RICE COMPUTER-2

GENERAL SPECIFICATIONS

by

The Research Staff of
The Rice University Laboratory
For Computer Science and Engineering

Rice University
Houston, Texas

This work was supported in part by the United States Atomic Energy Commission, Contract # AT-(40-1)-2572, to the Rice Computer Project, Rice University, Houston, Texas, and by NSF University Science Development Grant GU-1153.

TABLE OF CONTENTS

INTRODUCTION

The Rice Computer was constructed on the Rice Campus during the years 1958 and 1959, and was placed in limited operation early in 1960. It was made available as a computing facility on a campus-wide basis in 1961. The Project was directed by Dr. Martin Graham, now Professor of Computer Science at the University of California, Berkeley. The software complement of the system was prepared under the direction of Mr. John Iliffe, now with International Computers, London. The computer architecture and the software system are described in the publications listed at the end of this section.

The system became widely known because of the advanced concepts in its operating system, particularly for its fully generalized dynamic storage allocation. These techniques were made attractive, at least in part, because the hardware has unusually versatile addressing facilities. It can form addresses with accumulated index registers. It uses multiple level indirect addressing with indexing at all levels. It has a set of four storage registers addressable by short address fields in the instruction.

The system has always been known simply as the Rice Computer, without benefit of either an acronym or an exotic sounding number. After all that it has taught about the management of data structures, and all the work it has done, its arithmetic and logical unit is about to be supplanted by a successor, which is completely new in its architecture. The change has been made, first of all, to explore new and interesting ideas in computer architecture. Secondly, the old computer was not well suited to handle real time interrupts, so a new organization is needed to do experiments in time sharing or any regimen involving multiple independent resident programs. A third motivation is the advantage to be had from contemporary circuit technology, for the sake of speed, reliability, simplicity, and a reasonable degree of compatibility with any new peripherals which might be acquired.

The following text describes the architecture of this new processor. Since the processor plays the largest role in characterizing the system, it will be called the Rice Computer-2 or for brevity R-2. The architecture of R-2 has been influenced in part by features incorporated into the software operating system of the Rice Computer. This influence may also be noted in John Iliffe's Basic Language Machine, a model of which was constructed at International Computers Ltd. during 1967-68. Finally, the architecture of the Basic Language Machine served as a starting point for the design of R-2, with Mr. Iliffe present to guide the development.

The outstanding features of R-2 are briefly stated in the following points. 1. Storage protection both between programs, and within them. 2. Convenient hardware aids to addressing multi-dimensional arrays. 3. Fully automatic stack arrangements, with provision for access by conventional addressing to items within the stack. 4. A system of data tags which simplifies the order codes, permits correct computation with mixed operands, and prevents errors due to operations inappropriate to the data.

PUBLICATIONS

Iliffe, John and Jodeit, Jane. "A Dynamic Storage Allocation Scheme", The Computer Journal, Vol. 5, No. 3, October 1962.

Jodeit, Jane. "Storage Organization in Programming Systems", Comm. of the ACM, Vol. 11, No. 11, November 1968.

Iliffe, John. "The Use of the Genie System in Numerical Calculation", Annual Review in Automatic Programming. 2. Pergamon Press 1961.

Iliffe, John. "Basic Machine Principles", American Elsevier Publishing Co.

I

CLASSIFICATION OF DATA IN STORAGE

The use of a single, undifferentiated store to contain various classes of data and programs has been common practice for decades. The segregation and identification of these classes has been universally a software function, usually implemented by a system of named arrays, with reference tables, etc. A means for differentiation which applies to each storage location is described in the following section.

The R-2 is modeled in part on the Basic Language Machine [1][2]. The Computer accommodates four major categories of words in central storage, and some of these categories include several subclasses. The four major categories are: 1. Instruction Words, 2. Numeric Words, 3. Address Words, 4. Control Words. Instruction and numeric words serve their usual purposes. Each instruction word contains two instructions. Control words contain the information necessary to locate entry points to programs.

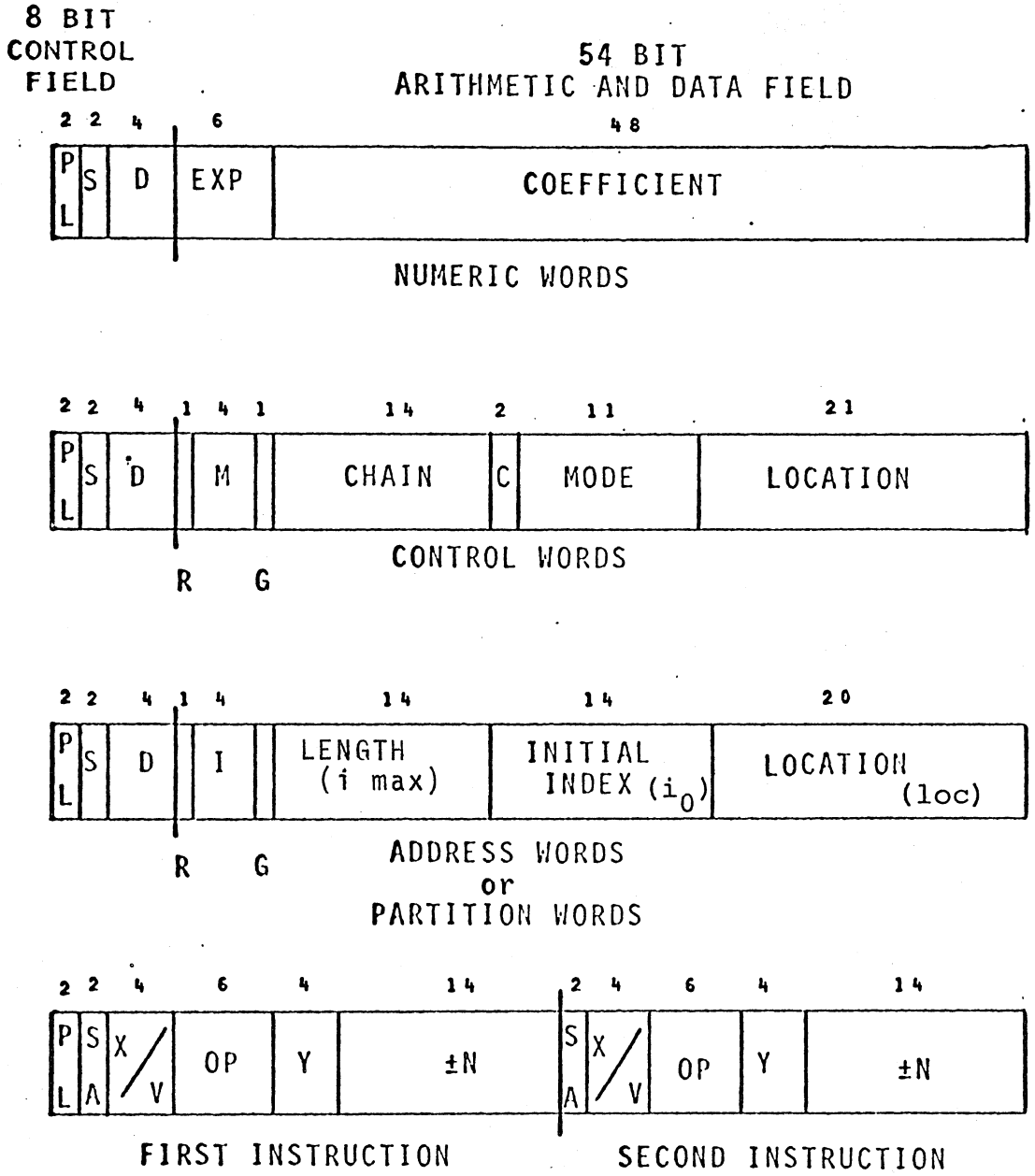
Word categories 2, 3, and 4 all contain subclasses, and they may be organized into ordered sets in storage. Address words contain the information necessary to locate any stored word of categories 2, 3, and 4. Access to storage for such words can be made only through the use of address words, which include the first word location, length, and initial index of data arrays.

A proliferation of data types is characteristic of modern

computer systems. Operation types are provided to match the requirements of operand types. It is a considerable challenge to an operating system to keep the types all properly segregated and matched with appropriate operations [3]. The structure of R-2 is designed to aid this process through the use of data tags. Tags are understood to be symbols which are associated with items of data in memory, but are not a part of the data, and are not processed along with the data. They must therefore be interpreted by the control hardware.

In the R-2 design, all words except instruction words have a data field which is 54 bits long. There are ten additional bits provided as a control field. Seven of these function as tags, in one manner or another. The 8th indicates word parity, and the 9th and 10th are spares, for the time being. In principle, the seven tags could be used to categorize individual register entries into 128 distinct classes. In practice, it is convenient to maintain relationships between major classes and subclasses, so it may not be appropriate to use all possible tag codes as class designators. Four of the seven data tag bits are so used, providing for 16 classes. Figure I illustrates the format of each major word category. The fifth and sixth control bits provide for classification that is defined by program. The seventh is a write lockout bit.

Tags associated with individual locations, as described above,



S---Software-defined tags
D---Direct tags
I---Indirect tags
M---Mark
P---Parity
L---Write lockout
A---Literal value of ±N
N---Displacement of Location
R---Restricted access to array
G---Array present in core
X---1st operand reg. selector
V---Variant on operation
Y---2nd operand reg. selector
OP---Operation code
C---Condition code

DIAGRAM OF R-2 WORD FORMATS

Figure I-1

may be considered as direct tags, since they are physically associated with individual data items and may be interpreted as the items are read. They serve as identifiers for data which is not organized by category, and in fact they make it possible to organize data in storage on any desired basis.

Tags for Arrays

Since every item of data of category 2, 3, or 4 must be accessed in the store by way of an address word, it may be convenient in many cases to organize the data into arrays consisting of a single category or class, and in fact that is the required organization for most conventional computer systems. The R-2 address word contains a four bit field which identifies the class of the items in the array to which it refers. This is called the indirect tag, since the tag is not physically associated with the words in store but is effectively assigned when the words are called. Referring to Table I-1, consider an address word which refers to a vector of real numbers. It may have a direct tag 1110, meaning that it contains an absolute address to some memory location. Its indirect tag field will be 0100, indicating that the memory locations it refers to contain real numbers.

If the address word has 0000 in its indirect tag field,

this means that the word refers to an array containing elements of more than one type. The direct tag associated with each item then determines its class. Otherwise, the tags must agree.

Control words can refer only to instructions, so they do not need an indirect field for a class designator. The corresponding field is used to carry a mark which defines the position of the control word in some hierarchy of control words. They do include direct tag fields, since their own identity as control words must be made manifest when they occur in a mixed array.

With the above hardware tagging system, it is clear that any given item of data will be described by two separate identifiers. These must agree if they identify specific types of data. If the array contains elements of more than one type, the indirect tag will be 0000. In this case the direct tag identifies the element. If both tags are 0000, a program exception occurs. On storing into an unmixed array, the tag of the item to be stored must agree with the indirect tag for the array or a program exception occurs.

The uses developed thus far for the tags which are interpreted by hardware are:

1. to give warning, or corrective action when an attempt is made to combine items from different classes which are incompatible;

2. to modify the execution microcode, as needed, to properly combine items from different classes when such combination is permissible;
3. to provide an identification for a result which does not fit the classes from which it was derived; for example, a number produced as a result of overflow, or underflow.

Tags Interpreted by Software

The fifth and sixth tag bits in words other than instruction words are reserved for identifying classes which are defined at users option by software. A single tag of similar purpose is associated with each of the two instructions that comprise an instruction word. Response to these tags is accomplished by an automatic program branch to a reserved location in core. This is called trapping. There are separate reserved core locations corresponding to the instruction tag, and to configurations 01, 10, 11 of the data tags. The value 00 is regarded as an untagged item. In usage, each reserved core location will be loaded with an entry to a routine which is programmed for whatever action is desired in connection with the tag.

Trapping occurs only if there is a certain correspondence between a tag configuration and a pattern of bits in the MODE register. Three bits are assigned to the data tags, one bit being

associated with each of the three possible tag values. A trap to the corresponding core location will occur if there is a correspondence between a tag value and the presence of a 1 in the MODE register bit associated with that tag value. One bit will initiate a trap to the location corresponding to instruction tags if a tagged instruction appears. Thus, the system can trap on any selection of the software tags.

Software-defined tags provide for the arbitrary definition of any item of data as an element of a distinct class, and the associated program can carry out any desired action with respect to this element. Thus an array that is made up of a single hardware-defined category of data may be divided into four categories by the two software tag bits. Trapping on instruction tags provide an easy way to control a tracing operation.

The two software-defined tags have a counterpart in Rice Computer #1, and other computers [5][6]. The greatest merit of this arrangement lies with its utility in interactive operation. By use of such tags and appropriate trapping procedures, any program may have a control regimen superimposed on it after it is written, without making any changes in the program itself, except for insertion or deletion of tags at appropriate points.

Microcode Control by Interpretation of Hardware Tags

The instruction set contains only one set of arithmetic and logical operations. The data tags are interpreted by the control, and the microcode for these operations is altered to suit the data types involved. Thus the operation type within these classes is determined by the data itself, and not by the compilation process. An example of this is the treatment of double length operands. These are stored in sequential address pairs. Stepping through such an array by increments of two is automatically provided by hardware. Operations applied to such operands will be appropriately altered by interpretation of the tags, and the correct procedures will be applied. Thus complex numbers, or numbers represented in double precision may be combined using the one operation set.

If a binary operation is directed to incompatible operands, one of them will be automatically converted, if possible. If it is not possible, or if any illegal combination is attempted, a program exception occurs. An appropriate tag for a result is derived from the tags of the operands, and from the outcome of the operation.

A null element, as described in [1], is "any element which cannot be directly represented by hardware, such as an out-of-range numerical result, or an undefined item". Such elements carry a

specific tag in the Basic Machine, and they normally cause a trap when they appear.

It is immediately evident that this device is useful for dealing with mishaps such as overflow, or programming errors, and of course, as a means of provoking trapping actions. M. B. Wells has disclosed the real power of this concept [7]. He points out that many results of computer operations may be perfectly valid results, but they are not included in the number representations defined for the computer. He suggests that "non-existent number" is a valid result, "indeterminate" is another and "infinite" is yet another, and that these results should not lose their identity due to lack of representation. He suggests a representation much like the one proposed for the Rice Processor.

A word with tag 1000 is an element "undefined for normal operations". For such a word, a 4 bit field internal to the word is used to distinguish the species of elements, from a total of 16 possible species. The three suggested by Wells will be included, along with others which may appear to be useful.

The appearance of any operand with tag 1000 will cause a trap. The program initiated by the trap will examine the indirect tag field of the operand which caused the trap, and will jump to a routine appropriate for the species indicated. The portions

of such words apart from the tag fields will contain such information as might be meaningful for the species, and this can be extracted by the routine.

The seventh of the bits available for tagging is interpreted in every word as a "read only" tag. Instructions will be stored in "read only" status when they are used in recursive and other reentrant programs.

Reference

[1] Iliffe, J.K., "Basic Machine Principles," Elsevier Publishing Company, 1968.

[2] Iliffe, J.K., "Elements of BLM," Comp. J., Vol. 12, #3, August 1969.

[3] Hauck, A. and Dent, B.A., "Burroughs B/6500 and B/7500 Stack Mechanism," Spring Joint Computer Conference, 1968.

[4] McKeeman, W., "Language Directed Computer Design," Proc. FJCC, 1967.

[5] "IBM 1401 System Summary" IBM Form A24-1401-1, September 1964.

[6] Gram, C. et al, "Gier - A Danish Computer of Medium Size," IEEE Trans. on Electronic Computers, EC-12, December 1963.

[7] Wells, M.B., "Elements of Combinatorial Computing," Pergamon Press (to be published).

II. ADDRESSING

The Addressing Problem

Storage systems for modern computers designed for multi-programming operation must have a means for mapping and remapping program name space into physical memory space. Commonly used systems are oriented toward conceptual and physical simplicity, and they provide hardware aids for mapping linear name space into linear memory space, with a minimal set of constraints on the user.[1][2]

In practice, the name space for a program needs to be organized into structures more complicated than a simple linear sequence. The system described here has been derived from earlier storage mapping systems, in which arrays of data in storage are delineated by codewords. These are interpreted by software routines included in the operating system.[3] Since codewords can themselves be organized into arrays, the relationships within high order data structures of any degree of complexity may be readily represented and interpreted.

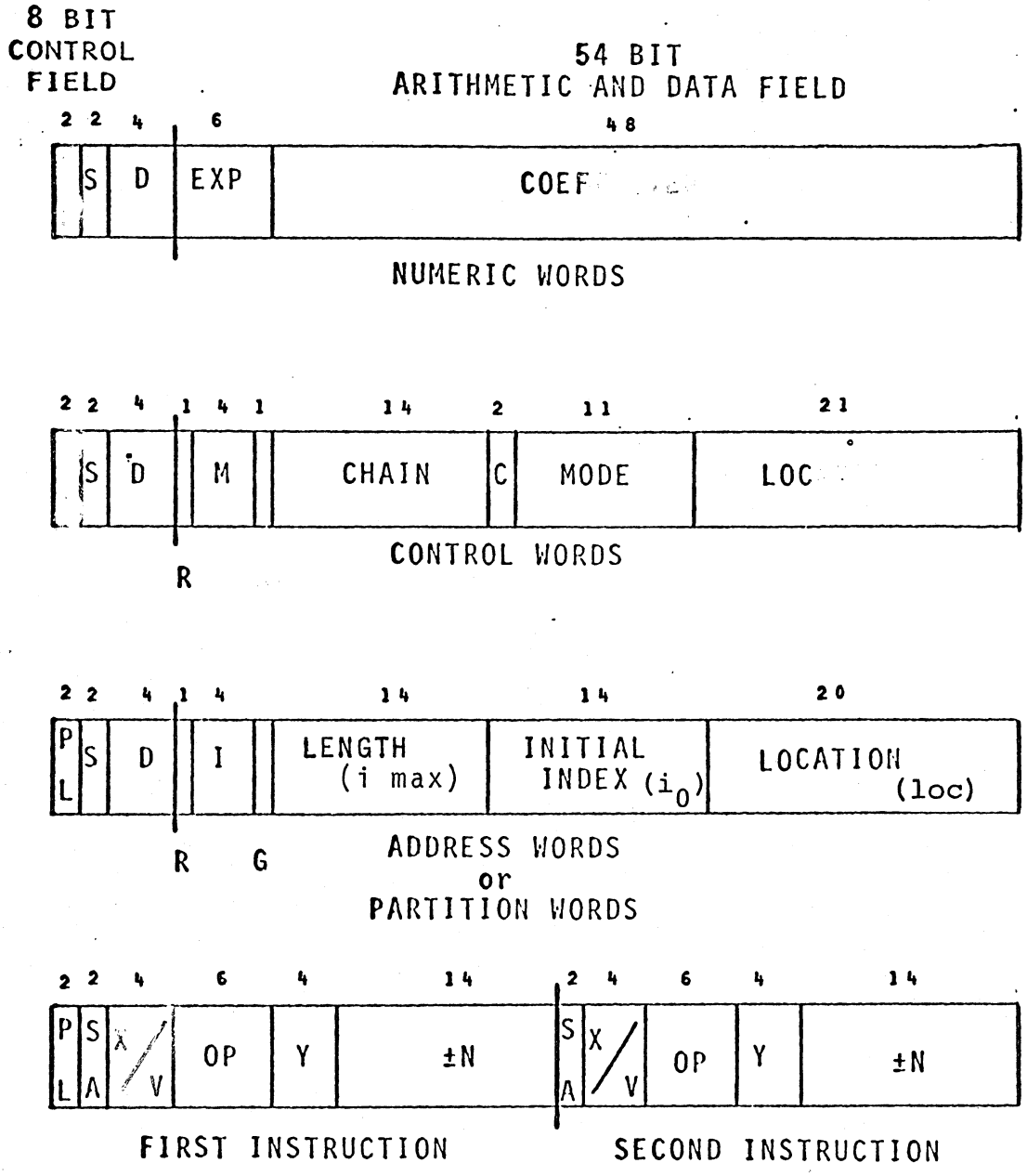
The system described is based on a similar approach, but the software is simpler. In this system, the interpretation of the words involved in the storage mapping function is done by hardware provided specifically for that purpose. This hardware generates all storage addresses. It can operate concurrently with computations in process in the numeric section of the computer.

Hardware Structures

The addressing system of the Rice Computer (R-2) has an effective address length of 20 bits. Such addresses refer to storage locations, each of which contains a 64 bit word. Although addresses are thus directed to words, each word includes tags which can identify the nature of the data stored in the remainder of the word. By this device, the correct location and length of data entities longer or shorter than a storage word may be derived. Accordingly, the system is designed to operate on data entities whose length is specified for each particular array. At present, operations have been implemented for full word and for double word entities.

A set of sixteen 64 bit general purpose registers is provided in the processor structure. Each register includes a tag field which identifies register content as explained in Section 1. Registers may contain Numeric Words, Address Words, or Control Words. The registers are not used for storing instructions. Figure II-1 shows the format of the various word types recognized in R-2, and Table I-1 indicates the assignment of data tags.

The computer instruction occupies half of a computer word. Each instruction word contains two completely independent instructions. Program entrance or exit may be made at either instruction in a word, because storage addresses which designate instruction locations are 21 bits long, and therefore can designate half-words.



S---Software-defined tags
D---Direct tags
I---Indirect tags
M---Mark
P---Parity
L---Write lockout
A---Literal value of ±N
N---Displacement of Location
R---Restricted access to array
G---Array present in core
X---1st operand reg. selector
V---Variant on operation
Y---2nd operand reg. selector
OP---Operation code
C---Condition code

DIAGRAM OF R-2 WORD FORMATS

Figure II-1

The instruction format includes the three address fields, labeled X, Y, and ±N. X and Y are each 4 bit fields which designate a register from the sixteen included in the processor. Register 0 is reserved for use as the stack pointer. Register 1 is the two word arithmetic register combination which functions as accumulator in arithmetic and logic operations and contains the final result of any such operation. The high order member of this pair is called U; the low order member is called R. Registers which are specifically committed, as these are, may be addressed implicitly in appropriate operations. Registers 2 through 15 are available to the program as general purpose registers. The Field ±N is a 14 bit field used according to rules to be described in following sections.

The computer instruction provides for, at most, two explicit operand address designators. Some functions may require no explicit addresses. Others may require one, or may make one explicit and one or more implicit. A few may be viewed as "wired macros", because they may imply a sequence of elementary functions from the set, and may utilize both explicit and implicit addresses. This is not a new idea. Existing computers, for example, have instructions for testing, for indexing, and for program jumps, and then a set for doing all three by a single instruction.

Arithmetic and logic functions of two arguments take the first argument from Register 1. The second argument is specified by the

the Y field, according to addressing rules to be discussed in a later section. For these functions, the X field serves as a modifier of the operation code, rather than as a register designator. For some functions, the four data tags also operate as inflections on the operation code. This leads to the notion that the vocabulary list is enormous. There are, in fact 6 bits in the function code, so there can be 64 primitive operations. This is a generous allotment, since these can be fitted to the data as required, by using information in the operand tags to modify the operation microcode as required.

Addressing Rule for Operand Access

Binary operations are understood to be of the form x op y → X. For operations other than arithmetic or logic, the operand x is found in the register designated by instruction field X, and the operand y is found in the register designated by instruction field Y. If the content of either register is tagged as an Address Word, and the operation is not one involving address modification or any other process which may be applied to Address Words, the word in the register will be used to form a memory location number.

For the operand designated by field X, the unmodified LOCATION field of the register is taken as the memory location number.

The corresponding word in memory will be used as an operand if it bears an appropriate tag. If it bears the tag of a Chained Address Word, another access will be made, using the new address. If it is neither a Chained Address nor a legitimate operand, a program exception will be indicated.

For the operand designated by field Y, the formation of the effective memory location number is done according to an addressing rule which involves interpretation of the special tag in the instruction, called the "a" bit, and an evaluation, with interpretation, of the fields Y and ±N in the instruction. The "a" bit determines whether or not the register designated by field Y is to be used in obtaining the operand.

If "a" = 1, the register is not used. In that case, the Y field is used to designate the choice of one of several options:

1. If Y=0, the value of the ±N field itself is used as the operand. This is the familiar immediate addressing option, available on many computers.
2. If Y = 1, the number in the ±N field is used as an absolute core location number, without further modification, and the operand is fetched from, or stored into this location. The first 16K memory locations may be addressed in this way. This region is available for parts of the operating system whose location must be invariant.
3. If Y=2 or 3, the number in the ±N field is added to the location number of the memory word containing the current

instruction, to form the effective location number required by the operation. This provides addressing relative to program location. It is useful in programs which are expected to be relocated. Jump addresses for branching within a program segment are formed in this way, and constants stored within the segment are referenced by this means. Store operations to such locations can be prevented by setting the Write Lock-Out bit.

