1974

Storage linking techniques for the automatic management of dynamically variable arrays

Luis Manuel Alarilla Jr.
Iowa State University

Storage linking techniques for the automatic management of dynamically variable arrays

by

Luis Manuel Alarilla Jr.

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Electrical Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1974

TABLE OF CONTENTS

I. INTRODUCTION

The arrangement of numbers and/or symbols into arrays provides a natural and convenient way of representing data for algorithmic processes to be performed by a computer (1). This comes about from the ability to reference array elements by specifying the array name together with a set of subscripts. Thus general purpose computers from the earliest models to the present have had array handling capabilities. These capabilities have grown from the ability to work with a one dimensional array or a vector of equal length elements in the early computers to a machine which allows the most general arrays. The changes in array handling capabilities are tied to the changes in the way the memory resource of the computer is organized.

Most present day computers follow the pattern of the von Neumann model of central processor design which was presented in the pioneering study by Burks, Goldstine, and von Neumann (2). The most important characteristic of the von Neumann model is the concept of a single linear store addressed by consecutive location numbers. In this model instructions are indistinguishable from data. Memory is allocated either a location at a time or in blocks of contiguous locations. This view of memory is very well suited to the data representation of linear arrays or vectors since the array elements are stored in contiguous locations in memory with the length

of the elements being limited by the computer word size.

A multidimensional array must exist in conventional storage as a linear array. To locate an array element, the set of indices or subscripts must be transformed to the location (address) of the element in its storage vector. The computation of the index address transformation is performed by the central processor. Hellerman has described several ways of implementing this transformation (3). The use of index registers has been the path followed by modern conventional machines in the attempt to speed up the address calculation of the location of an array element.

Most algorithmic languages and compilers provide means for working with arrays. Through the use of a dimension or a type declaration statement in the language, a collection of primitive variables of identical type can be identified by a single variable name. The compiler will respond to such instructions by allocating a contiguous group of memory words for the entire array. A set of subscripts supplied with the array name invokes the mechanism for selecting a particular element in the array. By remembering the initial location of the block allocated to the array and the size of the individual components, the compiler calculates the location in memory of the particular element specified by the subscripts (4).

In systems using static allocation the maximum dimensions are declared by the programmer and the compiler allo-

cates the corresponding amount of memory. This results in a certain amount of wasted memory space. In more dynamic block structured languages the array dimensions may be left as variables and they can vary in the transition from block to block during execution time. This is achieved by using "dope vectors" which contain the parameters necessary for doing address calculation within each block.

Most conventional computer systems provide means for working with arrays of multiple dimensions in which the elements are scalars. Problems arise when it is desired to represent an array in which the elements are themselves arrays (1). The PL/1 language provides a means of defining some hierarchical structure of arrays which may be viewed as a multidimensional array. However there is an additional requirement that every subarray must be given a label in the original definition of the structure (5). This restriction makes this structure essentially different from the arrays being considered.

A system, which allows the elements of multidimensional arrays (called generalized arrays) to be arrays themselves, has been devised and implemented in the operating system of the Rice University computer (1). This computer does not view the memory resource as a linear store. In this computer system physical storage is issued in blocks of specified length to the program which requested it. Associated with

each block of storage is a codeword which contains the location of the first word of the block, its length, and a symbol to indicate whether the block contains other codewords or data. If other codewords are contained, then these will in turn point to other blocks of storage, so that a storage hierarchy is constructed. This type of store has been described as approximating a tree structure (6).

Blocks of storage under this mode of control can represent arrays of some complexity. Accessing an element of a high order array may require several intermediate accesses through the codeword hierarchy. Consider matrices as an example. Each matrix is an array structured from a primary codeword which defines an array of codewords. The ith codeword of this array points to an array whose elements are scalars of the same length. These scalars form the ith row of the matrix. The jth element of this row corresponds to the (i,j)th element of the matrix. The above system can be extended to generalized arrays, since the elements of the matrices may themselves be matrices or arrays of any dimensions. Accessing an element is achieved by tracing through the storage hierarchy with the subscripts supplied. The system permits a total of five subscript levels (1). Another computer system organization which treats arrays in a similar manner is used in the Burroughs B5700/B6700 series (7).

The APL language allows subscripted expressions like A(I;J;K), where A is an array and I, J and K may be scalars, vectors or arrays of any size and shape. APL has been implemented on an IBM/360 model 25 with the use of a microcoded interpreter (8, 9). This interpreter allocates a block of memory for each variable name in the program and uses an address table to point to the blocks of memory assigned. The address table entry also contains information regarding the name. The entry specifies whether the variable and the block contains a value and if so whether the value is character, logical, integer or real. The address table entry also specifies whether the variable is a scalar, a vector or an array. This interpreter essentially reorganizes the conventional memory into something like the Rice computer (1) and the Burroughs machines (7) with the address table entry corresponding to the codeword. It is interesting to note a statement that was made by Hassitt and Lyon (8). They stated that the construction of an efficient APL interpreter led to some complex problems.

An examination of a selection of mathematical and data-handling problems that has to be solved by a computer will reveal that their data and instruction storage requirements will not fall easily into the mold of the linear store. The programmer then has the task of fitting the data representations that he uses to the machine and probably restricting

his range of operations as a result. Iliffe (6) has commented that the conventional means of translating and obeying programs acts as a filtering mechanism which admits to the computer only those problems whose solution justifies the cost of translation into a rigid store structure.

Due to the above mentioned difficulty the memory resource has received considerable attention from people interested in computer systems architecture. Barton (10) proposed a structure for storage devices which, though they have physical words, appear at the machine language level to have no structure at all beyond a single large field upon which programmed structural definitions can be imposed. He suggested that field size be associated with a field or sets of fields of identical sizes so that it is encountered in accessing the object. This necessitates making mapping a machine function and through it hopefully systematizing problems of locating and relocating data and programs. He proposed looking upon all storage as a tree of fields. Barton's proposals were motivated by a desire for economy of representation and ease of expression.

More recently Dennis (11) has described the reluctance of computer architects to adopt a view of memory better suited to the data representation needs of contemporary computer systems. Dennis was in favor of architectures which provide means for building large data structures from small

ones. He has come out in favor of tree-like objects as the fundamental storage structure since he considers them better matched to applications requirements than the linear store.

An existing computer system which represents a step in the direction suggested above is the SYMBOL 2R computer system. The SYMBOL 2R computer system was first described in the 1971 SJCC (12, 13, 14). It was described as a system which provides variable word length processing and a storage which allows the explicit representation of structures that are variable in size, shape and field length. Type and size declarations are unnecessary since conversion and space management are handled automatically. This relieves the programmer of the necessity of declaring data base sizes and attributes. Very few limits are placed on data structures. Data fields may grow to the size of main memory and no restriction is placed on the level of subscripts in an array.

The ease with which the above features were implemented is a consequence of the way SYMBOL views its virtual memory. It looks at the memory resource as an effectively unlimited number of data-storage strings with each one having the capacity to store an arbitrarily large and dynamically variable amount of information. Richards and Zingg (15) have called this the Logical Storage level. The basis upon which the Logical Storage is constructed is the virtual storage. Virtual storage is simpler and more conventional since it is

essentially a linear virtual memory store. SYMBOL allocates its virtual memory space in quanta called groups, with each group consisting of eight contiguous 64 bit (8 character) words. This allocation is done dynamically upon demand. These groups can be connected logically together to form strings of arbitrary length. SYMBOL makes use of list processing techniques to handle these storage strings. SYMBOL represents arrays in virtual memory in a manner analogous to a tree structure.

The above paragraphs have traced how arrays have been handled in general purpose computers. The first computers could handle only one dimensional arrays or vectors since that data structure corresponded very well with the linear store model of viewing memory. Means of handling multidimensional arrays were introduced to be able to take care of more complex data structures. Multidimensional arrays are handled in present day general purpose computers through address calculation with the aid of index registers. The Rice University computer system represented a step in complexity since it was able to handle arrays of arrays, although with a maximum bound on the level of subscripting. All of the above systems are restricted to having uniform field length for the lowest level elements. The SYMBOL 2R computer system and the APL implementation on the IBM/360 model 25 represent major steps in array handling capability since there are no restrictions

on the array size, shape and field length of elements.

Early in the history of the SYMBOL 2R computer system it was realized that the present scheme of implementing dynamically variable arrays resulted in slow accessing of array elements. Thus there was a desire for an implementation scheme which allows faster element accessing. The people who developed the SYMBOL 2R computer system have done some preliminary work towards the development of another scheme of implementation which will be able to access array elements faster.

This investigation is motivated by the same desire for faster element accessing in an implementation scheme for dynamically variable arrays. The basic problem of this research is primarily a search for an implementation scheme with this desired characteristic. This investigation is also directed towards evaluating the present implementation of the concept of dynamically variable arrays. Other implementation schemes which are possible within the constraints of the way memory is organized in the SYMBOL 2R computer system are developed and studied to determine how the concept is implementation limited.

There can be no argument against the desirability of being able to accomodate dynamic data structures in a computer system. But what is the cost of having this feature in a computer system? The existence of this feature in the SYMBOL 2R system provides a means for getting some answers to this

question.

This study is also directed towards finding the cost of having dynamic variability in one type of data structure--arrays. This paper hopes to provide some measure of the cost, in terms of memory space and memory access time, of having dynamically variable arrays in the SYMBOL 2R computer system. This should provide some idea of the cost, in terms of time and space, of the ease of use and generality incorporated in a system. It is also hoped that the search for implementation schemes with faster element access time will reveal some of the trade-offs involved between memory space and access time.

The approach taken is to simulate the present implementation scheme and the alternative schemes which were developed. These simulations will allow a comparison of the performance of these schemes within the same framework of memory. The results of these simulations should reveal how the present implementation limits the concept of dynamically variable arrays. These results provide the means for measuring the cost of having this feature in the system.

The recent paper by Richards and Zingg (15) has proposed that any evaluation of the SYMBOL 2R Logical Storage system must be addressed to two types of questions, namely,

--are the benefits that accrue from Logical Storage worth its cost?

--can those benefits be obtained by other means at lower cost?

This investigation indirectly touches on the first question by hopefully leading to a better appreciation of the cost and benefits of Logical Storage. For this investigation, the second type of question can be rephrased into,

--can the benefits of dynamically variable arrays be obtained by other schemes at lower cost?

This study hopes to provide an answer to this question in terms of another scheme which implements dynamically variable arrays at a lower cost. Hopefully this paper would provide some help in the search for an organization of general purpose computer systems memory resource which would conveniently accommodate contemporary data representation.

II. THE SYMBOL 2R COMPUTER SYSTEM

The SYMBOL 2R computer system will be described briefly with special emphasis on the processors which perform the storing and accessing of data structures. There will also be special attention on the way the memory resource is organized in the system. This material on the system is necessary for a better understanding of the different schemes for storing and accessing elements of dynamically variable arrays.

The SYMBOL 2R computer system is composed of eight substantially independent processors, each dedicated to a particular set of functions. The Translator (TR) translates the source program supplied by the user into an internal object program, which is then executed by the Central Processor (CP). The Input/output Processor (IP) and the Channel Controller (CC) provide the communication between the user and his program. The System Supervisor (SS) coordinates the activities of the various processors. The user-oriented processors do not communicate directly with the core memory or the disk. Instead requests for memory service are addressed to the Memory Controller (MC). The Disk Controller (DC) and the Memory Reclaimer (MR) assist the MC in providing memory services to the other processors. More information about these processors can be found in (12, 13, 14).

The Central Processor consists of four sub-processors which are as follows: the Instruction Sequencer (IS), the

Arithmetic Processor (AP), the Format Processor (FP), and the Reference Processor (RP). The IS scans the object string and controls the overall execution process by calling on the other sub-processors as necessary. Among the functions of the RP are performing assignment operations, and manipulating and accessing data structures (16).

The original papers (12, 13, 14) describing the SYMBOL 2R system and the documentation (17, 18) accompanying the system have used the term "structure" to represent dynamically variable arrays. An attempt has been made to consistently use the word "array" instead of "structure" in this paper. However, there will be unavoidable lapses especially when describing actual routines and their flow charts. And so it should be noted that "structure" will generally mean "array" especially in the sections dealing with routines and their flow charts.

The Reference Processor bears the major responsibility with regards to array storing and referencing. The RP may be divided into three major sections. The first is the Get Address Simple section. On request from the IS this section accesses the name table for a variable by taking the variable reference from the IS stack, analyzing that name table entry for the various possible cases, and finally returning the address together with the information on what case was found to the stack. The second major section is the Get Address Sub-

scripted section which takes the previously obtained simple address to which the subscripts have been appended during IS processing. In the RP the Get Address Subscripted section traces out the path denoted by the subscripts, expanding vectors if required, and returns the final address to the IS stack. Several utility functions are used by the Get Address Subscripted section. Among these utility functions are the routine which scans through the words of a storage string in order to find the desired component of the vector represented by that storage string, and two routines which are used to speed up this scanning process. The last major section is the Assign section which assigns some item to another item. There are three basic utility routines in this section. The first is the simple variable assignment routine where a scalar value is assigned to a variable. The second is an assignment to a component of a structure. This takes care of expanding the data space if the original space does not have enough room to accommodate the new data element. The last utility routine is the structure assign section which assigns at any point in data space a complete structure and also makes up the various link words pointing to lower structure elements (17).

The SYMBOL 2R computer system memory resource organization will now be described. The following description will essentially be based on the Richards and Zingg paper (15) on

the structure of the memory resource.

The SYMBOL storage hierarchy contains four levels, namely; user-level storage, logical storage, virtual storage, and physical storage.

User-level storage consists of the storage facilities furnished to the user and under user control. These are his Transient Working Area (TWA) and his program storage. The TWA is a one dimensional character string of arbitrary length used as a work space for program loading and editing. Program storage consists of the storage cells corresponding to each variable in a program. Each cell can contain a value which can be either a character string of any length or a tree structured collection of such character strings, i.e., an array. A variable's value, or any component of a composite value can be replaced at any time with another value, with no length or shape restriction on the new value.

Logical storage is the level upon which the user-oriented processors rely. Logical storage is seen by the processors as an endless supply of storage strings, which are allocated at their request by the Memory Controller. The MC operations which can be requested by the other processors can be roughly classified into four categories, which are the following:

a. Initial allocation. The MC allocates a new storage string and returns the address of its first word to the

processor.

b. Alterations. Included here are the operations which change a storage string's length, its contents, or both. Also included are operations for inserting words inside a storage string.

c. Reference. Operations under this category are the different fetch operations. Some of these permit a scan backwards and forwards over the words making up the storage string.

d. Deallocation. This covers the operations which return strings to the available space pool.

A very important characteristic of all logical storage operations is that only the MC generates addresses. The other processors just present addresses which were previously returned by the MC. Therefore it is not necessary for logically contiguous words in a storage string to have numerically consecutive addresses.

The next level in the memory structure hierarchy is the virtual storage. This is the basis upon which logical storage is constructed. This level is far simpler and more conventional since it is essentially a linear virtual memory consisting of some 16 million (2^{24}) words, each denoted by a unique 24 bit address. It is divided into 64K pages, each containing 256 words (2048 characters). At present, only the first 4K of these pages have been installed.

