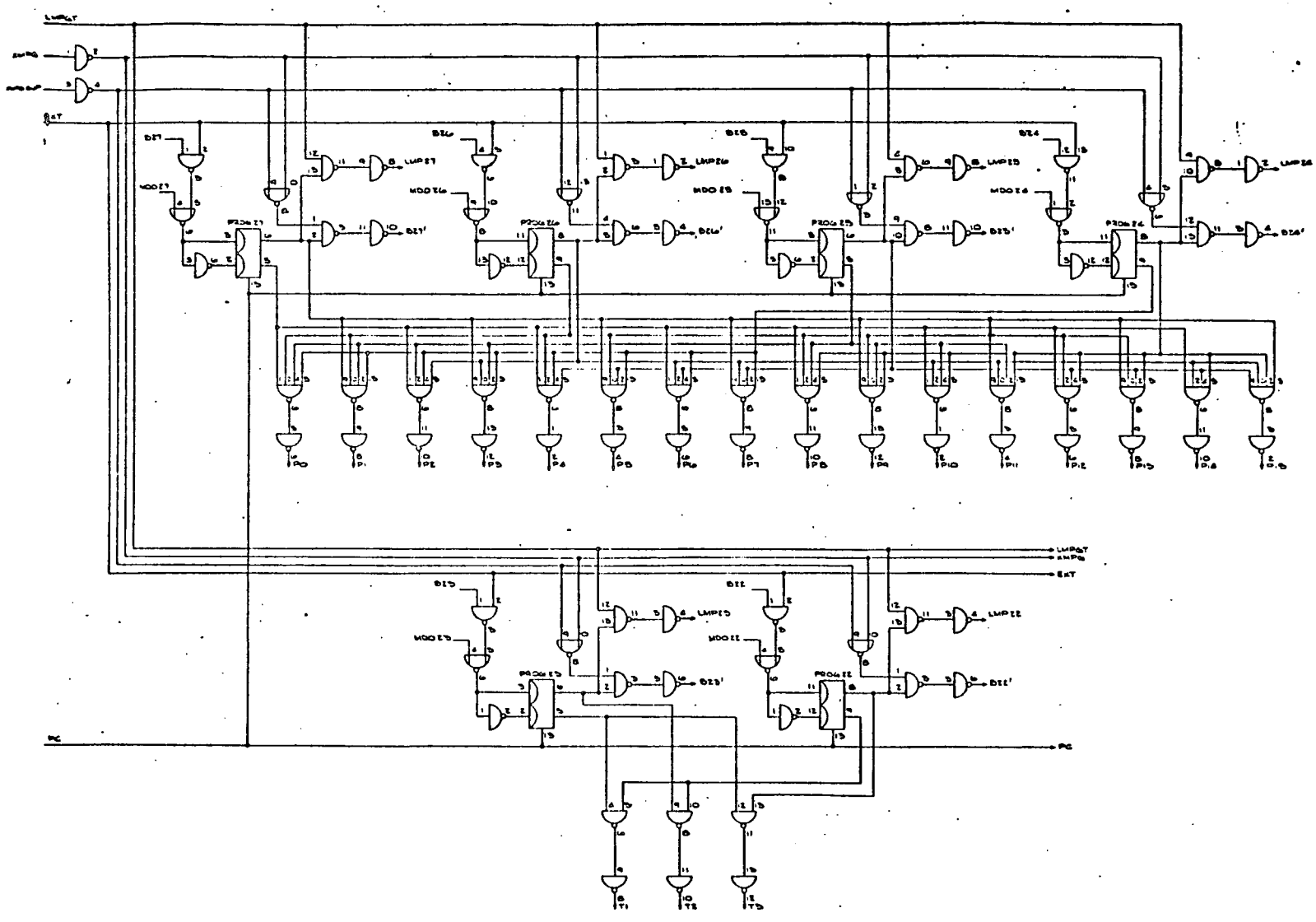


Figure A-13. PROGRAM REGISTER (Sheet 4 of 4)

Table A-1. INSTRUCTION SET

CODE	LOAD GROUP (T1)	SEND-RECEIVE GROUP (T2)	CLEAR GROUP (T3)
P0	Input Decoder W1	Address	Memory
P1	Input Decoder W2	D mask	Input decoder
P2	Input Decoder W3	Data X	Output decoder
P3	Input Decoder W4	Data I	D mask
P4	Input Decoder W5	I mask	I mask
P5	Output Decoder W1	Inhibit	Inhibit
P6	Output Decoder W2	Selector	Address
P7	Output Decoder W3	Cycle	Data
P8	Output Decoder W4	Input-output decoder	Load memory from data
P9	Output Decoder W5	Clear	Load selector
P10	Address	Load address from data	Send ID
P11	Data	Transfer control	
P12	D mask	Get out bus 1	
P13	I mask	Get out bus 2	
P14	Inhibit	Get out bus 3	
P15	Memory	Get out bus 4	

A load memory command, when received by a module which is not in control, will load an address from bits 15—20 and will set the module flip-flop. The module will then gate each load of the program register into successive memory locations until the bus line is dropped. (Note the control state will be described later.)

The clear group, when decoded from the program register, clears the specified registers. However, the last 3 instructions in the group are not clear commands. The load memory from data command contains an address in bits 15—20. This address is gated into the address register and the data register is gated into the lower 20 cells of that memory word leaving the upper 7 bits unchanged. Since this instruction affects the address register, it is normally

accompanied by a stop command which is bit 21 set. The output of program register bit 21 is ANDed with RUN to form STOP. This will reset the RUN flip-flop which will inhibit the address register from stepping.

The load selector command will gate the lower 16 bits of the program register (signal LSEL) into the selector register (signals SEL16 to SEL1). This command can be used to preset the selector register prior to doing a functional read. Since the selector is also used in the addressing system this command must contain a stop.

The send ID command alerts the bus system to connect other modules to this one. It can only be issued by a module that has control. In this command, bit 20 signals the end of an ID sequence, bits 17—19 specify which 16-bit sector is being addressed, and bits 1—16 represent a 16-bit sector specifying which of 16 modules is to be connected. In this manner, up to 128 modules can be simultaneously connected to a single module. After this command is executed, all data transferred over the bus will be sent to all connected modules. To change the connect configuration, simply send another ID command since this command sequence operates in a clear-connect mode.

The send-receive group is very dependent upon the control flip-flop. If the module has control then it is a transmitter and all the other modules connected to it are receivers. When one of these commands is received, it is decoded and the indicated action is taken. All of the transfers except the data transfers are from the program register to the program register.

The receive address command causes signal LDA to be activated which gates bits 20 through 15 to the address register through gates AD1—AD6. The receive D register command activates signal G5 which gates program register bits 1 through 20 out of the gates IDAT1—IDAT20. It also activates the load mask (LMASK) line to gate these bits into the data mask.

The data X and data I commands are used to send data between modules. For these commands, the upper 6 bits of the program register are sent by signal XMPGUP along with the 20 data bits sent by RDR. For the receiving module the program register bits are decoded and a LDAT signal is generated to place the data in the data register going through the input decoder and input mask.

In the transmitting module a data I command will also cause a LDAT signal. This will feed the data bits back into the data register through the input decoder. In this manner a shifting operation can be performed.

The receive inhibit and I mask register commands operate in the same manner as the receive D mask register command, except in these cases the load signals activated at LI and LIM, respectively.

The transmit selector command is used when a function has too many inputs and outputs to fit within one module. In this case the module containing the inputs does a functional search and then transmits the resulting selector to the module containing the outputs which then does a read cycle. The manner in which this is accomplished is that control will reside with the outputs module. This module will send a search cycle command to the input module and then it will send the transmit selector command and wait. The input module will complete the search cycle and then decode the program register. This will activate a read selector (RSEL) which will gate the selector register into the lower 16 bits of the program register. Then a transmit program register (SMPG) is issued to send the selector back to the output module. When the output module receives this information, it the generates a load selector (LSEL) signal to gate the lower 16 bits of the program register into the selector. It then generates a functional (FUN) read (RDEO), RDTO, or RDFO according to its configuration. After the read cycle is completed the next instruction is read from the address in the address register.

The receive cycle command specifies the cycle type and the read type to be executed. The decoded command will activate either a functional operation (FUN), or a conventional operation (CONV). Program register bits 18 and 19 specify the read types as:

CONV			FUN		
Bit 18	Bit 19	Type	Bit 18	Bit 19	Type
0	0	run	1	0	Read false only
1	0	read	0	1	Read true only
0	1	write	1	1	Read exclusive OR

Program register bits 16 and 17 specify the cycle type as:

Bit 16	Bit 17	Type
0	0	Conventional
1	0	Search only
0	1	Read only
1	1	Search-read

Bit 20 specifies whether or not the transmitting module is to cycle (SLF2). The module will then execute a cycle.

The receive input-output decoder command will load either the input or output decoder as specified by bit 20 (I/O) in groups of 3 cells at a time (as specified by bits 16—18) from the program register bits 1—15. In this command, bit 19 specifies whether the transmitting module is also to load its own decoders or not (LDWN). The cells to be loaded are specified as:

Bit 16	Bit 17	Bit 18	Cells	Signal
1	0	0	1, 2, 3	GL1
0	1	0	4, 5, 6	GL2
1	1	0	7, 8, 9	GL3
0	0	1	10, 11, 12	GL4
1	0	1	13, 14, 15	GL1
0	1	1	16, 17, 18	GL2
1	1	1	19, 20	GL3

In the receive clear command, the lower 8 bits of the program register specify which registers are to be cleared. This is specified by:

Bit	Register	Signal
1	All	MC2
2	Input decoder	IDC2
3	Output decoder	ODC2
4	Data mask	MSKC2
5	Input mask	IMKC2
6	Inhibit	IHC2
7	Address	ADC2
8	Data	DC2
9	Clear your own	SFC

In this command, the transmitting module can also clear its own registers by setting bit 9.

The load address from data command is not a transmitted command. It is used to transfer a computed address from the data register bits 15—20 to the address register. Bit 20 specifies whether a 4- or 6-bit address is to be loaded (DAD6).

The transfer control command is used to transfer control to another module and clear the bus system. Bits 15—20 are the address to which control is to be transferred. Bit 13 specifies whether or not the address is to be loaded (NLAC). Bit 14 specifies whether or not control is to be retained by the transmitting module (SPTC). This is used to split the control into 2 or more independent operations to be done simultaneously. A control sequence may be terminated by transmitting control to a bus line which has no modules connected to it with bit 14 reset since this action will reset the control flip-flop.

Bits 1 and 2 specify the cycle type, 3 and 4 specify the read type, and 5 specifies self-cycle in the same manner as bits 16—20 did in the transmit cycle command.

The last 4 get-bus commands are used to access 1 of the 4 major bus systems for the purpose of transferring data and control to the various functional areas in the system. A functional area is a set of up to 128 modules connected to a minor bus system as discussed in the above presentation. These modules are programmed into distinct functions such as CPU's, fast Fourier transform, etc. Hence, data can be separately sent to these functional areas and they can be simultaneously activated. In this manner, the system can become as parallel as the problem being solved dictates.

I. The Address Register

The address register shown in Figure A-14 is a 6-bit counter capable of being loaded either from the program register (AD1—AD6) or from the data register (ADD1—ADD6). The data register can load either a full 6-bit address (line DAD6 active) or a 4-bit address leaving the lower 2 bits zero since all loading operations are part of a clear (CLR) load sequence (CLK).

Once loaded, the LSELA signal gates the address into 2 decoders. The output of decoder 2 (SD0—SD15) is gated into the selector register. The output of decoder 1 is sent to the read control A1—A4. In this manner, when the module is used conventionally, full addressing of all cells is achieved.

After reading the selected address, a step signal (STPD) is issued. This causes the address register to count up by one and, thus, prepares the module to read the next word of memory. If the current instruction contains a STOP, this will be the next address read when a run is issued, unless the address register is reloaded.

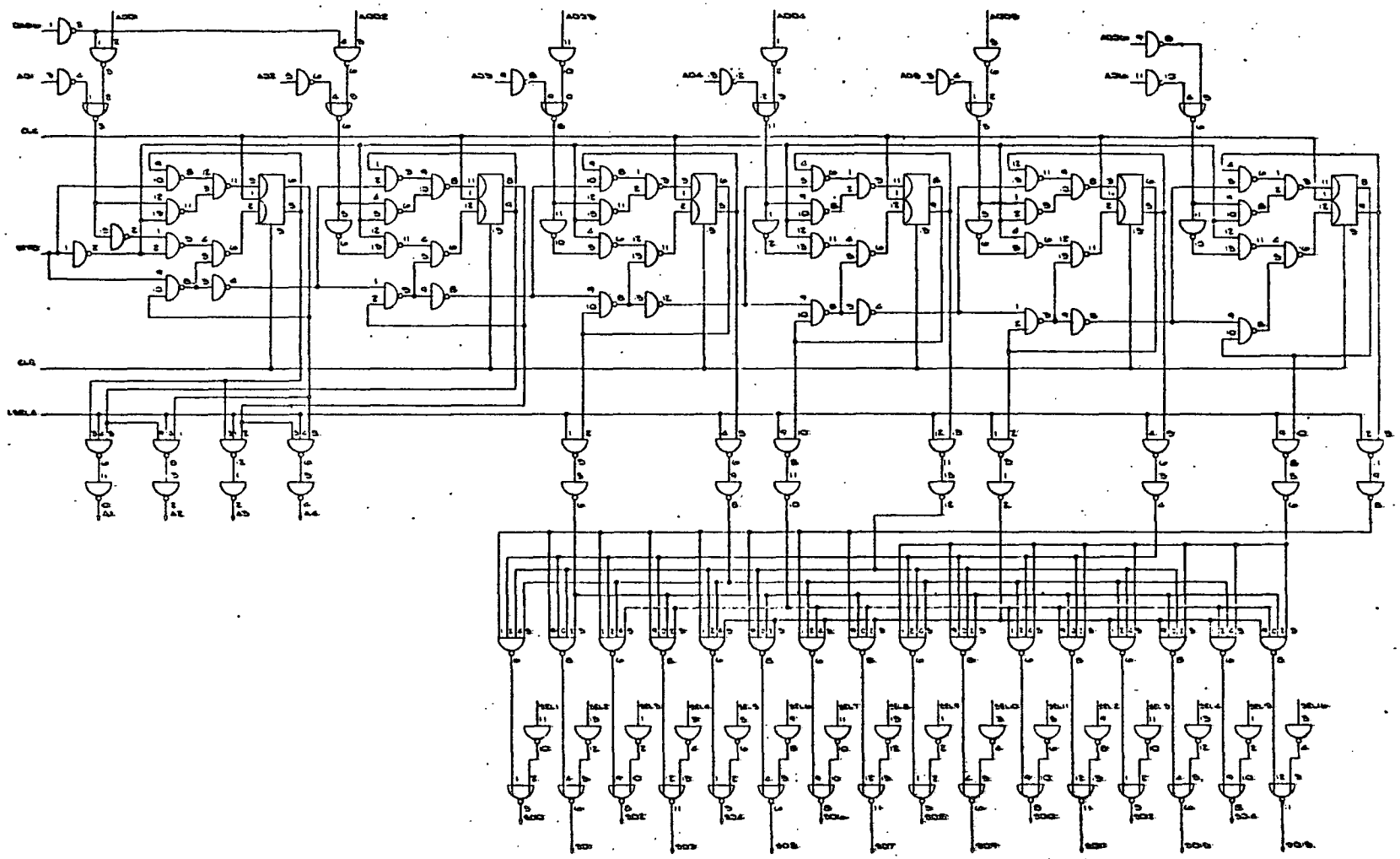


Figure A-14. ADDRESS REGISTER

J. The Memory Control

The memory control section generates all the necessary control sequences for the module. Since a large number of signals are generated, this section is presented in several parts. Many of the signals generated here are used in executing the various microinstructions of the module. Figure A-15 shows the communications controls, Figure A-16 shows the I/O decoder controls, and Figure A-17 shows the clear controls and the internal module controls.