4. If Y=4 and the instruction is a jump, the ±N field is added to the current location number, as in Y=2 or 3. The word fetched from that location must be an address word; it must point to a vector of control words. The address word is then modified by the contents of Register 1 (U), a previously set integer, and control is transferred to the instruction addressed by the designated control word. A program exception occurs if any of the conditions are not met.

If "a"=0, the register designated by the Y field will be used in obtaining the operand. The register may contain the operand or an Address Word; this is made manifest by the tags. If the register contains an Address Word, that word will be used to form a memory location number for the operand. If the register content, regardless of tags, is required, the ±N field will contain all ones. This is equivalent to minus zero, in the one's complement system, so it is an

unused element in the range of values of ±N. When this designation appears, the content of the register designated by Y is used as the operand, whatever it may be. Of course, the tags associated with the operand are interpreted according to the rules, and no illegal combinations are permitted. This option eliminates the need for special function codes for register to register transfers, etc.

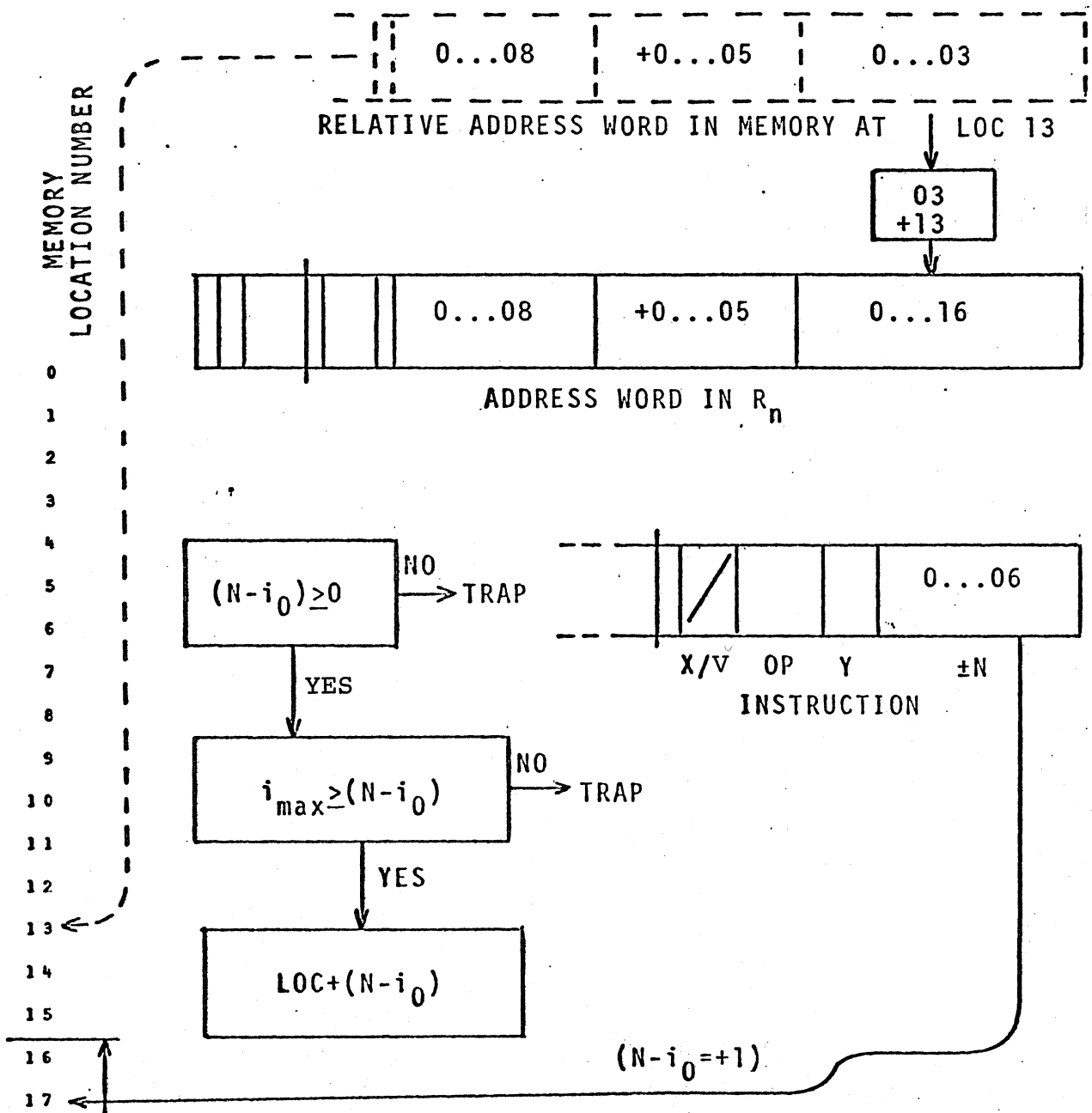
If "a" = 0, and ±N ≠ (-0), a normal storage access is to be made. If the register Y is not tagged as an address, its content is taken as the operand, and ±N is ignored, unless Y is tagged as an integer. In that case the operand is (Y) + (±N). If Y is tagged as an address, the memory location number is formed from the fields i_0 and loc of the address word, as illustrated in Fig. I-1, and field ±N of the instruction word, as $[(\pm N) - (\pm i_0)]$ + loc.

From the above explanations of the function of the X and Y field in the instruction, it is evident that accesses outside the first 16K words can be made only by addressing through one of the 16 registers, and the contents of the register must be tagged as an Address Word. The rules of tag interpretation are drawn for all operations in such a way that words tagged as addresses cannot be inadvertently spoiled by mistakenly combining them in arithmetic operations, or overwriting them with data.

Addressing Rule for Data Arrays

Each address word may define an array of data. The 20 bit location number specifies the location of the first stored word of the array. The field of 14 bits, labeled i_{max}, specifies the length of the array. If all arrays were simple lists of numbers, it might be reasonable to consider the first stored word as being the first word of the array. However, there are cases in which a given array is related to one or more other arrays. The rows or columns of a matrix are examples of such a grouping of related arrays. The numbering of the elements in each array is usually conceptually important. An irregular matrix, or a triangular matrix may be conceived as a rectangular matrix with all its positions numbered, but with many of these positions empty. It is, of course, desirable to store only the values for the non-empty positions. Thus, for a particular vector, all elements up to the nth may be empty, so the nth element must be the first stored element of the array. The difference between the conceptual numbering of the nth element, and its physical position as the first stored element in the array, can be accounted for simply by assigning the value n to the field i_0 in the address word which describes the array.

In referencing an element by an instruction, if the ith element is wanted, the i will be placed in the ±N field of the instruction. The address is formed as $[(\pm N)-(\pm i_0)]$+LOCATION. For example, if the



A RELATIVE ADDRESS DESCRIBING AN ARRAY OF 12 ELEMENTS

It is known that elements 1, 2, 3 and 4 do not exist in memory. The first stored element is 5. It is stored at location 16.

The address is stored at location 13. The array which it describes is 3 spaces up from its own location. Thus its location field is stored with the value 3. When it was brought to register R_n, its own location number, 13 was added into its location field 3 to give a value 16, the absolute address of the first stored word of the array.

The instruction is to operate on element 6 of the array. This is stored at location 17.

Figure II-2

first 7 elements of a conceptual array are null, and element 8 is the first stored, i_0 will be given value +8. A reference to this element will be of the form op(Y±N) where N = +8. The quantity $N-i_0$ will be formed as 8-8 = 0, which added to the loc field, gives the first stored word. A valid address must satisfy the rule $i_{max} > (N-i_0) \geq 0$, to insure that it lies within the defined limits of the array. This test is applied automatically by hardware to every address before it is used. The use of addresses is illustrated by diagram in Figure II-2.

Special operations are available for setting the location number to point to successive words of the array. This is done by the MOD function, which adds an increment to the loc field, and subtracts this same quantity from the length field. When the length field goes negative, the bound has been exceeded, and a program branch or a program exception occurs.

Addressing Rule for Storage Management Functions

The inclusion of the initial index value, i_0 in addresses requires that the value of ±N be consistent with i_0 for all instructions referencing this array. This is a convenient arrangement, when both code and data are created under known conditions, as is the case in many applications programs.

It sometimes happens, however, that a particular system program must access several arrays belonging to user programs. The initial indices may not be known, and even if they were, alteration of the ±N value or any other field in an instruction is not permitted in a program, once it has been loaded. For this reason, a set of load, store and move functions is included in the vocabulary set which interpret the field i_0 in an address as being zero, regardless of its content. A value of zero in the ±N field will then reference the first stored word of an array, etc. For storage allocation, and other system processes, this is a convenient and natural way to reference blocks of storage.

Dynamic Relocation of Selected Blocks of Storage

By means of addressing rules developed thus far, Address Words can be used to describe the detailed structure of the programmer's name space. Storage relocation procedures must preserve this structure, but they will not always need to be applied to data arrays individually. In many cases, larger blocks of storage, including several arrays, and perhaps several program segments, may be taken as a unit for a storage relocation procedure.

It is obvious that if an array is relocated in store, any Address Word which designates the actual physical location of the array must have its location field updated to reflect that change. The updating of all Address Words so involved must be a part of

the relocation procedure. Address Words of this form are tagged as absolute.

Alternatively, the location field of an Address Word may specify just the displacement of the beginning of its array with respect to its own location. It is then easy to derive the absolute location of the array as required, because the absolute location of the Address Word will be at hand when the word is loaded into a register. Such Address Words are tagged as relative. When they are loaded into registers, the absolute location of the beginning of the array is automatically determined and written into the location field of the register, as a hardware function.

If a relocation procedure involves the movement of arrays described by relative Address Words, and the Address Words are moved together with the arrays, with no alteration of their relative locations, then the Address Words need not be updated. Obviously, Address Words which are absolute must be updated in any case. Should a relocation procedure be performed which does result in a change of the relative locations of relative Address Words, their location fields must likewise be updated.

The use of relative Address Words provides the opportunity to simplify and shorten storage relocation procedures if suitable conventions and strategems are observed in the designation of blocks to be moved. This is entirely a system function, and there are no hardware imposed conventions on the block structure for storage allocation.

Chaining of Addresses

Address Words which describe arrays are normally brought from storage to one of the sixteen general registers before they are used for referencing memory. This is done because alterations required in indexing may be performed on copies held in registers, while the Address Word in storage remains in storage in the original form, properly describing the full array.

Address Words are frequently used to obtain an item from a single location in storage, so that no alteration of the Address Word is involved. An instance of this usage is the process of transmitting arguments to subroutines. In the simplest case, the subroutine would be supplied with an Address Word pointing to a sequence of Address Words, each of which points to one of the arguments of the subroutine. The Address Words pointing to the arguments would be most conveniently used as Indirect Addresses, so that the subroutine could reference an argument in a single operation. Notice that there is no occasion to index any of the argument Address Words. Such addressing procedures are referred to as Chained Addresses in this discussion, since R-2 does not use direct addressing except in a highly restricted sense.

The R-2 chaining procedure avoids the necessity of loading addresses into registers before using them. This saves time, reduces register occupancy and avoids the use of specific

instructions for the process. It is implemented by tagging Address Words as either Chained or Unchained. The rule of tag interpretation for chaining may be stated as follows:

> For any access, location information will be extracted and used from, at most, one Unchained Address Word, but it may be extracted and used from as many Chained Address Words as occur in any sequence.

It follows from this that if the Address Word in the register which is used to start the sequence is Unchained, the next Unchained Address Word, or any word not tagged as an Address, will be taken as the final operand. If the initial Address Word in the register is Chained, the datum to which it points is extracted and the process is repeated until either data is encountered, and the sequence ends, or an Unchained Address Word is found This word will have its location information extracted and used for a further access, and the chaining will continue until a non-Address, or a second Unchained Address Word is encountered and taken as the operand.

Conclusions

It has been deemed important in this system to provide a means for keeping storage references within the defined boundaries of an array, whatever its length. This requires that the computer

be able to evaluate datum and limit information for every reference. Given the hardware for doing this, it is natural to express lengths of storage blocks in words, not in pages. Variable length blocks are natural in this system, so blocks can be arranged to correspond to the form of arrays as they are conceived by the programmer, or developed during execution.

Most storage control systems are at their best when the program calls for many successive references to a particular region of storage, so that the amount of updating of the block index is reduced [1]. Similarly, this system is at its best when the program calls for many successive references to a particular array. Once an array has been entered, the system can derive addresses for successive references without a table look-up.

The components involved in the bounds checking hardware represent a trade-off against the associative memory used for the block index look-up used in some other systems [2]. It gains the advantage of easy monitoring of the very common programming error which produces out-of-bounds references. It provides storage protection by blocking the source of storage violations. This not only blocks references to other programs, it blocks references which might be valid in form, but are directed to the wrong array in the current program.

This system avoids the loss of storage space due to partially

filled pages. It makes addressing of structured data simpler, since the storage addresses are formed in the addressing unit according to the structure, and the arithmetic processor does not need to be committed to evaluation of storage mapping functions. The result should be simpler and shorter programs which run faster.

III. STACK OPERATIONS

The Stack Concept

A stack is generally understood to mean an addressing arrangement organized in such a way that the last item to be stored will be the next one to be called. Stacks are often implemented in software by an appropriate algorithm for the formation of the addresses.

Stacking procedures may also be implemented in hardware. In such machines, the formation of storage addresses which form the stack is done by hard-wired sequences. In one well known hardware arrangement, operation codes are associated with operands derived from the stack, and they are not, in general, associated with explicit addresses [1].

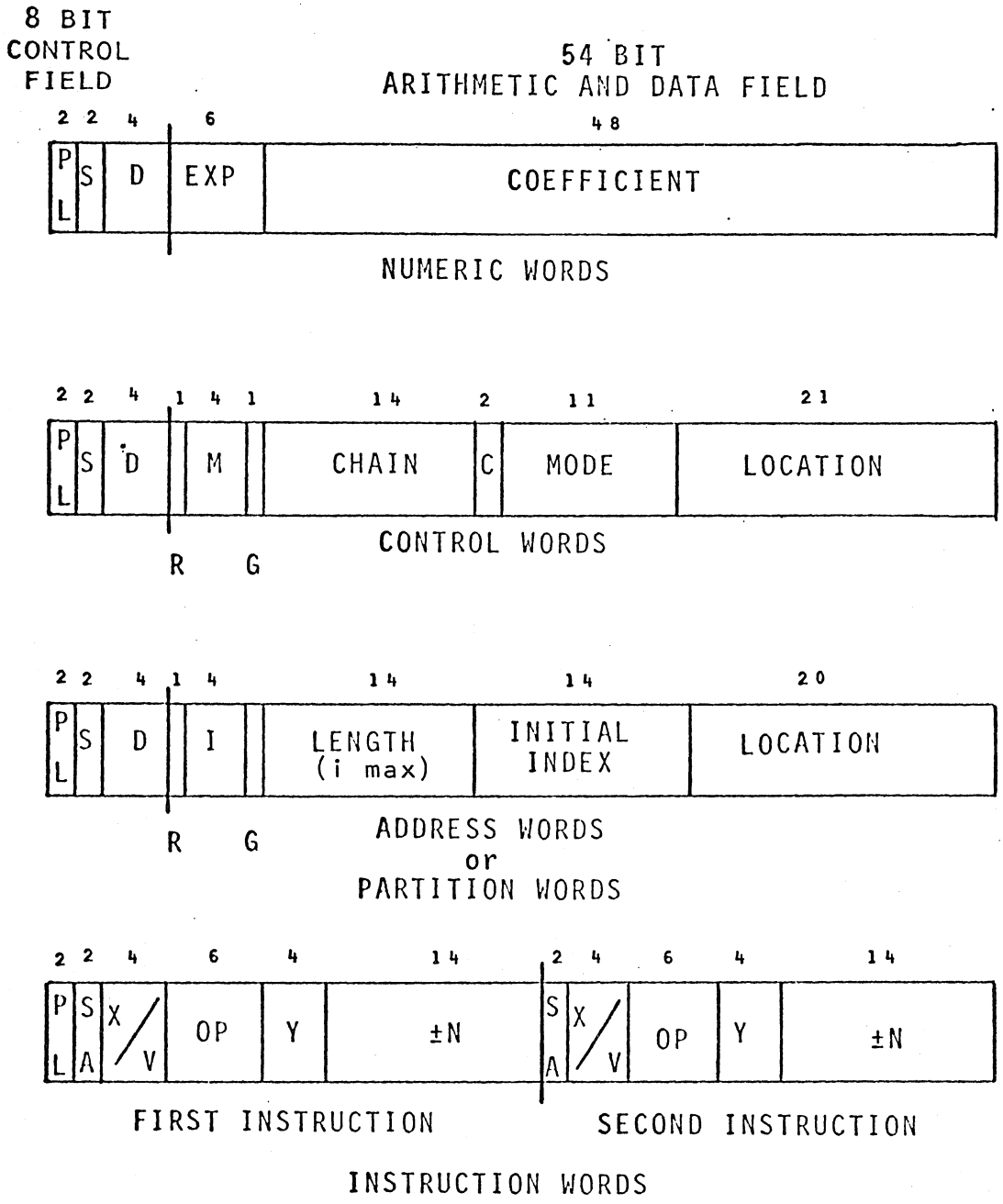
The concept of stacking developed from the fact that data storage locations must be chosen according to some well defined and easily interpreted convention to minimize the amount of computing effort and storage space needed to keep track of where things are stored. The stack is one of the most efficient and powerful tools for this purpose. As a result, during the past decade it has been customary to characterize computer architecture as stack oriented or conventional, in a mutually exclusive sense [2]. The stack convention is a powerful one, but it can be a handicap if the commitment is so inflexible that data thus stored cannot be retrieved by alternative means.

The system to be described is included in a new arithmetic and logic unit made for the Rice Computer System, known as R-2. It provides the options of both the stack convention, with automatic generation of the stack addresses, and the familiar arrangement of operations associated with explicit operand addresses. Both options are available at all times, with no mode switching required. As a consequence any element of the stack is accessible, simply by associating its address explicitly with an operation.

In this processor, one unique address is used to designate the stack domain. When this address is used storage accesses follow the stack convention, with address generation being done by hardware. Since, for all instructions, operations are associated with at least one explicit address, any operation may be associated with the address designating the stack domain. This is expressed in the second operand address field of the instruction. (See Figure III-1). When the configuration "a"=0, Y=0000, N=0...0 appears as an effective address in an instruction, the operand used will be the top element of the stack. Stacking or unstacking, with adjustment of the pointer, occurs as a part of the execution.

The General Addressing Rule

Operations in the R-2 System are of the form X op Y → X, where X and Y may be the contents of any of sixteen registers



S---Software-defined tags
D---Direct tags
I---Indirect tags
M---Mark
P---Parity
L---Write lockout
A---Literal value of ±N
N---Displacement of Location
R---Restricted access to array
G---Array present in core
X---1st operand reg. selector
V---Variant on operation
Y---2nd operand reg. selector
OP---Operation code
C---Condition code

DIAGRAM OF R-2 WORD FORMATS

Figure III-1

designated by two four bit fields, the X and Y fields in the instruction. Figure III-1 illustrates the format of instruction words and of other word types. Each register, and each location in store, includes a tag which indicates the word type occupying the rest of the location.

The content of either or both of the designated registers may be tagged as Address Words. If so, the LOCATION field of the address word designates the memory location to be accessed when the register is selected by the instruction field X. When the register is selected by the instruction field Y, the memory location is formed from the LOCATION field and the INITIAL INDEX field of the address word, and from the ±N field of the instruction, by the summation

$$A = [\pm N - (\pm i_0)] + \text{LOCATION}$$

where INITIAL INDEX is represented as i_0.
Since Address Words describe data arrays, they include the LENGTH field, which specifies the length of the array. A hardware bounds check is made to insure that the relation

$$i_{max} > (N - i_0) \geq 0$$

is satisfied, where LENGTH is represented as i_{max}.

Structure of the Stack

The physical arrangement of the stack in memory in its simplest form is a sequence of contiguous storage locations, with a distinctive marker stored at each end of the sequence. A word type defined and tagged as a Partition Word is used for this purpose. The format of a Partition Word is identical to that of an Address Word. (Figure III-1). The I field of the Partition Word is used to categorize it as to subtype. The subtypes used for defining stack structures are: 1. "Beginning", 2. "End", or "Forward Reference", 3. "Change Stack Mode", 4. "Back Reference". The beginning of the stack will contain a Partition Word of the subtype "Beginning". It will occupy the highest location number in the region of memory assigned to the stack. Items added to the stack occupy successively smaller location numbers. The end of the sequence of locations available for stack building will contain a word tagged as a Partition Word of subtype "End". On every stacking operation, the resident word is first tested. If the end of stack symbol is detected, an error trap will occur. Otherwise, normal stacking will proceed. The end of stack symbol will occupy the lowest location number in the stack sequence.

The special register R_0 is dedicated to serve as the stack pointer; it is not a full length register, but its content is tagged as an address and interpreted according to the common addressing rules. Its LOCATION field, α, points to the location of the next

vacant address in the section of memory available for stack building. By convention, the top of the stack is address $\alpha+1$ and unstacking of the top element includes the operation $\alpha+1\rightarrow\alpha$.

The content of register R_0 may be used as an ordinary address, simply by using a value other than 0...0 in field N of the instruction. Such a configuration is not interpreted as a normal reference to the stack domain, so the hardware operation $\alpha\pm1\rightarrow\alpha$ does not occur. For example, the configuration a=0, y=0, N=0...01 will form an effective address $\alpha+1$ and will access the top element of the stack without changing the value of α in R_0. Such an operation is convenient for copying the top of the stack without unstacking. Similarly, the configuration a=0, y=0, N=k will access the element of the stack $k^{\underline{th}}$ from the top, without unstacking. In general, any element within the range delimited by the bounds check as applied to R_0 may be accessed. This covers all of the topmost segment of the stack, as further explanation will show. All elements of the stack may be accessed by use of suitably constructed address words held in registers other than R_0.

Allocation of Storage to the Stack

A region of memory for the stack is assigned initially with a length specified by the input program. Dynamic storage allocation is applicable to all storage, including the region occupied by the stack. The space initially assigned may not meet the maximum

required by the running program. In the event of a trap on the end of stack symbol, a recovery procedure may be taken which first calls for the allocation of an additional region, anywhere in memory, to the stack. Then the Partition Word of subtype "End" will be replaced by subtype "Forward Reference", with the LOCATION field pointing to the first location of the new region. This location will receive a subtype "Back Reference" which points to the last item before the "Forward Reference". Normal stacking and unstacking is performed by the hardware across such a link.

The use of Partition Words thus provides hardware accommodation for a dynamically varying stack with a length up to the maximum available memory. Although the required length of stack is typically short for some computations, there are many which require an amount which is unpredictable and may at times be very large.

Double Length Operands

The computer can operate on double word operands, and generate double word results. These must be stored as single items occupying two storage locations. By convention, the two locations are consecutive, with no restriction on the odd-even relationship.

For stacking operations, single word storage is taken as the normal mode when any program segment is entered. Upon occurrence of a double word operand, a hardware procedure sets "MODE" bit 8

to double word mode and then stacks a Partition Word of subtype "Change Mode". The LENGTH field of this word will show the length of the segment of single word items immediately preceding it. A subtype "Change Mode" may be stacked, at the programmers option, to return to single word mode. Alternatively, single word items can be stored while in double length stacking mode, with the waste of the extra storage location. The LENGTH field of the "Change Mode" word will show the number of single word locations occupied by the sequence of double words preceding it. Indexing is by items, not by words. The length of the partition immediately preceding the top of the stack is expressed in the LENGTH field of the stack pointer as a number of words, designated as i_{max}. To begin the indexing operation, the index value is converted to words according to the state of the double word mode bit. If $i_{max} \geq$ index value, the desired item lies within the partition. The index value is added to the LOCATION field of the stack pointer, and the item is taken from the location so formed.

If i_{max} < index value, the value of i_{max} is converted to items, and the index value in items is reduced by this amount. Then the value $i_{max} + 1$, in words, is added to the stack pointer and the 0^{th} word, which identifies the end of the partition, is accessed. If it is a Partition Word of the subtype "Beginning", or if it is a Control Word, a bounds check error is indicated. If it is any other

subtype of Partition Word, its LENGTH field is used to form a new value for i_{max}, and the above process is iterated until the index falls within the partition.