Each of these pages is partitioned into overhead and data-space regions. The first 32 words of each page are devoted to overhead while the remaining 224 words comprise the data space. The data space is further subdivided into 28 groups of eight consecutive words each. Each data-space group will have an associated group link word in the overhead region of the same page. Of the other four overhead words, one is unused and the other three are used by other system functions. It can be seen that the quantum of storage allocation as performed by the MC is the eight word group.

Group link words are used to link together groups which form the storage strings of logical storage. This is made possible by the two addresses contained in a group link word. These addresses correspond to forward and backward pointers in the string. Group link words imply an extra memory access every eight data storage access when scanning through storage strings.

The last level in the SYMBOL memory resource hierarchy is the physical storage. This is presently made up of an 8192 word, 2.5 microsecond core memory and a 4096 page drum. A demand paging scheme is used for transferring pages between the drum and the core. Of the core memory, 28 pages (7168 words) are used as the buffer memory while the remaining 1024 words are reserved for system supervision overhead.

III. DYNAMICALLY VARIABLE ARRAY IMPLEMENTATION SCHEMES

A. Present Scheme

The ability to handle vectors, matrices, and higher dimensional arrays is a standard feature of general purpose computer systems. In conventional systems the elements of these structures are usually restricted to scalars of one or two words. This restriction imposes cumbersome data representations for sophisticated applications. In the SYMBOL 2R system there are no restrictions on these data structures except for an upper limit of 9999 for each subscript. The elements of multidimensional arrays could be scalars, character strings, vectors or even multidimensional arrays. Each data element within the structure could be as long as necessary.

Dynamically variable arrays are presently implemented as vectors in string form. The components of a vector are stored in a storage string one after another without regard for their length. Components which are themselves arrays are stored in a different manner. In this case the component stored in the original vector is a pointer to the substructure. This component is marked with a special code character from the SYMBOL 2R internal code set. The end of each vector is also marked by a word containing a special code character.

An example of how a one dimensional array looks in storage can be seen in Figure 1. The figure shows the array

occupying twenty words in a storage string made up of three groups. It can be seen from the figure how the group link words link the groups forming the storage string. The 21st

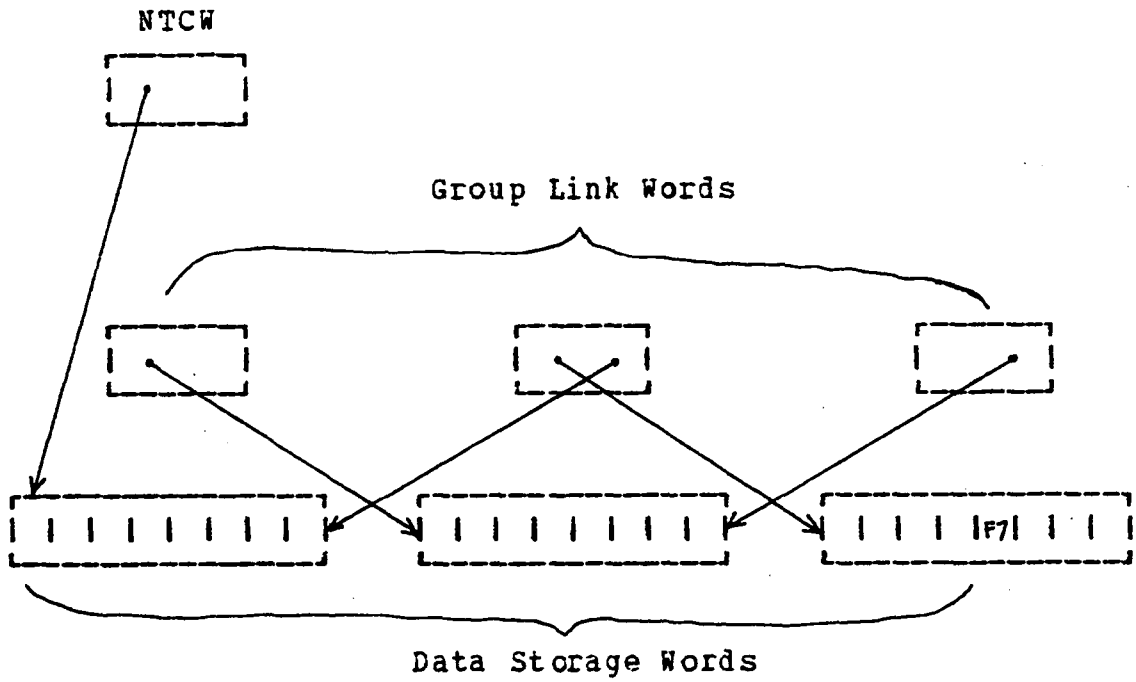


Figure 1. One Dimensional Array

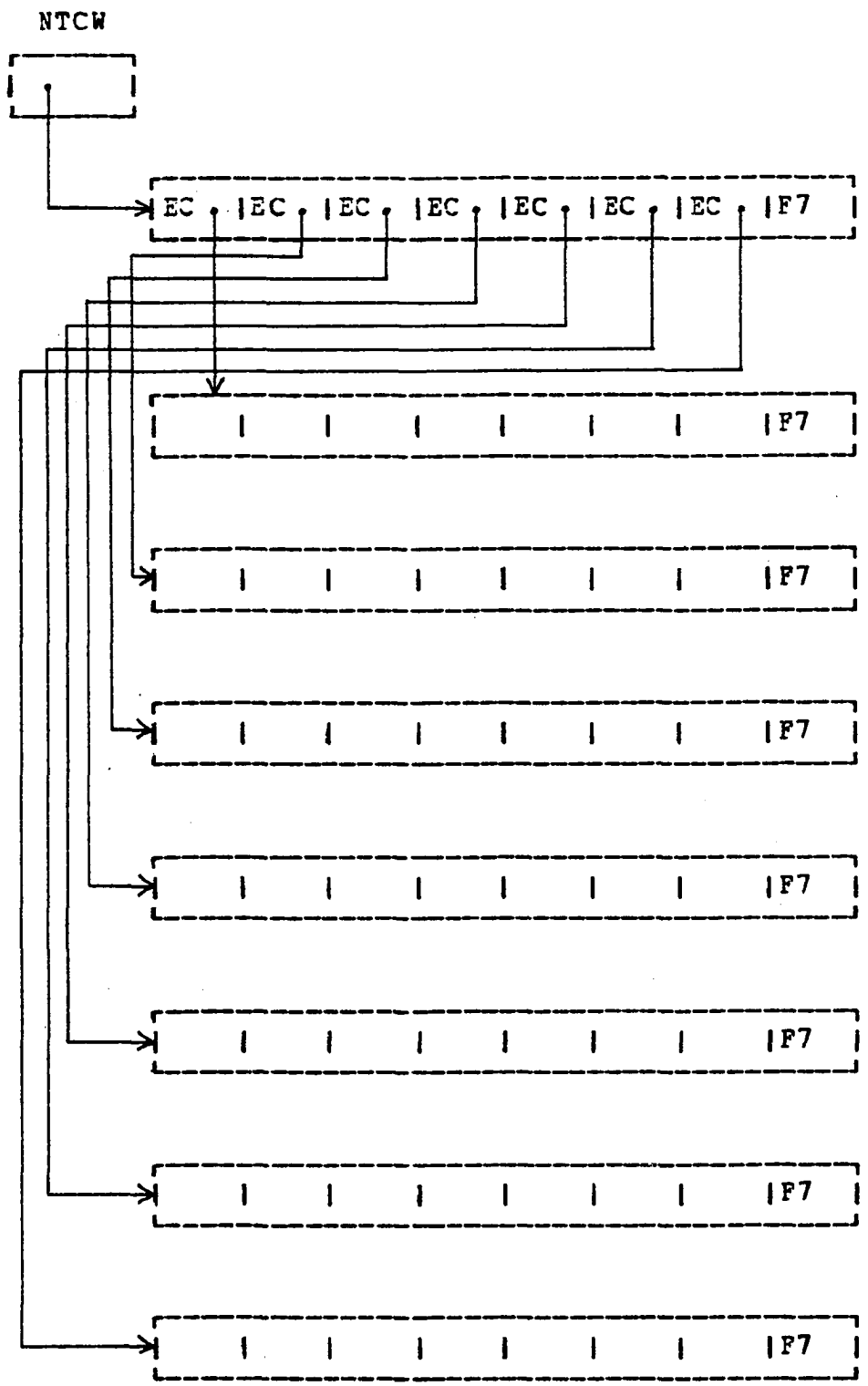
word of the string contains an F7 which is the end vector character. This signifies the end of the vector and that the remaining three words in the string are unused. The first 32 bits of the Name Table Control Word (NTCW) will contain the link to the array. The first 8 bits contain various flags and the remaining 24 bits contain the address of the first word of the vector. This vector could contain 20 one word long components or two 10 word long components or any combination of number of components and component length as long

as the total space occupied is 20 words. The components could be of string or numeric data values of variable length. Special code characters are used to mark the start and end of each component.

Any of the components of a vector could be another array or structure. In this case the component stored in the vector is a pointer to the subarray or substructure. This linking to other arrays could go on with no limit except perhaps the amount of physical storage available. This scheme is what makes possible the storage of arrays of irregular size and shape.

An example showing how a 7 x 7 array would look like in storage is shown in Figure 2. Here the Name Table Control Word points to the origin of a vector which contains 7 components. Each of these components is itself a vector of 7 components, so the first vector contains seven links to substructure and an end vector. Each of the vectors pointed at contains seven data components and an end vector character in the eighth word. Irregularity in shape comes in if the vector contains components which are not arrays or if the subarrays contain pointers to other subarrays.

A subscripted reference in the object string consists of a reference to the subscripted identifier, followed by a list of subscript expressions. The expressions in the list are separated by Integerize operators and the list is terminated



one word long data elements

Figure 2. A 7 x 7 Array

by the "]" instruction (16). The Integerize operations are performed as they are encountered and the subscript values are placed on the stack. When the "]" is detected, the last subscript is integerized and stacked and then the stack is searched backwards for an entry which is not an integer. Each subscript was integerized so the noninteger entry should be the subscripted identifier. The RP is called to replace the entire subscripted reference in the stack with the address of the desired component. The identifier address is passed to the RP when it is called to resolve the reference.

The manner in which the array elements are stored in the system allows the arrays to dynamically vary in size and shape. The price paid for this feature in the present system is the length of time it takes to find the address of a desired component. Since the elements could be of any length, the address calculations used in conventional systems to quickly find addresses of array elements will not work. Another reason is that the array elements will not in general be stored in contiguous memory locations.

What the RP basically does at present is to start at the address pointed to by the Name Table Control Word and then it scans the storage string word by word, testing for component starts. Every time a component start is found the subscript register is decremented and this goes on until the right element is found. This is indicated by a zero subscript regis-

ter. It can be seen that the number of memory accesses to get the address of an array element is highly dependent on the location of the element in the string and sizes of the elements preceding the desired component. The number of memory accesses required to retrieve an array element is also heavily dependent on the shape of the array. As an example, it may take longer to access member (2,99) of a 2 x 100 array than to access member (99,2) of a 100 x 2 array. The reason behind this is that the first case requires a much longer serial search over variable length fields than the second case. In the second case only one word links to substructure are passed over to get to the pointer to the start of the vector corresponding to the second subscript.

Since the number of memory accesses needed to get the address of an element of an array could become a considerably large number for big arrays, two techniques for speeding up the structure scanning process are utilized. The first technique is made possible by the fact that processing involving arrays can usually be done by accessing components in ascending order. If a vector has been scanned through to find the Ith component at some address A, the value of I and A can be stored. Then if at some later time the (I+K)th component is desired, the search starts at A and then scans over K components to get the proper address.

Words which point to vectors are the NTCW for a structure and the pointer to substructure for vectors within structures. These words contain the code character in the first 8 bits and the starting address of the vector in the next 24 bits of the word. Since each word has 64 bits, the last half is available for storing the subscript and the address of the most recently accessed component of that vector. The subscript is stored in BCD in the first 8 bits and the address of the last accessed component occupies the last 24 bits of the halfword. This halfword of each pointer to a vector is called the Current Pointer.

Since there are only 8 bits available for storing the BCD subscript, the Current Pointer is good only up to the value 99. Whenever the desired component has a subscript greater than or equal to the Current Pointer, the search for the desired component starts from the address contained in the Current Pointer instead of the starting point of the vector. If the subscript of the last accessed component is greater than 99 then what is saved is the value 99 and the address of the starting point of the 99th component of the vector.

The second technique is called the Rapid Search and this makes use of the unused bits in the group link words. The first 8 bits of each halfword of the group link words are available since the forward and backward link addresses only

occupy 24 bits each. The first 8 bits of the group link word correspond to each word in the group and these 8 bits are known as the Rapid Search Field. If a word contains the start of a component, then the corresponding bit in the Rapid Search Field is set to ON. Otherwise the bit remains OFF. When scanning over components, the component starts can be detected and counted from the Rapid Search Field of the group link word. Thus if the 16th component is wanted, the Rapid Search Fields of successive group link words in the storage string can be examined and the ON bits are counted. When the 16th ON bit is found, then the 16th component must start in the corresponding word. The resulting performance is an 8 to 1 saving in the number of words scanned and hence in the number of memory operations over the basic word scan process.

It has been mentioned before that the Reference Processor is the particular sub-processor in the CP which is responsible for array storing and referencing. A closer look will now be taken at the sections of the RP which perform these functions. This will show how the present scheme of dealing with dynamically variable arrays has been implemented in the RP.

One of the major sections of the RP is the Assign section and its function is the assignment of some item to another item. There are three basic routines used in this section. The first is the simple variable assignment routine,

which has the function of assigning a scalar value to a variable.

The second basic routine's function is the assignment to a component of a structure. This takes care of expanding the storage string if the original data space does not have enough room to accommodate the new data. The expansion is performed when a new component start point is detected in the process of assigning a data field to a structure element. The field expansion subroutine performs the algorithm for rewriting the area and assigning a new group to the area so that it is expanded.

The last basic routine is the Structure Assign section. This section has the function of assigning a complete array at any point in data space and making up the various link words pointing to lower structure elements. The Structure Assign section is described in detail in section A of the Appendix with the help of a flow chart from the Reference Processor Flow Charts (18). Both the field expansion and the Structure Assign section call the rapid search generate routine to create or update the rapid search field in order to properly rapid search through the data space on subsequent access (17).

The other major section of the RP that is of interest is the Get Address Subscripted section. This is the section which takes the subscripted reference in the stack and re-

places it with the address of the desired component. This is achieved by tracing out the path denoted by the subscripts, expanding vectors if required, till the desired component is reached; then the starting address of the component is returned to the IS stack. This routine uses several utility functions in the referencing process.

The first one is the current pointer routine which checks the current pointer halfword in the vector link to see if the reference subscript is greater than the subscript stored in the current pointer. In that case, the current pointer address is used as the starting address for the scan after the subscript in the current pointer halfword has been subtracted from the reference subscript.