The communications controls (Figure A-15) generate all the necessary control signals for the transmission and reception of module-to module data. The send/receive group of commands are detected by signal T2. This group is activated if the memory is not being loaded at this time ($\overline{\text{LMPG}}$). Under this group (refer to column 2 of Table A-1), the load address instruction is detected by signal P0. If at this time the module is in control (CONT), then the instruction is to be transmitted and the transmit program register (XMPG) signal is activated. All the other instructions of this group act in the same manner when the module is in control. If the module is not in control, this implies that the load address instruction has been received. In this case, the signal P0 is gated with the appropriate timing control (T) to form the load address signal (LDA).

When a load data mask instruction (P1) is received ($\overline{\text{CONT}}$), it generates a load mask signal (LMASK) and a send mask data signal (G5). Signal G5 is used to gate data from the program register onto the internal bus which connects to all the other registers of the module. Thus, by activating this signal along with the proper load signal, data is transferred from the program register to the register to be loaded. Thus, the reception of a load input mask instruction (P4) and load inhibit (P5) generate signal G5 and the respective load signals LIM and LI.

The above signals are also generated when the load group is detected (T1') and the appropriate load instructions are decoded: P12 for the data mask, P13

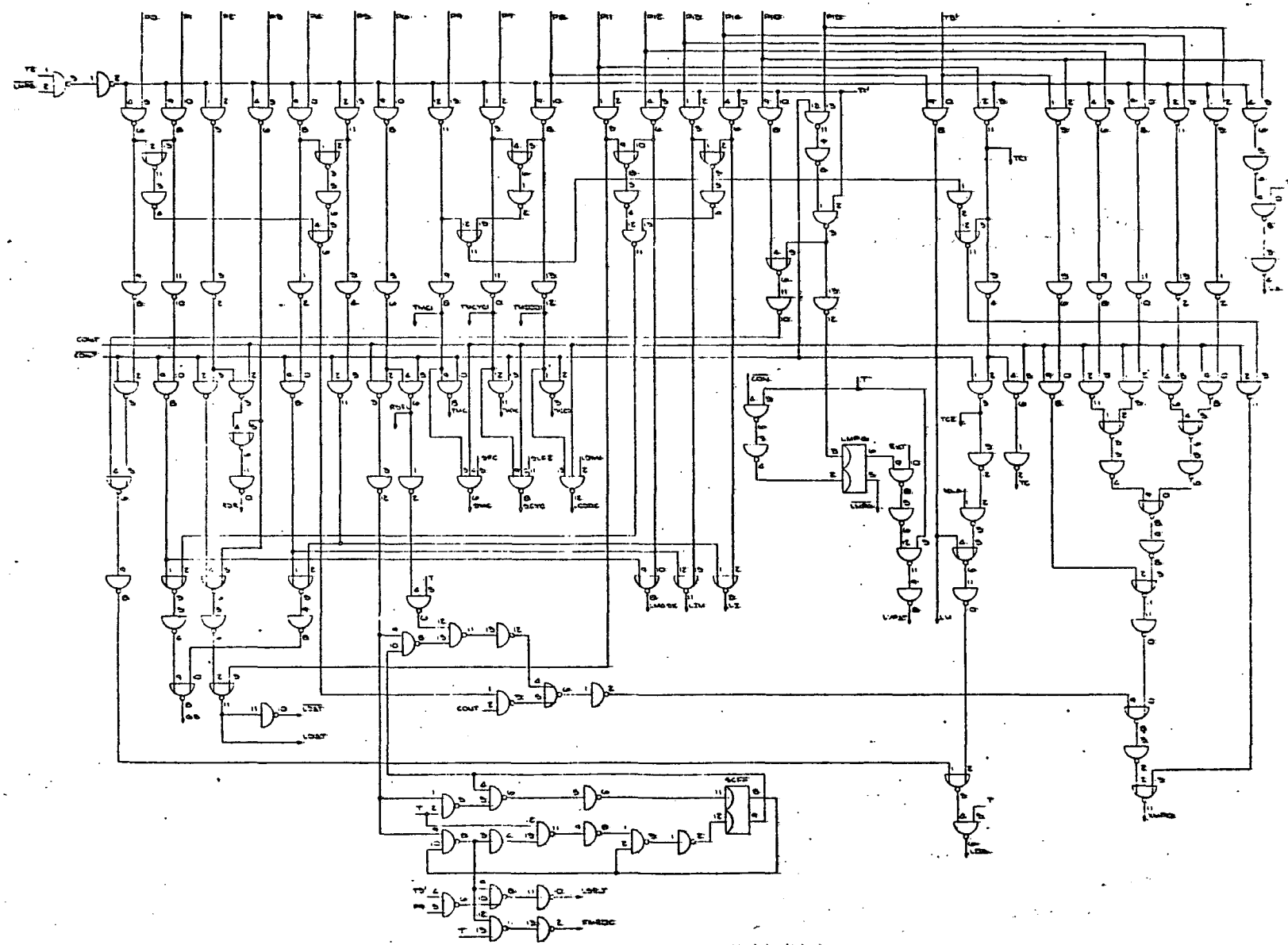


Figure A-15. COMMUNICATIONS CONTROLS

for the input mask, and P14 for the inhibit register. Further, under this group, a load data instruction (P11) generates a G5 and the load data register signal LDAT.

The LDAT signal is also generated by the send/receive group data X instruction (P2) and data I instruction (P3). The difference between these two instructions is that the data I instruction, in addition to transmitting the data to another module with the XMPG signal, can read the data register (RDR) and load it (LDAT) without regard to the control state. The data X instruction can only read data (RDR) if the module is in control and can only load data (LDAT) if the module is not in control.

The send/receive clear (P9), cycle (P7), and load I/O decoder (P8) instructions generate gating signals (TMC1 for clear, TMCYC1 for cycle, and TMIOD1 for I/O decoder) for the program register. These signals are present because a module may act upon itself in these cases as well as transmit action to another module. These signals gate the various fields of the above named instructions from the program register into the control unit. In the clear instruction, if the module is not in control, TMC is generated. If the module is in control, SFC is gated from the program register to produce signal SMC. As will be seen when the clear controls are discussed, either signal, TMC or SMC, can activate the clear signals. In the cycle instruction, if the module is not in control, TMCYC is generated. If the module is in control, SLF2 is gated from the program register to produce SCYC. Either signal, TMCYC or SCYC, can activate the cycle controls as will be seen in the internal control section. In the I/O decoder instruction, if the module is not in control, a load instruction has been received and TMIOD is activated. If this instruction is being sent and signal LDWN is gated back from the program register, then LODDC is generated to load the decoders.

The send/receive selector instruction (P6) is the most complex instruction and is used in cases where a function is too large to fit in one module. The instruction is designed to be used after a functional-search-only cycle. It is sent to the module which just completed the search. It reads the selector from that module and sends it back to the requesting module. This module then loads its own selector and does a functional read, thereby, completing the functional cycle. This instruction, when received by a module not in control, generates a read selector signal (RSEL). Then, after the selector has been read at time T, a transmit register (XMPG) is generated.

For the module in control, there is a slight problem, since when this instruction is first read, it must generate a XMPG, but when it is received back it must generate a load selector (LSELT). This problem is solved by the flip-flop SCFF. When the instruction is first read, this flip-flop is reset and a XMPG is generated. After a time delay to allow the transmission to take place, SCFF is set. Then, when the data is received from the search module, a load selector (LSELT) is generated and SCFF is reset. Also a functional read cycle is initiated (FMRDC).

In the load group (T1), the load address instruction (P10) is gated to generate an LDA signal. The load memory instruction (P15) also generates an LDA signal if the module is not in control. This instruction also sets the load memory flip-flop. This flip-flop inhibits any instructions from being decoded and allows an external signal EST to gate data into the memory via signal LMPGT. This signal will gate the program register into the memory and step the address.

There are two instructions which allow data to be placed in module registers. The load memory from data (P8 and T3) generates an LDA signal to load the desired address and an LM signal to gate the data register into the lower twenty

cells of the memory word. The load address from the data instruction (T2 and P10) generates signal LA which gates bits 15 to 20 of the data register into the address register.

The transfer control instruction (T2 and P11) is used to generate control sequences. This instruction generates signal TC1 to gate the instruction fields from the program register to the control unit. If the module is in control, signal TC is generated to clear the control flip-flop. If the module is not in control and an address load is requested (NLA), then an LDA will be generated.

The instruction set allows for 5 communications bus systems to be implemented. These instructions are: send ID (T3 and P10), get bus 1 (T2 and P12), get bus 2 (T2 and P13), get bus 3 (T2 and P14, and get bus 4 (T2 and P15). These instructions generate a transmit program register (XMPG). This data is decoded by the various bus systems and decoded in order to form a data path between a module in control and several other modules.

The input/output decoder controls (Figure A-16) generate the signals which are used to load the input and output decoders. Those from the load group T1 (load input decoder words P0—P4 and load output decoder words P5—P9) and one instruction from send/receive group T2 (load input-output decoder P8) form the inputs to this section.

In the send-receive group, if this module is not in control ($\overline{\text{CONT}}$), signal TMIOD is generated and, if it is in control and a self load is requested, the LODDC is generated. Then, as can be seen in Figure A-13 (the program register), these signals cause I/O and IDW1-3 to be sent to the control unit. As can be seen in Figure A-16, these signals are used to enable the decoders and gate out the proper load signals. Since the decoders have 20 5-bit wide cells and since the program register has 20 bits available for communications, 2 signals (GL1 and G6) are required to gate the data from the program register to the

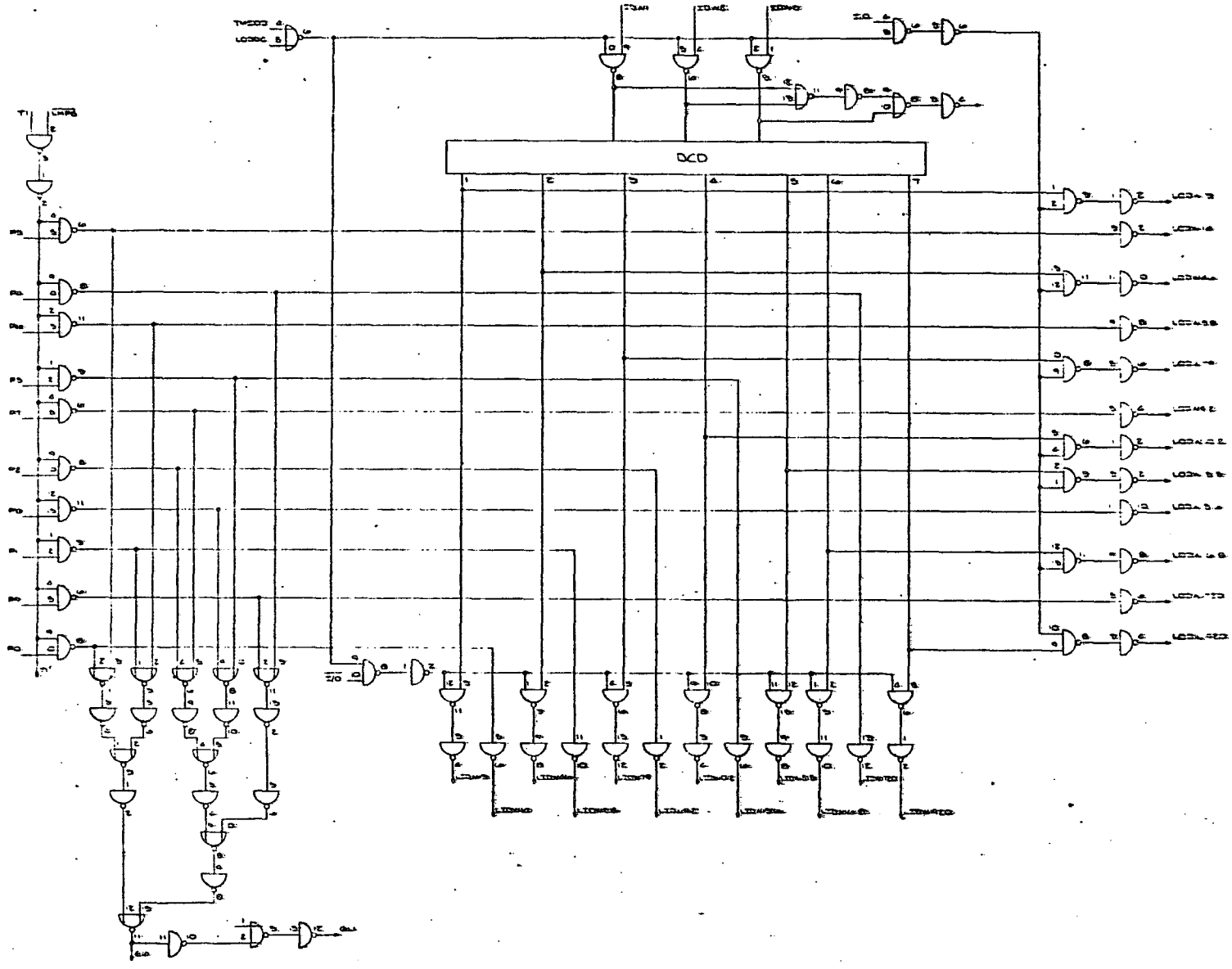


Figure A-16. I/O DECODER CONTROLS

decoder. This is because, in transmission of the decoder values from one module to another, the upper 5 bits are needed to contain other information. Hence, depending on which instruction is issued, the decoders are loaded either 3 or 4 cells at a time. Hence, G6 is required to gate the 4-cell case and GL1 is used to gate the 3-cell case.