Program Branching

A jump which requires linkage must find its jump destination in a word called a Control Word. (The word format is illustrated in Figure III-1).

A Control Word has some similarity to the program status word in certain contemporary computers. It contains a twenty-one bit LOCATION field which is used as a jump destination. This provides for jumping to either of the two instructions in a word. It also contains a field for displaying a condition code which records the result of test operations, and a MODE field which displays the operating mode of the processor. There are two other fields, labeled MARK and CHAIN. The use of these fields will be explained in a later section.

The normal addressing rule is applied to determine the destination of jump instructions. If the instruction "a" bit has the value 1, the Y field is decoded to distinguish between absolute addresses given by the displacement field N, and addresses relative to the program counter, given by CN±N. If the "a" bit is

zero the jump destination is taken from a Control Word which either resides in the designated register, or is taken from memory by the ordinary addressing process. If this process does not reference a word tagged as a Control Word, an error will be indicated.

Control Words will be created when a program is compiled, to link various program segments. All the fields will appear in an initialized form, with the appropriate value appearing in the jump address.

Segmentation of the Stack

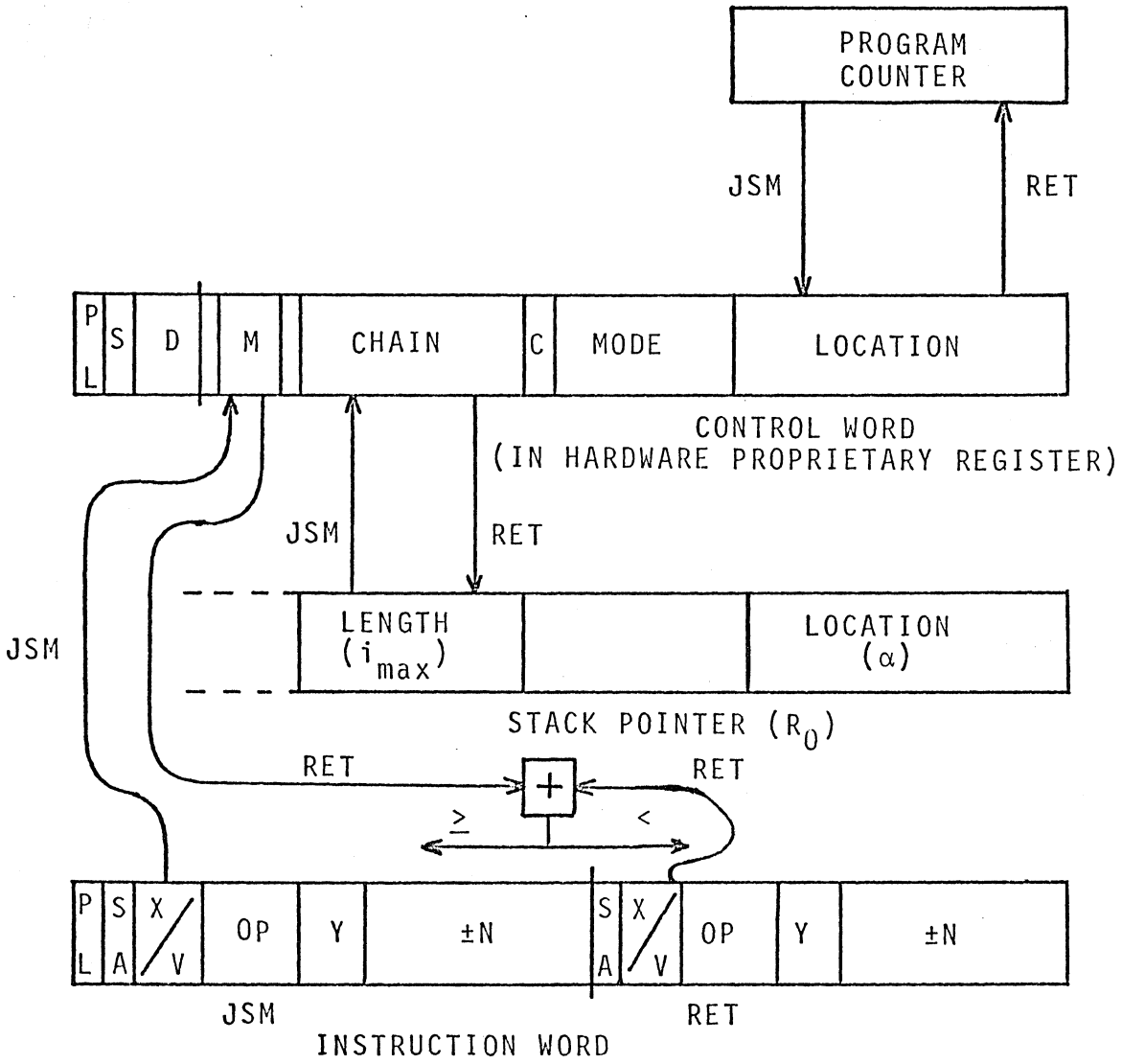
Although the stack concept is a simple one, the stack structure is not a simple structure. The stack grows as a succession of segments because the program which uses it traverses a succession of program segments. The program may need to retrace its path, or to re-enter it in some other fashion. If a program is to be re-entered, operations must be resumed with its stack segment as it existed when the previous exit was made. Since the stack pointer has all the features of an address word, the i_{max} field can be used to indicate the length of a stack segment. By convention, stack building is accompanied by the operation $\alpha-1\rightarrow\alpha$, whereas unstacking includes the operation $\alpha+1\rightarrow\alpha$. Along with this, stack building includes the operation $(i_{max}+1)\rightarrow i_{max}$ and unstacking includes the opposite. Thus i_{max} always shows the current length of the

stack segment. When the stack pointer, R_0, is used either in an unstacking operation, or as an ordinary address, the normal hardware check is made to see that the effective address is within this segment.

Two specific jump instructions are provided for maintaining a succession of stack segments corresponding to a succession of program segments. These are the operation JSM (JUMP and SET MARK), and the operation RET (RETURN TO MARK). The JSM operation creates a Control Word and stacks it by a normal stacking operation before the JUMP is made. Figure 2 illustrates the information transfers that are involved.

The Control Word is formed by the following procedure. The LOCATION field of the Control Word is loaded with the address of the instruction following the JUMP instruction. This is called the CONTROL NUMBER (CN). It is just the content of the program counter after fetching the jump but before its execution. This is the natural point for re-entry to the program segment.

The CHAIN field is loaded with the present value of i_{max}, from R_0. Finally, the MARK field of the Control Word is loaded from the X/V field of the jump instruction. The Control Word is then stacked, the i_{max} field of R_0 is set to 0, and the jump is executed. A new stack segment will be constructed as the new program segment proceeds, until another jump occurs. This may be



INFORMATION TRANSFERS FOR MICROPROGRAM
OF THE OPERATIONS JSM AND RET

Figure III-2

another JSM, or it may be RET (RETURN TO MARK).

When RET is to be executed, the X/V field of this instruction will contain a value related to the MARK of the Control Word at the top of the desired segment. The return may go to any segment in the stack, by a proper choice of marking conventions. The desired segment is found by successive examination of the Control Words, and only the Control Words, in the stack. No reference table is required.

This process begins by fetching the top Control Word or Partition Word in the stack. The location number of this is formed simply by forming $\alpha+1+i_{max}$ from the stack pointer, R_0. This location number is loaded into the location field of R_0. When the word is obtained, the i_{max} field in R_0 is loaded from the CHAIN field of the Control Word, or the LENGTH field of a Partition Word. The stack pointer is thus initialized for the partition which precedes the word. When this is accomplished, the X/V field of the RET instruction is compared to the MARK field of the word, if it is tagged as a Control Word. If the MARK value is $\geq$ the value in ±N, this is the desired entry point. The program counter is loaded from the LOCATION field of the Control Word, and the program resumes at this point. Note that the stack pointer is set to overwrite the Control Word on the next stack building operation, and will continue to overwrite subsequent locations. Its LENGTH field

defines the length of the partition which occupied the top of the stack at the time the Control Word was written, and, of course, will be adjusted as stacking or unstacking occurs. The bounds check is applied on all unstacking operations, so a warning is provided when the limit of a stack segment is reached.

If MARK value is < the value in ±N or if the word is a Partition Word, this is not the desired entry point. In this case, the next Control or Partition Word is addressed and tested by exactly the same procedure as before, using the stack pointer with its new values for α and i_{max}. This iterative process continues until the desired entry point is found, or until a Partition Word of subtype "Beginning" is read out. This indicates that the entire stack has been traversed, so an error trap occurs. Usually subsequent Control Words would be marked in a descending sequence, although several Control Words can have the same MARK. In such a case, the first one encountered which satisfies the test will be used. In a RET sequence, Partition Words are addressed just as Control Words are, taking i_{max} from the LENGTH field, but they are not tested.

It should be noted that the return to a particular MARK does not involve an unstacking procedure. The return requires one fairly simple hardware process for each segment traversed, to get to any desired point. The time required is independent of the number of data items that are entered in the stack. It is therefore

a much faster process than the usual unstacking procedure.

The operations JSM and RET, aided by the marking facility, are a great convenience for any computational task which has a tree-like form. Such tasks are very common in modern computing practice. In general, any process that can be recursively defined can be arranged in this form.

A simple example pertinent to the use of these operations is the problem of determining whether a tree contains a given item. The comparison process continues as long as the items do not match, until the elements have all been examined. Upon any instance of a match, whether it occurs early or late in the sequence of tests, the comparison process is terminated, and control must go to a different program segment, perhaps at a higher node in a tree structure. The RET operation is designed to aid this process. Such procedures are used frequently in programs for implementing compilers for high level computer languages. These programs should benefit greatly from the marking facility.

Conclusions

In normal programming practice, a stack is an attractive storage structure for operands which are to be processed on a last in-first-out basis, such as arguments for recursively called procedures. It is also attractive for data of arbitrary ordering

which must be simply stored and recovered. Such a case arises when several registers must be saved to accommodate an interrupt.

For all systems in which stacks are formed and operated by a programmed algorithm, the option of explicit addressing is retained and used extensively. In the hardware implementation, where the choice between explicit addressing or stacked operands has been made as one or the other, but not both, explicit addressing has the greatest number of adherents. This is understandable, because explicit addressing has the greatest generality, at least for conceptually simple storage arrangements. The system described here has brought the advantage of automatic stack manipulation to a machine which is fully committed to explicit addressing, thus bridging the choice which has led to two widely differing concepts in computer architecture.

Reference

[1] Burroughs Corporation, "The Descriptor", Bulletin 5000-20002-P, February, 1961.

[2] Amdahl, G.M., Blaauw, G.A., and Brooks, F.P., "Architecture of the IBM System 360", IBM J. of Res. and Dev., Vol. 8 No. 2, April, 1964.

[3] Iliffe, J.K., "Basic Machine Principles", (Book) American Elsevier Publishing Company, New York, 1968.

[4] IBM System 360 System Summary, Form A22-6810-2.

PROGRAM BRANCHING

Program branching within a program segment will use the literal value of the location field in the jump instruction. ("a" bit must be = 1.) This is modified by the value of the control counter (instruction counter) and used as the jump destination. The jump address is thus relative to the present location of the program. Such jumps do not require recording of the status of the processor, nor any return linkages.

A jump to a different program segment must find its jump destination in a word called a control word (tagged 1010 or 1011). The jump instruction will contain an address in the Y $\pm$ N field. The register Y may be tagged 1010. If so, $\pm$N is ignored and this is taken as the desired control word. If not, it must be tagged as an address, in which case the addressing rules are followed to find the control word. If it is not tagged either as a control word or an address, or if the addressing rule does not locate a control word, a program exception occurs.

Control words have some similarity to the program status word in contemporary computers. It contains a twenty bit location field which is used as a jump destination. It also contains a field labeled "Mark", which identifies the position of this control word in a sequence of control words. Another field called "Chain" indicates the number of address spaces to the location of the next

related control word. It contains a field for displaying a condition code which records the result of test operations, and a field which displays the operating mode of the processor.

Control words will be created when a program is compiled, to link various program segments. All the fields will appear in an initialized form, with the appropriate value appearing in the jump address.

Control words will also be created when a program interrupt occurs, or on a jump in which a return link is desired. In these cases the location field will be loaded with the value of the control counter + 1, and the other fields will be loaded with the conditions that exist at the time of the interrupt or jump.

Control words are related to instruction arrays the way addresses are related to data arrays, and instruction arrays are subject to change of location, just as data arrays are. Accordingly, control words stored in core point to instruction lists relative to their own location. When they are brought to registers to be used, they are made absolute by modifying their location field by the value of the location at which they were stored. If needed, control words may be tagged as absolute. In this case they are used as found, without any modification.

V

MACHINE FUNCTION LIST

This section consists of a brief description of the R-2 instruction set. A detailed description is given by the microprograms in section VI.

1. Operations on Numeric Classes

Number representation, and the results of arithmetic operations are the same as their counterparts in the Rice Computer and will not be elaborated here. In the following descriptions, B is a general buffer register which contains the second or Y operand fetched by means of the general addressing rule.

00 - EXTRACT/MASK COMPARE (XTR/MCP)

The operation is specified by bits 3 and 4 of the X field of the instruction:

00 - Replace bit i of U with bit i of B if bit i of R is 1.

01 - Replace bit i of U with bit i of B if bit i of R is 0.

10 - Bit i of U is 1 if the i-th bits of U and B are different and bit i of R is 0.

11 - Bit i of U is 1 if the i-th bits of U and B are different and bit i of R is 1.

The Boolean condition code is set in register CC and the store option is applied.

SEPT. 22, 1970

00	3 VAR. BOOLEAN	10		20 LD	30 STO
01		11	ARITHMETIC	21 CBL	31 CBS
02		12		22 LDI	32 STI
03	2 VAR. BOOLEAN	13		23 XFA	33 XCH
04		14	TEST	24 XFB	34 RFA
05		15	USRGB	25 SVF	35 RFB
06	UNUSED	16	SHIFTA	26 MV	36 TAG
07	NUMBER CONVERSION	17	SHIFTB	27 BTR	37 UNUSED

40	JSM	50	DOT	60 IOA	70
41	JSL	51	GET	61 UNUSED	71
42	JCC	52	PUT	62	72
43	JPT	53	MOD	63	73
44	JL	54	LIM	64	74
45	JNL	55	MEM	65	75
46	JLT	56	CTR	66	76
47	JGE	57	RET	67	77

NOTES ON R4 FUNCTIONS

SEPT. 22, 1970

(1.) FUNCTION: $I(F) = 15_8$ = USRGB

I(x)	FUNCTION	CC	
0000	Z→	UNCHANGED	} STORES "0" OR "1" (INTEGER) TO "y"
0001	I→	"	} USING ADDRESSING RULE. X1 IS NOT DISTURBED.
0010	-Z→	"	
0011	-I→	"	
0100	STR	"	
0101	LDR	ARITHMETIC	
0110	SUR	"	$y \to S$, $S \to U$, $U \to R$
0111	XUR	"	$U \rightleftarrows R$, NO SECOND OPERAND
1000	CLA	"	$y \to (X1)$, LIKE LD., BUT SETS CC ARITHMETICALLY
1001	NEG	"	$-y \to (X1)$
1010	ABS	"	$\lvert y \rvert \to (X1)$
1011	NBS	"	$-\lvert y \rvert \to (X1)$
1100	CLA→	"	} SAME AS ABOVE, WITH RESULT STORED BACK TO LOCATION OF OPERAND.
1101	NEG→	"	
1110	ABS→	"	
1111	NBS→	"	

(2.) $I(F) = 07_8$ = NUMBER CONVERSION

THESE FUNCTIONS SET ARITHMETIC CONDITION CODES.

I(x)		
0000	INC	} ADDRESSING RULE OBTAINS y (MUST BE SINGLE PRECISION N TYPE)
0001	INC→	} IF y IS NOT AN INTEGER (0101) CONVERSION TO INT OCCURS AND $y \to X1$

(3) $I(F) = 56_8$ = CTR (CONTROL)

THIS FUNCTION IS USED TO READ AND LOAD THE OTHERWISE UNADDRESSABLE REGISTERS MODE, CC, AND ST (SOFTWARE TAG).

I(X)				
0000	SCC	y = INT.	$y_{53} \uparrow y_{54}$	REPLACE CONTENTS OF CC.
0001	SMD	y = INT.	y_{44-54}	" " " MODE
0010	SST	" " .	y_{53-54}	" " " ST
0011	MON	" " .	"1's" IN y_{44-54}	TURN ON CORRESP. BITS IN MODE.
0100	MOF	" " .	"1's" IN y_{44-54}	" OFF " " " MODE.
0101	GCC	CC $\rightarrow X1_{53-54}$,	TAG = "INT.",	OTHER BITS ZERO.
0110	GMD	MODE $\rightarrow X1_{44-54}$,	" = " ,	" " " .
0111	GST	ST $\rightarrow X1_{53-54}$,	" = " ,	" " " .
1000 ↓ 1111	UNUSED			

(4) $I(F) = 57_8$ = RET

I(X)		
0000	RTL	RETURN TO MARK EQUAL TO OR LESS THAN I(N).
0001	RTG	" " " " " " GREATER " " .
0010 ↓ 1111	UNUSED	

(5) $I(F) = 33_8$ = EXCH $\quad X \neq y$

IF $I(Y) = 0$ AND $I(N) = +0$, THE CONTENTS OF $I(X)$ ARE STACKED (AND SP ADVANCED) AND AN INTEGER ZERO IS LOADED INTO X.

(6)

$I(F) =$ 34_8 = RFA (FIELDS $0 - 15_{10}$)
35_8 = RFB (FIELDS $16_{10} - 31_{10}$) .
36_8 = TAG (FIELDS $0 - 15_{10}$)

SINCE THE STACK POINTER CONSISTS OF FIELDS 12 AND 14 ONLY, AN ATTEMPT TO MODIFY ANY OTHER FIELD WITH THESE INSTRUCTIONS RESULTS IN AN ERROR. FOR ALL THREE FUNCTIONS, IF $I(Y) = 0$, AND $I(N) = +0$, AN INTEGER ZERO IS STACKED WITH THE SPECIFIED FIELD INSERTED.

(7) $I(F) = 25_8$ = SVF

IF $I(Y) = 0$ AND $I(N) = +0$, THE TOP WORD ON THE STACK IS EXCHANGED WITH THE CONTENTS OF X.

(8) MOD $(I(F) = 53_8)$ CAN BE APPLIED TO X0 TO SHORTEN THE STACK, BUT LIM $(I(F) = 54_8)$ FAILS ON X0.

(5) $I(F) = 33_8$ = EXCH $\quad X \neq y$

IF $I(Y) = 0$ AND $I(N) = +0$, THE CONTENTS OF $I(X)$ ARE STACKED (AND SP ADVANCED) AND AN INTEGER ZERO IS LOADED INTO X.

(6) $I(F) =$ 34_8 = RFA (FIELDS $0-15_{10}$)
35_8 = RFB (FIELDS $16_{10}-31_{10}$) .
36_8 = TAG (FIELDS $0-15_{10}$)

SINCE THE STACK POINTER CONSISTS OF FIELDS 12 AND 14 ONLY, AN AITEMPT TO MODIFY ANY OTHER FIELD WITH THESE INSTRUCTIONS RESULTS IN AN ERROR. FOR ALL THREE FUNCTIONS IF $I(Y) = 0$, AND $I(N) = +0$, AN INTEGER ZERO IS STACKED WITH THE SPECIFIED FIELD INSERTED.

(7) $I(F) = 25_8$ = SVF

IF $I(Y) = 0$ AND $I(N) \neq 0$, THE TOP WORD ON THE STACK IS EXCHANGED WITH THE CONTENTS OF X.

(8) MOD $(I(F) = 53_8)$ CAN BE APPLIED TO X0 TO SHORTEN THE STACK, BUT LIM $(I(F) = 54_8)$ FAILS ON X0.

01 - LOGICAL PRODUCT (AND)

The operation is determined by bits 3 and 4 of the X field of the instruction as follows:

00 - Replace U by the bitwise logical product (AND) of registers U and B.

01 - Replace U by the complement of the bitwise logical product of U and B.

10 - Form the logical product of U and B and the Boolean condition code is set but U is unchanged.

11 - The bitwise complement of U replaces U.

The Boolean condition code is set and the store option applied.

02 - LOGICAL SUM (OR)

This operation is the same as AND except that the logical sum (OR) is used for inflections 00, 01, and 10 and the complement of B replaces U for inflection 11.

03 - SYMMETRIC DIFFERENCE (SYD)

This operation is the same as AND except that the symmetric difference (exclusive OR) is used for inflections 00, 01, and 10. Inflection 11 is not used.

04 - IMPLICATION (IMP)

This is the same as SYD except that the result of the logical expression "U implies B" ($\bar{U} \vee B$) is used instead of symmetric difference.

05 - REVERSE IMPLICATION (RIMP)

This is the same as IMP except that "B implies U" ($\bar{B} \vee U$ is used.

10 - ADDITION/SUBTRACTION (ADD/SUB)

If bit 4 of the X field of the instruction is 0, U and B are added and the result placed in U, R. If it is 1, B is subtracted from U. In both cases, if bit 3 of the X field is 1, the result is negated. The arithmetic condition code is set and the store option applied.

13 - REMAINDER DIVIDE (REM)

If bit 4 of the X field is 0, the double length operand in registers U, R is divided by the operand in register B. If bit 4 of the X field is 1, the operand in B is divided by the one in U. The quotient is put in R and the remainder in U.

If bit 3 of the X field is 1, the result is negated.

11 - MULTIPLY (MPY)

When bit 4 of the X field is 0, the operand in U is multiplied by the operand in B and the product placed in the extended register U, R. When bit 4 is 1, B is multiplied by U. If bit 3 of the X field if 1, the result is negated.

SHIFTING

16 - DOUBLE REGISTER LOGICAL SHIFTS (SHA)

The U and R registers are shifted logically, the number of places being determined by the operand in B. Variant bits one and three determine the shift directions of U and R respectively, a "1" indicating a left shift, a "0" a right shift. The indicated shift directions are reversed if the operand in B is negative. The absolute value of the contents of B mod 128 is the shift length.

Variant bits two and four control spill from R to U and from U to R respectively, a "1" enabling the spill from one register to the other, a "0" forcing all vacated bits to be filled with "0"'s.

Boolean condition codes are set based on the final contents of U.

17 - SHB

This op code encompasses several functions related to shifting. They are listed below with their corresponding variant codes in octal and three character mnemonics.

0	U logical left	LUL
1	U logical right	LUR
2	R logical left	LRL
3	R logical right	LRR
4	U, R arithmetic left	DML
5	U, R arithmetic right	DMR
6	(not assigned)	
7	(not assigned)	
10	bit count	BCT
11	bit count accumulating	BCA
12	zero count	ZCT
13	zero count accumulating	ZCA
14	most significant bit	MSB
15	least significant bit	LSB
16	most significant zero	MSZ
17	least signigicant zero	LSZ

The single length logical shifts all fill vacated bits with "0"'s. Shift direction will be reversed if the operand in B is negative. Boolean condition codes are set.

The double length arithmetic shifts fill vacated bits with the sign of U mantissa (U_{MS}). Exponents are not disturbed. The sign and overflow bits of R mantissa are always set to U_{MS}. The remaining bits of U and R are shifted and handled as if arranged $U_{MOV}, U_{M1}, U_{M2}, \ldots, U_{M46}, U_{M47}, R_{M1}, R_{M2}, \ldots, R_{M46}, R_{M47}$. Any bits not equal to U_{MS} which spill into or beyond U_{MOV} on a left shift will cause a condition code of OVERFLOW to be set. The direction of arithmetic shifts will also be reversed for a negative operand in B. Arithmetic condition codes are set based on the final contents of U. data is in R originally

The bit and zero count operations shift R right logically the number of places indicated by the operand in B, negative numbers being taken as zero. As "1"'s or "0"'s as appropriate spill from the low order end of R they are counted in U. U is initially cleared to zero for BCT and ZCT, but the counted bits are added to the previous U for BCA and ZCA. Arithmetic condition codes are set.

The most and least significant bit and zero operations set in U the number of shifts needed to, for example, in the case of MSB, just shift the most significant "1" out of the exponent sign. The word is interpreted logically, the exponent and mantissa bits being combined to form a single 54 bit boolean quantity. If the

operand in B is null for MSB or LSB, or all "1"'s for MSB or LSB, the result will be the integer 55_{10}. The R register is used for the shifting in these operations and at completion will contain the result of the final shift. Arithmetic condition codes are set.