The next routine used is the rapid search routine which is used to scan over the rapid search field of the group link words to ascertain the number of component start points within the group. In this way, groups are quickly counted over through the group link words. Two corollary subroutines are used to generate or update these speed-ups. The Current Pointer Generate updates the current pointer in the vector link word whenever there is a reference to that vector. The Rapid Search Generate sets up the rapid search fields in the group link words whenever a structure is created or a component expanded.

The final utility routine is the word scan routine which scans through the words of the storage string in order to test for component starts. Whenever a component start is detected the subscript register is decremented. A zero subscript register will indicate that the desired component has been located. This routine also has the function of creating new structure when oversubscripting is started by scanning past the end of the existing structure.

A more detailed description of the Get Address Subscripted section and all its utility functions can be found in section A of the Appendix. All the necessary flow charts (18) are part of this section of the Appendix.

B. Modulo 16 Scheme

The biggest disadvantage of the present scheme of handling dynamically variable arrays is the considerable number of memory cycles spent in order to reach the desired component in a subscripted refernce. However, no address computation scheme is possible in this system where the array elements are allowed to vary dynamically. A way of speeding up the accessing of array elements is to associate a pointer with each component. This is the basis of the Modulo 16 scheme.

The author came up with the basic idea for this Modulo 16 scheme in the course of the search for an alternative scheme which would allow faster array element accessing and

which would be feasible within the framework of memory of the SYMBOL 2R computer system. It was subsequently developed and expanded so as to allow a comparison of its performance characteristics with other schemes.

In this scheme halfword pointers are assigned for each array element. In accessing an element, the search for the desired element is performed in the storage string containing the pointers instead of the storage string containing the actual data elements. Searching through the pointers is faster than through the actual data space since the pointers are all of uniform length. This uniformity helps speed up the process of accessing elements at the cost of consuming more space for linking overhead. With this scheme there should be no difference that will affect the user except a speed up in access time and a need for more storage space. There will still be no restrictions of any kind on the arrays that can be represented in the system (except for the 9999 limit on subscripts).

Dynamically variable arrays will be implemented in this scheme in a manner similar to the present scheme. Arrays will still be vectors in string form, but in this case vectors will only contain pointers. The pointers could either be a link to data or a link to substructure. The end of vectors will still be marked by the same end of vector code characters. Each halfword pointer will consist of an 8 bit

link code field and a 24 bit address field. In addition to the previously mentioned type of links, null elements will be denoted by an all zero link field.

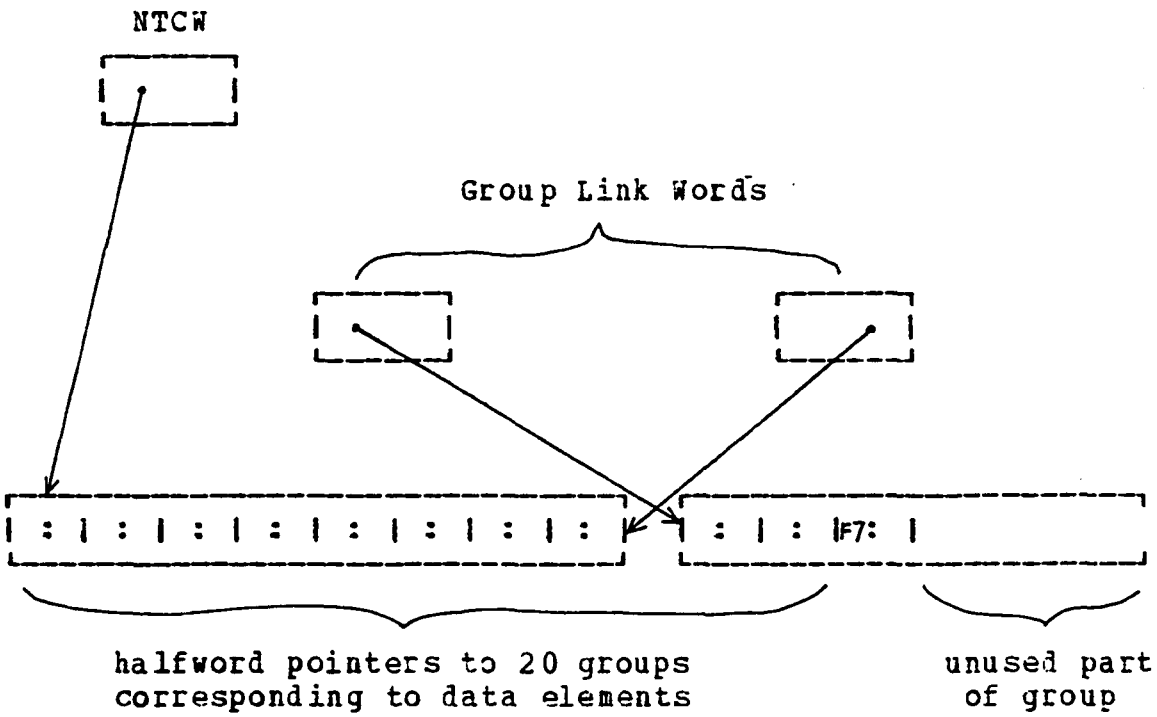


Figure 3. One dimensional array stored using Modulo 16 scheme

The way a twenty element one dimensional array would look in storage is shown in Figure 3. Here the NTCW points to the storage string of links instead of the data elements themselves. There are 20 halfword pointers so these occupy 10 words in a storage string made up of two groups. These 20 halfword links to data point to the locations of the 20 data elements belonging to the vector. In the storage string consisting of pointers, the 21st halfword or the first halfword of word 11 will contain the end vector character

signifying the end of the vector. Since no link to data code characters presently exist in the SYMBOL 2R internal code set, code characters must be added to take care of this kind of pointer. Allocating one group for each data element will result in allowing data elements to grow and shrink without affecting each other. There will be no need for a field expansion routine. It will also mean a greater amount of wasted space if the data elements are less than 8 words.

The storage string of pointers could contain links to data or links to substructure which in turn point to other storage strings consisting of other links. This allows the representation of an array of any size or shape. As an example a 7 x 7 array would be stored in the manner shown in Figure 4. The NTCW points to a storage string of one group containing 7 links to substructure and an end vector mark. Each of these links to substructure points to a storage string consisting of 7 links to data storage strings.

The investment in additional space overhead for the pointers to the data elements results in some improvement in the speed of finding the address of a desired element. Due to the uniform length of the link words in the pointer storage string, the search for the component is done by examining the group link words of the pointer storage string. As long as the subscript is greater than 16, the search is performed by getting the address of the next group from the group link

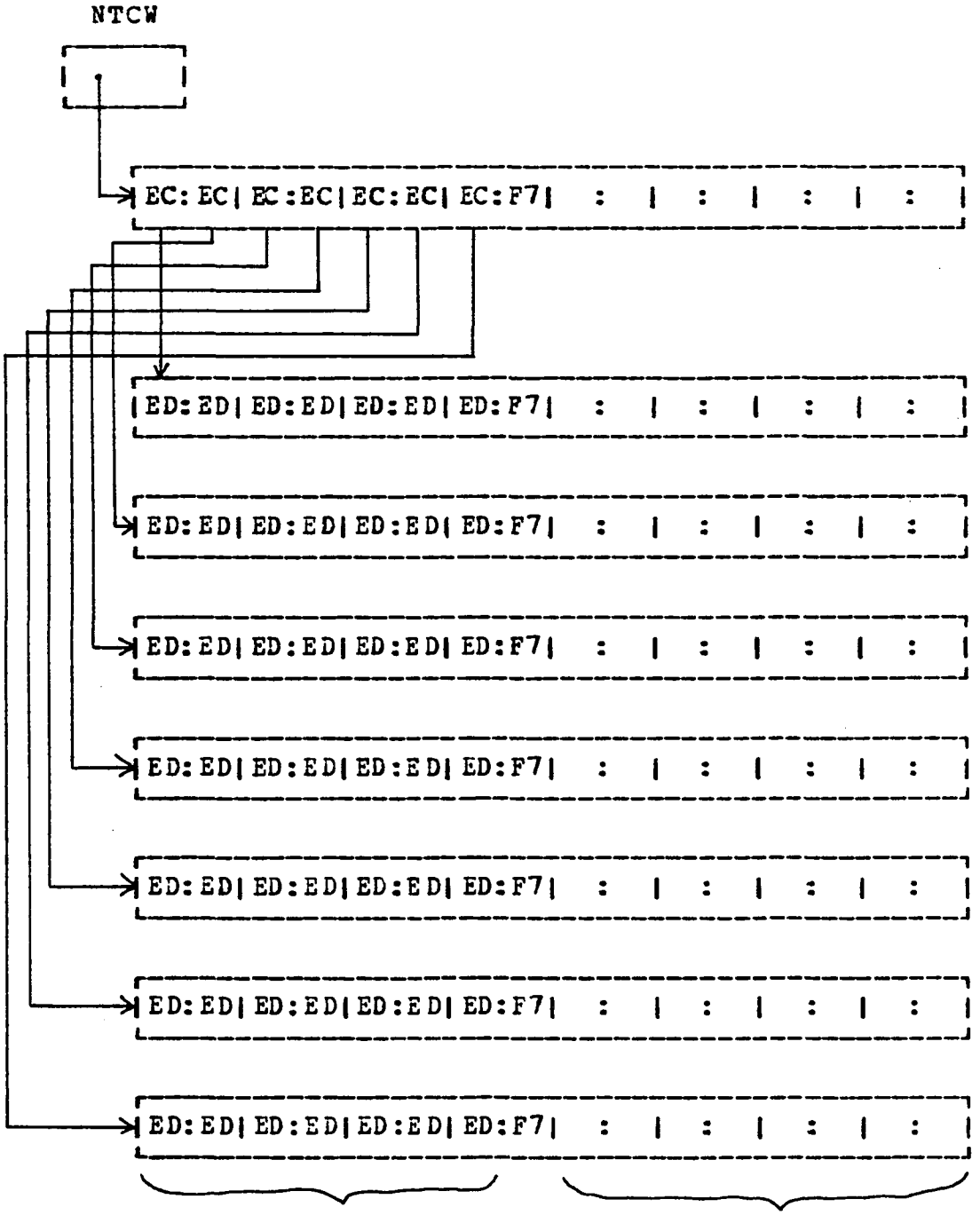


Figure 4. A 7 x 7 array stored using the Modulo 16 scheme

word and then decrementing the subscript register by 16. After this, the subscript is checked to see if it is greater than 16, if it is, then the sequence repeats. This is the reason behind the choice of Modulo 16 as the name of this scheme. If the subscript becomes equal to or less than 16, then the address of the desired element can be obtained by incrementing the address register by 1 while the subscript is decremented by 2. This is done repeatedly as long as the subscript is greater than or equal to 2. When the subscript is less than 2, then the address is contained in the word pointed at by the address register. If the subscript register contains a 1, then the address is in the left half of the word. A zero subscript means the address is in the right half of the word.

The resulting performance using this scheme is at least 16 times faster than the word scan and at least 2 times faster than the rapid search of the present scheme. The performance becomes better than 2 times faster than rapid search or 16 times faster than word scan, if the length of the data elements becomes greater than one word. This is possible because in this scheme the access is independent of the field length of the elements. In the present scheme the data elements are stored one after another in one storage string. Thus any increase in the length of the data elements results in more words to be scanned through when searching for a com-

ponent. One negative effect of this scheme is the increase in group wastage or number of words unused in the group if the space the data elements need is less than one group. In addition, the increase in space overhead for links is one halfword for each data element. The total number of words used for linking can be a very significant number for very large arrays.

To implement this scheme certain parts of the RP have to be modified. These are the portions which perform the storing and accessing of the array elements. The two sections of greatest interest are the Structure Assign section and the Get Address Subscripted section. The modification in the Structure Assign section takes care of the extra store operations to maintain the link words and the assignment of a new group for each data element. In the Get Address Subscripted section the big change is the replacement of the current pointer, rapid search and word scan routines by a single routine to be called the search routine. The flow charts for these modified sections together with their respective descriptions can be found in section B of the Appendix.

C. Pseudo Level Vectors Scheme

Another scheme of handling dynamically variable arrays permits the desired component in a subscripted reference to be reached faster than the two previously described schemes. This is the Pseudo Level Vectors scheme. Pointers to the start of each data element are also used here. In addition, an extra amount of space is used to organize and link these data element pointers. These additional links and the way they are organized allow the search for the desired component of the subscripted identifier to depend on the digits of the subscript instead of using the whole subscript and counting down through the elements. The search for the desired component uses each digit, starting with the most significant and proceeding in the direction of the least significant digit, to trace through the linking until the desired element is actually located with the help of the least significant digit of the subscript.

The basic ideas for the Pseudo Level Vectors scheme originated with the people involved in the SYMBOL 2R computer project in the Digital Systems Department of the Fairchild Semiconductor Research and Development Laboratory. Permission to use this scheme has been granted through communications with Dr. William R. Smith of Fairchild Semiconductor.

As originally conceived by the people at Fairchild Semiconductor, this scheme was supposed to have been implemented

in a second version of the SYMBOL computer. They were planning to implement a 5 word group in this proposed version of the computer. With a 5 word group, each halfword in a group would be assigned a decimal digit. In this way the actual BCD subscript could be used digit by digit to locate the desired component of the array.

However, this original Pseudo Level Vectors scheme did not mesh very well with the the framework of memory of the existing SYMBOL 2R computer. The use of the original Pseudo Level Vectors scheme in the existing computer memory organization would have meant wasting the last three words in every 8 word group that is used as a vector for linking purposes. However, at an early stage of this investigation Dr. Roy J. Zingg came up with the idea of converting the BCD subscript to hexadecimal. The use of hexadecimal subscripts in the element location process meant that the scheme could be used with the present framework of memory without any group wastage in the level vectors. The expansion of these basic ideas and the development of the algorithms using the scheme were done by the author.

Since there are 8 words in a group, there will be 16 halfwords in a group which can be used as pointers. Each of these halfwords can be made to correspond to a hexdigit in the hexadecimal number system. The assignment of hexdigits to halfwords is shown in Figure 5. Here the left halfword of

the first word corresponds to the hexdigit 0, while the right halfword of the 8th word corresponds to hexdigit F. Each halfword of 32 bits will be divided into a link code field of 8 bits and an address field of 24 bits.

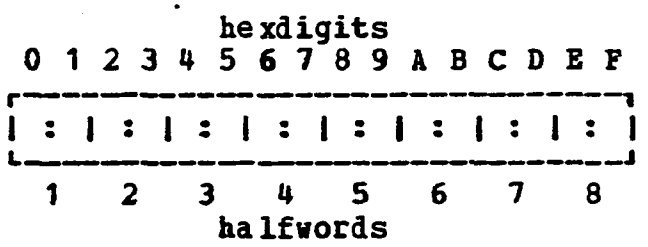


Figure 5. Assignment of hexdigits to halfwords

As the array is created, the number of elements is counted so that every time 16 of them have been counted a new group or vector must be assigned to contain additional links. At the same time links are created with higher level vectors. The groups or vectors are organized into four levels. The level number corresponds to the power of 16 that the links have in that level. So level 0 corresponds to vectors containing 16 to the zero power. The highest level is level 3 and this will not be completely populated since the maximum allowable subscript is 9999 in base 10 or 270F in hexadecimal.