The LIDW (load input decoder word) and LODW (load output decoder word) signals control this process. The signal names are coded to indicate what cells they load. The codes are:

Signal	Cells Loaded
W14	1 to 4
W13	1 to 3
W46	4 to 6
W58	5 to 8
W79	7 to 9
W912	9 to 12
W1012	10 to 12
W1316	13 to 16
W1315	13 to 15
W1618	16 to 18
W1720	17 to 20
W1920	19 to 20

Figure A-17 shows the clear controls. These controls can be activated by the clear group. The first 8 instructions of this group will clear the memory (P0), input decoder (P1), output decoder (P2), data mask (P3), input mask (P4), inhibit register (P5), address register (P6), and data register (P7), if the module is not being loaded. The memory clear is used as a master clear instruction. Another way to activate these controls by a module not in control is to apply a transmit clear instruction (TMC). Further, a module in control and sending a TMC instruction can also activate these controls by requesting a self clear (SMC). These last 2 methods of clearing the register specify what register to clear by

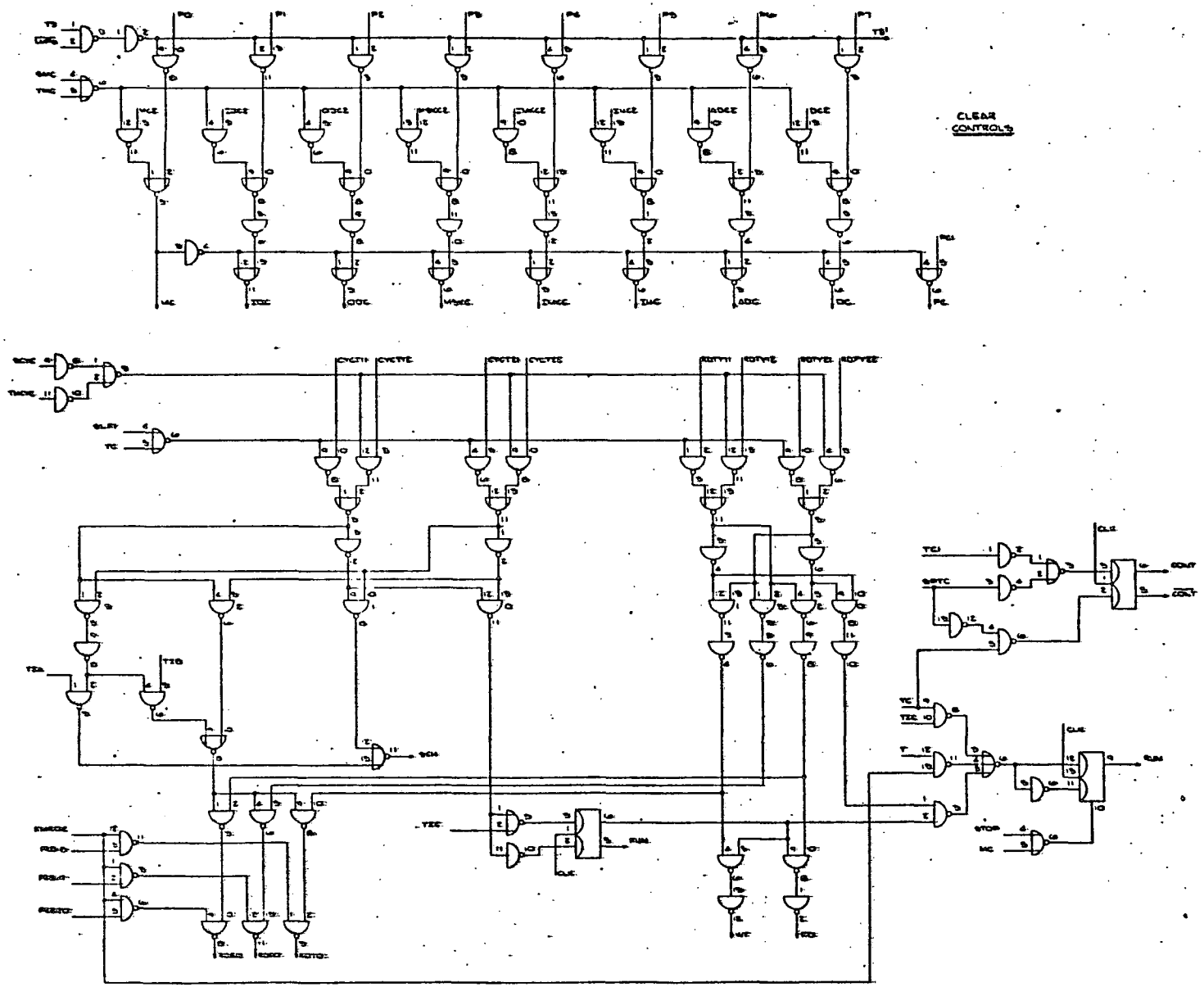


Figure A-17. CLEAR AND INTERNAL MODE CONTROLS

using bits 1 through 8 of the program register, which are gated into the controls as signals MC2, IDC2, ODC2, MSKC2, IMKC2, IHC2, ADC2, and DC2, respectively.

As mentioned previously, there are 2 types of cycles that this module can perform, a functional cycle and conventional one. There are 2 instructions that can activate a cycle, the transfer control instruction (TC) and the cycle instruction (TMCYC), and each of these can activate a self cycle (SLF1 and SCYC). A cycle instruction has 2 types of fields, the cycle type and the read type. The cycle type can be a conventional cycle (decoded value 0), a functional-search-only (decoded value 1), a functional-read-only (decoded value 2), or a full functional cycle (decoded value 3). In a full functional cycle, three timing values are required. At time TIA, a funotional search is performed. Then at time TIB, a functional read is performed. At time TIC, several actions occur. The conventional signal is set to enable the module to read microinstructions. Also, if a transfer of control instruction is present, the control and run states are set. The conventional signal is also set by the conventional cycle type.

There are several types of reading operations that the module can perform. During a functional read, the read can be done using the true-bit-line-only (RDTO), the false-bit-line-only (RDFO), or as the exclusive OR of the two (RDEO). These signals can be selected by decoding the read type where 1 means read true only, 2 means read exclusive OR, and 3 means read false only. Further, a transmit selector can also activate these read types by using the program register bits 18, 19, and 20 for RDTO, RDFO and RDEO, respectively.

During a conventional cycle, reading type 1 writes the contents of the data register into a specified memory address (WT). Reading type 2 will read the lower 20 bits of a specified memory location (RD) into the data register. Reading type 0 is a normal cycle instruction and will set the RUN flip-flop which will

start the module reading and executing microcode. This RUN flip-flop can be reset either by a stop bit or by a master clear.

The control flip-flop can be set by a receive control instruction (TC1) or by a retain control signal (SPTC). It is reset if control is being transferred (TC) and not retained ($\overline{\text{SPTC}}$).

The external pin assignments for this module are:

Pin	Function
1–27	I/O
28	Connect (CON)
29	External data request (EXT)
30	Data received reply (RCD)
31	Module busy
32	General bus request
33	Bus 1 request
34	Bus 2 request
35	Bus 3 request
36	Bus 4 request
37	Power
38	Ground

The connect signal (CON) is used to connect a module to the requesting bus system. Once connected, a busy signal is sent to the other bus systems, inhibiting any further connect requests. The external data request (EXT) is used to gate data from the bus into the program register. The data received reply (RCD) is used to acknowledge the receipt of data. The five bus requests are used by a module in control to gain access to a communications bus.

The bus systems are used to connect modules together and to coordinate replies from the connected modules to the control module. In this manner, the replies from the connected modules to a data transmission are grouped together to form a single reply to the transmitting module. Hence, timing conflicts are resolved within the bus system.

APPENDIX B

This appendix describes, in a formal manner, the telemetry preprocessing language developed in this thesis. The language (reference 12) by which the telemetry preprocessing language is being described is termed a metalanguage and is uniquely distinguishable from the preprocessing language. To formalize the definitions in the metalanguage, each definition is given the form of a statement or construct, which is analogous to a formula. However, to accomplish some unique features of such a specification, the operators define a mode of construction, or concatenation. In this text we shall employ the following symbols in the metalanguage:

< x >	the variable name x
: : =	can be formed from
|	or
$\{z\}_i^j$	z is to be repeated at least i times but not more than j times. When i is omitted, its value is assumed to be 1, and when j is absent, its value is assumed to be infinity.

The format of a metalanguage construct will be as follows. The variable named in the corner braces may be formed from the variables named or specified on the right. This definition specifically avoids any reference to concatenation on the right-hand side of the construct, since not all constructs contain the operation of concatenation, and where desired, the concatenation operator is specified. In fact, concatenation is implied by the juxtaposition of names or objects in the construct. Thus the metalanguage construct A < x > ; is intended to symbolize the linear concatenation of the object A, the variable named x, and a semicolon. Another example of this concept is the following metalanguage statement:

<object>: : = < part 1 > | A' < part 2 >

This statement reads as follows: "the variable object can be formed from the variable part 1 or from A and apostrophe and the variable part 2 in that order of occurrence."

Throughout the text, variables will be defined in their order of occurrence in the definition statement; i.e. in the above example, part 1 would be defined first followed by part 2. Then, any new variables created in these definitions would be defined in their order of occurrence. This ordering of definitions will be followed unless written text intervenes.

The definition of the telemetry language begins with the definition of the character set used in the language.

<element>: : = <letter> | <digit> | <special character>

<letter>: : = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit>: : =<binary digit> | <octal digit> | <decimal digit>

<special character>: : =<arithmetic operator>|/|(|)|, |.| =

|<blank>

<binary digit>: : = 0|1

<octal digit>: : = 0|1|2|3|4|5|6|7

<decimal digit>: : = 0|1|2|3|4|5|6|7|8|9

<arithmetic operator>: : = <additive arithmetic operator>|* | ÷

<blank>: : = <the absence of <letter> | <digit> | <special character>>

<additive arithmetic operator>: : = + | -

Thus, it is seen that the elements of the telemetry language consist of all the capital letters, the digits 0 to 9, and the listed special characters. Further, the binary digits 0 and 1 and the octal digits 0 to 7 have been defined separately from the decimal digits and can now be used to define other variables. The arithmetic operators have also been separately defined within the special

character set, and the additive arithmetic operators have been further separated from them.

Having defined the basic elements of the language, these elements may be used to form variables in the telemetry language in the following way:

<variable> : : = <name> | <decimal number>

$$\text{<name>: : = } \left\{ \text{<letter>} \mid \text{<name><letter>} \mid \text{<name><digit>} \right\}_{0}^{n \leq 9} \left\{ \text{<blank>} \right\}_{9-n}^{9-n}$$

$$\text{<decimal number>: : = } \left\{ \text{<digit>} \mid \text{<decimal number><digit>} \right\}_{1}^{4}$$

A variable in the telemetry language can be either a name or a decimal number; but, as is shown, limits exist on the size of either. A decimal number must range from 0 to 9999, and a name may not have more than 9 allowable elements in it. Note that a name when not blank must begin with a letter but may then contain a sequence of letters and digits; e.g. L101A, B47, ABLE, etc.

The general structure of the telemetry language is:

<telemetry language>: : = <valid expression> FINIS

<valid expression>: : = <format definition> | <frame definition>

<format definition>: : = <format identifier> <format specifiers> END

<frame definition> : : = <frame identifier> <instructions> END

Hence, the language has two major divisions: the format definition and the frame definition. Each of these divisions terminates with the word END and the language specification terminates with the word FINIS. Before the word FINIS is encountered, however, every format identifier must be matched by an equivalent frame identifier; otherwise, an improper specification has occurred.

Now the elements of the format definition will be described.

<format identifier> : : = <name> FORMAT

< format specifiers > : : = $\left\{ \text{< necessary format specifier >} \right\}_{3}^{3}$

$\left\{ \text{< optional format specifier >} \right\}_{0}^{3}$

< necessary format specifier > : : = < sync code > | < word size > |
< frame size >

< optional format specifier > : : = < bit rate > | < modulation code > |
< tape speed >

< sync code > : : = FSP < number >

< word size > : : = BITS/WORD < number >

< frame size > : : = WORDS/FRAM < number >

< bit rate > : : = BIT RATE < number >

< modulation code > : : = CODE < code type >

< tape speed > : : = TAPE SPEED < speed factor >

< number > : : = < decimal number > | < octal number > |
< binary number > | < binary number > < octal number > |
< octal number > < binary number >

< code type > : : = SPPH|BIPH|RZ|NRZ|NRZM|NRZL|NRZC

< speed factor > : : = 120|60|30|15|71/2|33/4|17/8

< octal number > : : = 0 < octal digit > | < octal number > < octal digit >

< binary number > : : = (< binary digit > | < binary number >
< binary digit >)

All of the necessary format specifiers must be present in the format specification; otherwise, the format specification is invalid. The only parameter representation restriction is that the binary equivalent of the FSP parameter must be the exact frame sync code including all leading zeroes.