12 - DIVIDE (DIV)

This operation is the same as RDIV except that the quotient is put in register U and the remainder in register R.

14 - TEST (TEST)

The X field of the instruction is used to specify one of 16 possible arithmetic or logical operations to be performed. The result is used to set the CC register but no other registers are changed.

2. DATA MANIPULATIONS

40 - LOAD (LD)

The Y operand address is formed by means of the general addressing rule. The contents of that address is then transferred into register X. (If register X contains an address, it will be over-written.) The condition code is set on the basis of the direct tag of the data fetched.

26 - CHAIN BREAK LOAD (CBL)

The same as LD except that no chaining of addresses is allowed.

34 - LOAD INITIAL (LIZ)

The same as LD except that the I_0 field of the address is taken to be zero.

41 - SAVE AND FETCH (SVF)

The same as LD except that the contents of register X is saved on the top of the stack before the register is loaded.

43 - STORE (STO)

The Y operand address is formed according to the general addressing rule. The contents of register X is then transferred to this location.

27 - CHAIN BREAK STORE (CBS)

The same as STO except that no chaining of addresses is allowed.

35 - STORE INITIAL (SIZ)

The same as STO except that the I_0 field of the address is taken to be zero.

42 - MOVE (MV)

The Y operand address is formed using the general addressing rule. The contents of this address are then copied into register X, or into the address specified by register X if it is tagged as an address.

37 - BLOCK TRANSFER (BLT)

The same as MV except that register X must contain an address and relative address and control words are not derelativized or relativized as they are transferred.

50 - EXCHANGE (EXCH)

The general addressing rule is used to form the Y operand address. Then the contents of this address is exchanged with the contents of register X.

24 - GET ELEMENT (GET)

The general address rule is applied to form the Y operand address. The contents of this location must be an integer and the contents of register X must be an address. Then the y-th element of the array pointed to by register X is loaded into the U register.

25 - PUT ELEMENT (PUT)

The same as GET except that the contents of U is put into the y-th element of the array pointed to by register X.

52 - DOT (DOT)

The same as GET except that the contents of the y-th element of the array pointed to by register X is loaded into register X.

3. PROGRAM BRANCHING

51 - JUMP AND SET LINK (JSL)

This is an unconditional branch operation. Control is transferred to the location formed by the general addressing rule after a return link (control word) has been created and stored in register X.

20 - JUMP AND SET MARK (JSM)

This is similar to JSL. The return link is stored on the stack and has the effect of saving the contents of the stack and creating a new stack region.

22 - RETURN (RET)

This instruction is used to restore the stack after a JSM instruction and return to a previous program segment.

21 - JUMP ON CONDITION CODE (JCC)

This is a conditional branch instruction. It is conditioned by the state of the condition code (CC) register which has previously been set by another instruction (e.g. logical, arithmetic, or data handling). The CC register has two bits and so may be viewed as storing a number from 0 to 3. A correspondence between the X field of the instruction and these numbers may be made by numbering the X-field bits from 0 to 3. If bit j of the X field is 1 and cc=j, then control is transferred to the location specified by the Y operand using the general addressing rule. Otherwise the

next instruction in sequence is executed. Note that by setting more than one bit of the X field to 1 the branch can be conditioned on more than one condition.

54 - JUMP ON PROGRAM TAGS (JPT)

This is the same as JCC except that the software tag register (ST) is used in place of the CC register.

44 - JUMP IF LAST (JL)

Register X must contain an address. The program will branch to the location specified by field Y if the LENGTH field of register X is 0. Otherwise the length field is decremented by 1 and the LOCATION field is incremented by 1. (This is equivalent to a MOD (56) by 1). Execution continues in sequence.

45 - JUMP IF NOT LAST (JNL)

This is the same as JC except that the branch is taken if the LENGTH field is not zero.

46 - JUMP IF LESS THAN 0 (JLT)

Register X must contain an integer. The program branches if this number is less than zero. Otherwise the number is decremented by 1 and the next instruction is executed.

47 - JUMP IF GREATER THAN OR EQUAL TO) (JGE)

This is the same as JLT except that the branch is taken when register X is greater than or equal to zero.

4. ADDRESSING OPERATIONS

56 - MODIFY ADDRESS (MOD)

The Y operand address is computed by means of the general addressing rule. It must contain a nonnegative integer and register X must contain an address. If this integer is less than or equal to the LENGTH field of the address, it is added to the LOCATION field and subtracted from the length field. This has the effect of shortening the array addressed by X, from the end with the lowest indices.

57 - SET LIMIT (LIM)

This operation is the same as MOD except that the Y operand replaces LENGTH field of the address. This shortens the array addressed by X from the end with the highest indices.

53 - TEST MEMBERSHIP (MEM)

Register X and Y must contain addresses. The operation checks to see if the LOCATION field of register X is pointing to a location which is a member of the array defined by register Y. The result is indicated by the CC register and if it is in the array, its index is stored in register X.

5. WORD BUILDING OPERATIONS

30,31 - EXTRACT FIELD (XFA, XFB)

A word is fetched using the general addressing rule. The X field of the instruction is then used to select a field of this word which is extracted and put into the U register. Fields 0 through 15 (as listed in Table VI-2) can be selected using XFA and fields 16-31 can be specified using XFB.

Is U cleared first?

32, 33 - REPLACE FIELD (RFA, RFB)

These operations are the opposite of XFA and XFB. The contents of the U register replace the specified field.

55 - FIELD REPLACEMENT (TAG)

The general addressing rule is not used and register Y contains the word to be altered. The X field of the instruction is augmented by the "a" bit so that any one of the 32 fields can be selected. The specified field is replaced by as many bits an needed from the low-order end of the ±N field of the instruction. This is an "immediate" operation, which allows it to be monitored at assembly time. The general user will use this operation instead of the more general RFA and RFB operations.

(CONTROL WORD BIT NO. (OCT) FIELD)	23	24	25	26	27	28	29	30	31	32	33
(MODE REGISTER BIT NO. →)	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11
	INSTRUCTION TRACE MODE	DOUBLE WORD STACK MODE	DISABLE WRITE PROTECT	I.O. INTERRUPT ENABLE	I.O. INTERRUPT ENABLE	(SPARE)	(SPARE)	MODIFIED SOFTWARE TAG MODE	SOFTWARE TAG 1 TRAP	SOFTWARE TAG 2 TRAP	SOFTWA TAG TRAP

MODE BITS #4 AND #5 CONTROL EXTERNAL INTERRUPTS AS FOLLOWS:

BIT #4	BIT #5	
0	0	NO EXTERNAL INTERRUPTS
0	1	ALLOW LEVEL 1 INTERRUPTS ONLY
1	0	" " 1 AND 2 " "
1	1	" " 1, 2, AND 3

ASSIGNMENT OF DEVICES TO EACH PRIORITY LEVEL IS DONE BY SOFTWARE IN THE PDP/11.

MODE BIT #9 CAUSES THE SOFTWARE TAG REGISTER (ST) TO HOLD THE TAG OF OPERANDS LOADED INTO X1 ONLY. WHEN MODE BIT #8 IS OFF, ST IS SET BY THE SECOND OPERAND OF ALL INSTRUCTIONS.

RESERVED LOCATIONS IN MEMORY

CONTAINS LOC$_8$	WHEN AN INTERRUPT OCCURS AND X15
	ANYTHING BUT AN ADDRESS: HARDWARE USES LOC. NO. BELOW AS ABS. MEM. LOCATION
	AN ADDRESS: HARDWARE USES LOC. NO. BELOW AS MEM. LOC. REL. TO X15
0-17	REGISTER EXTENSIONS
20	ARITHMETIC INTERPRETATION TRAP
21	SOFTWARE TAG #1 TRAP
22	SOFTWARE TAG #2 TRAP
23	SOFTWARE TAG #3 TRAP
24	ERROR TRAP A
25	ERROR TRAP B
26	ERROR OPERAND
27	TRACE OPERAND #1
30	" " #1 (ONLY USED FOR DOUBLE WORD ITEMS)
31	TRACE OPERAND #2
32	" " #2 (" " " " " ")
33	TRACE TRAP
34-46	STACK OVERFLOW BUFFER (11_{10} WORDS)
47	STACK OVERFLOW TRAP
50	TIMER INTERRUPT
51	LEVEL A INTERRUPT
52	LEVEL B INTERRUPT
53	LEVEL C INTERRUPT
54	
55	
56	
57	(SPARE)
60-75	COMMAND-STATUS WORD PAIRS FOR LOC 51_8-57_8
76-136	32_{10} WORD MPX DATA BUFFER

VI. MICROPROGRAMS

The microprograms which describe the behavior of the various instructions are presented in the form of flow charts. No attempt has been made to accurately represent that part of the hardware of the machine which is hidden from the user. Instead, registers and register transfers were selected and used to facilitate the descriptions. However, the effect of these instructions on the programmable registers and memory is accurately described.

All registers are referred to by a name which has been selected to indicate its function wherever possible. A complete list of these registers is given in Figure VI-1. It is also possible to refer to various subfields of the registers and these are listed in Figure VI-2. The convention which is used is that the subfield is identified, either by number or name, in parentheses following the register name. For example, U(15) denotes the LOCATION field of the U register and IN(OP) denotes the OP code field (field 3) of the instruction register. Individual bits of registers and subfields can be referenced by means of subscripts. For example $MODE_7$ indicates bit 7 (counting from the left and starting with 1) of the MODE register and $IN(X_{1,2})$ is the first two bits of the X field of the instruction register.

Registers can be thought of as 1-dimensional arrays. There

are two sets of registers which are conveniently viewed as 2-dimensional arrays or arrays of registers.

M_i	i-th location (absolute) of main core memory
R_i	i-th general purpose register (i=0,1,...,15)
B	Buffer register
IN	Instruction register
P	Memory pointer
CN	Control Number (Program counter)
T	Temporary storage register
U	R_1, general purpose register Number 1
R	Extension of U
CC	Condition code register
ST	Software tag register
MODE	Mode register
TPR	Temporary store for R field
TPG	Temporary store for G field
ITAG	Temporary store for Indirect tags
OL	Operand location flag
CH	Chain register

Figure VI-1. Register Designations

These are the sixteen general registers and the core memory

which are referred to by the names R and M respectively. For these arrays, subscripts are used to indicate a particular register of the set. Thus M_{258} is word number 258 of the memory, and R_0 is the first general purpose register. The subscripts in this case can also be a register name. Then $R_{IN(X)}$ is the general purpose register indicated by the number stored in the X field of the instruction register.

There are two types of register transfers that are used in these diagrams. The first may be thought of as a bitwise transfer and is indicated by a left pointing arrow. Thus if the operation $U \leftarrow R_2$ is executed, U would be set to contain the same bit pattern as R_2. The second type of transfer is indicated by a left pointing double arrow. In this case, $U \Leftarrow R_2$ means that U is to be set equal to the number stored in R_2.

Two types of transfer are the same if the lengths of the registers involved are equal. To illustrate the differences let S and T be two registers and assume that S has more bits than T. Then the transfer $S \leftarrow T$ would cause the contents of T to be set into S, right justified and all other bits of S set to 0. If on the other hand T contained a negative number, then $S \Leftarrow T$ would cause the left most positions of S to be set to 1 since 1's complement representation is used.

Several (micro) subroutines are used in the flow charts and these are briefly described below.

NAME	DESCRIPTION	FIELD NUMBER
TAG	Direct tag field	8
S	Software tag field	7
WP	Write Protect bit	6
R	Restricted access bit	9
G	Array present in Core bit	11
N	±N field of instructions	5
Y	Second operand field of instructions	4
OP	Operation code field of instructions	3
X	First operand field of instructions	2
A	"a" bit of instructions	18
L	Location field of addresses	14
IO	Initial index field of addresses	13
IM	Length field of addresses	12
I	Indirect tag field of addresses	10
L	Location field of control words	15
MD	Mode field of control words	23
CC	Condition code field of control words	16
CH	Chain field of control words	12
MK	Mark field of control words	10

Figure VI-2. Subfield Designations

ADDR - This routine defines the general addressing rule explained previously. There are four possible exits:

P - This exit is taken whenever a memory access, other than to the stack, must be made. A pointer to the memory location is returned in register P. A routine such as FETCH must be used to actually make the access.

S - This exit is taken whenever a stack access is to be made. No pointer is returned; the stack routines calculate it depending on the function.

V - This exit is taken whenever a memory access is not needed. The operand itself is returned in the buffer register B.

E - This exit is usually taken to indicate an error but some operations make special use of it.

FETCH - This routine is used to fetch the word pointed to by register P. It is returned in the buffer register B. No chaining is allowed.

CHAIN FETCH - This is the same as fetch except that chaining is allowed and the last word in the chain is returned in register B.

STORE - This routine stores the contents of register B into the location indicated by the contents of register P.

CHAIN STORE - This is the same as STORE except that the word is stored in the last location of the chain.

DESTIN - This routine is used to determine the destination of a JUMP instruction. It also restores the status of the machine according to the control word it finds there and modifies the control number (CN) so that transfer of control can be made.

PUSH STACK - The contents of buffer register B are stored on the top of the stack and the stack pointer is incremented automatically.

READ STACK - If the ±N field of the instruction is zero, the top word of the stack is returned in register B (exit V) as long as that word is not an address word. Otherwise, a pointer to memory is returned (exit P). This pointer is determined by the address word on the top of the stack if ±N is zero otherwise it points to the N-th word on the stack.

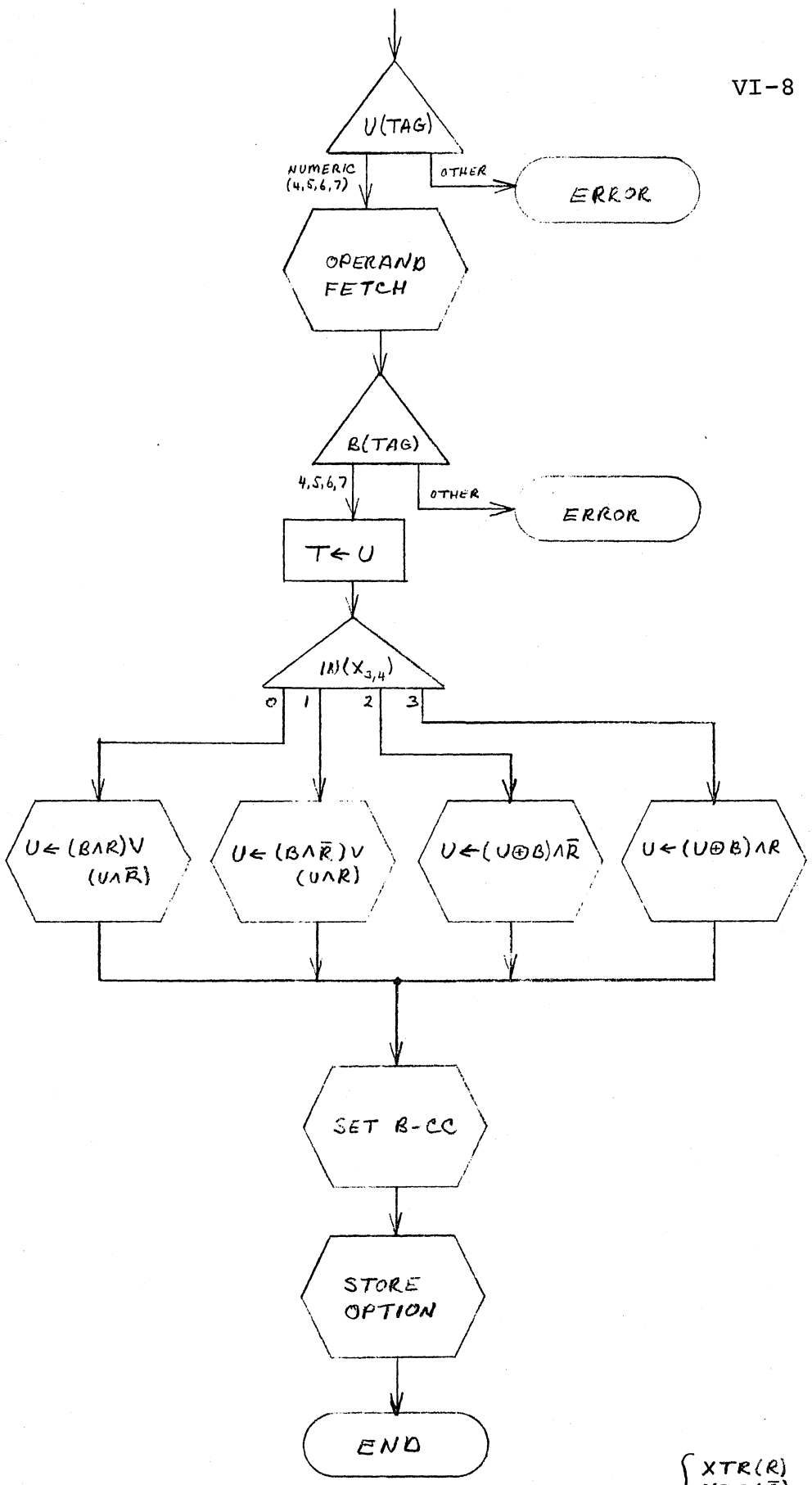
WRITE STACK - The contents of register B are stored in the stack. If the ±N field of the instruction is 0, they are stored on the top of the stack. Otherwise, a pointer to the N-th location of the stack is returned.

OPERAND FETCH - This routine is used to fetch the second (or Y) operand for arithmetic operations.

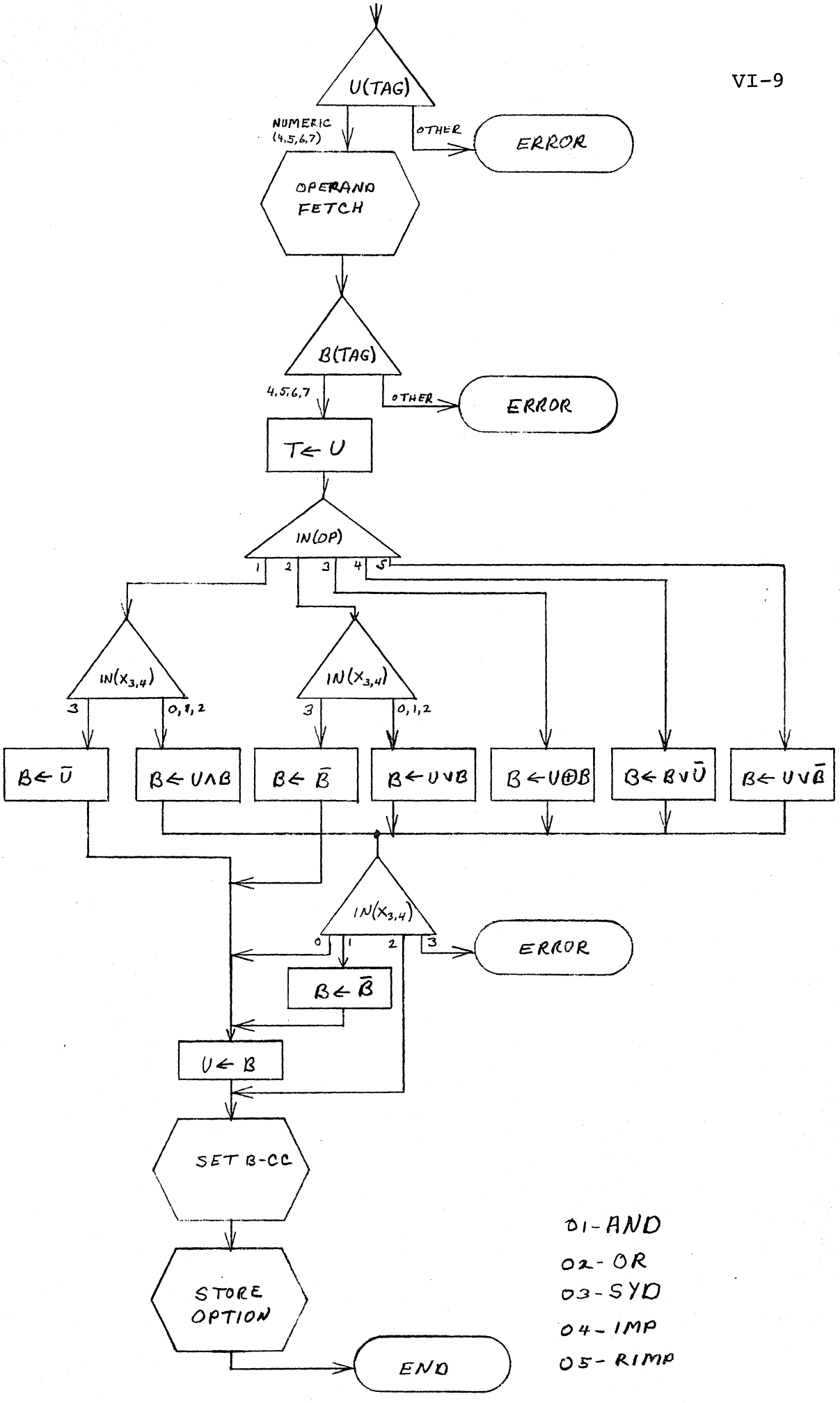
SET A-CC - This routine sets the condition code register after numeric operations.

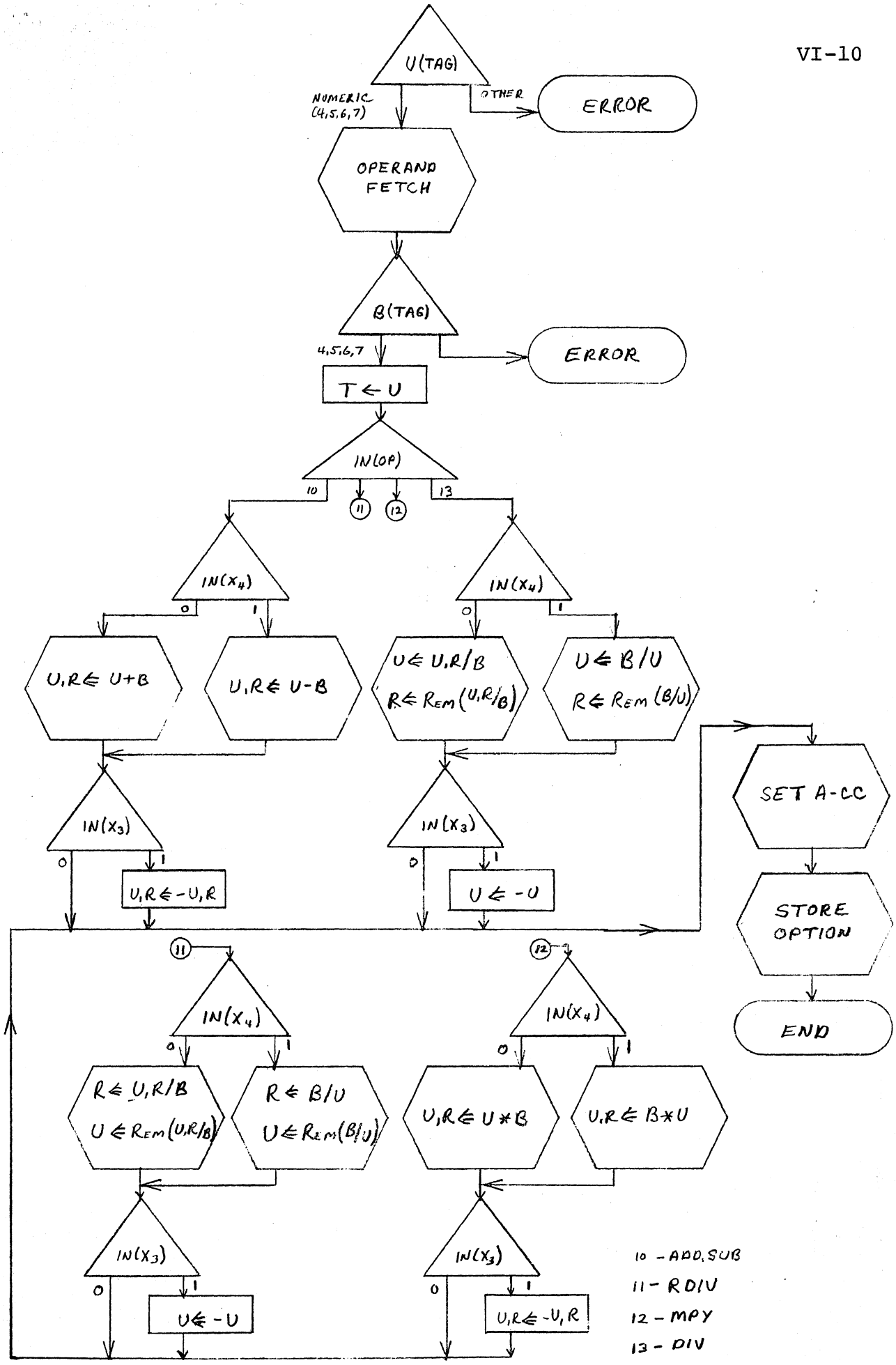
SET B-CC - This routine sets the condition code register after Boolean operations.