An example showing how the links are organized in the Pseudo Level Vectors scheme is shown in Figure 6. Most of the different level vectors for a one dimensional array of 9999 base 10 elements are shown. The pointers in the level 3

vector point to level 2 vectors. The pointers in level 2 vectors point to the next level vectors. This linking goes on until the level 0 vectors are reached. The pointers in the level 0 vectors point to the locations of the actual data elements.

To illustrate how a particular element is reached, two cases will be considered with regards to the example shown in Figure 6. In both cases the subscript is assumed to have been transformed to hexadecimal from decimal. For the first case, the address of the first element of the one dimensional array is desired. The NTCW will point to a vector which contains one element signifying a single dimensional array. This element will in turn point to the highest level vector which in this case is level 3. The level number of a vector will be stored in the first two bits of the group link word for that group. It may be recalled that the first 8 bits of the group link comprise the rapid search field of the present scheme which will not be of any use in this scheme. The halfword of the level 3 vector corresponding to the hexdigit 0 is examined. This link will point to a level 2 vector. Again the first halfword of the vector will be accessed to see where it points. The process will repeat until the level 0 vector is reached. In this case the right halfword of the first word is accessed since this corresponds to hexdigit 1. The address part of this link should point to data element

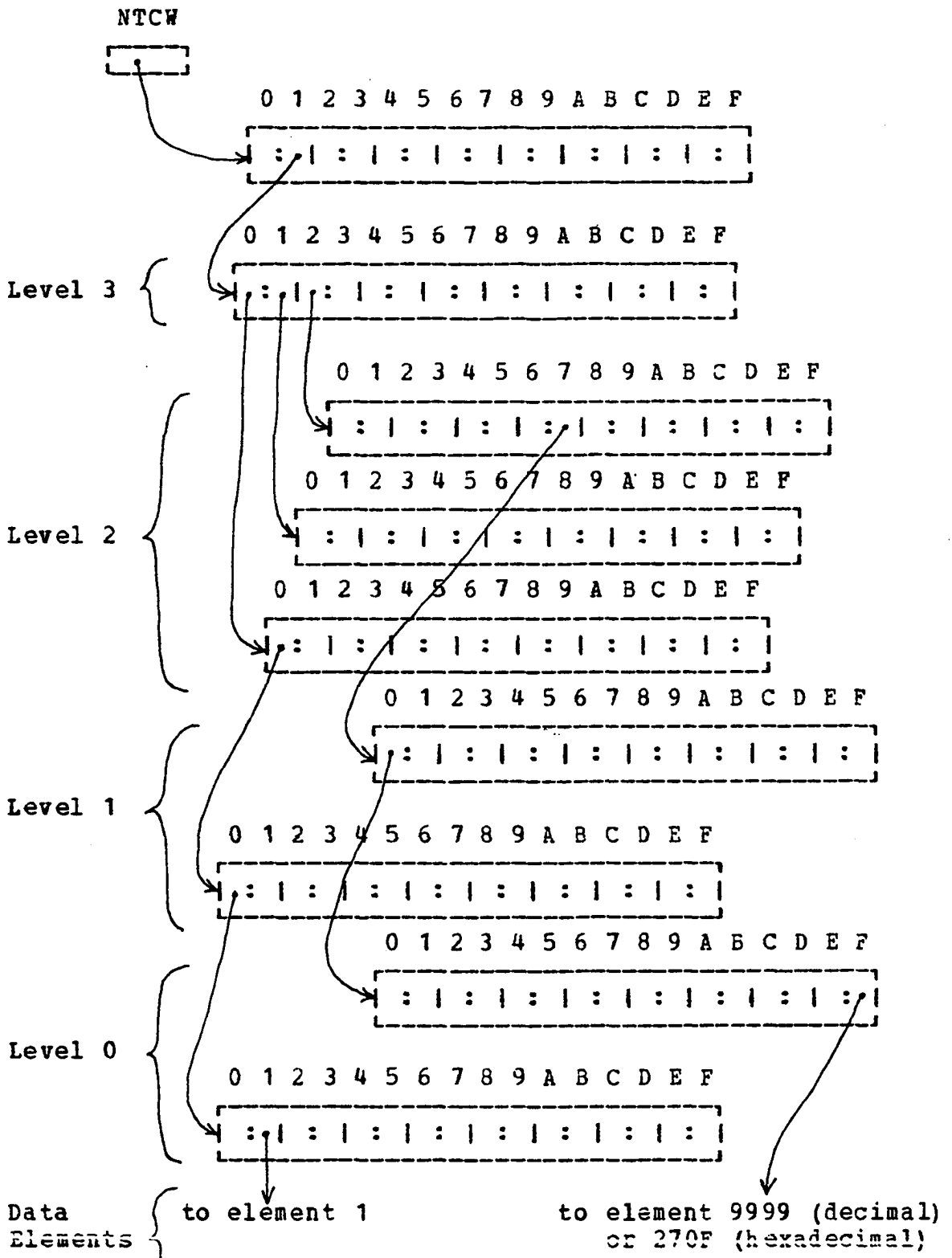


Figure 6. One dimensional array of 9999 elements stored using Pseudo Level Vectors scheme

number 1 of the array. It can be seen that to reach the element from the NTCW five memory accesses are used. Actually six are used since an additional access is used to check the highest level number which is contained in a group link word.

For the second case the address of element number 9999 in base 10 is desired. The subscript in hexadecimal is 270F. Again the NTCW will point to the vector containing only one element. This will in turn point to the level 3 vector. Here the left halfword of the second word in the group is accessed since this link corresponds to the hexdigit 2. Again this points to a level 2 vector. From this vector the 7th pointer is accessed and the address of the level 1 vector can be obtained from this pointer. The link corresponding to hexdigit 0 of the level 1 vector will yield the level 0 vector address. Finally, in the level 0 vector, the last halfword which corresponds to hexdigit F, will contain the address of element 270F in hexadecimal or 9999 in decimal.

In the case of multidimensional arrays, the level 0 vectors will contain pointers which can point to the starting location of data elements or the starting address of another array with same organization of level vectors as in Figure 14. Since there is no limit on the number of subscripts, this repetition of the same organization of level vectors can go on as desired. All of the last set of level 0 pointers should point to data elements.

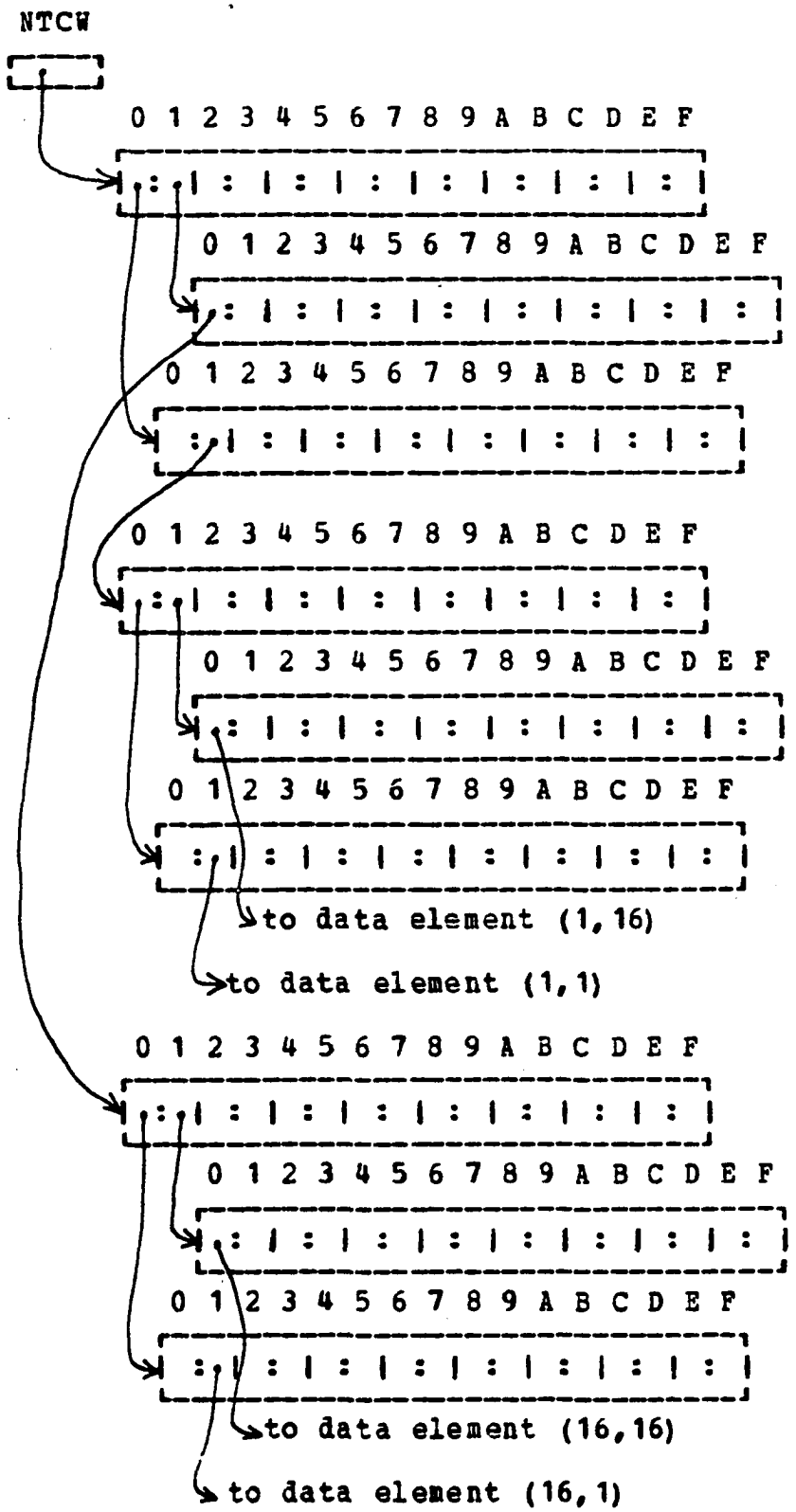


Figure 7. A 16 x 16 array stored using Pseudo Level Vectors scheme

To show the case of arrays with more than one subscript, a 16 x 16 array is shown in Figure 7. Since a decimal 16 transforms to 10 in hexadecimal, the highest level vector will be a level 1 vector. That is why in Figure 7 the NTCW points to the level 1 vector of the vectors corresponding to the first subscript. Only the first word of this level will contain pointers since the array is only 16 x 16 (decimal) or 10 x 10 (hexadecimal). The two level 1 pointers will each point to a level 0 vector. These level 0 vectors will each contain 16 pointers to substructure. There will be 15 links in the vector pointed at by linkword 0 of the level 1 vector, while linkword 1 points to the last link. Each of these 16 links to substructure points to another set of level 1 and level 0 vectors. These correspond to the second subscript. In all these 16 sets the level 0 vectors contain only pointers to data. When referencing a desired element, three memory accesses are used to locate the link to substructure corresponding to the first subscript and another three are used to find the element corresponding to the second subscript. The same number of memory accesses will be used to locate any of the members of the 16 x 16 array.

It can be seen from the first example that the same number of memory cycles, six, are used to get the address of element 1 and element 9999. This scheme is definitely much faster than the two previous schemes described since no

serial accessing and counting down are performed. The address of words within a group may be computed so that no serial accessing of any sort is performed, even within a group. The disadvantage of this scheme is the large amount of overhead space used for linking. A halfword pointer is assigned for each data element. These pointers use up the same amount of overhead as the Modulo 16 scheme. However, over and above this overhead an additional amount of space overhead is imposed by the need for links which correspond to the vectors of level higher than 0.

Implementing this scheme means modifying certain parts of the RP. These will be the same sections modified in the discussion of the Modulo 16 scheme--the Structure Assign section and the Get Address Subscripted section. These are the sections of greatest interest since they respectively perform the storing and accessing of the array elements. The Structure Assign section is modified so that it will take care of creating and organizing level vectors and the associated links in each vector in addition to storing the data elements of the array. The Get Address Subscripted section will transform the decimal subscript to hexadecimal and then search for the desired component through the use of each hexdigit in the transformed subscript in following the path shown by the corresponding pointers in the different level vectors. Again flow charts and detailed descriptions of what

is happening in these sections can be found in section C of the Appendix.

One of the things that needs to be changed is the way the end of a vector is marked. In the two other schemes an extra word in the data storage string of the vector for the present scheme and an extra halfword in the pointer storage string for the Modulo 16 scheme are used to contain the code characters for the end of vector, the F7. In the Pseudo Level Vectors scheme all the halfwords in the group correspond to a hexdigit, therefore using an extra halfword to take care of the end vector is out of the question. What is done is to mark the link code field of the last pointer with an end vector mark, F7, if the pointer points to data, or with an end structure mark, F8, if the pointer points to another vector or structure. In this scheme an ordinary pointer to a data element will be marked with a link to data field, ED, in the link code field while an ordinary pointer to substructure will be marked with a link to substructure, EC. Pointers which do not contain any address should have an all zero link code field. Since pointers in vectors of level greater than 0 can only point to lower level vectors, then they will be marked by setting the first bit of the link code field to one while making the other seven bits zero. This bit will be called the activity bit of the link code field. With this coding any pointer which has not been assigned should have

all zeros in the link code field and in the address field.

IV. SIMULATION OF THE THREE SCHEMES

Three implementation schemes for dynamically variable arrays have been described. One objective of this investigation is to evaluate the present scheme of implementation by comparing it with other possible schemes. The quality of an implementation will be assessed on the basis of two criteria, namely,

1. speed of element access
2. efficiency of storage utilization.

It is desired to find out the cost of having dynamically variable arrays in the SYMBOL 2R computer system. The above mentioned criteria will provide a measure of the cost of dynamic variability in arrays. It is also desired to investigate the trade-offs involved between the two criteria.

It was decided that the approach to take in this investigation is to simulate the three implementation schemes. The simulation of the different schemes will show how the scheme will perform under the same conditions. The simulation will also provide a check on the two alternative schemes of implementation at the flow chart level.

The simulation of the three schemes was achieved by writing programs which simulated the storing and accessing of array elements of the different schemes. A program simulating the Structure Assign routine and a program simulating the Get Address Subscripted were written for each scheme. These

programs were based on the flow charts described in the previous chapter.

The programs were written in the SYMBOL Programming Language (SPL) and run in the SYMBOL 2R computer system. The Structure Assign programs simulated what happened when an array was stored using the three different schemes. The Get Address Subscripted programs did the same thing for accessing a particular array element in arrays which were stored using the different schemes.

The simulation programs followed the previously described flow charts very closely. However, the programs did not actually perform the SYMBOL memory operations that appeared in the flow charts. Instead, whenever a memory operation occurs, the programs only increment counters that keep track of the number of times that particular operation was performed in the routine.