The frame definition division of the telemetry language will be described next. Recalling the syntax of the frame definition:

< frame definition > : : = < frame identifier > < instructions > END

< frame identifier > : : = < name > FRAME

< instructions > : : = < specifier > | < control instruction > | < data handling instruction >

All instructions are prefixed with a 9-element location field which is defined as:

< location symbol > : : = < simple location symbol > | < indexed location symbol >

< simple location symbol > : : = < name > | < blank >

< indexed location symbol > : : = < name > (< simple arithmetic expression >)

< simple arithmetic expression > : : = < variable > | < variable > < arithmetic operator > < variable >

Recalling the definition of name, it is seen that there is a restriction on the location field; if the location symbol is less than nine elements, it must be left justified in a blank field.

A specifier is defined as:

< specifier > : : = CONTINUE|DIMENSION < decimal number >

At this point, it should be noted that an assumption is being made concerning the memory space of the telemetry language. The assumption is that this memory space consists of two distinct and completely independent sections: the program memory and the data memory.

The only location symbol capable of addressing the program memory is the location field of a CONTINUE statement or those names which match the location

symbol of a CONTINUE statement. All other location symbols address the data memory.

Furthermore, all data memory references must be sized with a DIMENSION statement. As examples, consider the address of PL1 of the program memory and the array ELMT of the data memory. Address PL1 is defined by:

PL1 CONTINUE

Since instructions are sequentially addressed as they appear in the frame definition division of the telemetry language, the above instruction will have the effect of assigning to the symbol PL1 the next available address in the program memory.

Array ELMT is defined by:

ELMT DIMENSION 200

Since DIMENSION statements are similarily processed for the data memory, this statement will have the effect of assigning the next available 200 addresses of the data memory to the array ELMT.

Control instructions are those instructions used to alter the sequence in which instructions are executed, specify parameters, or form similar frame definitions. These instructions are defined by:

<control instruction> : : = <go to expression> | <repeat expression> | <equate expression> | <index expression> | <if expression>

<go to expression> : : = GO TO <name>

<repeat expression> : : = REPEAT <variable> , <name> = <simple arithmetic expression> , <simple arithmetic expression>

<equate expression> : : = EQUATE <name>

<index expression> : : = <name> =<arithmetic expression>

<if expression> : : = IF (<logical expression>) <name>

The GO TO, REPEAT, and IF instructions must have null location fields. The names in the GO TO and IF instructions must appear as a location expression of a CONTINUE statement. The GO TO instruction is an unconditional branch to the specified location. The IF instruction is a conditional branch instruction. If the logical expression (to be defined) is true, then the branch to the specified location is executed; otherwise execution continues with the next instruction.

The REPEAT instruction is used to perform a looping operation. The loop index is specified by name. The initial value of the index is the value of the first simple arithmetic expression. The final value which will cause the loop to be exited is the value of the second simple arithmetic expression. The length of the loop is specified by the value of the variable. Two important restrictions must be remembered: 1) If the variable is not a decimal number, it must be the name of a previously defined index. 2) The loop index may be altered within the loop but care must be taken to insure that equality will result at the end of loop test.

The EQUATE statement must have a name in the location field. This name must match the name of a frame identifier. This statement is used to define a frame which is simply the reverse of another defined frame; a condition commonly prevalent when spacecraft recorders are used.

The index definition instruction may have a location symbol. If it does, the value of the index as computed by the definition of that index will be placed in the specified data memory location.

D

Now the arithmetic and logical expressions will be defined.

<arithmetic expression> : : = <signed variable> | <signed variable> <arithmetic operator> | <arithmetic expression> <variable> | <arithmetic expression> <arithmetic operator> | (<arithmetic expression>) <arithmetic operator> <variable> | <arithmetic expression> (<arithmetic expression>) | (<arithmetic expression>)

<signed variable> : : = <additive arithmetic operator> <variable> | <variable>

<logical expression> : : = <simple logical expression> | <logical expression> <logical operator> <simple logical expression>

<simple logical expression> : : = <logical operand> <logical operator> <logical operand>

<logical operator> : : = . <logical relator> .

<logical operand> : : = <unary operator> <operand>

<logical relator> : : = AND|OR|LE|LT|EQ|NE|GE|GT

<unary operator> : : = .NOT. | <blank>

<operand> : : = <variable> | <arithmetic expression> | <logical expression>

Notice that in both arithmetic and logical expressions, any level of parentheses are allowed. However, one important rule of operation must be remembered, the order of evaluation is left to right with no precedence among the operators. Also in logical operations, the unary operator NOT means compliment the operand to its right. If that operand is a variable, the value of that variable will be complimented prior to comparison.

In using the logical relator AND, if the two operands are of unequal bit length, leading zeroes will be added to the shorter of the two to make them equal and then the operation will be done.

The data handling instructions will be described next.

<data handling instruction> : : = <location name operation > |
<word instruction> | <sync instruction> | <sub instruction>

<location name operation> : : = <name> <parameter set> | <location name operation > < linkage>

<word instruction > : : = WORD <parameter set > | <word instruction> <linkage >

<sync instruction> : : = SYNC <parameter set> | <sync instruction> <linkage>

<sub instruction > : : = SUB, <count mode > , <variable > <parameter set> | <sub instruction> <linkage>

<parameter set> : : = <modifiers> | <modifiers > (<word control >) | (<word control >)

<linkage> : : = < additive arithmetic operation >

<count mode > : : = B|F

<modifiers> : : = <modifier > | <modifiers > , <modifier>

<word control> : : = < simple arithmetic expression> | <simple arithmetic expression >,<simple arithmetic expression > , <simple arithmetic expression >

<modifier> : : = L|PET|PEA|POT|POA

In the location name operation, the name must be a defined data memory address. Hence, the parameter set will operate on the data memory locations specified by name; whereas in the other data handling, the operations are performed on the input data set.

In all of these instructions, the location field specifies a data memory address in which the defined data value will be placed. The combination of the parameter set and the linkage define the data value. The linkage symbols are fully distinguished from their arithmetic counterparts by their location in the instruction stream as shown in the above syntax.

The plus linkage implies that the bits specified by the instruction immediately following will be appended to the right of the bits already extracted. The minus linkage implies that the compliment of those bits will be appended. In this manner a new data value is formed from the input bit stream.

The bits to be manipulated in the above manner are specified by the parameter set. The word control specifies the bit locations and the modifier L specifies their end for end reversal. If the word control is a single parameter, the value of that parameter specifies the word of the designated bit stream to be manipulated. If the word control is a three parameter set, the first parameter specifies the word, the second specifies the starting bit, and the third the number of bits to take from that word. It is these bits which will then be manipulated.

Words and word sizes of the input data stream are specified in the associated format division of the telemetry language.

The modifiers PET, PEA, POT, and POA specify that the associated data value is to be checked for a parity error. The parity bit to be used for comparison is the bit specified by the word control section of the instruction containing one of these modifiers. The data value to be checked is specified by those instructions that are linked to this instruction. The type of check to be performed is specified by E for even parity or O for odd parity. The time that the check is to be performed is specified by T for before manipulation or A for after manipulation. The result of the parity check is to set the sign bit of the

specified data memory word to 1 if a parity error is detected and to 0 if not. The location name operation and the word instruction operate as described above. The other two instructions perform special operations. The sub instruction is designed to check a data counter. In this instruction, the count mode specifies forward or backward counting and the variable is the counter modulus. In addition to forming the data value as described above, the sign bit of this data word is altered. This bit has a value of 1 until the three previous values of this data word represent continuous counts. It then remains 0 until three consecutive counts occur out of sequence, at which time a new counting sequence will be searched for.

The sync instruction does not place the frame sync code in the data memory, instead it retrieves the frame sync code from the format word and compares it with the data value bit by bit. It then replaces this data value in the data memory by a set of words representing: 1) the total number of errors in the frame sync pattern, 2) the number of pattern ONEs in error, and 3) the logical product of the pattern and the received frame sync code.

Comments in the language, when punched on an 80 column card, can be entered either with an asterisk (*) in column 1 or after a blank at the end of a language statement. Comments may appear anywhere within the telemetry language. The syntax of comment statements is:

<comment> : : = * <textual stream> | <blank> <textual stream>

<textual stream> : : = <element> | <textual stream> <element>

A final note on card formats, if the linked data handling instructions or an index definition statement forms a character string too long to fit on one card, an asterisk (*) in column 80 signifies that the entire next card is a continuation

card. As many continuation cards as needed may be used. However, on a continuation card an * in column 1 does not signify a comment card. In this case the * is interpreted as an operator.

APPENDIX C

This appendix describes the APL programs and subroutines which comprise the compiler of the telemetry preprocessing language. In order to read and understand this appendix, the reader must have a working knowledge of the APL programming language (references 10 and 11).

This compiler is divided into two major segments, the frame segment and the format segment, as was described in Chapter 2. Compilation begins with the execution of the routine COMPILE. This routine initializes all of the tables required during compilation and reads the first statement of the language. The routine can recognize four statements. If the statement read is an EQUATE statement, the necessary linkages between the new and old frame identifiers are formed. If the statement read is a FINIS statement, the routine is exited with all of the tables properly loaded. If the statement read is a FRAME statement, the frame parser is entered. If the statement read is a FORMAT statement, the FORMAT parser is entered. Any other statement read by this routine generates an error message.

The FORMAT subroutine checks the name of the format identifier with the format table to insure that there are no multiple entries. If there are, an error message is generated and the subroutine is exited. Otherwise, the statements of the format section are read and the format table is filled.

The FRAMES subroutine checks the name of the frame identifier with the frame table to insure that there are no multiple entries. If there are, an error message is generated and the routine is exited. Otherwise, statements of the frame section are read and the operations table is formed.

The routine INPUT reads statements of the preprocessing language, ignores comment cards, and deals with continuation cards.

The routine CDAD computes the address of all symbols for both the program and data memory and assigns indices to those symbols which refer to index counters. These assigned values are placed in the operations table along with a code designating the type of variable represented.

The routine DECODE transforms the frame sync pattern into a bit string with the pattern right justified. It also determines how many bits are in the pattern and loads this value into the format table.

The routine ERROR generates all of the error messages that the compiler produces.

The routine ENTR places variable names into the symbol table, checks for multiple definitions, and generates appropriate error messages when required.

The routine LFORM parses simple arithmetic expressions.

The routine LIDEL recognizes and stores numbers, detects simple arithmetic expressions, and sends them to the routine LFORM.

The routine MEMP provides a compiled listing of the program. It lists all symbols used together with their addresses and provides a formatted output of the operations table.

The routine OPRA parses expression of the form M(A, B, C). Such expressions are found in the specifier protion of the data handling instructions.

The routine OUTERR lists all the compilation errors that the routine ERROR stores in the error table.

The routine PARSE translates the arithmetic and logical expressions of the preprocessing language into Polish string format and places these operators into the operations table.

The routine RESCD manages the internal temporary storage locations, assigns them as required, and reclaims them for reassignment.

The routine TRANS translates the three number systems of the telemetry preprocessing system (decimal, octal, and binary) into the decimal number system and places them in their assigned storage locations.