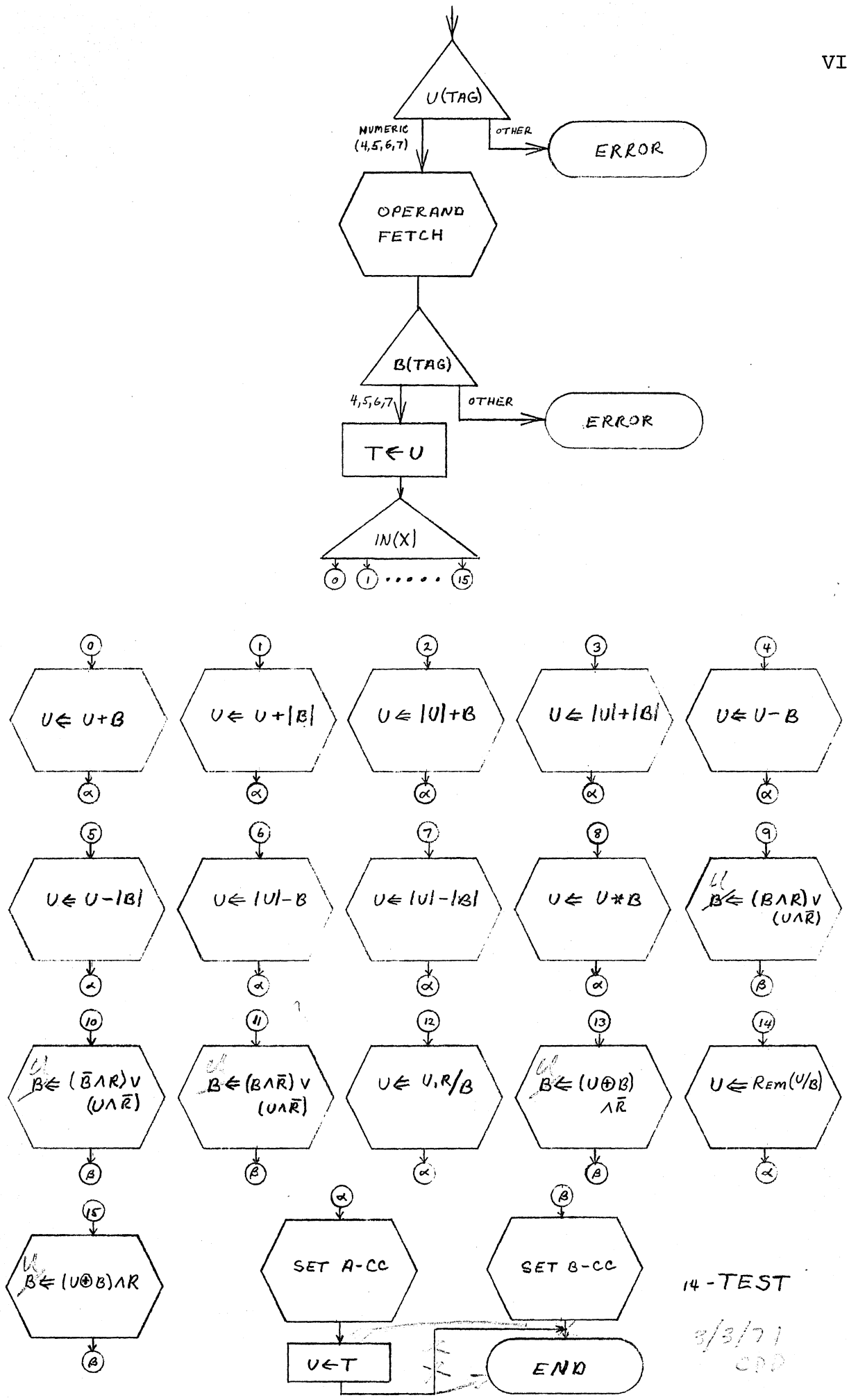
STORE OPTION - This routine is used to perform the store option operation after certain arithmetic and Boolean operations.



00- { XTR(R) / XTR(R̄) / MCP(R) / MCP(R̄) }

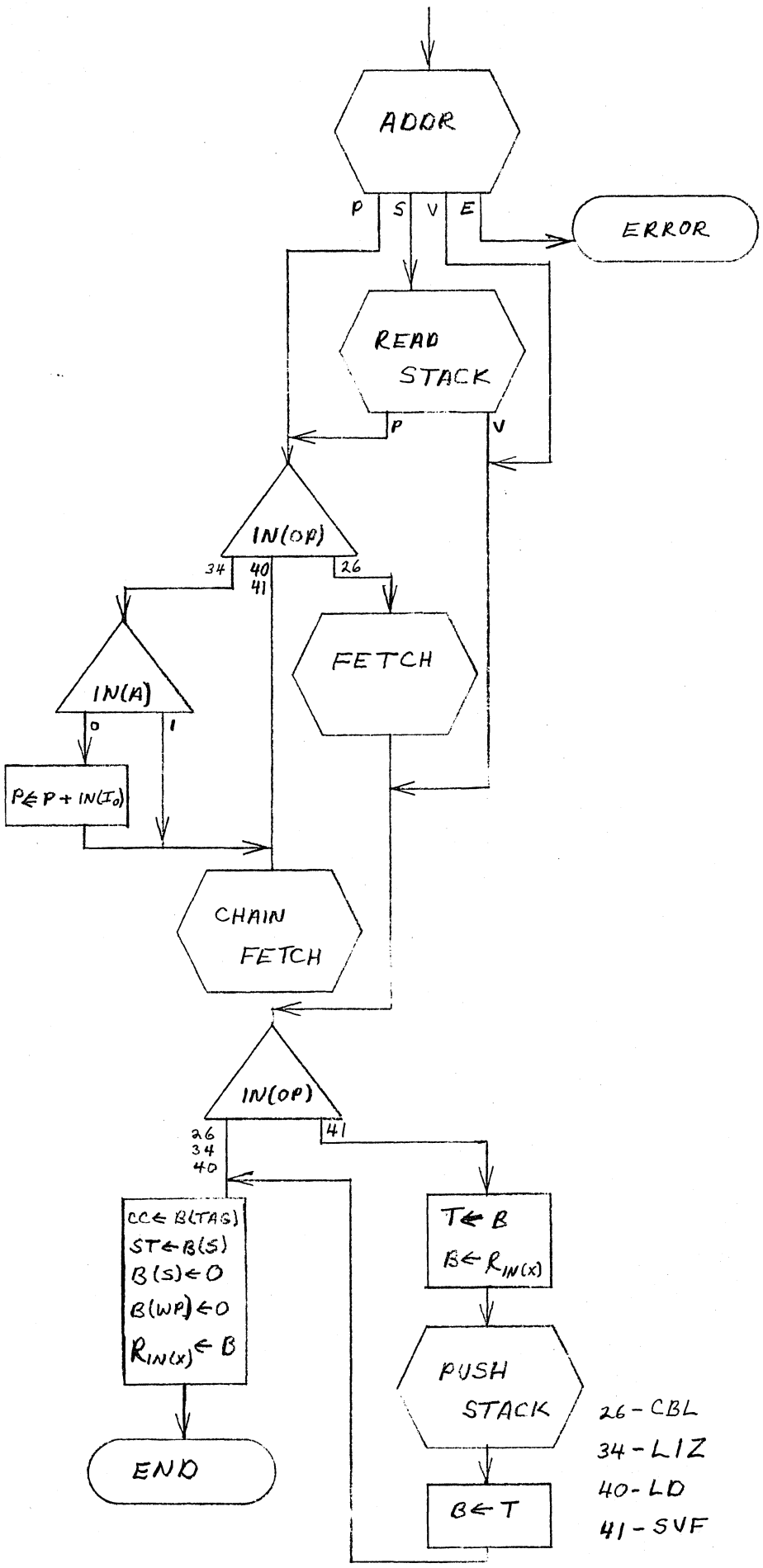

U(TAG)
NUMERIC (4,5,6,7)
OTHER
ERROR
OPERAND FETCH
B(TAG)
4,5,6,7
OTHER
ERROR
T ← U
IN(OP)
1
2
3
4
5
IN(X3,4)
3
0,1,2
IN(X3,4)
3
0,1,2
B ← Ū
B ← U∧B
B ← B̄
B ← U∨B
B ← U⊕B
B ← B∨Ū
B ← U∨B̄
IN(X3,4)
0
1
2
3
ERROR
B ← B̄
U ← B
SET B-CC
STORE OPTION
END
01-AND
02-OR
03-SYD
04-IMP
05-RIMP

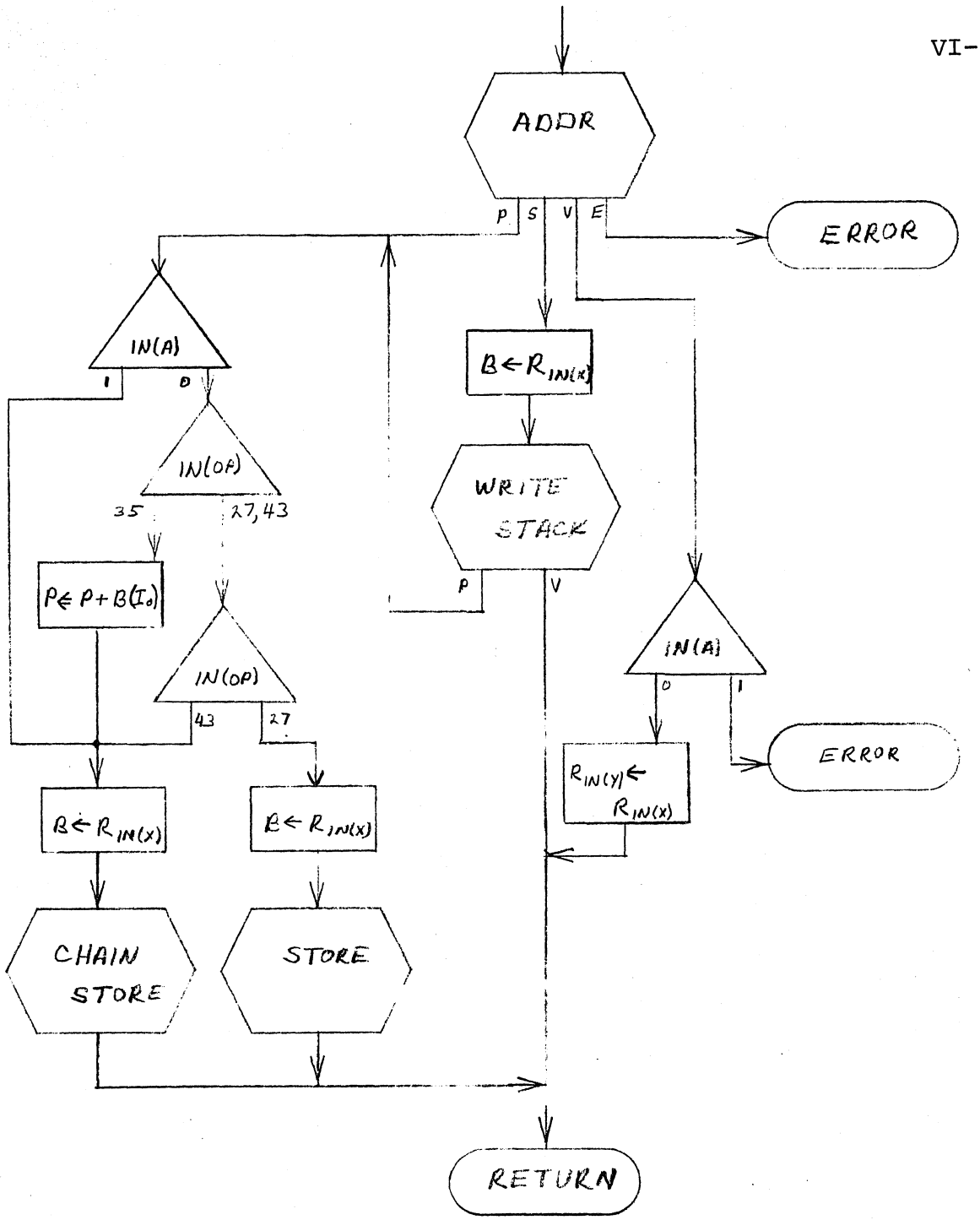

U(TAG)
NUMERIC (4,5,6,7)
OTHER
ERROR
OPERAND FETCH
B(TAG)
4,5,6,7
ERROR
T ← U
IN(OP)
10
11
12
13
IN(X4)
U,R ← U+B
U,R ← U-B
U ← U,R/B
R ← REM(U,R/B)
U ← B/U
R ← REM(B/U)
IN(X3)
U,R ← -U,R
U ← -U
SET A-CC
STORE OPTION
END
R ← U,R/B
U ← REM(U,R/B)
R ← B/U
U ← REM(B/U)
U,R ← U*B
U,R ← B*U
U ← -U
U,R ← -U,R
10 - ADD, SUB
11 - RDIV
12 - MPY
13 - DIV


U(TAG)
NUMERIC (4,5,6,7)
OTHER
ERROR
OPERAND FETCH
B(TAG)
4,5,6,7
OTHER
ERROR
T ← U
IN(X)
0
1
15
U ← U+B
U ← U+|B|
U ← |U|+B
U ← |U|+|B|
U ← U−B
U ← U−|B|
U ← |U|−B
U ← |U|−|B|
U ← U*B
B ← (B∧R)∨(U∧R̄)
B ← (B̄∧R)∨(U∧R̄)
B ← (B∧R̄)∨(U∧R̄)
U ← U,R/B
B ← (U⊕B)∧R̄
U ← Rem(U/B)
B ← (U⊕B)∧R
SET A-CC
SET B-CC
U ← T
END

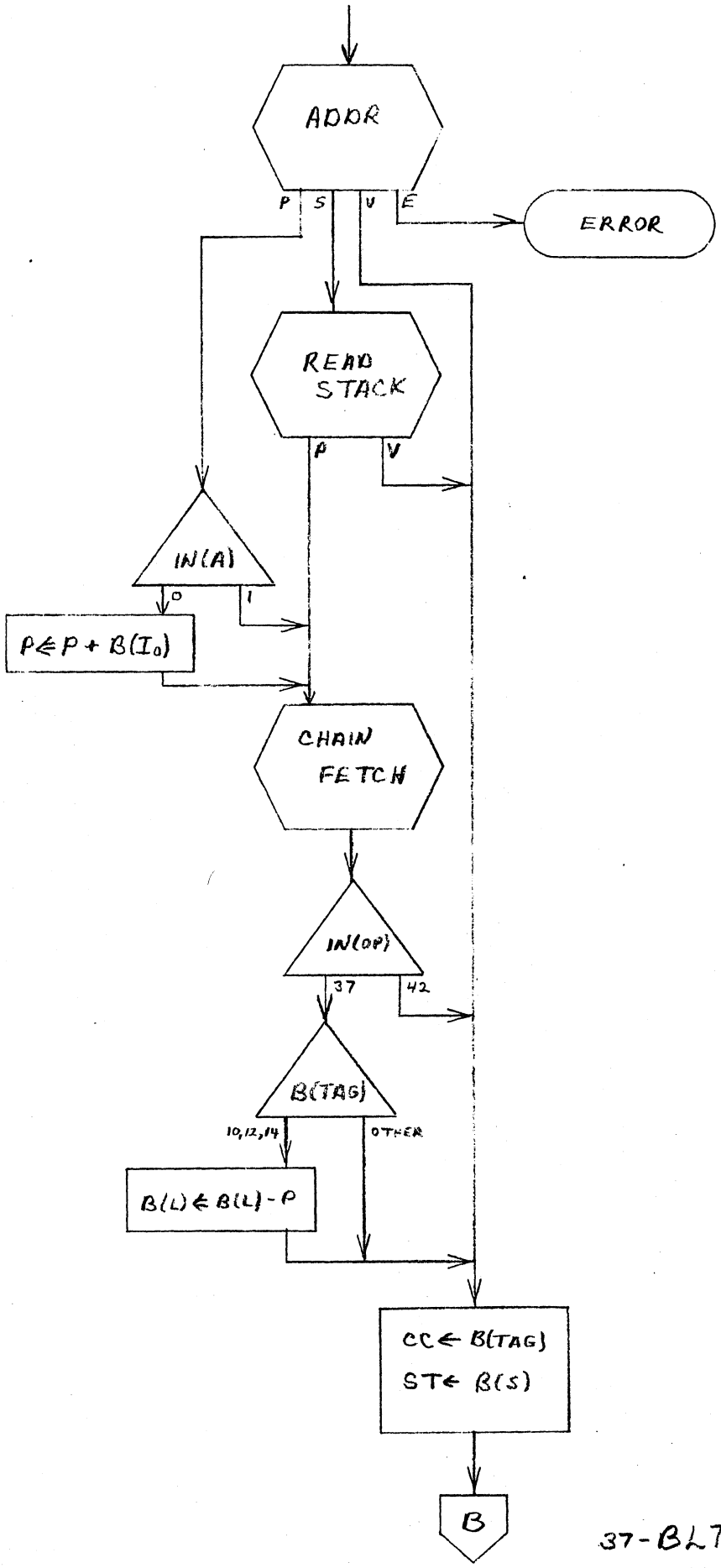
14-TEST

3/3/71

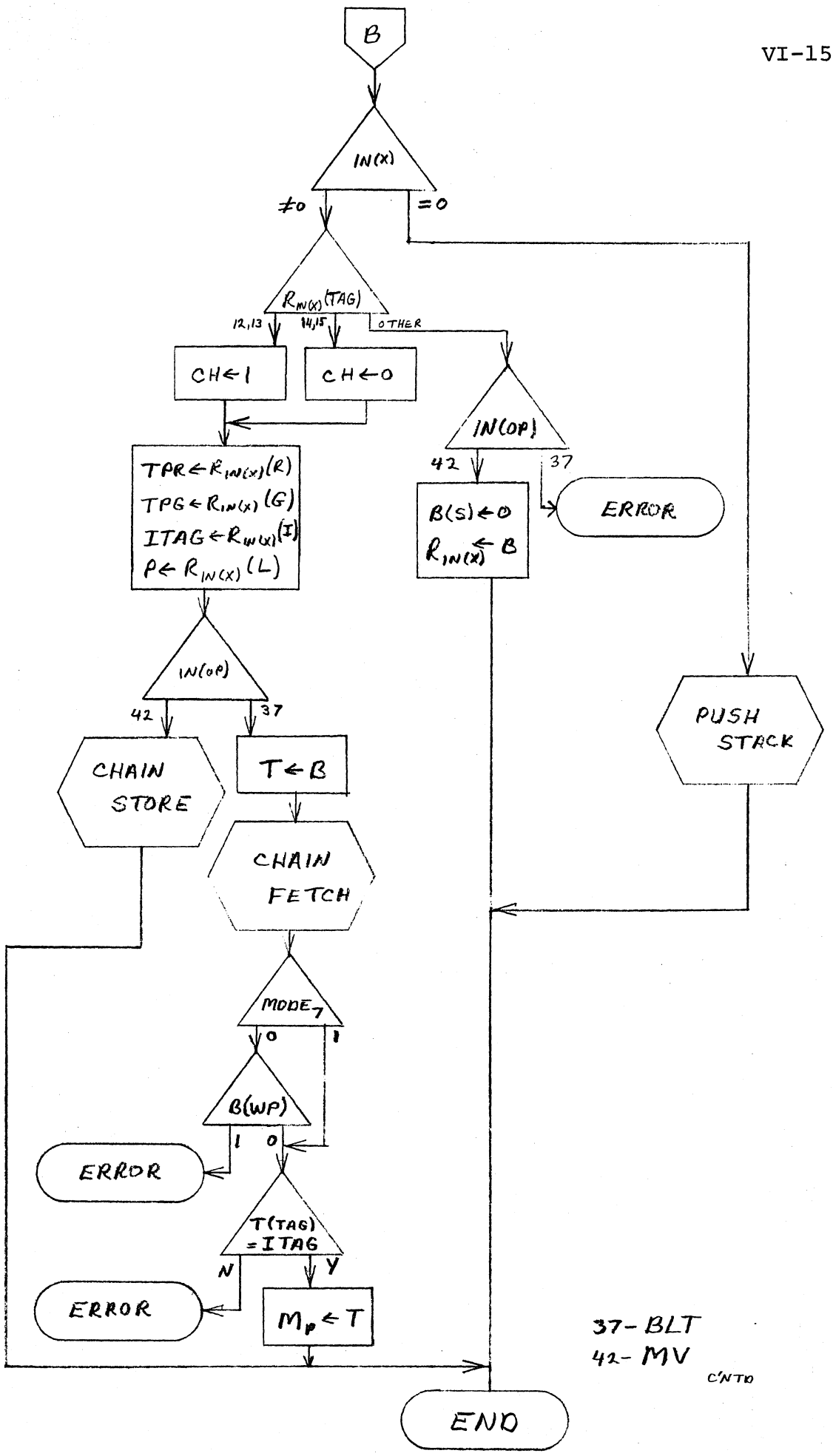

ADDR
P
S
V
E
ERROR
READ STACK
P
V
IN(OP)
34
40
41
26
IN(A)
0
1
FETCH
P ← P + IN(I0)
CHAIN FETCH
IN(OP)
26
34
40
41
CC ← B(TAG)
ST ← B(S)
B(S) ← 0
B(WP) ← 0
RIN(X) ← B
T ← B
B ← RIN(X)
PUSH STACK
B ← T
END
26 - CBL
34 - LIZ
40 - LD
41 - SVF