Aside from the different memory operations, the Structure Assign programs also keep track of the number of words and groups that have been assigned and used by data and by links. In this way, the amount of space overhead used when storing a particular array can be found.

This level of detail of the simulation gives all necessary information that is needed in this investigation. A more detailed simulation, where the actual memory operations are performed, could have been done. However, this kind of

detailed simulation would involve a lot more time and effort. The only possible advantage this detailed simulation would give is a more thorough check of the flow charts for the two alternative implementations. A simulation with this level of detail would be in order only if an alternative scheme is selected and is to be implemented in hardware. In this case, the increased checking derivable from the more detailed simulation is a necessity. Therefore for the purpose of this investigation, which is to study the performance of the different schemes, the present level of detail in the simulation has been deemed sufficient.

The simulations were checked and validated by printing out what had happened and what was contained in the relevant counters after each event. Another check was done for the two alternative schemes by using input data for which the results have been previously determined. For the present scheme of implementation, another method of validation was available. One of the convenient features of the SYMBOL 2R computer system is the capability of tracing the memory operations performed by the different processors in the course of processing programs. By running programs which stored and referenced array elements with Central Processor trace turned on, it was possible to get a printed output containing the correct sequence of memory operations performed in each case. The trace output contains the list of memory operations

together with the data transferred and the address from which the data came or the address to which it was going for each memory operation.

It has been mentioned that the simulation keeps track of the number of times each of the different memory operations is used in the course of storing an array or of referencing an array element. It is also of interest to know the total number of memory cycles spent in storing an array or in accessing an array element. Since each of the SYMBOL memory operations consists of from one to several memory cycles, it is necessary to transform each memory operation into memory cycles. Table 1 shows the number of memory cycles used in each memory operation for the different cases possible under each operation. This table is based on the SYMBOL 2R Memory Specifications (19) and the SYMBOL 2R Memory Controller Flow Charts (20).

In Table 1 the memory cycles were obtained by following the Memory Controller flow charts and counting the memory cycles used in each operation. In some memory operations, the number of memory cycles involved depends on where the next address is. The next address could be the next word in the group or the first word of the next group (obtained from the forward link of the group link word). In the case of the Store and Assign operation a third possible next address is obtained by selecting a group from one of the available space

Table 1. Memory cycles used by SYMBOL memory operations (assuming no paging involved)

Memory Operation	Next Address Case	Memory Cycles
AG, Assign Group		3
SA, Store and Assign	next word in group	1
	next group address	2
	select next group	9
FF, Fetch and Follow	next word in group	1
	next group address	2
FR, Reverse Fetch and Follow	next word in group	1
	next group address	3
FL, Follow and Fetch	next word in group	1
	next group address	3
FD, Fetch Direct		1
SO, Store Only		1
SD, Store Direct		1
DS, Delete String		4
DE, Delete to End of String		6

lists. In all these transformations from memory operations to memory cycles, the cases in the flow charts which involved paging were not considered. This is because the simulations were made under the assumption that no paging is involved. The inclusion of the paging cases in the simulation would just increase the number of variables and make the simulation more complicated without increasing the amount of information resulting from the simulation (from the point of view of this investigation).

In the computation of the total number of memory cycles involved in storing an array or in accessing an array element, it was assumed that the next address was the next word in the group except in the case of the eighth memory operation where the next address was assumed in the next group address. This was obtained by accessing the forward link in the group link word and hence involves more memory cycles. In the case of the Store and Assign operation the next address is the next word in the group except for the eighth operation when all the words in the group have been assigned and so the next address is the first address of a group selected from one of the available space lists in the same page.

V. RESULTS AND DISCUSSION

One of the objectives of this investigation is the evaluation of the present scheme of implementing dynamically variable arrays. An evaluation of the present scheme must start with studying the effectiveness of the two speed-up techniques used. These techniques are the current pointer and rapid search routines.

All the results of the simulations that will be shown in the graphs for the present scheme and for all the other schemes were obtained by assuming the field lengths of the array elements were all one word long. The effect of increasing field length of array elements in the access time characteristics of some schemes will be a linear increase in access time as the field length is increased. In other schemes the access time characteristics will be independent of the field length variation. Increasing the field length of the array elements will increase the amount of space occupied by data but will not affect the amount of space overhead used by an implementation scheme.

Data on the performance of the present scheme with or without the use of either or both speed-up techniques were generated using the Get Address Subscripted routine simulation program. The average times to access elements of a one dimensional array for varying array sizes were generated. Only a one dimensional array was used because it was enough

to show the access time characteristics of the scheme. Accessing an element of a multidimensional array only means going through the same algorithm several times, with the number of times depending on the number of subscripts. Since the use of the current pointer is dependent on the manner of accessing, data were collected for two cases whenever the current pointer was activated. These two cases are the best and the worst case possible. The best case takes place when the array elements are accessed in sequence starting with the first element, while the worst case happens when the array elements are accessed in reverse sequence starting with the last element. In the best case the current pointer is always used, while in the worst case the current pointer is useless in accessing from element 99 down to element 1.

Figures 8a and 8b show the average access time characteristics for the different cases mentioned above. Access time will be defined as the time needed to locate a desired element of the array and to return its address. Figure 8a shows the characteristics for array sizes 10 to 150, while Figure 8b shows the characteristics for array sizes 100 up to 1000.

In Figure 8a the case of no current pointer and no rapid search has an access time characteristic that increases linearly with array size and reaches the highest point among the cases shown. This characteristic reflects the condition that

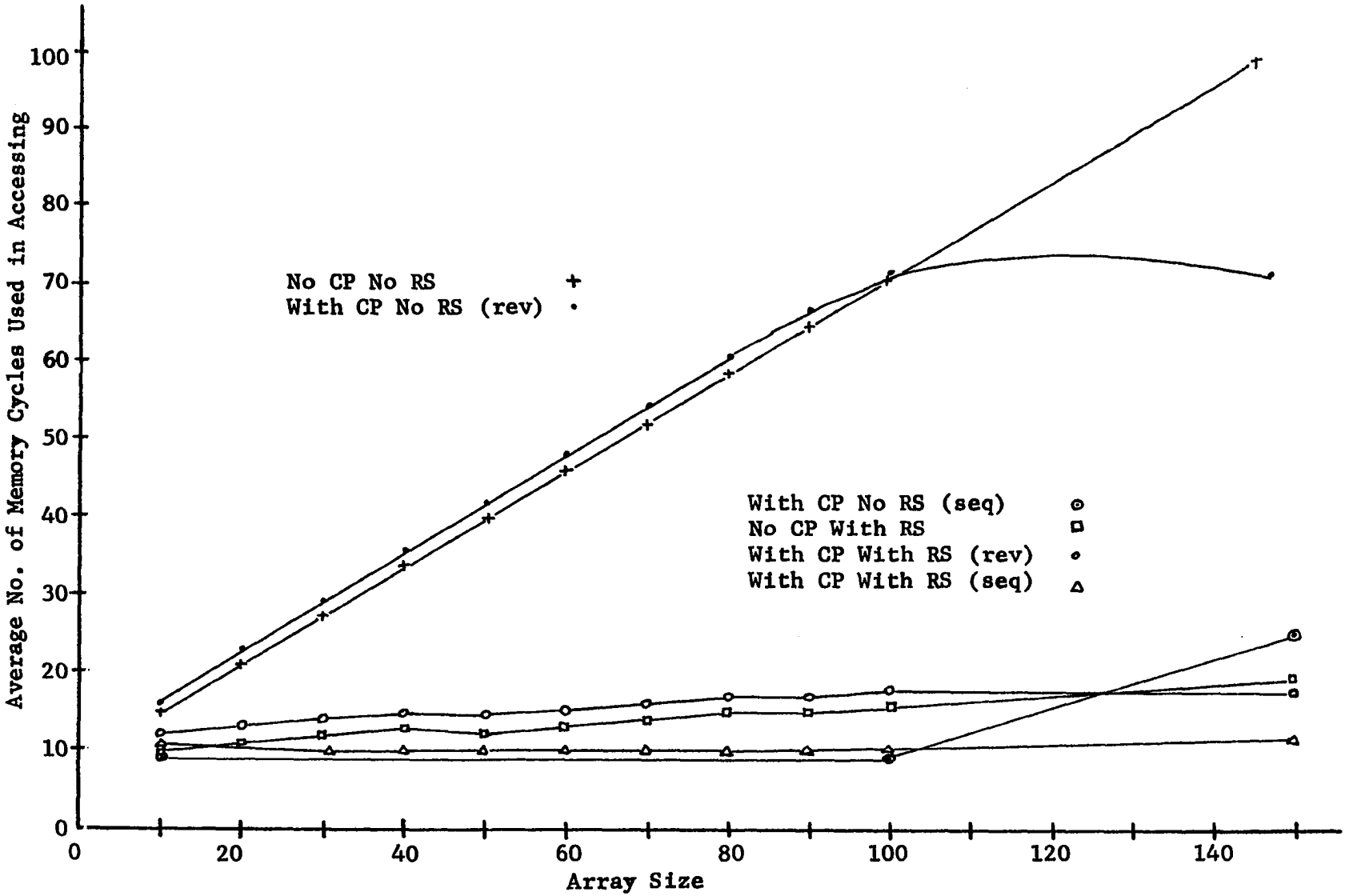


Figure 8a. Present scheme average accessing time characteristics

only the word scan routine is being used. The average access time increases as the array size increases since a larger array means more words to scan through. Above the no current pointer and no rapid search curve is the characteristic for the case when only the current pointer is used and the accessing is performed in the reverse sequence. For array sizes from 10 to 100 the reverse accessing current pointer only access time characteristic is higher than the word scan only case. This is so because accessing in reverse sequence with the current pointer on means memory cycles are spent in updating the current pointer which in turn does not lead to any speed-up. However, once the array size goes to 100 and above the current pointer can not go higher than 99 and so it helps by bypassing the first 99 elements when accessing beyond element 99. Thus the current pointer enabled reverse accessing characteristic becomes better than the word scan only accessing characteristic beyond array size 100.

In the region representing array sizes less than 100 the best characteristic is shown by the sequential accessing with only the current pointer enabled. The reason behind this is that the next element accessed is always located in the next word in the array storage string. Also no memory cycles are spent in looking at the group link words since the rapid search is not activated. The characteristic shows a constant average number of memory cycles used in sequentially access-

ing elements for array sizes up to 100. Beyond 100 the performance gets worse since the current pointer is only good up to 99.

With the current pointer turned off and the rapid search activated, the characteristic is much better than the word scan only mode. With only the rapid search on, it does not matter whether the accessing is done in sequence or not.

Now consider what happens if both current pointer and rapid search are activated. In the sequential accessing case, the characteristic with both speed-up techniques used is only about 2 memory cycles slower than the current pointer only mode in the region of array sizes less than and up to 100 elements. It becomes much better in the region beyond 100 elements. In accessing in reverse sequence, having both current pointer and rapid search on has a slightly poorer performance than the rapid search only mode. Again performance becomes better when array sizes greater than 100 are reached.

Figure 8b shows the access time characteristics for array sizes beyond 100 elements and up to 1000 elements. Here again the no current pointer and no rapid search mode has the worst characteristic. The next worst characteristics are those cases when only the current pointer is enabled. In this mode, when accessing is done sequentially, the characteristic is almost parallel to the word scan only character-

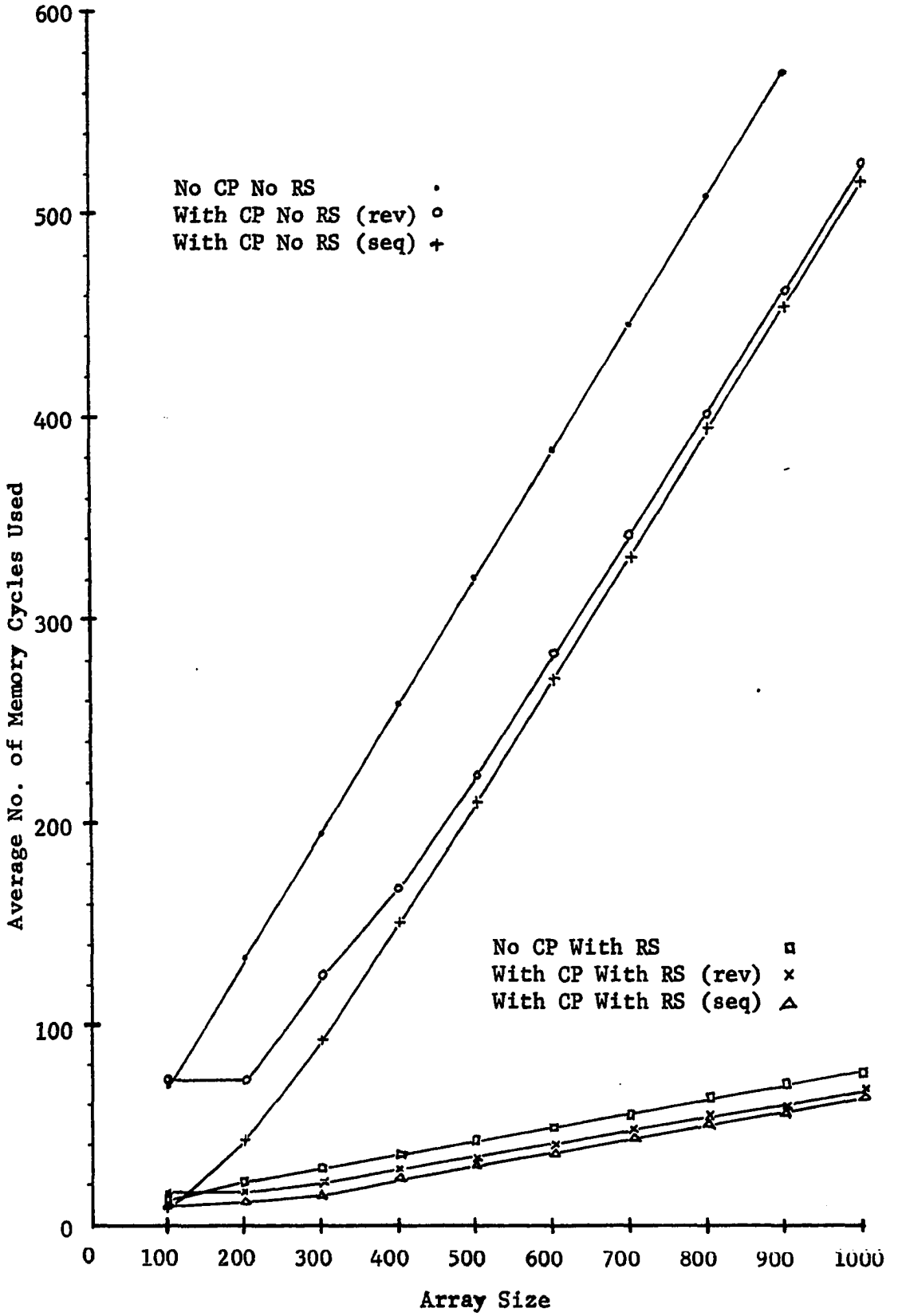


Figure 8b. Present scheme average accessing time characteristics

istic for array sizes beyond 200 elements. The offset can be accounted for by the saving in memory cycles resulting from the accelerated accessing of the first 100 elements when sequential accessing is done. When the accessing is performed in reverse sequence, the performance characteristic is bad compared to the sequential accessing characteristics for arrays up to 300 elements. Beyond that value the discrepancy between the two modes of accessing decreases as the effect of the first 100 elements goes down in the averaging. This is the reason why the two characteristics approach each other as the array size goes higher.