The routines just described comprise the compiler for the telemetry preprocessing language. The APL program listings for these routines follow.

```
      ∇COMPILE[□]∇
    ∇ COMPILE;LOC;FP;C;IMAD;DMAD;PRAD;TBL;OT;FRPT;FOPT;OP
[1]   IMAD←DMAD←1↑0,(PRAD←1),⍴FID←TBL← 1 9 ⍴' '
[2]   FP←AMP←AM←PT←FORMID←OT←FRPT←FOPT←⍳0
[3]  L0:OP←10↑9↓C←INPUT
[4]   →⍳0×⍴(C←19↓C),LOC←((9-⍴LOC)⍴' '),LOC←(LOC≠' ')/LOC←9↑C
[5]   A←(∧/'FRAME'=5↑OP),(∧/'FINIS'=5↑OP),∧/'EQUATE'=6↑OP
[6]   →(OP≠0)/OP←(L1,L2,L3,L4)×(∧/'FORMAT'=6↑OP),A
[7]   →L0,⍴□←'NOT A LEGAL STATEMENT ... RETRY'
[8]  L1:OT←FORMAT,OT
[9]   →L0,(IMAD←IMAD+17),FOPT←IMAD,FOPT
[10] L2:OT←(,C←FRAMES),OT
[11]  →L0,(IMAD←⍴OT),(FRPT←IMAD,FRPT),(PRAD←1+×/⍴C),FP←PRAD,FP
[12] L4:OP←',',∈C←(C≠' ')/C
[13]  →(L4E×⍳0='='∈C),(1+⍳26),⍴C←(¯1+C⍳',')↑C
[14]  LOC←(C⍳'=')↑C
[15]  →L4E×⍳0=LOC←+/(⍳1↑⍴FID)×,FID∧.= 9 1 ⍴((9-⍴LOC)⍴' '),LOC
[16]  →L4E×⍳0≠+/(⍳1↑⍴FID)×,FID∧.= 9 1 ⍴C←((9-⍴C)⍴' '),C←(¯1+C⍳'=')↑C
[17]  FID←(1 0 +⍴FID)⍴C,,FID
[18]  OT←(¯1,(1↑⍴FID),OP,14⍴0),OT
[19]  →L0,(IMAD←⍴OT),(FRPT←IMAD,FRPT),FP←FP[LOC],FP
[20] L4E:→L0,⍴□←'EQUATE NOT VALID'
[21] L3:→(L3+1),(⍴FID←FID[C;]),(FRPT←FRPT[C],IMAD),FP←FP[C←⍋FP]
[22]  PT←PT,OT[FOPT[+/(⍳1↑⍴TBL)×,TBL∧.= 9 1 ⍴,FID[1;]]+⍳17]
[23]  PT←PT,OT[FRPT[1]+⍳FRPT[2]-FRPT[1]]
[24]  →⍳0×⍴(FP←1↓FP),(FRPT←1↓FRPT),(AMP←AMP,FP[1]),⍴AM←AM,,FID[1;]
[25]  →(M0×⍳0=1↑⍴FID),(L3+1),⍴FID←FID[1+⍳¯1+1↑⍴FID;]
[26] M0:(((⍴PT)÷17),17)⍴PT
[27]  (((⍴AM)÷9),9)⍴AM
[28]  ((⍴AMP),1)⍴AMP
    ∇

      ∇FORMAT[□]∇
    ∇ FORMID←FORMAT;EV;TEMP;A;V;D;FTBL;FSP;OP;B
[1]   →L1×⍳0=+/(⍳1↑⍴TBL)×,TBL∧.= 9 1 ⍴LOC
[2]   →F9,⍴ERROR 2
[3]  L1:TBL←(1 0 +⍴TBL)⍴LOC,,TBL
[4]   FTBL←((1↑⍴TBL)-+/(⍳1↑⍴TBL)×,TBL∧.= 9 1 ⍴LOC),(5⍴0),7,0,0
[5]   →(1+⍳26),(EV←TEMP←6⍴B←0),FORMID←V←A←⍳0
[6]  F0:→(1+⍳26),(⍴C←(' '≠C)/C←40↑19↓C),⍴OP←10↑9↓C←INPUT
[7]   A←1++/(⍳1↑⍴INST)×,INST∧.=((1↑⍴INST),1)⍴(((1↑⍴INST)-⍴OP)⍴' '),OP
[8]   →(F1,F2,F3,F4,F5,F6,F7,F8)[A]
[9]  F1:→F0,⍴ERROR 1
[10] F2:→(F2A×⍳EV[1]=1),F2E×⍳TEMP[3]=1
[11] F2A:→F0,(⍴DECODE),(FSP←⍳0),TEMP[3]←1
[12] F2E:→F2A,(⍴ERROR 3),EV[1]←1
[13] F3:→(F3A×⍳EV[2]=1),F3E×⍳TEMP[4]=1
```

```
[14]  A←1+D←+/(⍳7)×,CODES∧.= 4 1 ⍴C←((4-⍴C)⍴' '),C
[15] F3A:→(F3B×⍳D=0),F0,(TEMP[4]←1),⍴FTBL[7]←(7,0,1,2,3,4,5,5)[A]
[16] F3B:→F0,⍴ERROR 2
[17] F3E:→F3A,(EV[2]←1),⍴ERROR 4
[18] F4:→(F4A×⍳EV[3]=1),F4E×⍳TEMP[5]=1
[19] F4A:→F0,(TEMP[5]←1),⍴FTBL[8]←TRANS
[20] F4E:→F4A,(EV[3]←1),⍴ERROR 5
[21] F5:→(F5A×⍳EV[4]=1),F5E×⍳TEMP[1]=1
[22] F5A:→F0,(⍴FTBL[3]←TRANS),TEMP[1]←1
[23] F5E:→F5A,(EV[4]←1),⍴ERROR 6
[24] F6:→(F6A×⍳EV[5]=1),F6E×⍳TEMP[2]=1
[25] F6A:→F0,(TEMP[2]←1),⍴FTBL[4]←TRANS
[26] F6E:→F6A,(EV[5]←1),⍴ERROR 7
[27] F7:→(F7A×⍳EV[6]=1),F7E×⍳TEMP[6]=1
[28]  A←1+D←+/(⍳7)×,TSPS∧.= 4 1 ⍴C←((4-⍴C)⍴' '),C
[29] F7A:→(F7B×⍳D=0),F0,(TEMP[6]←1),⍴FTBL[9]←(0,1,2,3,4,5,6,7)[A]
[30] F7B:→F0,⍴ERROR 2
[31] F7E:→F7A,(EV[6]←1),⍴ERROR 8
[32] F8:→F8A×⍳3=+/3↑TEMP
[33]  →(F9×⍳0≠⍴V),0,(⍴ERROR 9),⍴TBL←TBL[⍳(1↑⍴TBL)-1;]
[34] F8A:→(F9×⍳0≠⍴V),0,⍴FORMID←FTBL,(8⍴0),FORMID
[35] F9:→0,⍴OUTERR
     ∇

       ∇FRAMES[□]∇
     ∇ ITBL←FRAMES;A;V;D;D1;D2;P1;PTS;OT;P3;P2;C;PSY;DSY;ISY
[1]    →(L2×⍳0=+/(⍳1↑⍴FID)×,FID∧.= 9 1 ⍴LOC),(A←V←⍳0),⍳D←D1←0
[2]    →OUT,⍴ERROR 10
[3]   L2:FID←((1 0)+⍴FID)⍴LOC,,FID
[4]    SYTP← 1 1 ⍴' '
[5]    SYTB← 1 9 ⍴' '
[6]    →L0×⍳1↑(ITBL←⍳0),⍳0×1↑⍴SYPO← 1 1 ⍴0
[7]   L0:→OUT×⍳1=∨/((500+⍴ITBL)>⌶22),∧/'END'=3↑9↑C←INPUT
[8]    →(L7×⍳1=∧/' '=LOC),(⍳0×⍴LOC←9↑C),⍳0×⍴LOCA←40↑C
[9]    →L00×⍳0=∨/('CONTINUE'=8↑9↑C),('DIMENSION'=9↑9↑C),⍳0×⍴,P3←'1'
[10]   P3←'0'
[11] L00:→L7A,LOC←LFORM P3,LOC←(LOC≠' ')/LOC
[12] OUT:→(OUT+2)×⍳0=⍴V
[13]   →0,⍴OUTERR
[14]   ITBL←CDAD ITBL←(((⍴ITBL)↑17),17)⍴ITBL
[15]   →0,⍴MEMP
[16] L7:LOC←4⍴0
[17] L7A:C←9↓C
[18]   →GOTOINS×⍳∧/'GO TO'=5↑C
[19]   →REPINS×⍳∧/'REPEAT'=6↑C
[20]   →CONT×⍳∧/'CONTINUE'=8↑C
[21]   →DEM×⍳∧/'DIMENSION'=9↑C
[22]   C←((C⍳' ')-1)↑C
```

```
[23]   →HLT×⍳∧/'HALT'=4↑C
[24]   →IFINS×⍳∧/'IF('=3↑C
[25]   →L7A1×⍳'='∈D1←10↑C
[26] L000:→SUBINS×⍳∧/'SUB'=3↑C
[27]   →SYINS×⍳∧/'SYNC'=4↑C
[28]   →WDINS×⍳∧/'WORD'=4↑C
[29]   →L7A2×⍳∨/('('∈D1),(','∈D1),('+'∈D1),'-'∈D1←10↑C
[30] L7A3:→L0×2=⍴ERROR 15
[31] L7A2:→L7A3×⍳0=⍴C
[32]   P1←1↑LIDEL '2',D1←((D1[1↑⍸D1←C⍳'(,+-'])-1)↑C
[33]   ITBL←ITBL,LOC,5,0,P1,D2←OPRA C←(⍴D1)↓C
[34] I00:→(L0×⍳0=⍴C),L7A3×⍳0=∨/(1↑C='-'),1↑C='+'
[35]   →L000,(⍴C←1↓C),ITBL[(⍴ITBL)-11]←(1×C[1]='+')+2×C[1]='-'
[36] L7A1:PTS←1↑LIDEL '0',(⍳¯1=⍴C←(C⍳'=')↓C),((C⍳'=')-1)↑C
[37]   OT←PARSE C
[38]   →L0,RESCD LOC,4
[39] IFINS:D1←(⌈/(⍳⍴C)×C=')')↓C←3↓C
[40]   →IF1×⍳(9<⍴D1)∨0=⍴C←(0-1+⍴D1)↓C
[41]   PTS←1↑LIDEL '0',D1
[42]   →IF1×⍳(~','∈C)∨(+/C='(')≠+/C=')'
[43]   OT←PARSE C
[44]   →L0,RESCD LOC,3
[45] IF1:→L0,⍴ERROR 15
[46] GOTOINS:A←(1↓(D2←LFORM '0',(D2≠' ')/D2←((10↑C)⍳' ')↑10↓C)),7⍴0
[47]   →L0,ITBL←ITBL,LOC,1,0,(20000+D2[1]),A
[48] HLT:→L0,ITBL←I/BL,17⍴0
[49] CONT:→L0,ITBL←ITBL,LOC,9,12⍴0
[50] DEM:→DEM1×⍳4<⍴P1←(P1≠' ')/P1←(C⍳' ')↑C←10↓C
[51]   →L0,ITBL←ITBL,LOC,10,0,(P1←LIDEL '0',P1),10⍴0
[52] DEM1:→L0,⍴ERROR 11
[53] REPINS:D1←'='=C←¯1↓(C⍳' ')↑C←10↓C
[54]   →L7A3×⍳1=∨/(2≠⍴(D1≠0)/D1←',',=C),1≠⍴(D1≠0)/D1
[55]   D1←PARSE(⍳¯1=⍴C←(C⍳',')↓C),((C⍳',')-1)↑C
[56]   D2←1↑LIDEL '1',(⍳¯1=⍴C←(C⍳'=')↓C),((C⍳'=')-1)↑C
[57]   P1←PARSE(⍳¯1=⍴C←(C⍳',')↓C),((C⍳',')-1)↑C
[58]   A←(1≠⍴P1)×P1,0,0
[59]   P1←3↑P1[1],(|P1[3]),P1[2],P1←(3↑(1=⍴P1)×10000,P1,1)+3↑A
[60]   A←(1=⍴OT)×10000(OT←PARSE C),1
[61]   OT←3↑OT[1],(|OT[3]),OT[2],OT←(3↑(1≠⍴OT)×OT,0,0)+3↑A
[62]   →L0,ITBL←ITBL,LOC,2,0,D2,P1,D1,OT,3⍴0
[63] WDINS:→ICMN,(P3←8),(D2←4),P2←0
[64] SYINS:→ICMN,(P3←6),(D2←4),P2←0
[65] SUBINS:→ICMN,(P3←7),(D2←3),P2←0
[66] ICMN:→I00,ITBL←ITBL,LOC,P3,0,P2,D1←OPRA C←D2↑C
     ∇
```

```
      ∇INPUT[□]∇
    ∇ V←INPUT;WW
[1]   80ρ'123456789*'
[2]   V←⍬
[3]   →1×⍳V[1]='*'
[4]   V←V,(80-ρV)ρ' '
[5]   →0×⍳V[80]≠'*'
[6]   V←¯1↓V
[7]   80ρ'123456789*'
[8]   WW←⍬
[9]   →7×⍳WW[1]='*'
[10]  V←V,WW
[11]  →6×⍳'*'=¯1↓WW
[12]  V←V,' '
    ∇

      ∇CDAD[□]∇
    ∇ B←CDAD A;I;K;D
[1]   →L1,ρ(ρPAD← 1 1 ρ0),(ρDAD← 1 1 ρDMAD),(I←1),ρPSY←DSY← 1 9 ρ' '
[2]  L1:→L0×⍳0=ρK←(K≠0)/K←(⍳1↑ρA)×(10=,A[;5])
[3]   D←(D≠0)/D←A[K[1];]
[4]   A←(((1↑ρA)-1),(1↓ρA))ρ(,A[⍳K[1]-1;]),,A[K[1]+⍳(1↑ρA)-K[1];]
[5]   DSY←(1 0 +ρDSY)ρ(,SYTB[(1↑ρSYTB)-D[1];]),,DSY
[6]   DAD←(1 0 +ρDAD)ρ(DAD[1;]+D[3]-10000),,DAD
[7]   →L1,SYPO[(1↑ρSYPO)-D[1];]←¯1
[8]  L0:→L2×⍳0=ρK←(K≠0)/K←(⍳1↑ρA)×(9=,A[;5])
[9]   PSY←(1 0 +ρPSY)ρ(,SYTB[(1↑ρSYTB)-(1↑,A[1↑K;])];]),,PSY
[10]  PAD←(1 0 +ρPAD)ρ(PRAD+1↑K),,PAD
[11]  SYPO[(1↑ρSYPO)-1↑,A[1↑K;];1]←¯2
[12]  A←(1↑ρA)-1↑K
[13]  →L0,,A←(((1↑ρA)-1),(1↑ρA))ρ(,A[⍳(1↑K)-1;]),,A[(1↑K)+⍳A;]
[14] L2:ISY←SYTB[(D≠0)/D←(⍳1↑ρSYPO)×,0<SYPO;]
[15] L3:D←A[I;],K←2
[16]  →L4×⍳0=D[1]
[17]  A←(1↑ρSYTB)-D[1]
[18]  A[I;1]←50000+DAD[1+(+/(⍳1↑ρDSY)×,DSY∧.= 9 1 ρ,SYTB[A;]);]
[19] L4:→(L5×⍳20=⌊D[K]÷1000),1+⍳26
[20]  →(L4×⍳18≠K←K+1)
[21]  →(L3×⍳(1+1↑ρA)≠I←I+1),0,ρB←A
[22] L5:→(L6×⍳¯2=PTS),L7×⍳¯1=PTS←SYPO[(1↑ρSYPO)-D[K]-20000;]
[23]  A←(1↑ρSYTB)-D[K]-20000
[24]  →(L4+1),A[I;K]←20000++/(⍳1↑ρISY)×,ISY∧.= 9 1 ρ,SYTB[A;]
[25] L6:A←(1↑ρSYTB)-D[K]-20000
[26]  →(L4+1),A[I;K]←40000+PAD[+/(⍳1↑ρPSY)×,PSY∧.= 9 1 ρ,SYTB[A;];]
[27] L7:A←(1↑ρSYTB)-D[K]-20000
[28]  →(L4+1),A[I;K]←50000+DAD[1++/(⍳1↑ρDSY)×,DSY∧.= 9 1 ρ,SYTB[A;];]
    ∇
```

```
      ∇DECODE[⎕]∇
    ∇ Y←DECODE;B;D
[1]     Y←⍳3
[2]   L0:→(BIN×⍳∨/C[1]='(B'),(OCT×⍳C[1]='O'),OCT1×⍳C[1]∈'01234567'
[3]   ERR:→0,(⍴ERROR 2),FTBL[2 5 6]← 0 0 0
[4]   OCT:→ERR×⍳0=⍴C←1↓C
[5]   OCT1:→ERR×⍳~C[1]∈'01234567'
[6]     →(L0×⍳0≠⍴C←1↓C),L1,⍴FSP←FSP, 0 1[1+ 2 2 2 ⍳('01234567'⍳C[1])-1]
[7]   BIN:→ERR×⍳~∧/(B←(¯1↓D←D[1+⍋D←(C←1↓C)⍳')O'])↓C)∈'01'
[8]     →(L0×⍳0≠⍴C←D↓C),L1,⍴FSP←FSP,B='1'
[9]   L1:FSP←((32-FTBL[2]←⍴FSP)⍴0),FSP
[10]    FTBL[6]←+/B×10*(⍴B)-⍳⍴B←(6⍴8)⊤2⊥16↓FSP
[11]    B←(6⍴8)⊤2⊥16↓FSP
[12]    →(ERR×⍳32<FTBL[2]),0,FTBL[5]←+/B×10*(⍴B)-⍳⍴B
      ∇

      ∇ERROR[⎕]∇
    ∇ Z←ERROR X;Y
[1]     →(E1,E2,E3,E4,E5,E6,E7,E8,E9,E10,E11,E12,E13,E14,E15,E16,E17)[X]
[2]   E1:→E0,⍴Y←OP,' IS NOT A LEGAL OP CODE'
[3]   E2:→E0,⍴Y←'ILLEGAL ARGUEMENT FOR ',OP
[4]   E3:→E0,⍴Y←'WARNING MORE THAN 1 FSP INST'
[5]   E4:→E0,⍴Y←'WARNING MORE THAN 1 CODE INST'
[6]   E5:→E0,⍴Y←'WARNING MORE THAN 1 BIT RATE INST'
[7]   E6:→E0,Y←'WARNING MORE THAN 1 BITS/WORD INST'
[8]   E7:→E0,Y←'WARNING MORE THAN 1 WORDS/FRAM INST'
[9]   E8:→E0,Y←'WARNING MORE THAN 1 TP SPEED INST'
[10]  E9:→E0,Y←'FORMAT ',TBL[1;],' IS NOT PROPERLY DEFINED'
[11]  E10:→E0,⍴Y←LOC,' IS MULTIPLY DEFINED'
[12]  E11:→E0,Y←'INDEX ERROR IN LOCATION ',LOCA,' ILLEGAL NUMBER'
[13]  E12:→E0,Y←'INDEX ERROR IN LOCATION ',LOCA,' NUMBER > 999'
[14]  E13:A←'NOT YET DEFINED'
[15]    →E0,⍴Y←'INDEX ERROR IN LOCATION ',LOCA,' INDEX ',V3,A
[16]  E14:→E0,Y←'DEFINITION ERROR IN ',LOCA,' PARENS DONT MATCH'
[17]  E15:→E0,⍴Y←LOCA,' EXPRESSION IS ILLEGAL'
[18]  E16:→E0,Y←⍳0
[19]  E17:→E0,Y←'IN ',LOCA,'EXTRA PARENS SURROUND ',P3
[20]  E0:→0,(V←V,⍴Y),(⍴A←A,Y),⍴Z←'10'
      ∇

      ∇ENTR[⎕]∇
    ∇ D←ENTR X;V3
[1]     →L0×⍳0≠D←+/(⍳1↓⍴SYTB)×⍷SYTB∧.= 9 1 ⍴1↓X
[2]     SYTP←((1 0)+⍴SYTP)⍴(1↓X),,SYTP
[3]     SYTB←((1 0)+⍴SYTB)⍴(1↓X),,SYTB
[4]     D←(1↓⍴SYTB)-1
[5]     SYPO←((1 0)+⍴SYPO)⍴D,,SYPO
[6]     →(ER1×⍳'2'=1↓X),(DAJ×⍳'1'≠1↓X),0
```

```
[7]   DAJ:→0,ρSYTP[1;]←'2'
[8]   L0:→ERR×ι(SYTP[D;]=1↑X)∧'1'=1↑X
[9]     →0×ρ,D←(1↑ρSYTB)-D
[10]  ERR:→0×ρ(D←0),ρERROR 10
[11]  ER1:→0×ρ(D←0),(ρERROR 13),ρV3←1↓X
      ∇

        ∇LFORM[□]∇
      ∇ Q←LFORM X;Y
[1]     →A11×ι1='('∈X
[2]     →0×ρQ←(ENTR(1↑X),((9-ρ1↓X)ρ' '),1↓X←(X≠' ')/X), 0 0 0
[3]    A11:Q←ENTR('0'),((9-ρ1↓Y)ρ' '),1↓Y←(Y≠' ')/Y←((Xι'(')-1)↑X
[4]     Y←PARSE((Xι'(')-1)↑X
[5]     A←(1≠ρY)×Y,0,0
[6]     →0×ρQ←Q,3↑Y[1],(|Y[3]),Y[2],Y←(3↑(1=ρY)×10000,Y,1)+3↑A
      ∇

        ∇LIDEL[□]∇
      ∇ Q←LIDEL X
[1]     →A11×ι0=∨/11='0123456789'ιX
[2]     →0×ρ,Q←20000+1↑LFORM X
[3]    A11:→ER×ι4<ρX←1↓X
[4]     →0×ρ,Q←10000+10⊥('0123456789'ιX)-1
[5]    ER:→0,(Q←10000),ρERROR 12
      ∇

        ∇MEMP[□]∇
      ∇ Y←MEMP;SS
[1]     Y←ι2
[2]     'PROGRAM
        '
[3]     SS←'    V1       OP     V2       V1       OP     V2'
[4]     ' ADD   LOC   V1   OP   V2   I   L   A   V1   OP   V2   R   ',SS
[5]     ('12 345 ')$(((1↑ρITBL),1)ρι1↑ρITBL),ITBL
[6]     →L0×ι1=1↑ρDSY
[7]     'DATA SYMBOLS'
[8]     DSY
[9]     'DATA ADDRESSES'
[10]    (((1↑ρDSY)-1),1)ρDAD[1↓ι1↑ρDSY;1]
[11]  L0:→L1×ι1=1↑ρPSY
[12]    'PROGRAM SYMBOLS'
[13]    PSY
[14]    'PROGRAM ADDRESSES'
[15]    (((1↑ρPSY)-1),1)ρPAD[¯1↓ι1↑ρPSY;1]
[16]  L1:→0×ι1=1+1↑ρISY
[17]    'INDEX SYMBOLS'
[18]    ISY
[19]    'INDEX NUMBERS'
[20]    ((1↑ρISY),1)ρι1↑ρISY
      ∇
```