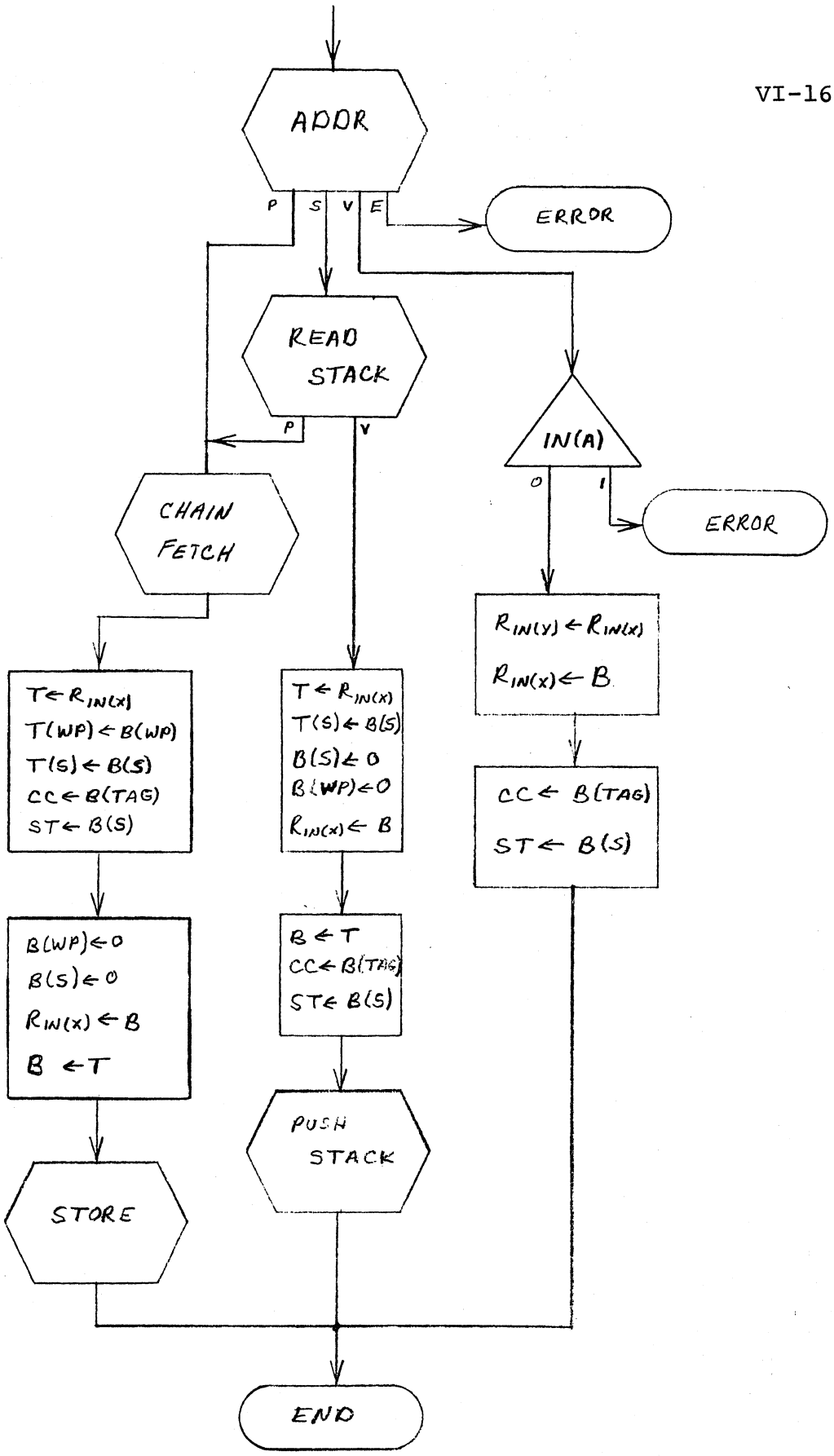


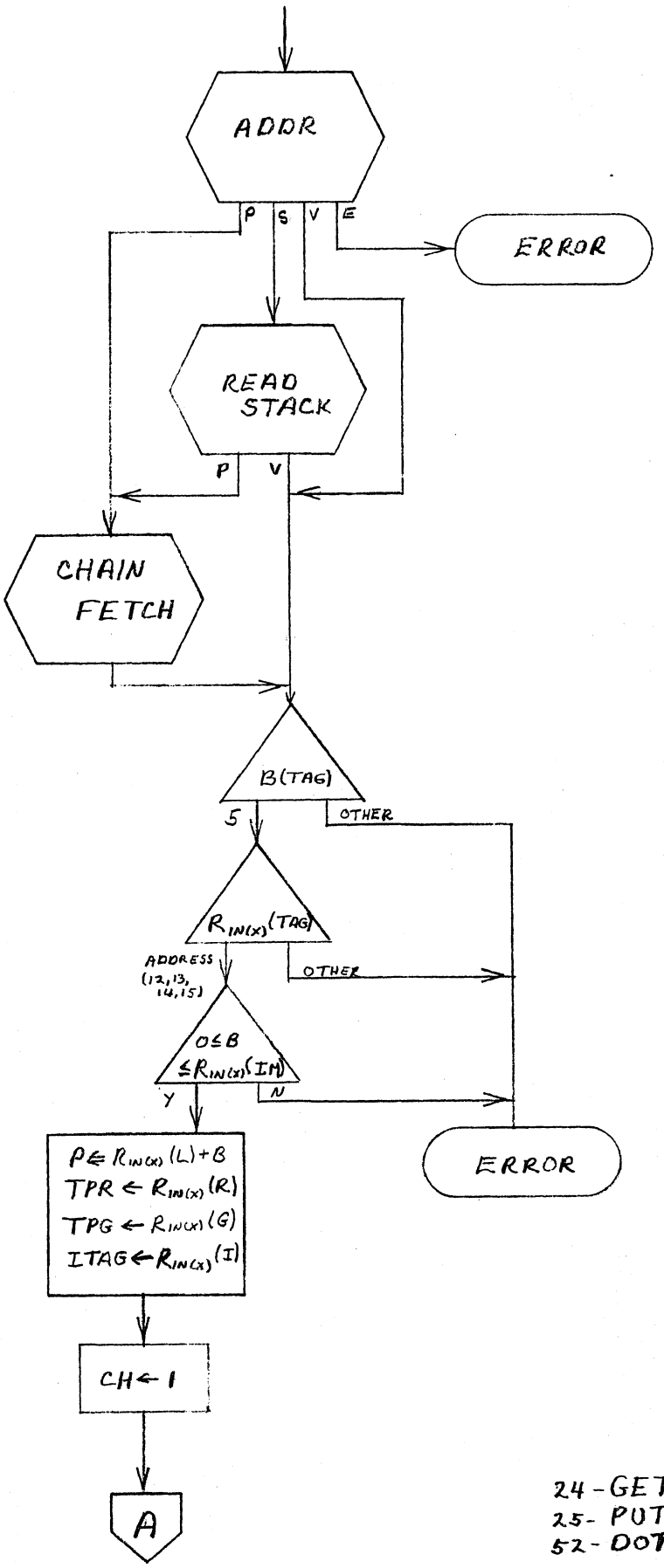
27 - CBS
35 - SIZ
43 - STO


ADDR
P
S
V
E
ERROR
READ STACK
P
V
IN(A)
0
1
P ← P + B(I0)
CHAIN FETCH
IN(OP)
37
42
B(TAG)
10,12,14
OTHER
B(L) ← B(L) - P
CC ← B(TAG)
ST ← B(S)
B

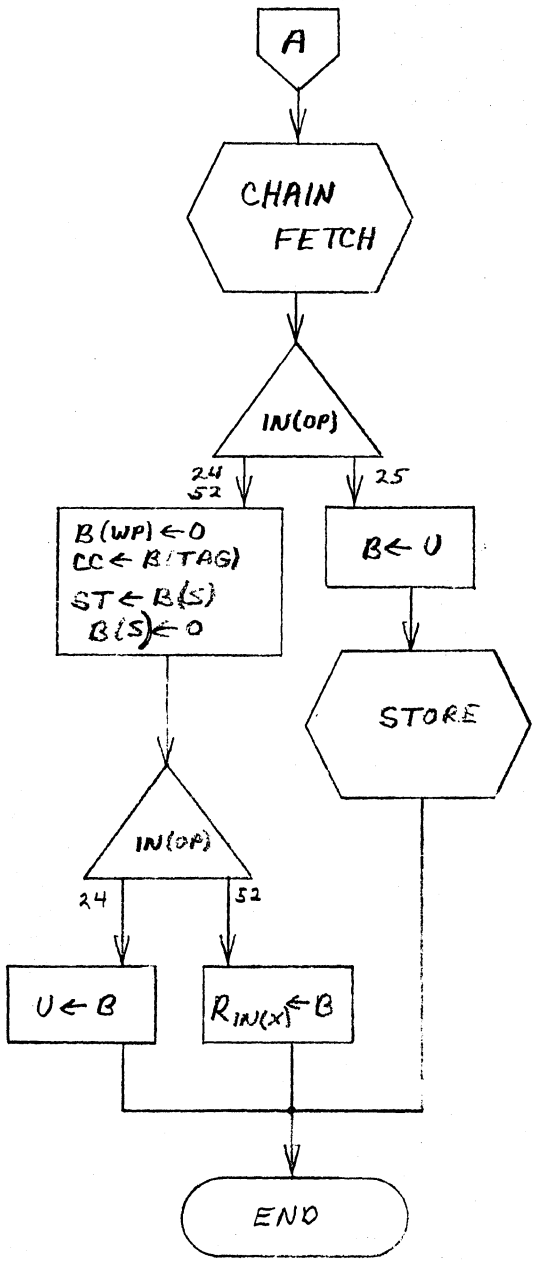
37 - BLT
42 - MV


B
IN(X)
≠0
=0
R IN(X) (TAG)
12,13
14,15
OTHER
CH←1
CH←0
IN(OP)
42
37
ERROR
TPR←R IN(X) (R)
TPG←R IN(X) (G)
ITAG←R IN(X) (I)
P←R IN(X) (L)
B(S)←0
R IN(X)←B
IN(OP)
42
37
CHAIN STORE
T←B
CHAIN FETCH
PUSH STACK
MODE 7
0
1
B(WP)
1
0
ERROR
T(TAG) = ITAG
N
Y
ERROR
Mp←T
37- BLT
42- MV
CNTD
END


ADDR
P
S
V
E
ERROR
READ STACK
P
V
IN(A)
0
1
ERROR
CHAIN FETCH
T ← R IN(x)
T(WP) ← B(WP)
T(S) ← B(S)
CC ← B(TAG)
ST ← B(S)
T ← R IN(x)
T(S) ← B(S)
B(S) ← 0
B(WP) ← 0
R IN(x) ← B
R IN(y) ← R IN(x)
R IN(x) ← B
CC ← B(TAG)
ST ← B(S)
B(WP) ← 0
B(S) ← 0
R IN(x) ← B
B ← T
B ← T
CC ← B(TAG)
ST ← B(S)
PUSH STACK
STORE
END

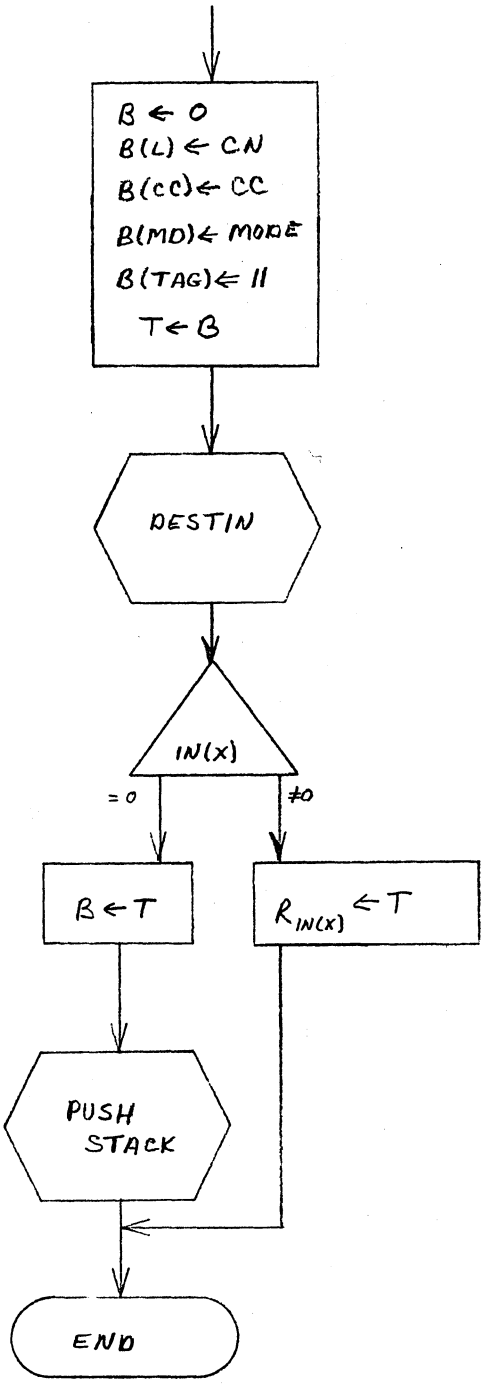


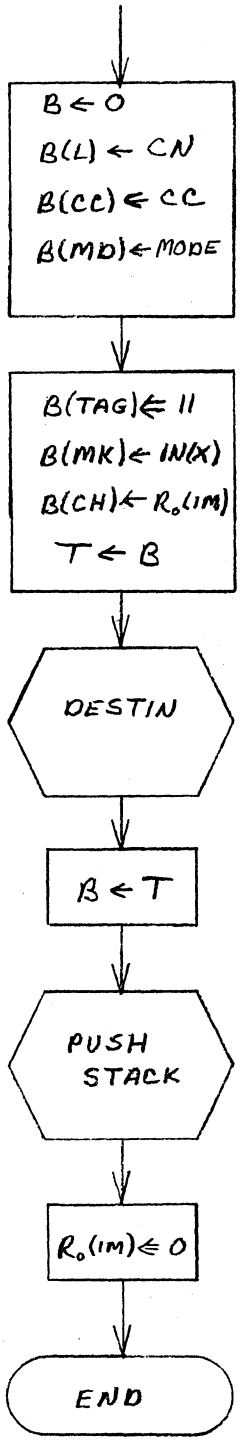
24 - GET
25 - PUT
52 - DOT

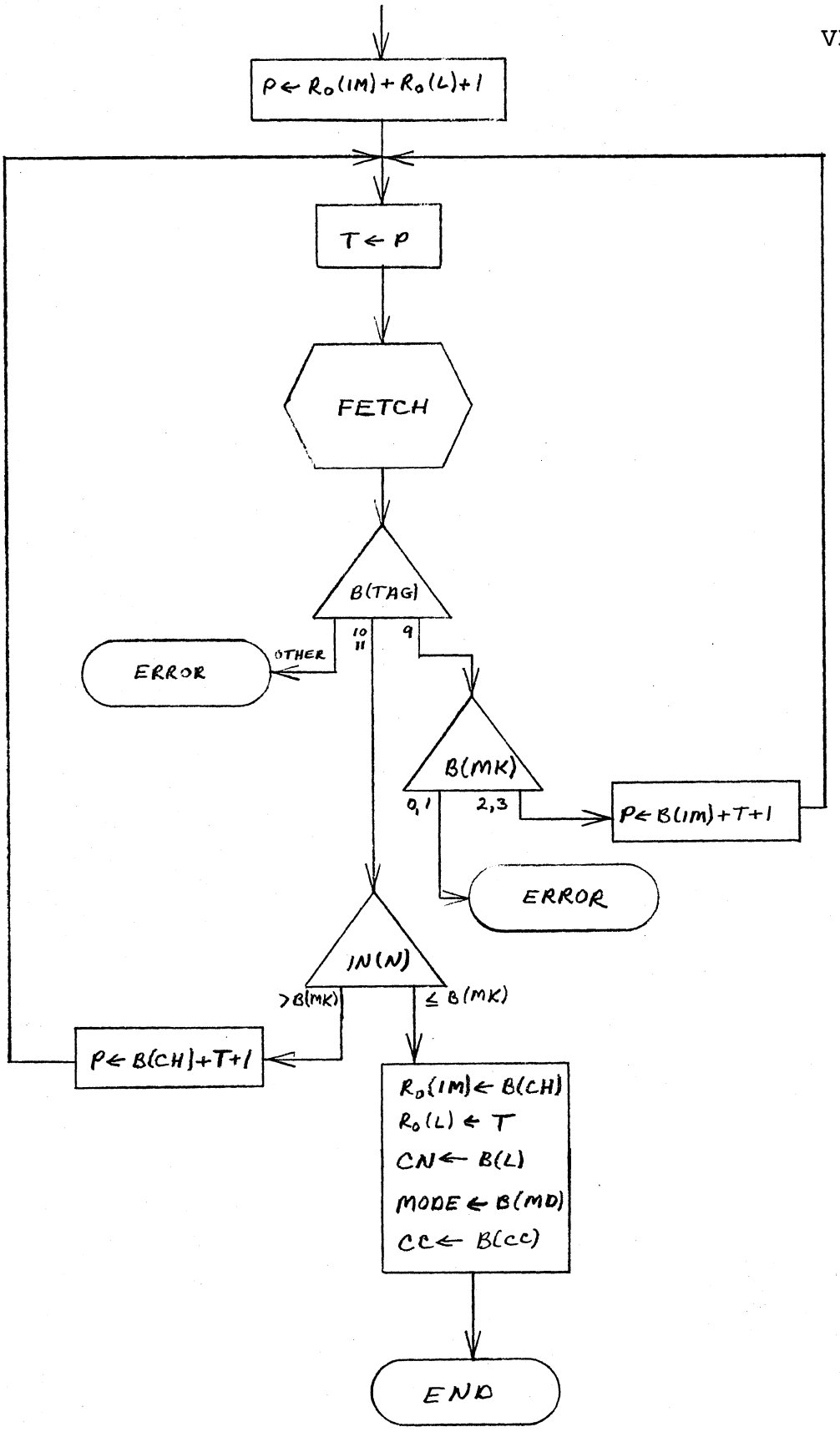


24 - GET
25 - PUT
52 - DOT

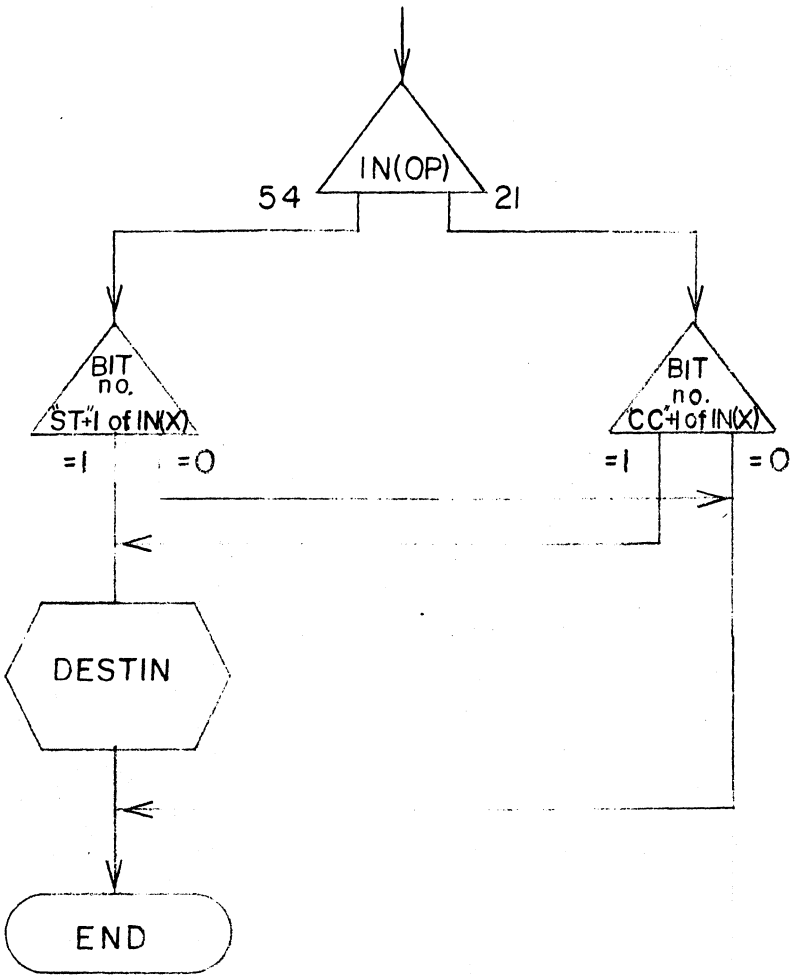
CNTD


B ← 0
B(L) ← CN
B(CC) ← CC
B(MD) ← MODE
B(TAG) ← 11
T ← B
DESTIN
IN(X)
=0
≠0
B ← T
R IN(X) ← T
PUSH STACK
END

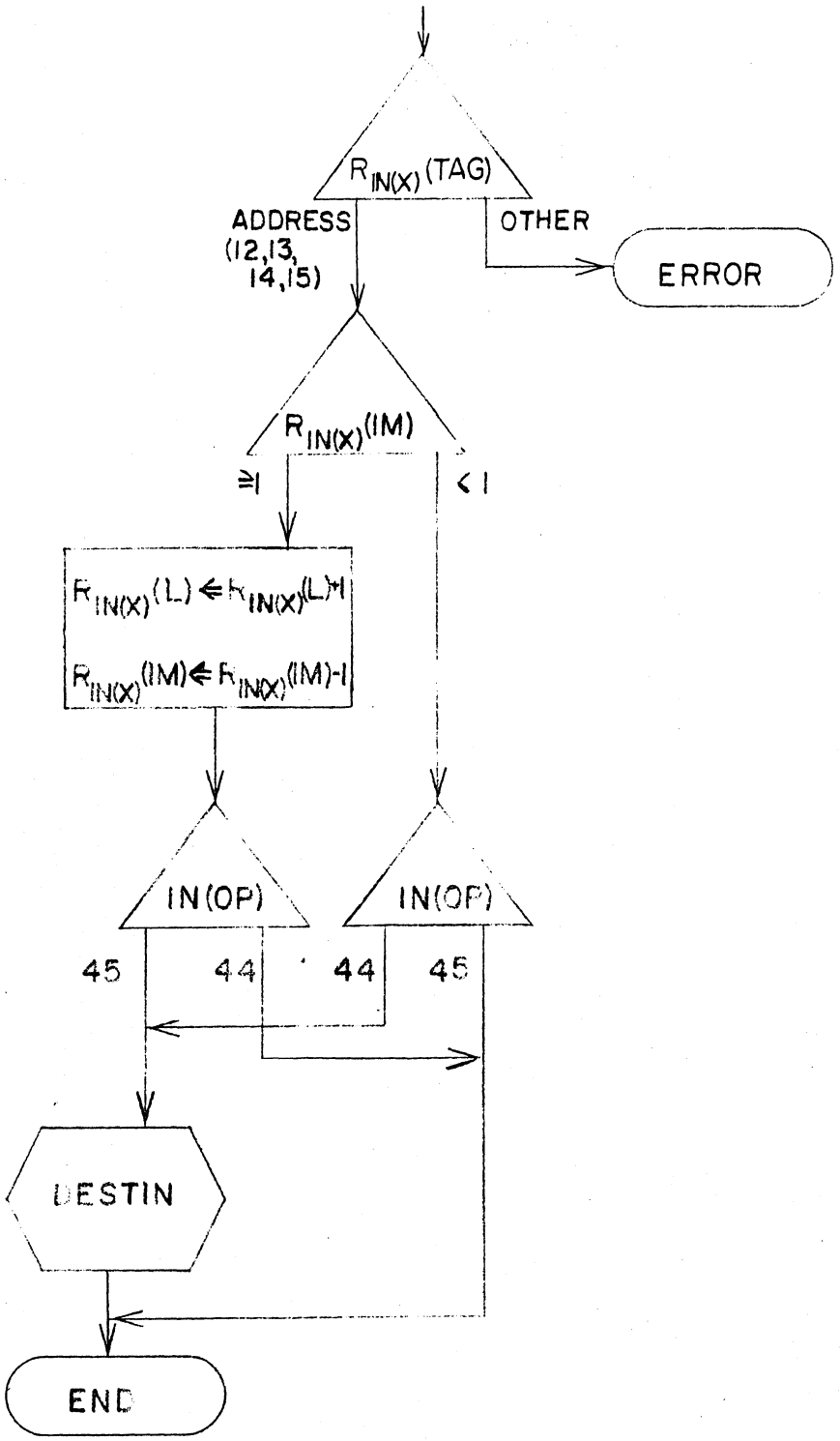

B ← 0
B(L) ← CN
B(CC) ← CC
B(MD) ← MODE
B(TAG) ← 11
B(MK) ← IN(X)
B(CH) ← R0(IM)
T ← B
DESTIN
B ← T
PUSH STACK
R0(IM) ← 0
END


P ← R0(1M) + R0(L) + 1
T ← P
FETCH
B(TAG)
10
11
OTHER
ERROR
9
B(MK)
0,1
2,3
P ← B(1M) + T + 1
ERROR
1N(N)
> B(MK)
≤ B(MK)
P ← B(CH) + T + 1
R0(1M) ← B(CH)
R0(L) ← T
CN ← B(L)
MODE ← B(MD)
CC ← B(CC)
END