Turning on the rapid search alone leads to a much better access time characteristic than any of the previous ones mentioned. The characteristic is approximately 8 times faster than the charcteristic for the no current pointer and no rapid search. This follows the predicted performance resulting from use of this technique.

Additional improvement in access time characteristics can be obtained if the current pointer is turned on at the same time as the rapid search. The improvement over the rapid search only mode is due to the speed up in the sequential accessing of the first 100 elements. Accessing in reverse order results in only a slightly poorer characteristic. Here again it can be seen that the difference between accessing in sequence and in reverse sequence becomes smaller

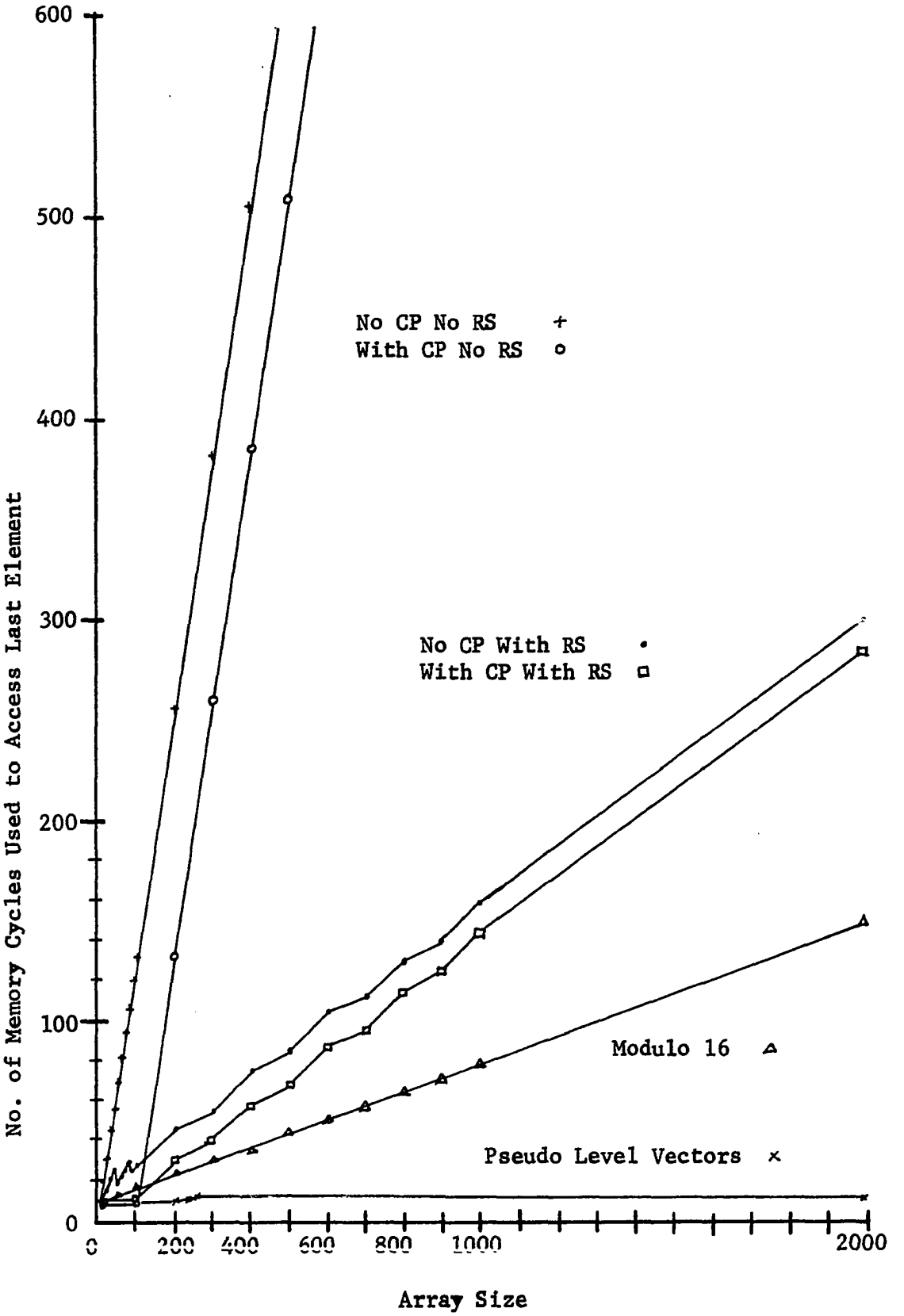


Figure 9. Last element accessing time characteristics

as the array size increases and the weight of the first 100 elements in the average goes down.

Now consider the two alternative schemes proposed in order to see how their performance compares with the performance of the present scheme. Figure 9 shows the last element access time characteristics of the present scheme, the Modulo 16 scheme and the Pseudo Level Vectors scheme. For the present scheme several cases are shown. These cases represent the different combinations of speed-up techniques used. Sequential access data is used for the present scheme since it represents the best possible performance. In the two other schemes the mode of accessing does not affect the access time characteristic. Remember that the access time here is not average but rather the actual number of memory cycles used in order to reach the last element of an array with the array size as the independent variable.

The present scheme characteristics are the same as those shown in the two previous figures. It should be noted however that the characteristics when the rapid search is enabled are not smooth. The reason behind this is that the last group in the array storage string is always word scanned instead of being rapid searched. The number of words that are word scanned depends on where the last element is stored in a group. And so for small arrays this causes the erratic behavior of the access time characteristic.

The best case possible for the present scheme is when both current pointer and rapid search are enabled. Comparing this characteristic with the Modulo 16 scheme will show that the alternative scheme has a better performance. The Modulo 16 scheme performance characteristic is about 2 times faster than the present scheme. However, a look at the Pseudo Level Vectors scheme access time characteristic will show that it is much better than all the other schemes.

The slopes of the characteristics on the access time against array size plane show the degree of dependence of a scheme's access speed on the array size. The present scheme using no current pointer and no rapid search is the most dependent on array size and hence it rises most rapidly. The use of the current pointer does not affect the slope, instead it only shifts the characteristic lower by a fixed amount. The rapid search routine cuts down the slope to 1/8 of the original value. The Modulo 16 scheme represents another cut by a half of the slope of the rapid search activated characteristics.

The Pseudo Level Vectors scheme represents the most significant reduction in slope since the slope of its characteristic is almost horizontal. It is not actually horizontal. The access time increases by one memory cycle for every increase in power of 16 of the number of elements in the array. Thus the access time characteristic increases by 1 at 16,

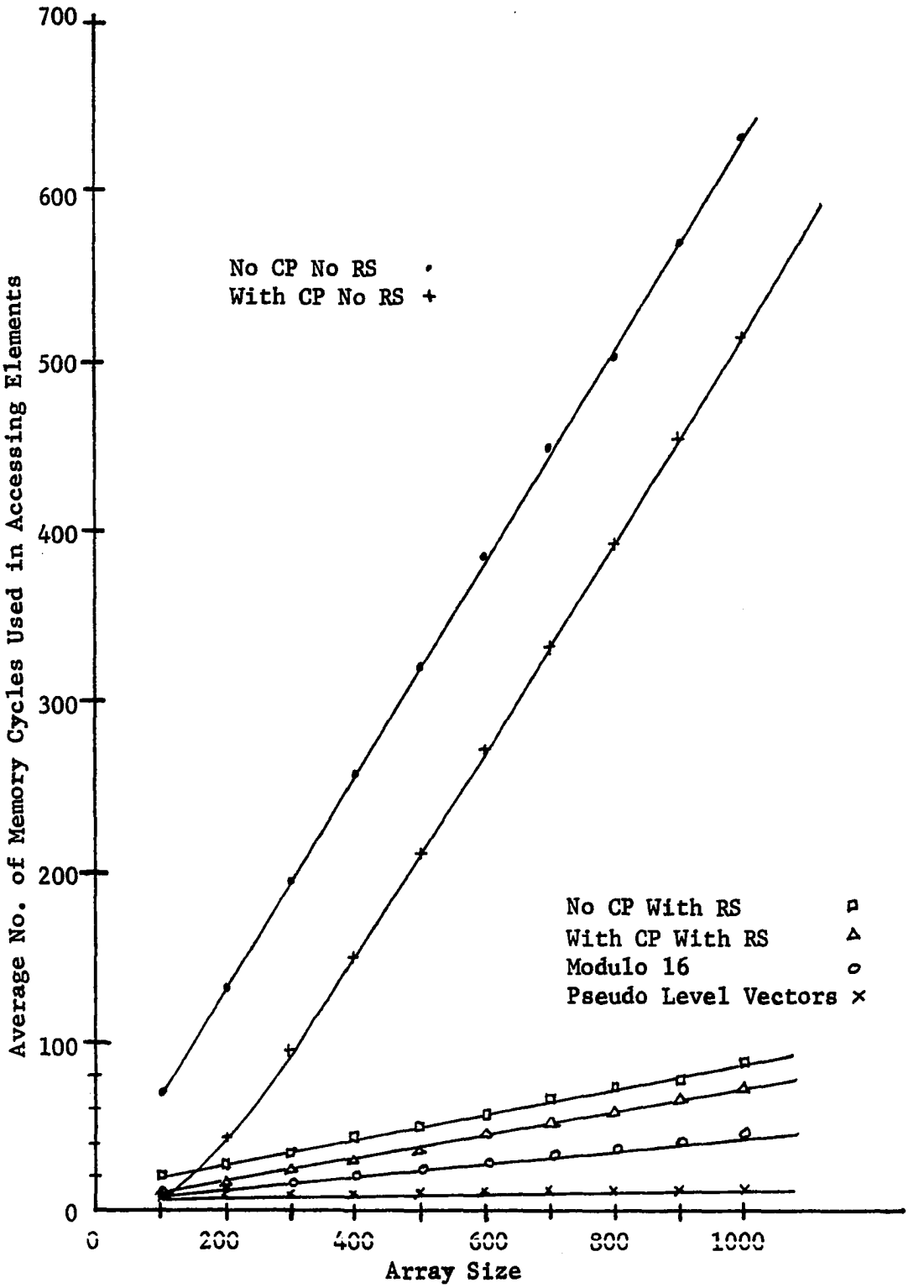


Figure 10. Average accessing time characteristics

256 and 4096. The Pseudo Level Vectors scheme is for all practical purposes independent of the array size. This scheme also guarantees that any array element from 1 to 9999 can be accessed within 14 memory cycles. Actually, all the elements of an array from the first to the last will be reached in a definite number of memory cycles. The actual number of memory cycles depends on the array size.

Figure 10 shows the comparison of average access time characteristics between several cases of the present scheme and the two alternative schemes. The characteristics with rapid search turned on are smoother and less erratic than in the previous figure as a consequence of the taking of averages. Again it can be seen that the beneficial effect of rapid search, Modulo 16 scheme and the Pseudo Level Vectors scheme is the reduction in the degree of dependence on array size as evidenced by the decline in slope of their respective characteristics.

It must be remembered in Figures 9 and 10 that the characteristics were obtained while assuming that the field length of each of the array elements is one word. The access time characteristics for the present scheme gets progressively worse as the array elements increase in field length. On the other hand, the Modulo 16 scheme and the Pseudo Level Vectors scheme have been set up so that the access time characteristics are independent of the field length of the array

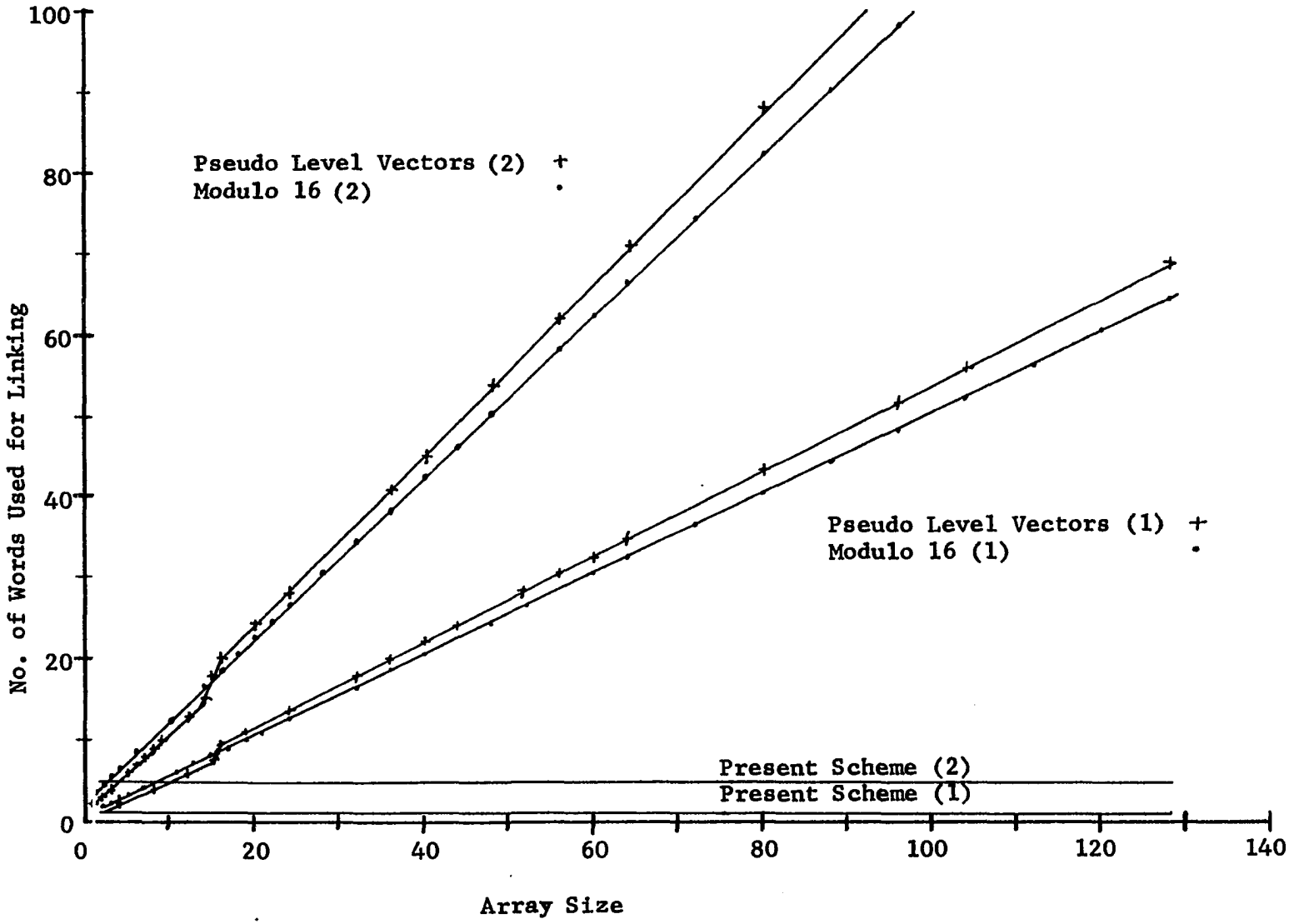


Figure 11. Space overhead characteristics

elements.