E

```
      ∇OPRA[⎕]∇
    ∇ Q←OPRA X;Y;K;L;S
[1]   Q←10⍴0,(K←0),0×⍴S←⍴X
[2]  L00:A←L1×⍳1↑X='.'
[3]   →(L10×⍳∨/(0=⍴X),(1↑X='+'),(1↑X='-')),(L0×⍳1↑X='(')⍴A
[4]   →0×⍴ERROR 15
[5]   →((L00+1)×⍳7≠P3),L00,P2←LIDEL '0',Y
[6]  L10:→0,⍴C←(S-⍴X)↓C
[7]  L1:Y←((Y[1↑⍋Y←X⍳'(,+-'])-1)↑X←1↓X
[8]   A←(3 1 ⍴3↑((3-⍴Y)⍴' '),Y,X←(⍴Y)↓X
[9]   →(L00+2)×⍳0=L←+/(⍳7)×,(7 3 ⍴'PETPOTPEAPOA  L  F  B')∧.=A
[10]  →L00,(K←K+1),Q[4]←Q[4]+L×10*4-K
[11] L0:→((L00+1)×⍳0=')'∈X),L2×⍳1='.'∈(X⍳')')↑X←1↓X
[12]  →L0A×⍳1=∧/'ALL'=3↑X
[13]  Q[3]←PARSE((X⍳')')-1)↑X
[14]  →L00,(X←(X⍳')')↓X),(Q[1]←10000),(Q[2]←1)
[15] L0A:→L00,(⍴X←4↓X),Q[2]←14
[16] L2:Y←(⍳~1=⍴X←(X⍳')')↓X),((X⍳')')-1)↑X
[17]  →(L00+1)×⍳2≠⍴(K=0)/K←Y='.'
[18]  K←(⍳~1=⍴Y←(Y⍳',')↓Y),((Y⍳',')-1)↑Y
[19]  L←PARSE K
[20]  A←(1=⍴L)×3↑L,0,0
[21]  Q[1 2 3]←3↑L[1],(|L[3]),L[2],L←((1=⍴L)×(3↑10000,L,1))↓A
[22]  K←(⍳~1=⍴Y←(Y⍳',')↓Y),((Y⍳',')-1)↑Y
[23]  L←PARSE K
[24]  K←PARSE Y
[25]  A←(1=⍴L)×3↑L,0,0
[26]  Q[5 6 7]←3↑L[1],(|L[3]),L[2],L←((1=⍴L)×3↑10000,L,1)↓A
[27]  A←(1=⍴K)×3↑K,0,0
[28]  →L00,Q[8 9 10]←3↑K[1],(|K[3]),K[2],K←((1=⍴K)×3↑10000,K,1)↓A
    ∇

      ∇OUTERR[⎕]∇
    ∇ Y←OUTERR;I;J
[1]   I←⍴V
[2]  L1:J←V[1]
[3]   V←1↓V
[4]   J↑A
[5]   A←J↓A
[6]   →L1×⍳0=I←I-1
[7]   A
[8]   Y←'10'
    ∇

      ∇PARSE[⎕]∇
    ∇ Y←PARSE IS;STK;STKV;LIC;HTSV;W;N;X
[1]   →0×⍳0=⍴(Y←⍳0),(,⍴IS←IS,')'),(STK←LIC←6),STKV←HTSV←0
[2]   A←(N×'↑'=IS)+(N×'('=IS)+(N←⍳⍴IS)×')'=IS
[3]   N←(N≠0)/N←(N×','=IS)+(N×'+'=IS)+(N×'-'=IS)+(N×'*'=IS)+A
```

```
[4]   L00:→0×⍳¯1=⍴(,⍴TS←(⍴W)↑TS),(N←1↑N-⍴W),⍴W←(1↑N)↑TS
[5]    →L0×⍳1=⍴W
[6]    →ER0×⍳1=(LTC=5)∨LTC=7
[7]    X←LIDEL '2',¯1↑W
[8]    →L5×⍳3=⍴(HTSV←7),(LTC←5),⍴W←¯1↑W
[9]   ER0:→0×⍴ERROR 15
[10]  L0:→(L1×⍳W='.'),(L2×⍳W∈'+-*÷'),L3×⍳W='('
[11]   →ER1×⍳1=∨/LTC= 1 2 3 4
[12]   →L5×⍳2=⍴(HTSV←0),(LTC←7),W←⍳0
[13]  ER1:→0×⍴ERROR 15
[14]  L2:→L2A×⍳1=∨/LTC= 1 6
[15]   →ER2×⍳1=∨/LTC= 2 3 4
[16]   X←-+/(⍳4)×W='+-*÷'
[17]   →L5×⍳2=⍴(HTSV←5),(LTC←4),W←⍳0
[18]  L2A:→ER2×⍳~W∈'+-'
[19]   →L5×⍳3=⍴(X←10000),(HTSV←7),LTC←5
[20]  ER2:→0×⍴ERROR 15
[21]  L3:→ER3×⍳1=∨/LTC= 5 7
[22]   →L5×⍳3=⍴(X←6),(HTSV←9),(LTC←6),W←⍳0
[23]  ER3:→0×⍴ERROR 15
[24]  L1:→0×⍳¯1=⍴(⍴TS←(⍴W)↑TS),(N←1↑N-⍴W),⍴W←(1↑N)↑TS
[25]   X←¯1×4++/(⍳9)×,(9 2 ⍴'EONELEGELTGTOPAHHO')∧.=(2 1)⍴2↑W
[26]   W←(1×11>|X)+(2×∨/ 11 12 =|X)+3×13=|X
[27]   →ER4×⍳1=((∨/W= 1 2)∧∨/LTC= 1 2 3 4 6)∨(W=3)∧∨/LTC= 1 4 5
[28]   →L5×⍳2=⍴(W←⍳0),(HTSV←+/(1×W= 1 2),3×W=3),LTC←W
[29]  ER4:→0×⍴ERROR 14
[30]  L5:→(L5A×⍳HTSV>1↑STKV),L5B×⍳HTSV<1↑STKV
[31]   →(L0×⍳1=⍴W),(L00×⍳0≠⍴TS),0×⍳0=⍴(STV←1↓STV),STKV←1↓STKV
[32]   →0×⍴ERROR 14
[33]  L5A:A←(STKV←((0×HTSV=10)+HTSV<10)×HTSV←HTSV+1),STKV)),LA5-1
[34]   →(L0×⍳1=⍴W),(L00×⍳0≠⍴TS),(0×⍳0=⍴(STV←X,STK),A
[35]  L5B:→L5,(STKV←1↓STKV),(STK←1↓STK),Y←Y,1↑STK
      ∇

        ∇RESCD[⎕]∇
      ∇ DD←RESCD OPCD;P1;L;V1;OP;V2;R;I
[1]     P1←(0≠P1)/P1←(⍳⍴OT)×OT<0
[2]     →(L1×⍳0=⍴P1),L0,DD←1↑I←⍳100
[3]    LC:→L2×⍳13=|OT[P1[1]]
[4]     A←(I←1↓I),R←30000+1↑I
[5]     →0×⍳0=⍴(V1←OT[P1[1]-2]),(V2←OT[P1[1]-1]),(OP←|OT[P1[1]]),A
[6]     →0×⍳¯1=⍴(P1←1↓P1-2),OT←((P1[1]-3)↑OT),R,P1[1]↓OT
```

```
[7]   L3:ITBL←ITBL,OPCD,1,PTS,V1,OP,V2,R,6ρ0
[8]    →(2+126)×ι0≠V/DD←30=|(V1÷1000),V2÷1000
[9]    T←((V1≠0)/V1←DD×(V1←V1-30000),V2←V2-30000),T
[10]   →L0×ι0≠ρP1
[11]   →0×ρ,ITBL[(ρITBL)-11]←0
[12]  L2:→0×ι0=ρ(V1←0),(V2←OT[P1[1]-1]),(OP←13),(T←1↓T),R←30000+1↓T
[13]   →L3,(P1←1↓P1-1),OM←((P1[1]-2)↑OM),R,P1[1]↓OM
[14]  L1:ITBL←ITBL,OPCD,0,PTS,0,0,0,OM,6ρ0
      ∇

      ∇TRANS[[]]∇
      ∇ X←TRANS;N
[1]     A←DEC×ι((C[1]='D')∨C[1]∈N)∧~∨\11=(N←'0123456789')ι1↓C
[2]     A←A,OCT×ι(C[1]='O')∧~∨/9=(8↑N)ι1↓C
[3]     →(BIN×ι(C[1]='B')∧~∨/3='01'ι1↓C),A
[4]     →0,(X←0),ρERROR 2
[5]    DEC:→0,X←10⊥(N⍳C←(C[1]='D')↓C)-1
[6]    OCT:→0,X←8⊥((8↑N)⍳C←1↓C)-1
[7]    BIN:→0,X←2⊥('01'⍳C←1↓C)-1
      ∇
```

APPENDIX D

This appendix describes the APL routines written to simulate the telemetry preprocessing language and presents listings of those routines.