22-RET

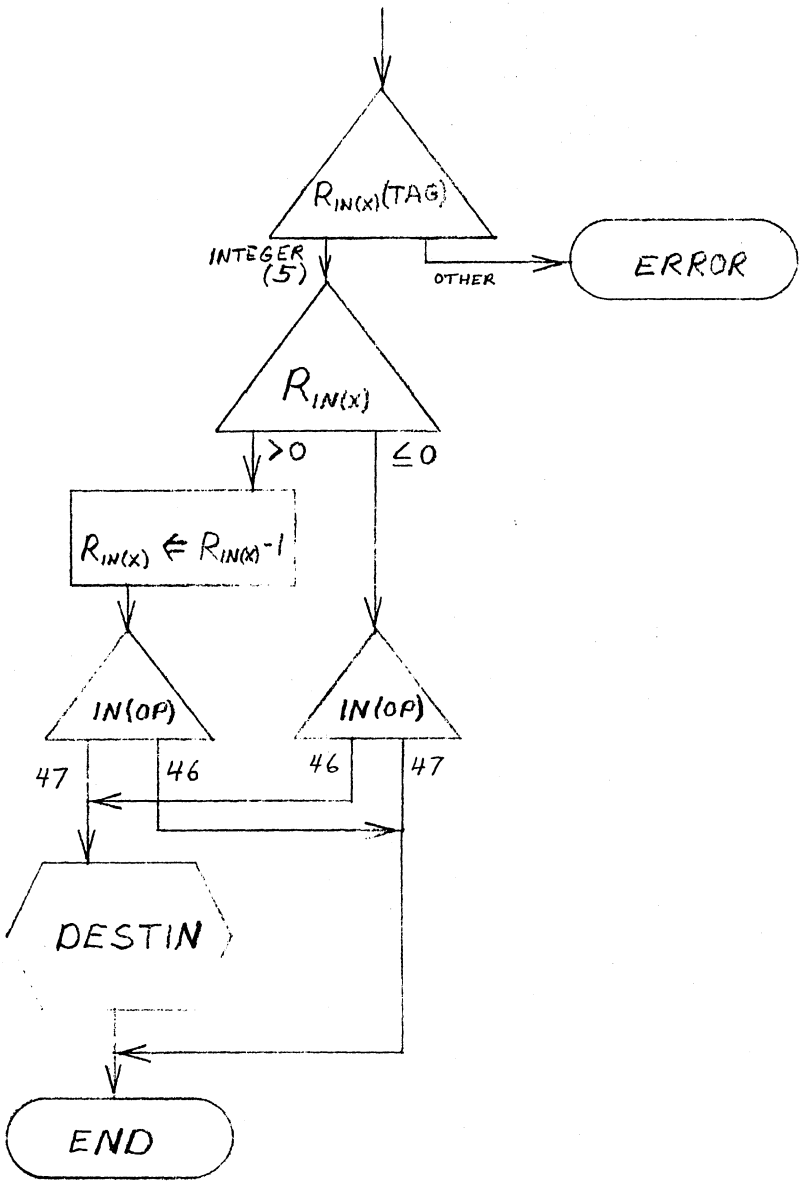


21-JCC
54-JPT

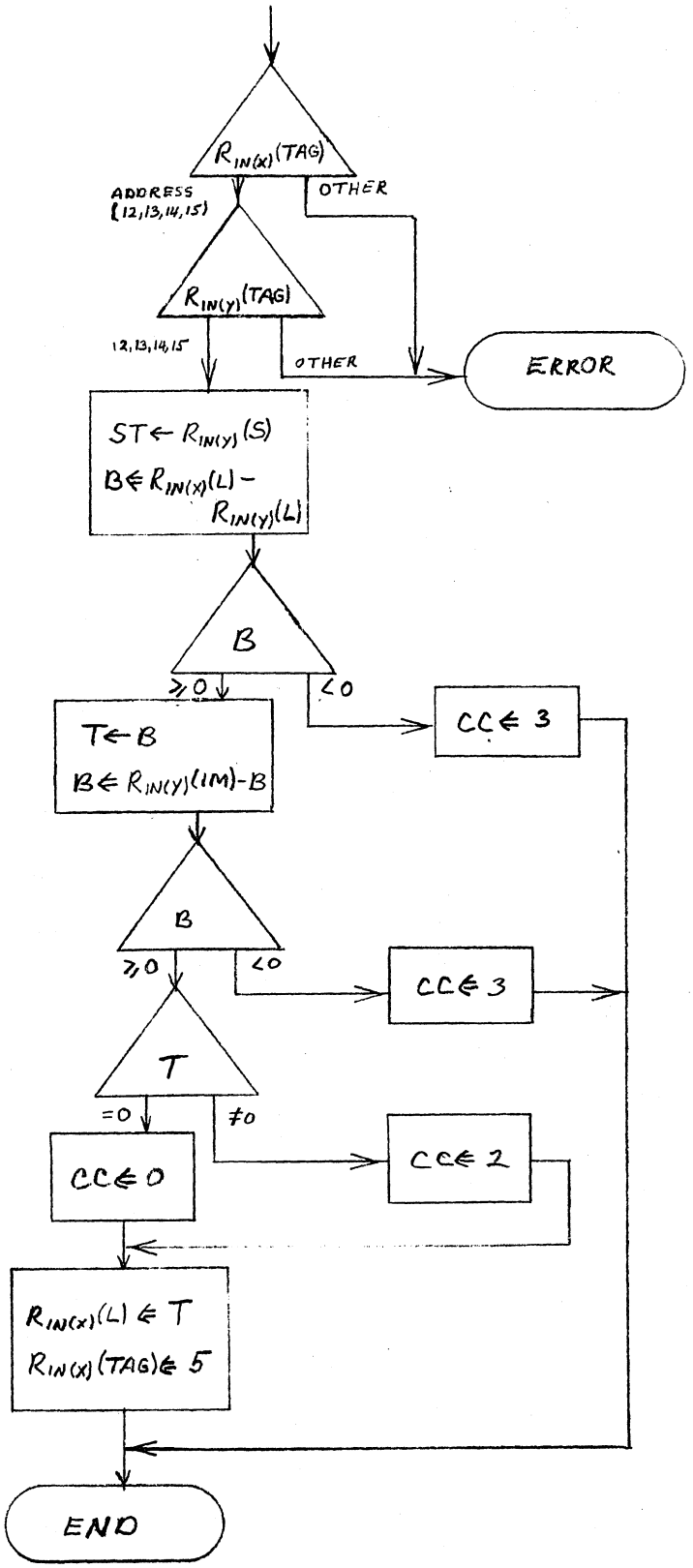

$R_{IN(X)}$ (TAG)
ADDRESS (12,13, 14,15)
OTHER
ERROR
$R_{IN(X)}$ (IM)
≥1
<1
$R_{IN(X)}$ (L) ← $R_{IN(X)}$ (L)+1
$R_{IN(X)}$ (IM) ← $R_{IN(X)}$ (IM)-1
IN (OP)
IN (OP)
45
44
44
45
DESTIN
END

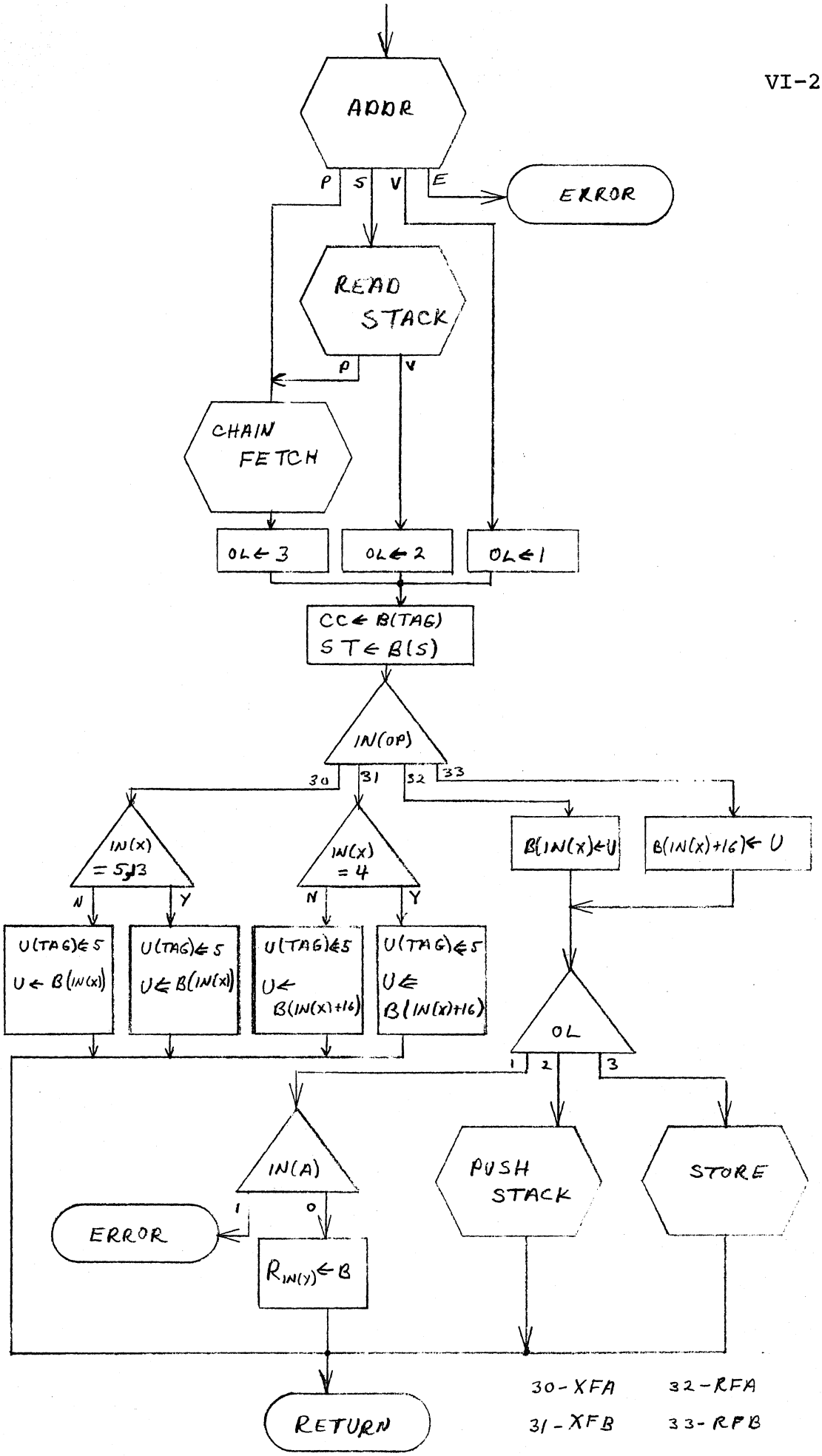
44 - JL

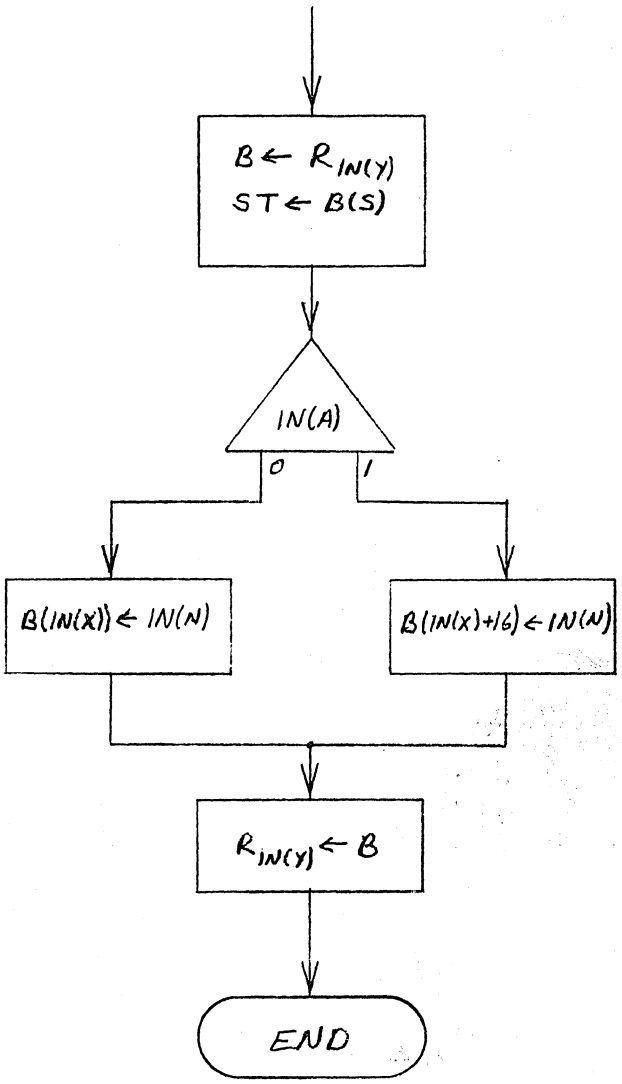
45 - JNL


$R_{IN(X)}$(TAG)
INTEGER (5)
OTHER
ERROR
$R_{IN(X)}$
>0
≤0
$R_{IN(X)} \Leftarrow R_{IN(X)}-1$
IN(OP)
IN(OP)
47
46
46
47
DESTIN
END

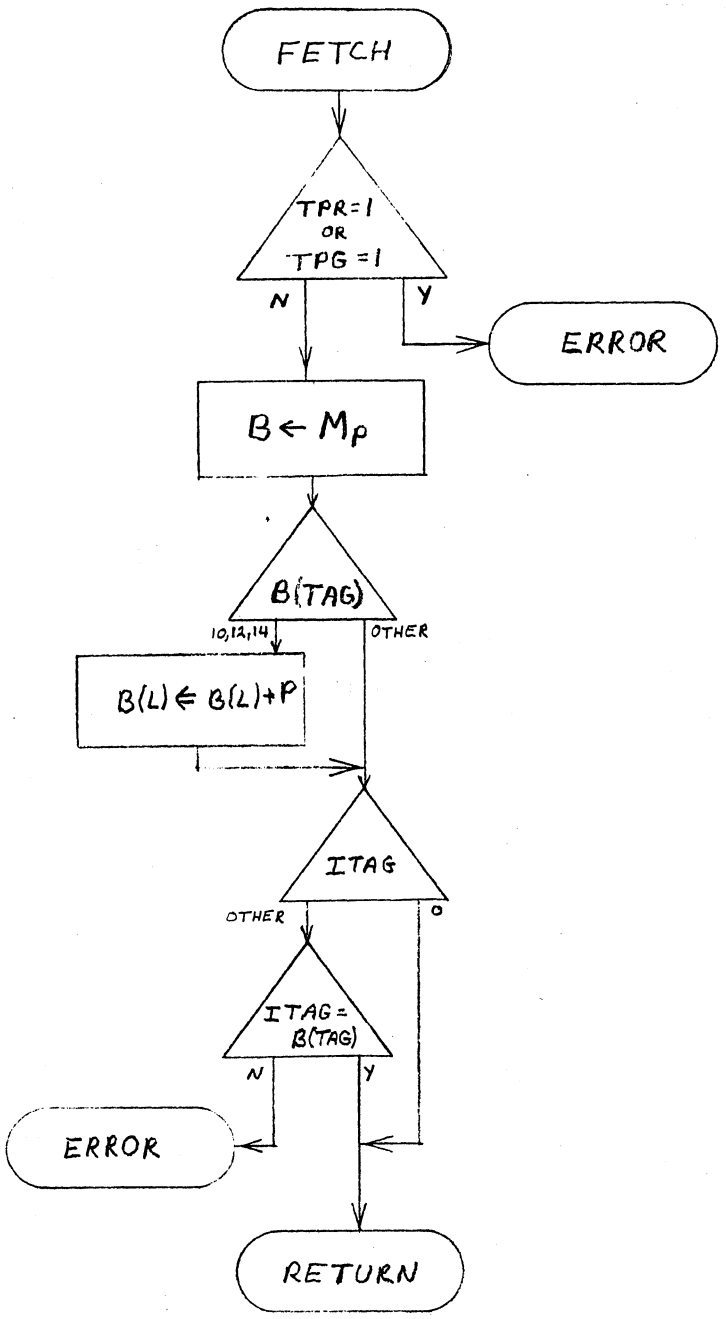
46 - JLT
47 - JGE

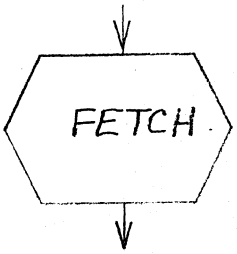

$R_{IN(X)}$(TAG)
ADDRESS (12,13,14,15)
OTHER
$R_{IN(Y)}$(TAG)
12,13,14,15
OTHER
ERROR
ST ← $R_{IN(Y)}$(S)
B ∈ $R_{IN(X)}$(L) − $R_{IN(Y)}$(L)
B
≥0
<0
CC ∈ 3
T ← B
B ∈ $R_{IN(Y)}$(1M)-B
B
≥0
<0
CC ∈ 3
T
=0
≠0
CC ∈ 0
CC ∈ 2
$R_{IN(X)}$(L) ∈ T
$R_{IN(X)}$(TAG) ∈ 5
END

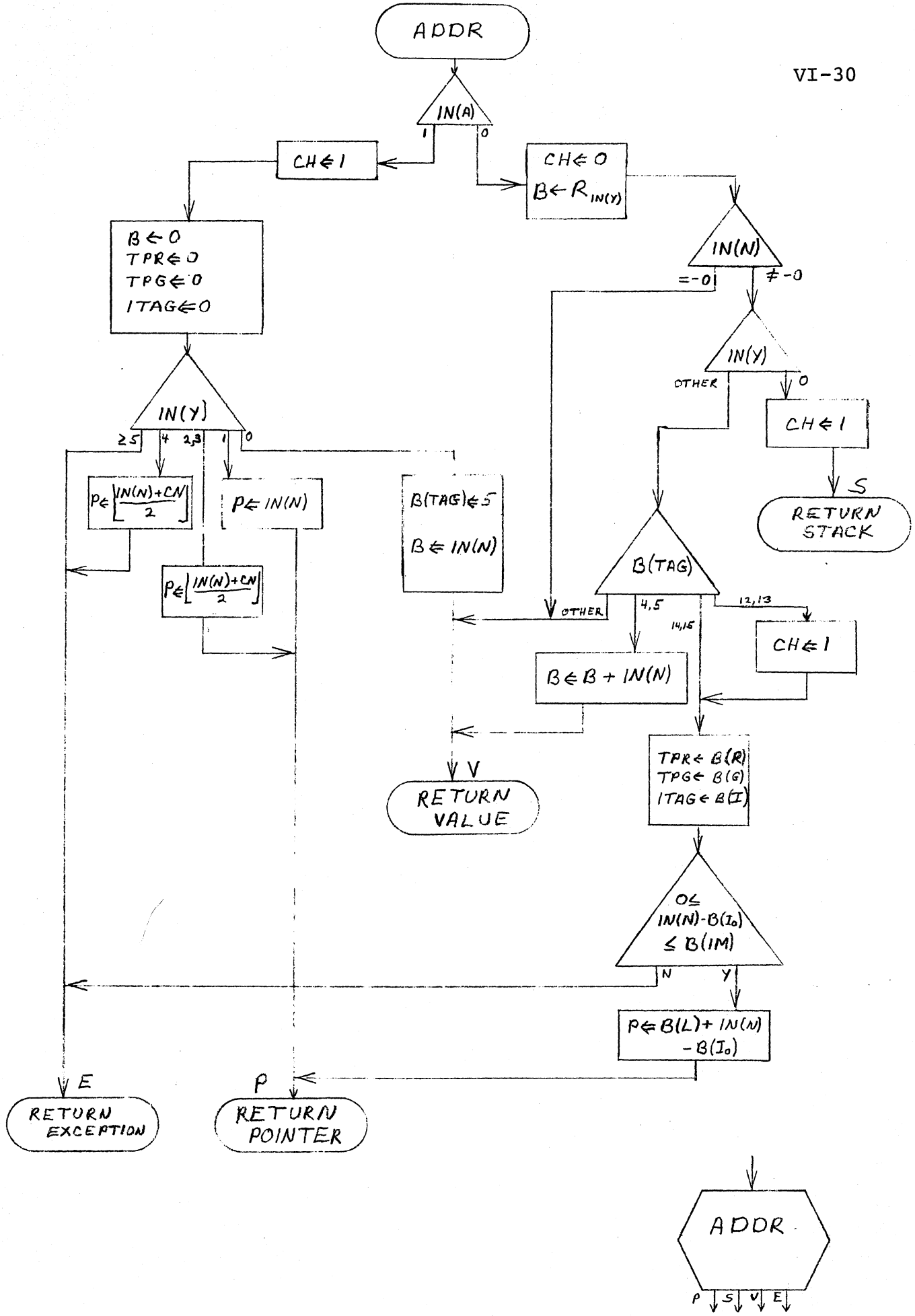

ADDR
P
S
V
E
ERROR
READ STACK
P
V
CHAIN FETCH
OL ← 3
OL ← 2
OL ← 1
CC ← B(TAG)
ST ← B(S)
IN(OP)
30
31
32
33
IN(X) = 5 or 3
N
Y
IN(X) = 4
N
Y
B(IN(X)) ← U
B(IN(X)+16) ← U
U(TAG) ← 5
U ← B(IN(X))
U(TAG) ← 5
U ← B(IN(X))
U(TAG) ← 5
U ← B(IN(X)+16)
U(TAG) ← 5
U ← B(IN(X)+16)
OL
1
2
3
IN(A)
1
0
ERROR
R IN(Y) ← B
PUSH STACK
STORE
RETURN
30 - XFA
31 - XFB
32 - RFA
33 - RFB