It seems from Figures 9 and 10 that the access time characteristics of the Pseudo Level Vectors scheme is the best among the alternatives. However, before immediately proclaiming the Pseudo Level Vectors scheme as the best implementation scheme the other criterion for determining the quality of an implementation should be considered.

Looking at Figure 11 will show how the different schemes perform from the point of view of this other criterion. This figure shows the space overhead characteristics for the three different schemes for varying array sizes. Two cases are considered for each scheme. The first case is for a one dimensional array, a 1 x n array. The other case is for a 2 x n array. In both cases the array size is increased by increasing n, the number of columns.

At first glance the space overhead characteristics seem to show the same thing as the access time characteristics. That is the different schemes cause a reduction in slope of the characteristics or that the degree of dependence on the array size is reduced by the different schemes. However, a closer look will reveal that the two extreme schemes have interchanged their positions. The present scheme of implementation has the best space overhead characteristic while the Pseudo Level Vectors scheme has the worst. The Modulo 16 scheme's utilization of space overhead is almost as bad as

the Pseudo Level Vectors scheme.

The space overhead characteristic of the present scheme is horizontal. It is independent of the number of elements. On the other hand, the two alternative schemes' space overhead characteristics are very dependent on the array size. The Modulo 16 scheme characteristic shows a linear variation of space overhead with array size. This is due to the assignment of a half word link for every element in the array. The Pseudo Level Vectors scheme space overhead shows about the same linear dependence on the number of elements in the array. A jump or discontinuity is noticed when the number of elements in the array is 16. This is accounted for by the creation of a new level vector for every power of 16. Other jumps will occur at 256 (16^2) and 4096 (16^3) if the axes of the graph are extended sufficiently.

Now consider the effect of increasing the number of subscripts or dimensions of the array. There seems to be a uniform effect for all three schemes. Increasing the number of subscripts to two results in approximately doubling the amount of space overhead used in each scheme. This is due to the creation of additional linking as the number of dimensions increase.

The storage efficiency of a scheme will be defined as the number of words assigned to data elements divided by the total number of words assigned including link words. The

Modulo 16 scheme will have a fairly constant storage efficiency since its overhead characteristic varies linearly with array size. The Pseudo Level Vectors scheme will also have fairly constant storage efficiencies at the linear portions of the characteristic. Changes in storage efficiency will occur at the jumps at 16, 256 and 4096. Finally, the present scheme will have a storage efficiency that will improve as the array size increases since the number of words used for linking remains the same with increasing array size. The storage efficiency mentioned here is with respect to Logical Storage. It must be remembered that Logical Storage by itself uses 12.5% of the virtual storage as overhead for Logical Storage administration (13).

The storage efficiencies for some array sizes were computed for the different schemes. The array sizes used were 10, 100 and 1000. As expected the storage efficiency of the present scheme started out very good for the small array and became excellent for the larger arrays. It started at 91% for the 10 element array and became 99% for the 100 element array and was 99.9% for the 1000 element array.

The storage efficiencies of the two other schemes were much lower since they use up a lot more space overhead for linking. The Modulo 16 scheme had storage efficiencies that ranged from 64.5% for the 10 element array and became 66.6% for the 1000 element array. The Pseudo Level Vectors scheme

has about the same efficiency. Its storage efficiencies ranged from a low of 64.9% to a high of 66.6%. These efficiencies are with respect to Logical Storage. In order to find the efficiency with respect to virtual storage the Logical Storage efficiencies must be multiplied by 87.5%. This is the efficiency of Logical Storage with respect to virtual storage.

It must also be mentioned that the present scheme minimizes group wastage since it stores the elements one after another in one storage string. On the other hand, in the two other schemes the group wastage can be considerable since each data element is assigned one storage string. The effect of this is to make the access time characteristic independent of the field length of the elements.

The results regarding the performance of the three schemes are summarized in Figure 12 which shows the space time characteristics. Here the access time is plotted against space overhead with array size as the third parameter. It must be kept in mind that these are all for a single dimensional array. Two cases are actually shown for the present case even though these two cases lie on the same straight line. The first case represents the present scheme with no current pointer nor rapid search, while the other case represents the present scheme with both current pointer and rapid search activated. The two cases lie on the same

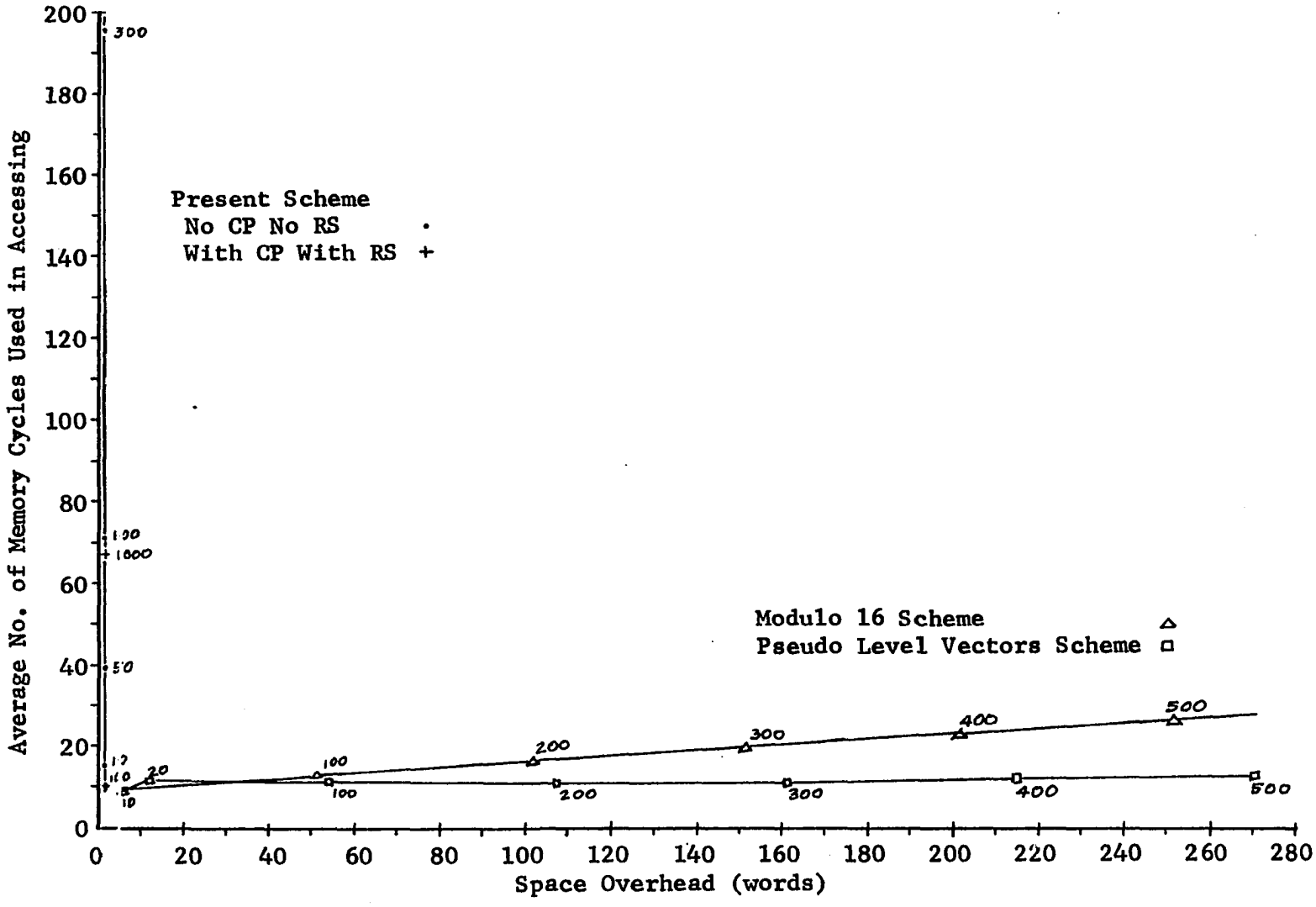


Figure 12. Space time characteristics

line since they use up the same amount of space overhead. However, it must be remembered that this is Logical Storage overhead. The rapid search technique uses the spare bits in the virtual storage overhead used for Logical Storage administration. The effect of the use of current pointer and rapid search is to collapse or shrink the characteristic into a shorter vertical line. Their effect is to approximately speed up the sequential access time by about ten times.

In spite of these two speed-up techniques the present scheme is very slow compared to the Modulo 16 and the Pseudo Level Vectors scheme. However, the speed in accessing of these two schemes results from the use of a large amount of space overhead. Clearly the two schemes have traded increased space overhead for faster element accessing speed.

The Pseudo Level Vectors scheme represents a very good implementation from the point of view of the speed of element access criterion, but it is a poor implementation from the point of view of storage efficiency. On the other hand, the present scheme is a very good implementation in terms of storage efficiency but it is poor from the other point of view. The present scheme can be characterized as a space saving and time consuming implementation, while the Pseudo Level Vectors scheme can be described as a time saving and space consuming implementation.

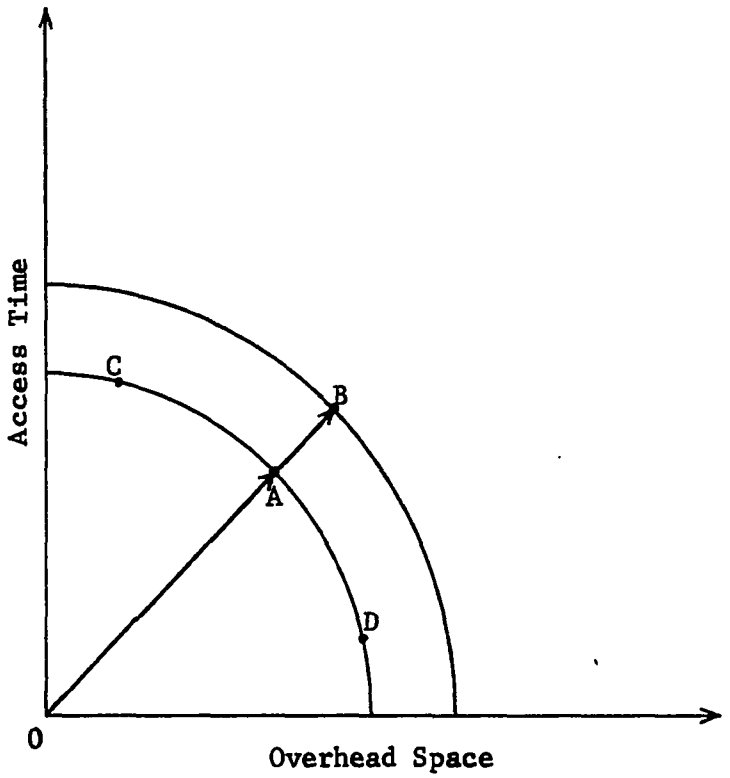


Figure 13. Sample space time characteristic points

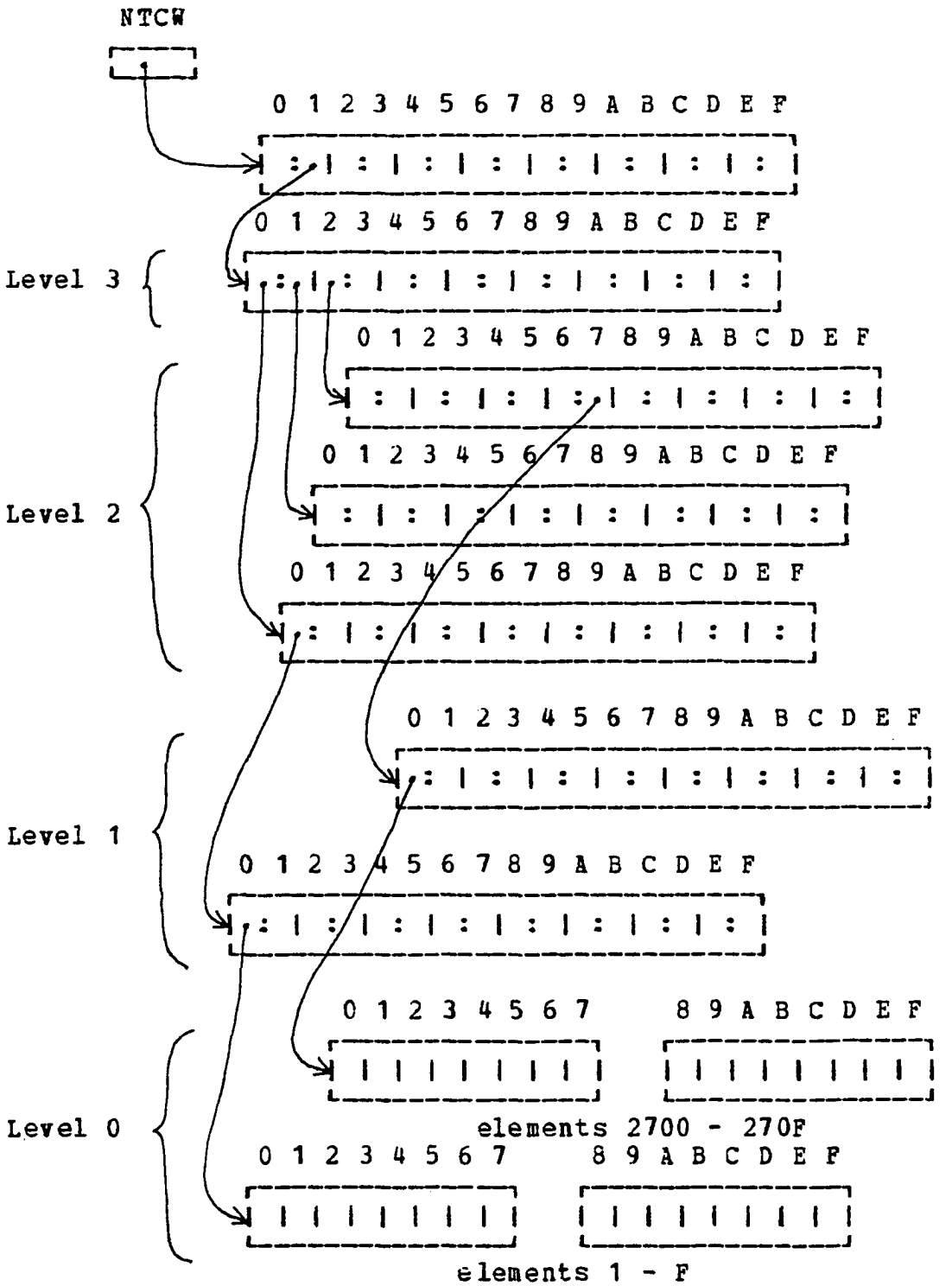
Since the access time and the overhead space are used to assess the quality of the implementation schemes, the graph of the access time against overhead space should reveal the characteristics of a desirable implementation. In this graph the distance of a particular array size point from the origin is the measure of the quality of an implementation. Consider Figure 13 as an example. In this figure assume points A and B are characteristic points of two different schemes for the same array size. Clearly, the shorter the distance from the origin the better is the implementation. Implementation A is better than B since it has faster access time and it consumes less space overhead.