The routine BUS directs and controls all data flow operations and control sequences within the simulation system. Control sequences can take up to eight ID commands in succession. This allows a total of 128 modules to be connected to the bus at one time. The BUS routine computes the record numbers of the module storage file on the disc system.

Prior to executing any functional sequences, the locations CONST, CNSQ, JQP, and CUMDND must be initialized to zero. CONST is the bus system connect indicator. CNSQ is the connect sequence control flag. JQP is a job queue pointer. CUMDND is the current working module number.

The entry to the connect sequence is controlled by the first branch instruction of the routine. Label L1 is the code to execute the initial connect sequence. It is entered if 1) a send ID command is issued by a module in control and 2) either the bus system is not connected and the connect sequence is just starting (CNSQ = 0) or a new connect sequence is being initiated by a connected module (CNSQ = 1).

Label L1A is the follow-on connect sequence and is entered if 1) a send ID command is issued by a module in control and 2) the module is already connected. This can arise because of successive send ID commands in a connect sequence.

The sequence labeled L2 is entered if 1) a transfer of data is requested, 2) the module is in control, and 3) the module is connected. This sequence is exited if no modules are connected. This would be the case when the send ID command is being used to clear the connects to the bus system.

Otherwise, a control vector of connected module numbers is set up, the modules are read in one at a time, and the data is transferred. Then, the updated modules are written onto the disc, the original module is restored, and the sequence is exited.

The sequence to drop a connect is labeled L3 and is entered if 1) it is requested and 2) the module is connected. This sequence simply drops the connect and exits.

The sequence to transfer a selector is labeled L5 and is entered if 1) it is requested, 2) the module is connected, and 3) the requesting module has control. The sequence checks to insure that only one module is connected. If there are more than one, it exits. The sequence then saves the current module parameters, reads the new module parameters, executes the instruction, retrieves the selector, restores the current module, updates the selector, and exits.

The control transferal sequence is labeled L4 and is entered if 1) it is requested and 2) the requesting module is connected. This sequence will create a job queue if a multiple control transfer is encountered. The form of this queue is a matrix whose number of rows equals the number of modules connected. The columns are composed of the current state of the system, i.e., the destination module of the transfer, the task to be performed, the request originator, the bus connect status, the connect sequence status, and the request type. This matrix then becomes a task table of jobs to be done. As long as there are entries in this table, every request using the bus system is stacked and the job entry of this table is the task performed. In this manner, the sequence of events is kept current.

Sequence L4 will drop the connect to the bus system and exit if no modules are connected. If a single module is connected, the current module status will be stored if it is in the stop state. Otherwise, execution will continue until the

stop state occurs and then it will be stored. The new module will then be read in and control will be transferred to it.

The routine CON emulates the control portion of a module. It initiates functional cycles, keeps track of the address of the current microinstruction, directs the reading of microinstructions, and controls the external communications to the module.

The routine FMSH emulates the search portion of a functional cycle.

The routine FMRD emulates the read portion of a functional cycle.

The routine INPUT is used to load the functional memory modules. It accepts octal data as input, converts it to binary, and stores it in the appropriate word of the memory array.

The routine INST decodes and emulates the operations of the forty-three microinstructions of the functional memory module.

The routine LIST is used to provide a formatted output of the contents from the functional memory module's memory array and from various registers.

The routine MCON is the initial entry point to the simulator. It initializes the various parameters of the simulator and receiver as input, the initial point of control, and the type of cycle to be run at the start of the simulation. It also prints out the results at the end of the simulation.

The routine PICK takes the top entry from the job queue matrix task table and initializes that task.

The routine PLACE adds tasks to the job queue.

The routine READ retrieves an image of a module from the disc system.

The routine WRITE places an image of a module on the disc system.