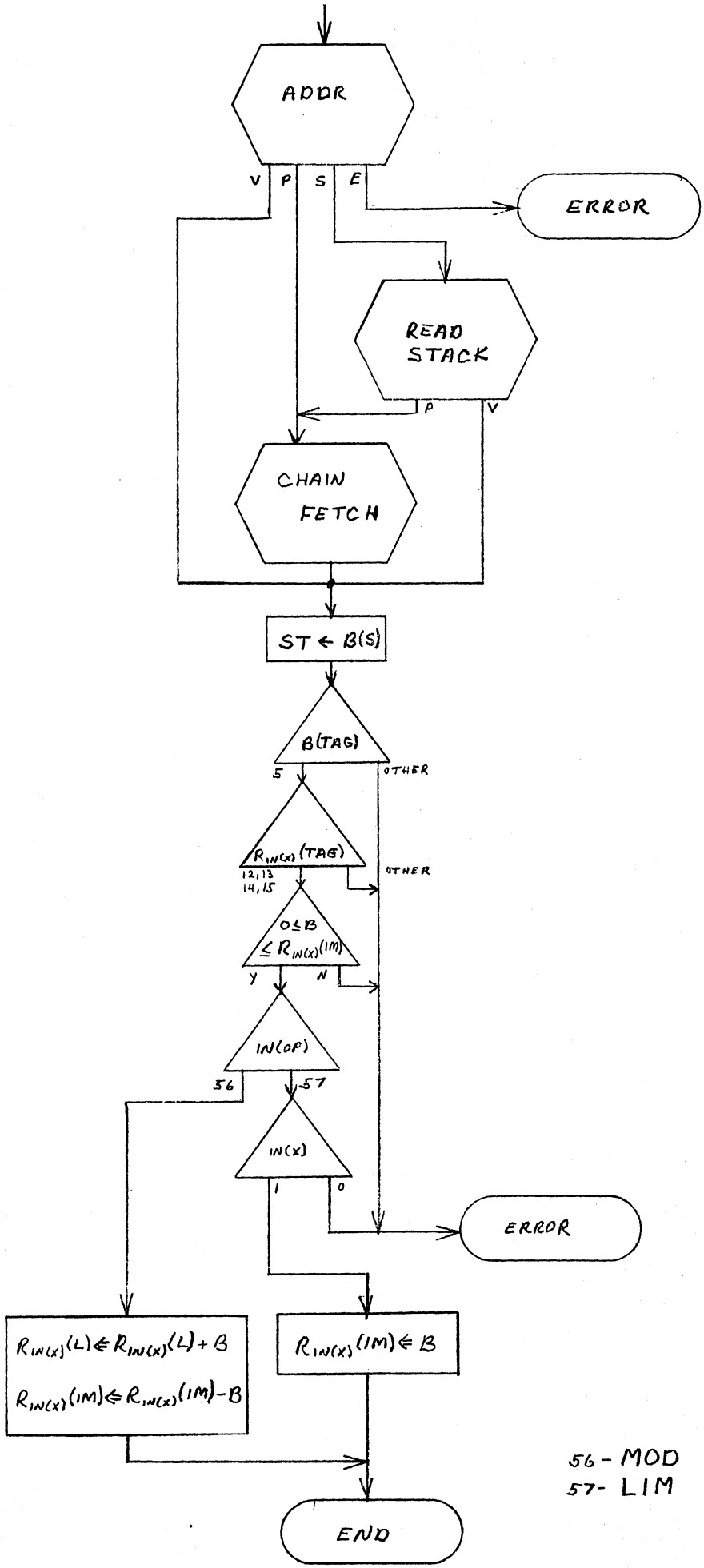


55-TAG

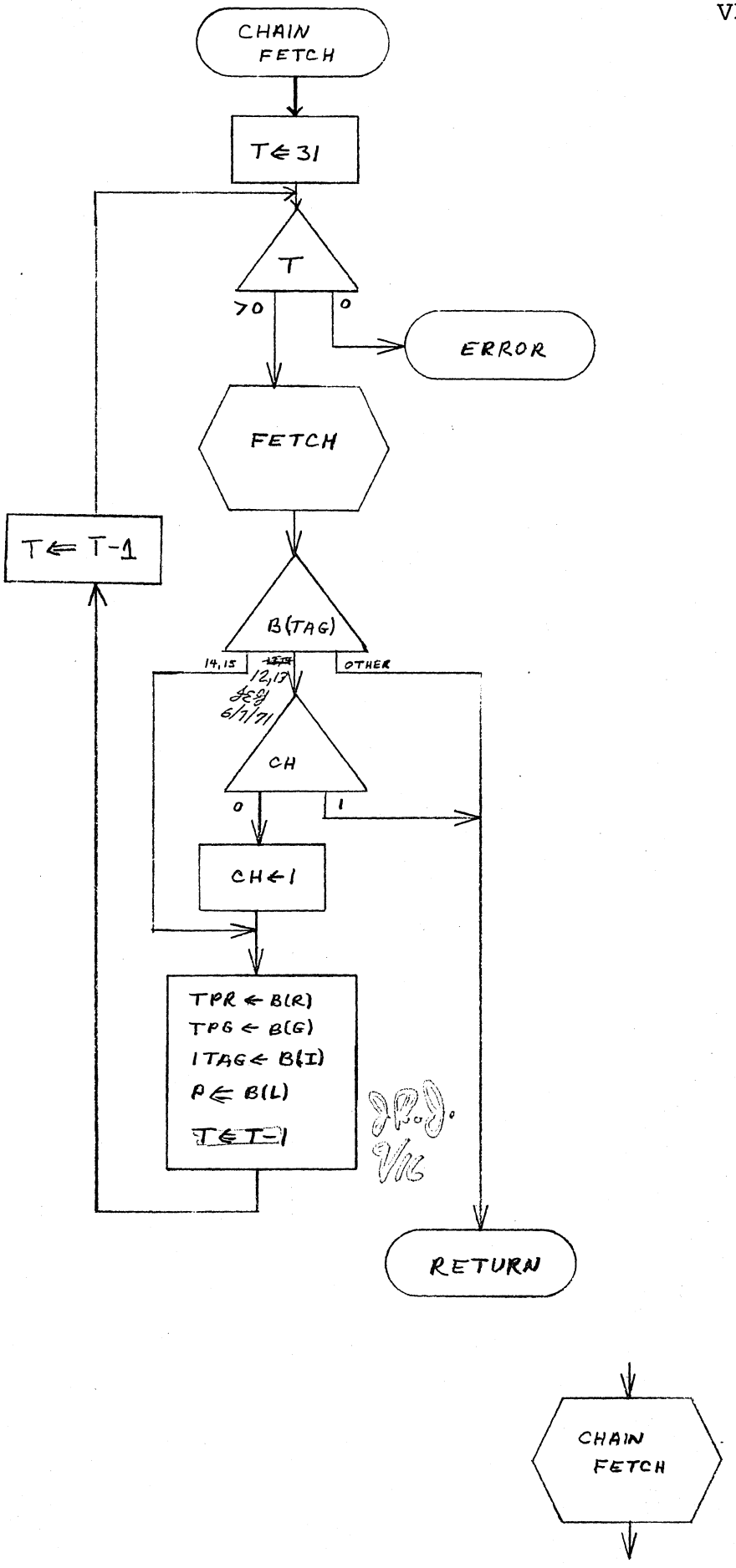

FETCH
TPR=1
OR
TPG=1
N
Y
ERROR
B ← Mp
B(TAG)
10,12,14
OTHER
B(L) ← B(L)+P
ITAG
OTHER
0
ITAG = B(TAG)
N
Y
ERROR
RETURN


FETCH

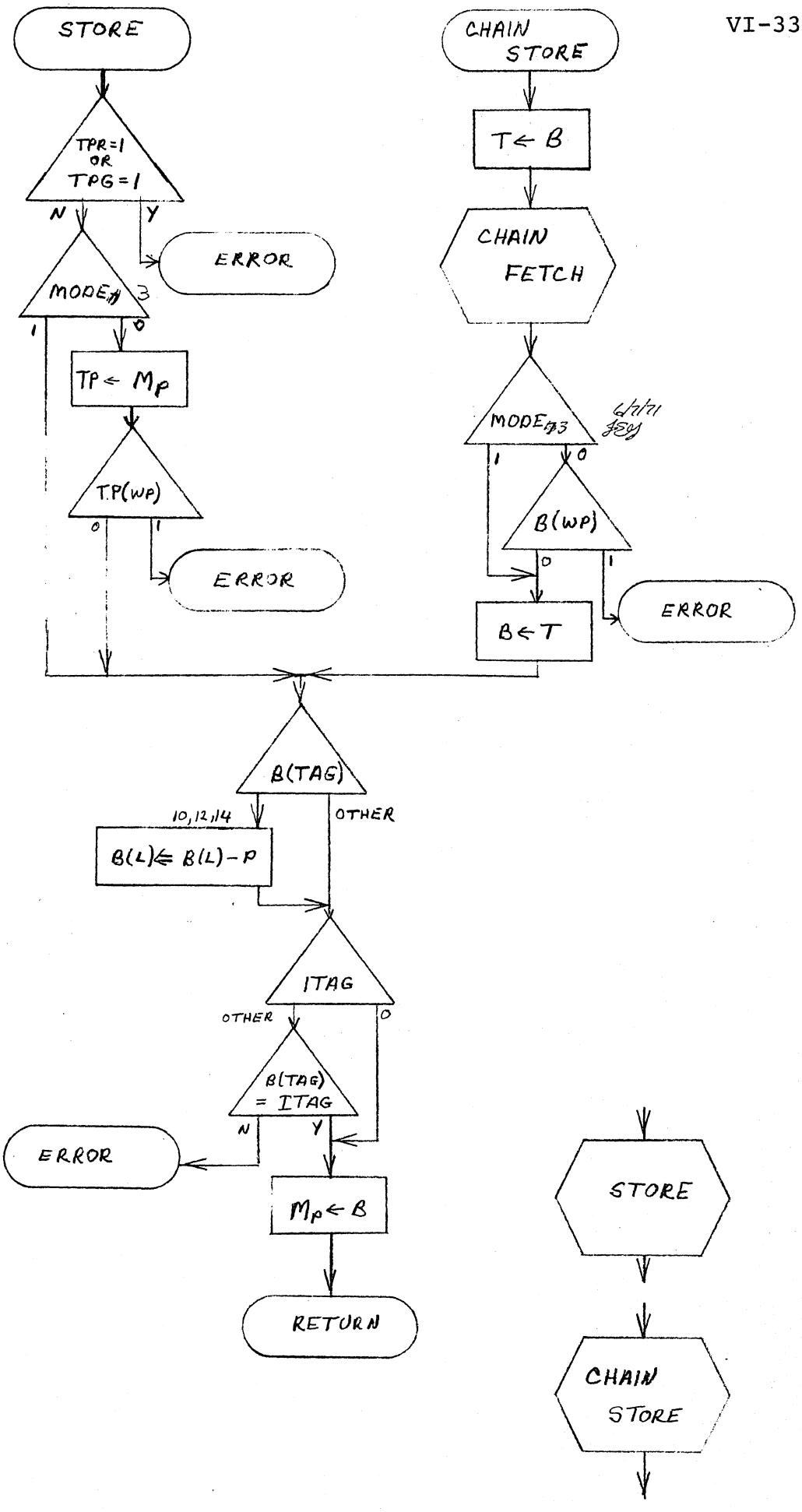

ADDR
IN(A)
1
0
CH ← 1
CH ← 0
B ← R IN(Y)
B ← 0
TPR ← 0
TPG ← 0
ITAG ← 0
IN(N)
=-0
≠-0
IN(Y)
OTHER
0
CH ← 1
S
RETURN STACK
IN(Y)
≥5
4
2,3
1
0
P ← ⌊(IN(N)+CN)/2⌋
P ← IN(N)
B(TAG) ← 5
B ← IN(N)
P ← ⌊(IN(N)+CN)/2⌋
B(TAG)
OTHER
4,5
14,15
12,13
CH ← 1
B ← B + IN(N)
V
RETURN VALUE
TPR ← B(R)
TPG ← B(G)
ITAG ← B(I)
0 ≤ IN(N) - B(I0) ≤ B(IM)
N
Y
P ← B(L) + IN(N) - B(I0)
E
RETURN EXCEPTION
P
RETURN POINTER
ADDR
P
S
V
E

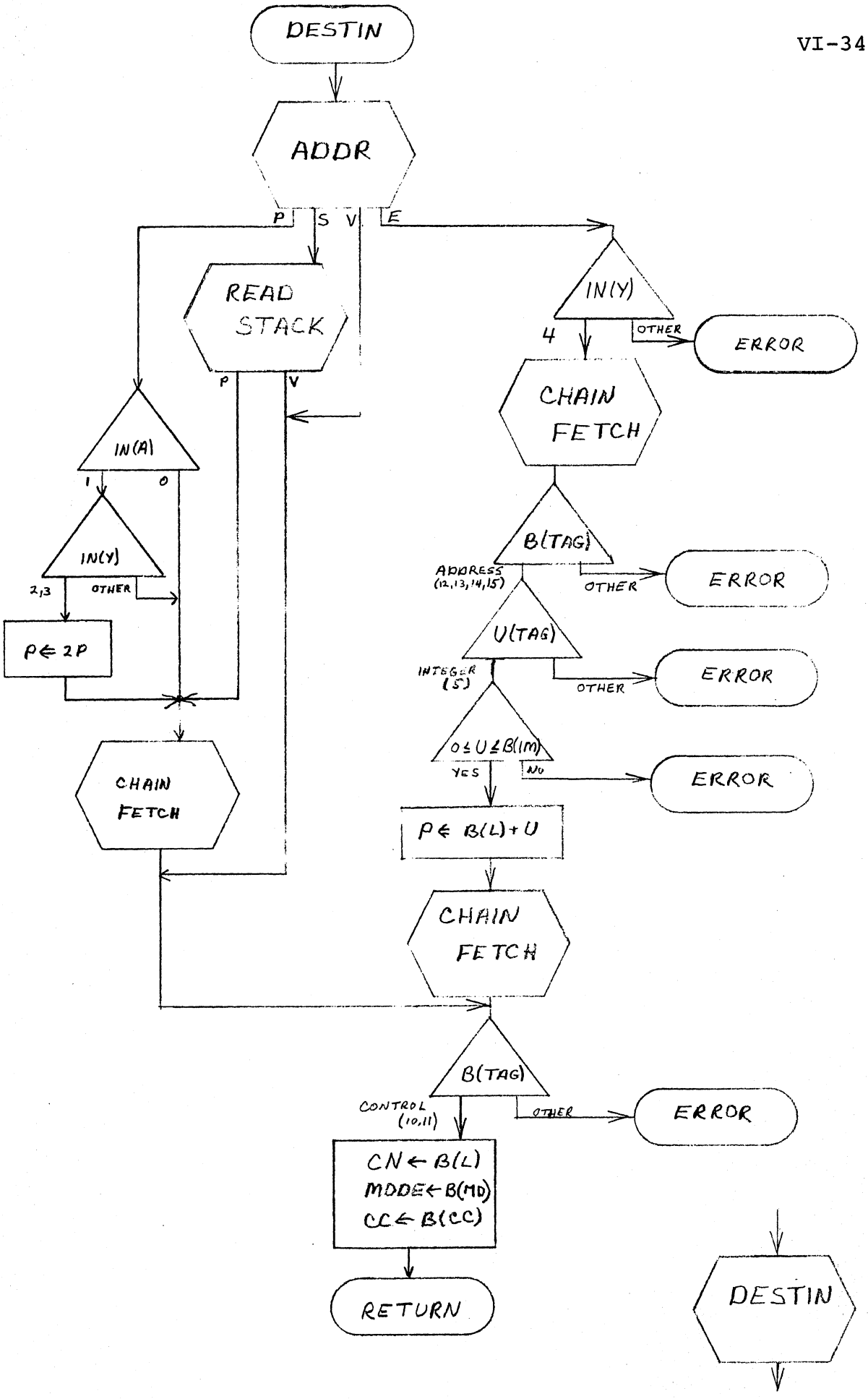

ADDR
V
P
S
E
ERROR
READ STACK
P
V
CHAIN FETCH
ST ← B(S)
B(TAG)
5
OTHER
R IN(X) (TAG)
12,13
14,15
OTHER
0 ≤ B ≤ R IN(X)(IM)
Y
N
IN(OP)
56
57
IN(X)
1
0
ERROR
R IN(X)(L) ← R IN(X)(L) + B
R IN(X)(IM) ← R IN(X)(IM) - B
R IN(X)(IM) ← B
END

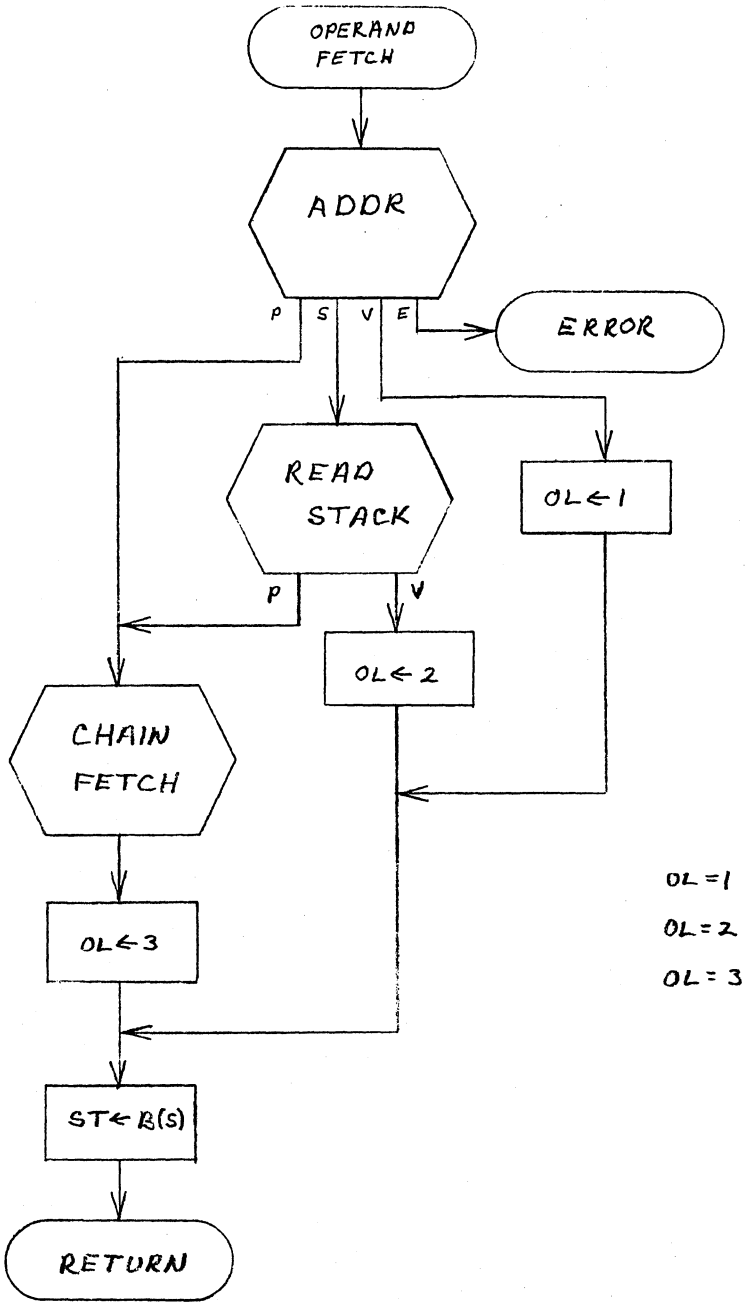
56 - MOD
57 - LIM


CHAIN FETCH
T ← 31
T
>0
0
ERROR
FETCH
T ← T-1
B(TAG)
14,15
12,13
OTHER
CH
0
1
CH ← 1
TPR ← B(R)
TPG ← B(G)
ITAG ← B(I)
P ← B(L)
T ← T-1
RETURN
CHAIN FETCH

Mode 3 is the W.P. Disable according to Mode Assignments
6/7/71
Y.S.



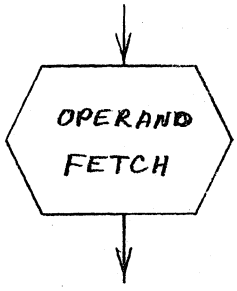

DESTIN
ADDR
P
S
V
E
READ STACK
P
V
IN(Y)
4
OTHER
ERROR
CHAIN FETCH
B(TAG)
ADDRESS (12,13,14,15)
OTHER
ERROR
U(TAG)
INTEGER (S)
OTHER
ERROR
0≤U≤B(IM)
YES
NO
ERROR
P ← B(L)+U
CHAIN FETCH
IN(A)
1
0
IN(Y)
2,3
OTHER
P ← 2P
CHAIN FETCH
B(TAG)
CONTROL (10,11)
OTHER
ERROR
CN ← B(L)
MODE ← B(MD)
CC ← B(CC)
RETURN
DESTIN

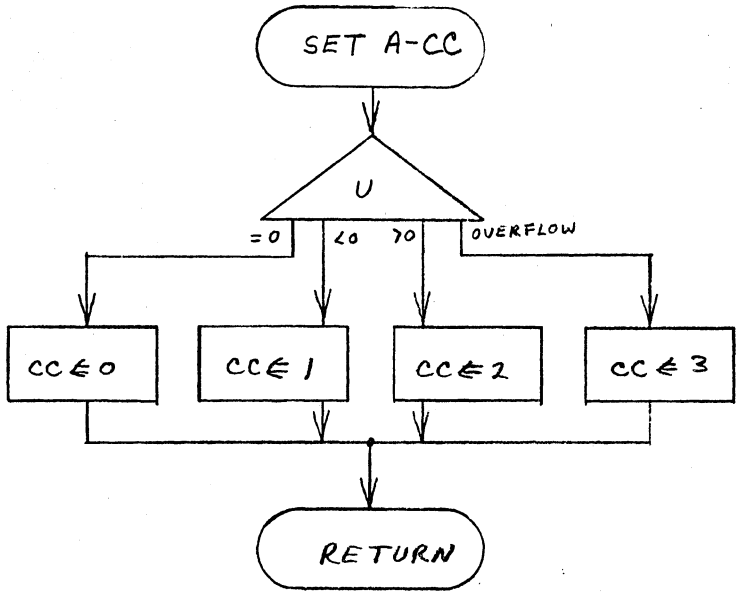


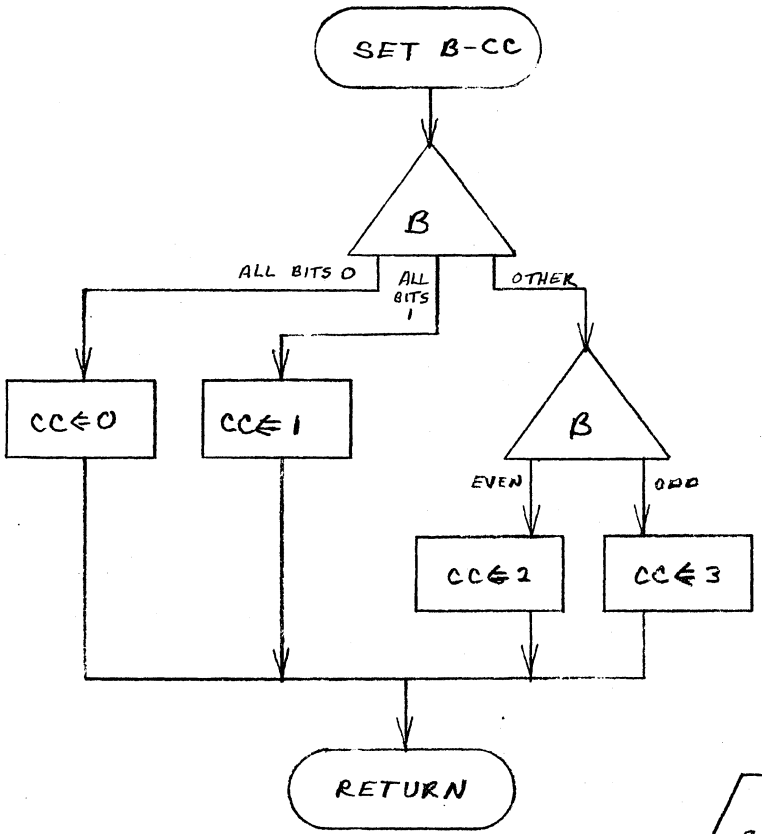
OL=1 — $R_{IN(Y)}$

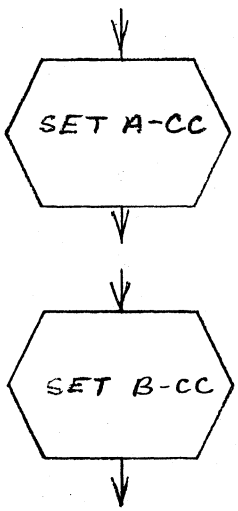
OL=2 — TOP OF STACK

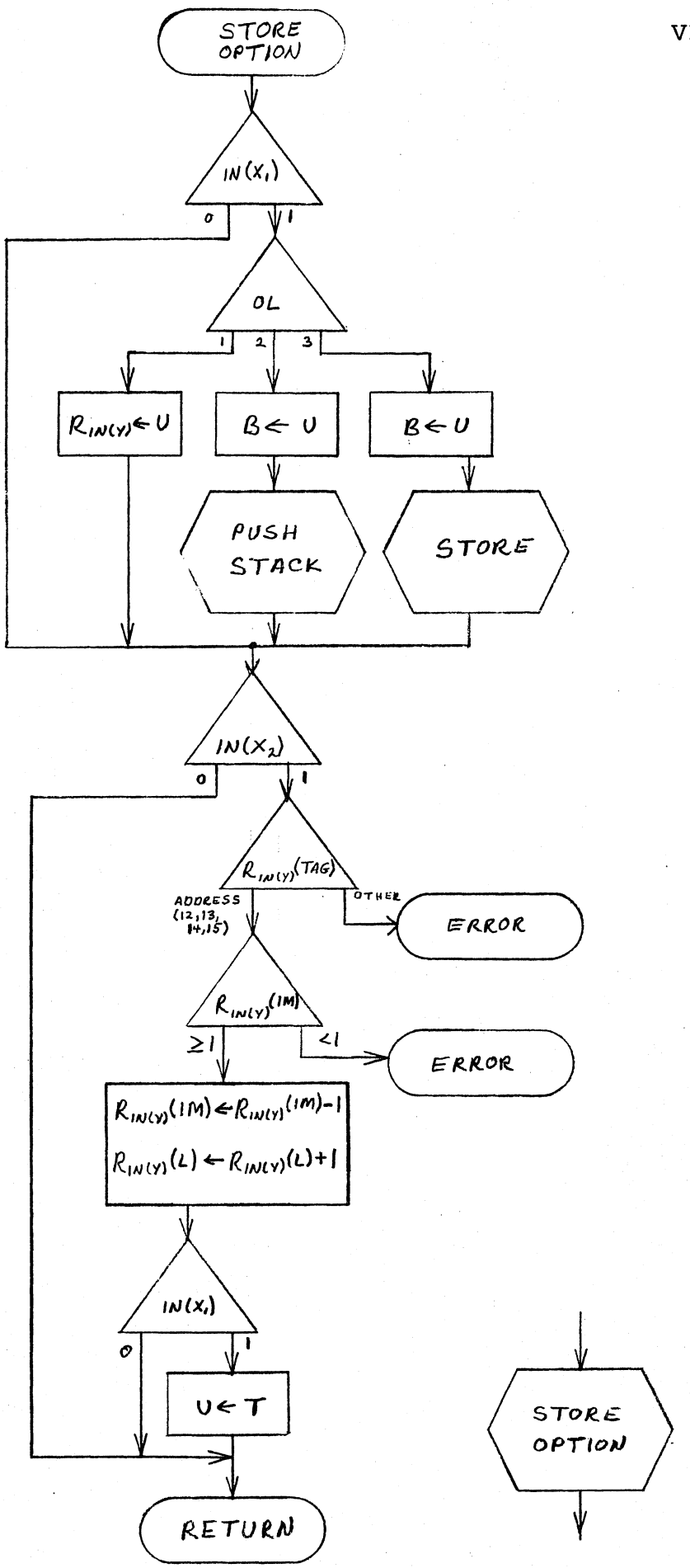
OL=3 — M_p




SET A-CC
U
=0
<0
>0
OVERFLOW
CC∈0
CC∈1
CC∈2
CC∈3
RETURN


SET B-CC
B
ALL BITS 0
ALL BITS 1
OTHER
CC∈0
CC∈1
B
EVEN
ODD
CC∈2
CC∈3
RETURN


SET A-CC
SET B-CC


STORE OPTION
IN(X1)
0
1
OL
1
2
3
R IN(Y) ← U
B ← U
B ← U
PUSH STACK
STORE
IN(X2)
0
1
R IN(Y) (TAG)
ADDRESS (12,13, 14,15)
OTHER
ERROR
R IN(Y) (1M)
≥1
<1
ERROR
R IN(Y) (1M) ← R IN(Y) (1M) - 1
R IN(Y) (L) ← R IN(Y) (L) + 1
IN(X1)
0
1
U ← T
RETURN
STORE OPTION