The routines that have just been described comprise the APL simulator of the telemetry preprocessing system. The APL program listings for these routines follow.

```
      ∇BUS[⎕]∇
    ∇ BUS EXT;X0;X1
[1]   B←BCOM
[2]   →(L5×⍳(EXT=5)∧(CONT=1)∧CONCT[B[1]]=1),JQB×⍳EXT=6
[3]   →(L3×⍳(EXT=3)∧CONCT[B[1]]=1),JQA×⍳(1=JQP)∧EXT=4
[4]  L0:→L1×⍳(EXT=2)∧(CONT=1)∧((CONCT[BCOM[1]]=0)∧CNSQ=0)∨CNSQ=1
[5]   →L1A×⍳(EXT=2)∧(CONT=1)∧(CNSQ=0)∧CONCT[BCOM[1]]=1
[6]   →L2×⍳(EXT=1)∧(CONT=1)∧CONCT[BCOM[1]]=1
[7]   →L4×⍳(EXT=4)∧CONCT[BCOM[1]]=1
[8]   →0,STOP←~∨/(21P[21+⍳6])= 35 39
[9]  L1:MDRN←(MDRN=0)/MDRN←(⍳16)×16+1↓BCOM
[10]  MDRN←(16×21 3+17↓BCOM)+MDRN
[11]  →0,(CNSQ←BCOM[21]),CONCT[B[1]]←1
[12] L1A:MDRN←MDRN,(16×21 3+17↓BCOM)+(X0≠0)/X0←(⍳16)×16+1↓BCOM
[13]  →0,CNSQ←BCOM[21]
[14] L2:→0×⍳0=⍴X1←,MDRN
[15]  WRITE 0
[16] L2A:READ X1[1]
[17]  CON
[18]  CONCT[BCOM[1]]←1
[19]  WRITE X1[1]
[20]  →L2A×⍳0≠⍴X1←1↓X1
[21]  READ 0
[22]  →0
[23] L3:→0,(CNSQ←1),CONCT[B[1]]←0
[24] L5:→0×⍳1≠⍴,MDRN
[25]  WRITE 0
[26]  READ MDRN
[27]  CON
[28]  CONCT[BCOM[1]]←1
[29]  READ 0
[30]  →0,P←1≠1↓BCOM
[31] L4:→JQI×⍳(1≠⍴,MDRN)∨(CONT=1)∨STOP≠0
[32]  →⍳0×⍴(CNSQ←1),CONCT[B[1]]←0
[33]  →(JQB×⍳JQP=1),0×⍳0=⍴,MDRN
[34] L4B:WRITE CUMDNO
[35]  READ MDRN
[36]  P←1≠1↓BCOM
[37]  P[21]←1≠0
[38]  INST
[39]  →LA44×⍳1≠FUN
[40]  MDRN,21ADD
[41]  DATA
[42]  →0,CUMDNO←MDRN
[43] JQI:→JQA×⍳JQP=1
[44]  MDRN
[45]  →0×⍳0≠⍴MDRN
[46]  A←BCOM,CUMDNO,CONCT,CNSQ,EXT
```

```
[47]  JBQ←(((⍴MDRN),1)⍴MDRN),((⍴MDRN),36)⍴Δ
[48]  →0,JQP←1
[49] JQA:PLACE
[50] JQB:WRITE CUMDNO
[51]  PICK
[52]  READ CUMDNO
[53]  →(L4A×⍳EXT=4),L0
[54] L4A:→⍳0×1+(CHSQ←1),CONCT[B[1]]←0
[55]  WRITE CUMDNO
[56]  READ MDRN
[57]  MDRN,2⊥6+15↑BCOM
[58]  DATA
[59]  →0,CUMDNO←MDRN
[60] LA44:→(LA45×⍳V=1),(LA46×⍳V=2),(LA47×⍳V=3),FUN←1=0
[61] LA45:→0,SEL←FMSH WV
[62] LA46:FMRD SEL
[63]  →0
[64] LA47:FMRD SEL←FMSH WV
     ∇

       ∇COR[⎕]∇
     ∇ CON;EXT;V
[1]  EX:P←1=1↓BCOM
[2]   →LP×⍳LPG=1
[3]   P[21]←0=1
[4]   INST
[5]   →(FC×⍳FUN=1),(CRD×⍳1=2⊥R⍍),(CWT×⍳2=2⊥R⍍),EOP
[6]  LP:M[1+2⊥ADD[⍳4];;1+2⊥ADD[5 6]]←P
[7]   →0,ADD←1+(7⍴2)⊤1+2⊥ADD
[8]  CRD:DATA←M[1+2⊥ADD[⍳4];⍳20;1+2⊥ADD[5 6]]
[9]   →0,(STOP←1=1),RT←1= 0 0
[10] CWT:M[1+2⊥ADD[⍳4];⍳20;1+2⊥ADD[5 6]]←1=DATA
[11]  →0,(STOP←1=1),RT←1= 0 0
[12] FC:FUN←1=0
[13]  →(EOP×⍳V=0),(FC1×⍳V=1),(FC2×⍳V=2),FC3×⍳3=V←2⊥CT
[14] FC1:→EOP,SEL←FMSH WV
[15] FC2:FMRD SEL
[16]  →EOP
[17] FC3:FMRD SEL←FMSH WV
[18] EOP:→EOP1×⍳0≠EXT
[19]  →EOP2×⍳(1=STOP)∧CONCT[PCOM[1]]=1
[20]  →EOP0×⍳1=STOP∧~FUN
[21]  →(FC×⍳FUN=1)
[22]  P←M[1+2⊥ADD[⍳4];;1+2⊥ADD[5 6]]
[23]  ADD←1=1+(7⍴2)⊤1+2⊥ADD
[24]  CYRDCT←CYRDCT+1
[25]  INST
[26]  →EOP
```

```
[27] EOP0:→0,CONT←1=0
[28] EOP1:BUS EXT
[29]  BST←BST+EXT≠2
[30]  BCN←BCN+EXT=2
[31]  →EOP,EXT←0
[32] EOP2:BUS 3
[33]  →EOP
     ∇

       ∇FMRD[□]∇
     ∇ FMRD SEL;RM;T
[1]    RM←20ρ0=1
[2]    →L2×ι0=ρSEL
[3]    RM←∨/[1] M[SEL;ι20;]
[4]    →L1+21RT
[5]   L1:→L2,RM←20ρ0=1
[6]    →L2,RM←RM[;3]
[7]    →L2,RM←RM[;4]
[8]    →L2,RM←(1=T)∨3=T←,+/[2] RM
[9]   L2:DATA←(DATA←DATA∧MASK)∨RM∧~MASK
[10]   FCYCT←FCYCT+1
     ∇

       ∇FMSH[□]∇
     ∇ SEL←FMSH WV;TL;TLI;C2;C3;C4;C6;I;J;O;A;EO
[1]    TL← 16 4 ρ0=1
[2]    TLI← 16 20 4 ρ0=1
[3]    SEL←16ρ0=1
[4]    C3←1+21 2 3 1 ⍉M[WV;ι20; 1 2]
[5]    A←M[WV;ι20;4]∧((ρWV),20)ρMASK∧~DATA
[6]    C4←(M[WV;ι20;3]∧((ρWV),20)ρMASK∧DATA)∨A
[7]    I←J←1
[8]   L10:TLI[WV[I];J;C3[I;J]]←C4[I;J]
[9]    →L10×ι21>J←J+1
[10]   J←1
[11]   →L10×ι(1+ρWV)>I←I+1
[12]   TL←∨/[2] TLI
[13]   TL[WV;]←~TL[WV;]
[14]   C2←C4←C6←0
[15]   I←1
[16]  L0:O←∨/TL[WV[I];]
[17]   A←∧/TL[WV[I];]
[18]   EO←∨/(1 3)=+/TL[WV[I];]
[19]   →L1+21M[WV[I];21; 1 2]
[20]  L1:→L2
[21]   →L2,(O←O∨C2),(A←A∧C2),EO←∨/1=+/EO,C2
[22]   →L2,(O←O∨C2∨C4),(A←A∧C2∧C4),EO←∨/(1 3)=+/EO,C2,C4
[23]   →L2,(O←O∨C2∨C4∨C6),(A←A∧C2∧C4∧C6),EO←∨/(1 3)=+/EO,C2,C4,C6
[24]  L2:→L3+21M[WV[I];21; 3 4]
```

```
[25] L3:→L4
[26]   →L4,SEL[VV[I]]←EO
[27]   →L4,SEL[VV[I]]←O
[28]   →L4,SEL[VV[I]]←A
[29] L4:A←6≤+/C2,C4,C6,⍷L[VV[I];]
[30]   C6←(2≤+/C2,C4,C6,⍷L[VV[I];]),(4≤+/C2,C4,C6,⍷L[VV[I];]),A
[31]   C2←C6[1]
[32]   C4←C6[2]
[33]   C6←2↓C6
[34]   →L0×⍳(1↑(⍴VV))>I←I+1
[35]   SEL←0↑(~∧/SEL=0)×(SEL≠0)/SEL×⍳16
     ∇

       ∇INPUT[⎕]∇
     ∇ INPUT A;V
[1]    P←(14⍴0),((6⍴2)⊤A),0,1= 0 1 1 1 1 1
[2]    INST
[3]    EXT←1
[4]   ER1:→ER×⍳~∧/(9↑V←⍞)∈'01234567'
[5]    BCOM←5,,⍉(3⍴2)⊤('01234567'⍳9↑V)-1
[6]    COM
[7]    →ER1×⍳0=⍴V←9↓V
[8]    ADD←1=ADD
[9]    M←1=M
[10]   →EXT←LPG←0
[11] ER:'FIRST 9 CHARS. NOT 0 TO 7. .....TRY AGAIN....'
[12]   →ER1
     ∇

       ∇INST[⎕]∇
     ∇ INST
[1]    EXT←0
[2]    CM←(16⍴L0),LID1,LID2,LID3,LID4,LID5,LOD1,LOD2,LOD3,LOD4
[3]    CM←CM,LOD5,LDA,LDAT,LMSK,LIM,LI,LMPG,SA,SDM,DX,DT,STM
[4]    CM←CM,SIM,SSEL,SCY,SDCD,SCL,LAD,TC,GR1,GR2,GR3,GR4,MC
[5]    CM←CM,IDC,ODC,MSKC,IMKC,IHC,ADC,DC
[6]    CM← 4 16 ⍴CM,LM,LSEL,TMID,5⍴L0
[7]    →CM[1+2⊥P[22 23];1+2⊥P[24 25 26 27]]
[8]   L0:→0,(STOP←1=1),CONT←1=0
[9]   LID1:→IDL,V← 1 2 3 4
[10] LID2:→IDL,V← 5 6 7 8
[11] LID3:→IDL,V← 9 10 11 12
[12] LID4:→IDL,V← 13 14 15 16
[13] LID5:→IDL,V← 17 18 19 20
[14] IDL:→OUT,,IDCD[;V]←1=⍉ 4 5 ⍴P[⍳20]
[15] LOD1:→ODL,V← 1 2 3 4
[16] LOD2:→ODL,V← 5 6 7 8
[17] LOD3:→ODL,V← 9 10 11 12
```

```
[18] LOD4:→ODL,V← 13 14 15 16
[19] LOD5:→ODL,V← 17 18 19 20
[20] ODL:→OUT,,ODCD[;V]←1=⍳ 4 5 ⍴P[⍳20]
[21] LDA:→OUT,ADD←P[14+⍳6]
[22] LDAT:→OUT,DATA←P[⍳20]
[23] LMSK:→OUT,MASK←P[⍳20]
[24] LIM:→OUT,IMASK←P[⍳20]
[25] LI:→OUT,WV←(WV≠0)/WV←(⍳16)×~P[⍳16]
[26] LMPG:→OUT×⍳CONT=1
[27]  LPG←1
[28]  →0,ADD←P[14+⍳6]
[29] IDC:→OUT,,IDCD← 5 20 ⍴0=1
[30] ODC:→OUT,,ODCD← 5 20 ⍴0=1
[31] MSKC:→OUT,MASK←20⍴0=1
[32] IMKC:→OUT,IMASK←20⍴0=1
[33] IRC:→OUT,WV←⍳16
[34] ADC:→OUT,ADD←6⍴0=1
[35] DC:→OUT,DATA←20⍴0=1
[36] MC:M[;;]←0=1
[37]  ODCD←IDCD← 5 20 ⍴0=1
[38]  MASK←DATA←IMASK←20⍴0=1
[39]  WV←⍳16
[40]  ADD←6⍴0=1
[41]  →OUT
[42] LM:ADD←P[14+⍳6]
[43]  →OUT,M[1+2⊥ADD[⍳4];⍳20;1+2⊥ADD[5 6]]←DATA
[44] LSEL:→OUT,SEL←0+(~∧/SEL=0)×(SEL≠0)/SEL←(⍳16)×P[⍳16]
[45] TMID:→OUT,(EXT←2),BCOM←5,P
[46] GB1:→OUT,(EXT←2),BCOM←1,P
[47] GB2:→OUT,(EXT←2),BCOM←2,P
[48] GB3:→OUT,(EXT←2),BCOM←3,P
[49] GB4:→OUT,(EXT←2),BCOM←4,P
[50] SA:→LDA×⍳CONT=0
[51]  →OUT,(EXT←1),BCOM←5,P
[52] SDM:→LMSK×⍳CONT=0
[53]  →OUT,(EXT←1),BCOM←5,P
[54] DX:→DX+1+2×⍳CONT=0
[55]  BCOM←5,(0×DATA),P[20+⍳7]
[56]  →OUT,(EXT←1),BCOM[1+(V≠0)/V]←(0≠V←(V≤20)×V←2⊥ODCD)/DA⍷A
[57]  A←P[((V=0)×⍳20)+V←(V≤20)↓V←2⊥IDCD]
[58]  →OUT,DATA←(DATA←DATA∧~IMASK)∨IMASK∧(V≠0)∧A
[59] DI:→DI+1+2×⍳CONT=0
[60]  EXT←1+0×1+BCOM←5,(0×DATA),P[20+⍳7]
[61]  BCOM[1+(V≠0)/V]←(0≠V←(V≤20)×V←2⊥ODCD)/DA⍷A
[62]  →(DX+3),P←1=1↓BCOM
```

```
[63] SIM:→LIM×⍳CONT=0
[64]  →OUT,(EXT←1),BCOM←5,P
[65] SIH:→LI×⍳CONT=0
[66]  →OUT,(EXT←1),BCOM←5,P
[67] SSEL:→SSEL+1+1×⍳CONT=1
[68]  →OUT,(BCOM←5,((16-⍴SEL)⍴0),SEL,P[16+⍳11]),STOP←0=0
[69]  BCOM←5,(16⍴0=1),P[16+⍳11]
[70]  BUS 5
[71]  SEL←(SEL≠0)/SEL←P[⍳16]×⍳16
[72]  →OUT,(FUN←1=1),CT←1= 1 0
[73] SCY:→SCY+1+2×⍳CONT=0
[74]  →⍳0×⍴(EXT←1),BCOM←5,P
[75]  →OUT×⍳P[20]=0
[76]  RT←P[18 19]
[77]  CT←P[16 17]
[78]  →OUT,(FUN←~0=2⊥CT),STOP←1=1↑(0×⍳(STOP=1)∧0=2⊥CT),STOP
[79] SDCD:→SDCD+1+2×⍳CONT=0
[80]  →⍳0×⍴(EXT←1),BCOM←5,P
[81]  →OUT×⍳P[19]=0
[82]  V←(V≤20)/V← 1 2 3 +3×(2⊥P[16 17 18])-1
[83]  →S1×⍳P[20]=1
[84]  →OUT,,IDCD[;V]←1=(5,(⍴V))⍴P[⍳5×⍴V]
[85] S1:→OUT,,ODCD[;V]←1=(5,(⍴V))⍴P[⍳5×⍴V]
[86] SCL:→SCL+1+2×⍳CONT=0
[87]  →⍳0×⍴(EXT←1),BCOM←5,P
[88]  →OUT×⍳P[9]=0
[89]  →MC×⍳P[1]=1
[90]  WV←(WV≠0)/WV←((0=P[6])×WV),(P[6]=1)×⍳16
[91]  ODCD←1=ODCD×~P[3]
[92]  IDCD←1=IDCD×~P[2]
[93]  IMASK←1=IMASK×~P[5]
[94]  MASK←1=MASK×~P[4]
[95]  DATA←1=DATA×~P[8]
[96]  →OUT,ADD←1=ADD×~P[7]
[97] TC:→TC+1+4×⍳CONT=0
[98]  →⍳0×⍴(EXT←4),BCOM←5,P
[99]  CONT←P[14]
[100] →OUT×⍳P[5]=0
[101] →TC1
[102] →⍳0×1↑(CONT←1=1),STOP←1=0
[103] ADD←(P[14+⍳6]∧~P[13])∨ADD∧P[13]
[104]TC1:CT←P[1 2]
[105] FUN←~0=2⊥CT
[106] →OUT,RT←P[3 4]
[107]LAD:A←DATA[14+⍳6]
[108] →OUT,ADD←1=((DATA[16+⍳4],0,0)×P[20]=1)∨(P[20]=0)×A
```

```
[109]OUT:STOP←STOP∨P[21]
[110] →0×ι25>((⌶21)-CPU)÷60
[111] CPU←⍬
[112] CPU←⌶21
     ∇

       ∇MCON[⍬]∇
     ∇ MCON;X;Y
[1]    CONCT←5⍴CNSQ←JQP←CUMDNO←FCYCT←CYRDCT←BST←BCN←0
[2]    CPU←⌶21
[3]    ⍞←'MODULE LEVEL NUMBER OF INITIAL CONTROL'
[4]    Y←(3⍴2)⊤⍞
[5]    ⍞←'SIXTEEN BIT MODULE CODE'
[6]    X←'1'=⍞
[7]    READ CUMDNO←(16×21Y)+(X≠0)/X←(ι16)×16⍴X
[8]    ⍞←'ADDRESS OF START'
[9]    Y←(6⍴2)⊤⍞
[10]   ⍞←'CYCLE TYPE'
[11]   X←(2 2)⊤⍞
[12]   ⍞←'READ TYPE'
[13]   X←X, 2 2 ⊤⍞
[14]   BCOM←5,1=X,(10⍴0),Y,0,1,0,1,0,1,1
[15] L2:CON
[16]   →(L2×ι~STOP=0),(L1×ι~JQP=1)
[17]   'NUMBER OF FUNCTIONAL CYCLES    ';FCYCT
[18]   'NUMBER OF READ CYCLES    ';CYRDCT
[19]   'NUMBER OF BUS TRANSFERS    ';BST
[20]   'NUMBER OF BUS CONNECTS    ';BCN
[21]   →0
[22] L1:BUS 6
[23]   →L2
     ∇

       ∇PICK[⍬]∇
     ∇ PICK
[1]    A←(MDRN←JBQ[1;1]),(BCOM←JBQ[1;1+ι28]),CUMDNO←JBQ[1;30]
[2]    →ι0×⍴(CONCT←JBQ[1;30+ι5]),CNSQ←JBQ[1;36]),EXT←JBQ[1;37],A←ι0
[3]    →L1×ι1=1↑⍴JBQ
[4]    JBQ←JBQ[1↓ι1↑⍴JBQ;]
[5]    →0
[6]   L1:→JQP←0
     ∇

       ∇PLACE[⍬]∇
     ∇ PLACE;X0
[1]    A←BCOM,CUMDNO,CONCT,CNSQ,EXT
[2]    X0←(((⍴,MDRN),1)⍴MDRN),((⍴,MDRN),36)⍴A
[3]    JBQ←(((1↑⍴JBQ)+1↑⍴X0),1↓⍴JBQ)⍴(,JBQ),,X0
     ∇
```

```
      ∇READ[⎕]∇
    ∇ READ X;Y
[1]   Q[7 8 9]←65536,(0 1)+. 20 20 ⊤X
[2]   Y←1=2061⍴0
[3]   Q⌷Y
[4]   →⍳0×⍴(CONT←Y[72]),(DATA←Y[72+⍳20]),(IMASK←Y[92+⍳20]),STOP←Y[133
[5]   →⍳0×⍴(WV←Y[6+⍳16]),(SEL←Y[22+⍳16]),(ADD←Y[38+⍳16]),P←Y[44+⍳27]
[6]   →⍳0×⍴(Y←133↓Y),(LPG←Y[1]),(FUN←Y[2]),(CT←Y[3 4]),RT←Y[5 6]
[7]   →⍳0×⍴(WV←(WV≠0)/WV←WV×⍳16),SEL←(SEL≠0)/SEL←SEL×⍳16
[8]   →⍳0×⍴(Y←100↓Y),,ODCD← 5 20 ⍴100↑Y
[9]   →⍳0×⍴(Y←100↓Y),,IDCD← 5 20 ⍴100↑Y
[10]  →0,(Y←⍳0),,M← 16 27 4 ⍴Y
    ∇

      ∇WRITE[⎕]∇
    ∇ WRITE X;Y;INH
[1]   INH←Y←16⍴0=1
[2]   →(2÷⍨26)×⍳1=⍴,0,WV
[3]   INH[WV]←1=1
[4]   ADD←1=ADD
[5]   →(2+⍨26)×⍳1=⍴,0,SEL
[6]   Y[SEL]←1
[7]   A←MASK,STOP,(,ODCD),(,IDCD),,M
[8]   Y←1=LPG,FUN,CT,RT,INH,Y,ADD,P,CONT,DATA,IMASK,A
[9]   Q[7 8 9]←1,(0 1)+. 20 20 ⊤X
[10]  Q⌷Y
    ∇
```

REFERENCES

1. W. B. Davenport and W. L. Root, "Random Signals and Noise," (New York: McGraw-Hill, 1958)

2. Harry L. Stiltz, "Aerospace Telemetry," (Englewood Cliffs, New Jersey: Prentice-Hall, 1961)

3. Telemetry Working Group and Inter-Range Instrumentation Group, Range Commanders Council, "Telemetry Standards," (IRIG Document 106-71, 1971)

4. The Telemetry Working Group and Inter-Range Instrumentation Group, "IRIG Standard Language for Describing Telemetry Data: Vehicle Independent Data Base," (IRIG Document XXX-69, May 6, 1969)

5. Ronald D. Cardwell, "Comet Design File," (NASA Document X-565-67-179, April 1967)

6. S. W. Hinkal, "TOPS Internal Reference Specification," (NASA/GSFC: Data Processing Branch, Information Processing Division, 1969)

7. M. V. Wilkes, "The Growth of Interest in Microprogramming — A literature Survey," (Computing Surveys, Vol. 1, pp. 139—148, Sept. 1969)

8. Peter L. Gardner, "Functional Memory and Its Microprogramming Implications," (IEEE Trans. on Computers, Vol. C-20, pp. 764—775, July 1971)

9. L. J. Koczela, "The Distributed Processor Organization," (New York: Academic Press, "Advances in Computers," 1968)

10. A. D. Falkoff and K. E. Iverson, "APL/360: Users Manual," (Ithica, New York: IBM, 1968)

11. Sandra Pakin, "APL/360 Reference Manual," (Chicago: Science Research Associates, Inc., 1970)

12. John A. N. Lee, "The Anatomy of a Compiler," (New York: Reinhold Book Corporation, 1967) Chapter 2, pp. 23—36

13. "Aerospace Data Systems Standards," (NASA/GSFC: Data Systems Requirements Committee, NASA Document X-560-63-2, pp. I-1.1 to I-3.9, Jan. 27, 1966)

14. Frances V. Shepherd, et. al., "Data Processing Plan for OAO-A2," (NASA Document X-565-68-427, November 1968)

15. F. R. A. Hopgood, "Compiling Techniques," (New York: MacDonald/ Elsevier Computer Monographs, 1969)

16. Peter Wegner, "Programming Languages, Information Structures, and Machine Organization," (New York: McGraw-Hill Book Company, 1968)

17. William M. McKeeman, et. al., "A Compiler Generator," (Englewood Cliffs, New Jersey: Prentice-Hall, 1970)

18. Samir S. Husson, "Microprogramming: Principles and Practices," (Eaglewood, New Jersey: Prentice-Hall Inc., 1970)

19. Motorola Inc., "MECL Integrated Circuits Data Book," November 1972.