However, for implementations whose characteristic points for the same array size have the properties of being equidistant from the origin and of having different access times and overhead spaces, deciding which is the better implementation is not as straight forward. As an extreme example consider points C and D in the figure. For some applications which place a high premium on memory space, implementation C is more desirable since it uses less space overhead even though it is slower in accessing elements. On the other hand, applications placing emphasis on fast processing times would find implementation D more attractive since it gives faster access time at the expense of higher memory space overhead. All this goes to show that the desirability of a scheme as

based on the two criteria is very much dependent on the weight assigned to the criteria. The weighting will also affect the trade-offs involved between time and space. As long as the criteria have the same weight, the closer the characteristic point of a scheme is to the origin the more desirable is the scheme.

Turning back to Figure 12, it seems that a desirable implementation should be able to access any array element within 20 memory cycles and yet avoid using up space as fast as the Pseudo Level Vectors scheme. A possible way to reconcile these two requirements could be obtained by combining the best features of the present scheme and the Pseudo Level Vectors scheme. The Pseudo Level Vectors scheme achieves its good access time characteristics by the use of more linking. The level 0 pointers are what use up most of the space overhead. The present scheme uses a minimum space overhead by storing array elements one after another in a storage string. If the level 0 links are dispensed with and if the present scheme is used in storing the array elements, then maybe a more desirable implementation will result.

By retaining only the level vectors higher than 0, the space overhead is reduced considerably. To satisfy the search procedure that is keyed by the hexdigits of the subscript, the array elements must be stored one after another in blocks of 16 in each storage string. Figure 14 shows how



Level 0 consists of storage strings of 16 one word long data elements (2 groups)

Figure 14. One dimensional array of 270F (hex) elements stored using the proposed scheme

this proposed scheme will store and link a single dimensional array of 9999 elements.

Under this proposed scheme, accessing an element is performed in the same manner as the Pseudo Level Vectors scheme until level 0 is reached. In this proposed scheme level 0 will contain actual data elements instead of another set of pointers. These data elements will be stored one after another in groups of 16 in each storage string. In this way the last hexdigit of the subscript is used to locate the data element in the storage string. This element can be obtained by use of the rapid search technique in that particular storage string. In Figure 14 the data elements are one word long so that the elements in the first group of the storage string can be found in one rapid search fetch of the group link word and those elements in the second group could be located in another fetch of the group link word. In the cases where there are more than one subscript, the level 0 can contain links to substructure in addition to actual data elements.

Since the level 0 pointers in the Pseudo Level Vectors scheme constitute the biggest user of space overhead, the removal of these pointers results in tremendous savings in space overhead. By grouping elements in blocks of 16 and using rapid search, an element can be accessed within 15 memory cycles provided all the elements are only one word in

field length. One difference between this proposed scheme and the Pseudo Level Vectors scheme is the dependence of this proposed scheme on the field length of the data elements. Increasing the field length of the data elements will mean increasing the number of words to be rapid searched. The effect of this linear dependence is minimized since there are only 16 elements per storage string to be rapid searched. This proposed scheme will need to use one word to mark the end of a storage string with less than 16 elements.

To compare the performance of this proposed scheme, the space overhead and average access time were computed for array sizes of 10, 100, 1000 and 9999. Figure 15 shows the performance of this scheme as plotted with the overhead space access time characteristics of the other schemes. The proposed scheme is only a couple of memory cycles slower than the Pseudo Level Vectors scheme and still much faster than the Modulo 16 scheme. The other good thing about it is that it consumes only about one tenth of the space overhead used up by the Pseudo Level Vectors scheme.

The storage efficiecies of this proposed scheme were computed for the array sizes of 10, 100, 1000 and 9999. The efficiencies ranged from a low of 91.7% to a high of 96.5%. Again these efficiencies were computed with respect to Logical Storage. They will have to be multiplied by 87.5% in order to obtain the efficiency with respect to virtual stor-

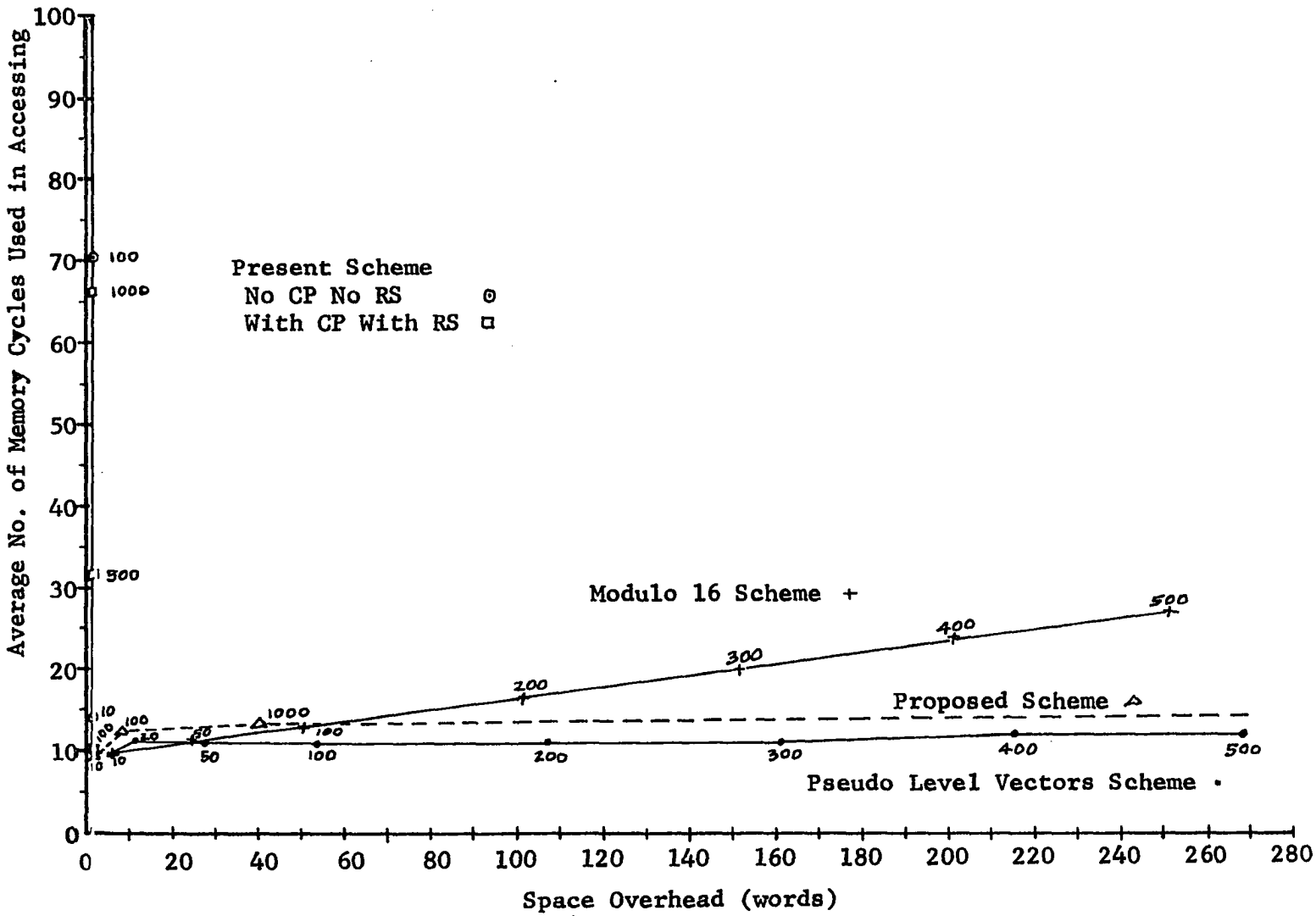


Figure 15. Space time characteristics including proposed scheme

age.

This proposed scheme does not have as much group wastage as the Pseudo Level Vectors scheme since data elements are stored one after another in their respective storage strings. Group wastage in this scheme will be more than that for the present scheme since every 16 data elements occupy one storage string instead of having all data elements in one storage string.

All the preceding paragraphs have shown that this proposed scheme is a much more desirable implementation than any of the other schemes since it is able to access any of the array elements within 15 memory cycles (for one word long data elements) and still keep its storage efficiency with respect to Logical Storage at around 96% or about 84% with respect to virtual storage.

Throughout this chapter the implementation schemes have all been assessed on the basis of speed of element access and storage efficiency. It must be kept in mind that no claim is made that these are the only two criteria that must be used in selecting a scheme of implementation. There are other considerations which can affect the choice of a desirable implementation scheme. For example, the present scheme of implementation is the most flexible among the schemes that have been considered. This flexibility is due to the fact that there are generally no links pointing to the array elements.

An additional element can easily be inserted in the storage string of array elements and the only effect of this will be to push back all the elements after the inserted element in the numbering sequence. In other words, the net effect is to increment the subscripts of the succeeding elements by one. On the other hand, an element can be deleted from the middle of the storage string and its sole effect will be to move all succeeding elements up by one in the numbering sequence. This means that the elements coming after the point of deletion will now be referenced by their old subscripts minus one. This insertion or deletion of elements is practically impossible in all the other schemes since a specific pointer is assigned to each element or blocks of elements. Associated with the location of this pointer is the subscript of the array element. And so insertion or deletion of elements within the array will put the numbering and locating system out of order.

Another consideration which may have some weight in the choice of a desirable scheme is the amount of hardware logic that will be necessary to actually implement the scheme. Judging on the basis of the amount of code needed for the simulations, the first two schemes may require roughly the same amount of logic to implement. The Pseudo Level Vectors scheme needs more hardware to keep track of all the linking going on. However, the proposed scheme will require more

logic to implement since it essentially combines the present scheme and the Pseudo Level Vectors scheme. As such, the proposed scheme must have the logic for each scheme in order to function properly.

Finally, another possible consideration could be the type of arrays that are going to be dealt with. Though the present scheme is generally the most efficient scheme in terms of storage utilization, the case of sparsely populated arrays is an exception. In the present scheme the representation of a null element takes up one whole word in the storage string. The Modulo 16 and the Pseudo Level Vectors schemes use a halfword to represent a null element, so that they are more efficient than the present scheme in this case. For extremely sparsely populated arrays the Pseudo Level Vectors scheme becomes more efficient since it might be able to stop the allocation of pointers at a level higher than 0. For these extreme cases even the proposed scheme is more efficient than the present scheme.

VI. SUMMARY AND CONCLUSIONS

In the SYMBOL 2R computer system an array has the ability to change in shape and the elements can grow or shrink in size. Because of this dynamic variability of the array elements and because of the way memory is organized in the system, it is impossible to predict the location of the next array element. As a result the address computation accessing scheme used in conventional computer systems is not applicable here.

The only way to reach the array elements in this kind of environment is to create paths that have to be traversed to reach the elements. In order to reach a point or an element as quickly as possible, a direct path from the root or starting point to that desired point must be created. An implementation which is minimizing the access time to reach the array elements must create separate direct paths to each of these elements. The paths are created by means of links and at the expense of memory space. An implementation scheme that seeks to minimize the memory space overhead must reduce the number of paths that have to be created.

The present scheme of implementation has taken the direction of minimizing the memory space overhead. As a result this scheme creates only a single path which passes through all the actual elements. It does not take too many memory cycles to reach the elements near the root of the path. How-

ever, as one gets farther down the line it takes more and more time to reach the elements. The obvious disadvantage of this from the point of view of speed of element access is that all the elements preceding the desired element must be passed through before the desired element is reached. Since the scheme passes through the complete element, the performance becomes worse as the length of elements increases.

Because of this disadvantage, two techniques have been used to speed up the accessing of array elements. The basic idea behind the current pointer technique is to skip through the elements that have been traversed in the previous access when accessing elements in sequence. However, the results in the previous chapter show that this technique is of limited effectiveness. Its effectiveness is limited by two conditions. It is only good up to 100 elements and it is useless if the accessing is not done in ascending sequence.

Much more effective is the rapid search technique. This technique essentially accelerates the traversal of the path to the elements. This is achieved by creating a shorter path composed of bits which have a one to one mapping with the words in the actual and longer path. The starting points of the data elements are marked in this shorter path. This rapid search technique is limited by the fact that a single path is traversed and so elements preceding the desired elements have to be passed through.

The Modulo 16 scheme is at least twice as fast as the rapid search technique in accessing speed. In this scheme the speed-up is attained by having the path to be traversed contain only the starting address of the data elements and not the actual elements themselves. Thus this scheme avoids going through the whole elements by just traversing the links to the actual elements. In spite of this faster speed, the performance of this scheme is also limited by the condition that there is only one path that can be traversed. This scheme probably represents the fastest accessing speed attainable with the use of a single path. It suffers from the same disadvantage of having to pass through the elements preceding the desired element. The Modulo 16 scheme also uses more overhead space than the present scheme.

In order to improve the access speed much more than that attained by the Modulo 16 scheme, it is necessary to create paths that lead to the elements without passing through the other elements. This is the approach taken by the Pseudo Level Vectors scheme. It creates a separate path from the starting point to each of the elements. The Pseudo Level Vectors scheme uses the hexdigits of each subscript to trace through the path to each element. Because of these separate paths to each element, this scheme has the best access time performance characteristic. However, this is achieved at the expense of considerable space overhead.

The proposed scheme, which was described in the previous chapter, is an implementation which looks good from both the speed of element access and the storage efficiency criteria. Faster element access is attained by more than one path to the elements. The storage efficiency is kept high by not creating a path for each element. The elements are grouped into blocks of 16 and each block has a separate access path. The rapid search technique speeds up the accessing once a block of 16 elements in one storage string is reached.

Several implementation schemes for dynamically variable arrays have been described. These schemes were simulated to get data on their performance characteristics. The resulting performance characteristics with respect to two criteria have also been presented. These two criteria are the speed of element access and the storage efficiency of each scheme. These are essentially measures of the cost of implementing dynamically variable arrays.

The present scheme has been shown to be the most efficient in the use of storage. However, this is achieved at the cost of access time. The Modulo 16 scheme displays both faster accessing and reduced storage efficiency. The Pseudo Level Vectors scheme shows that the access time cost can be significantly reduced at the expense of storage efficiency. These results show that there is a trade-off between the time and space cost as represented by the two criteria used in

assessing each scheme. Any improvement in performance with respect to one criterion is attainable only at the expense of some degradation in performance with respect to the other criterion.

However, a desirable implementation must have good performance with respect to both criteria. And so a scheme is proposed which gives fast element accessing speed and which also retains a high storage efficiency. This is clearly the most desirable among the schemes of implementations if the two criteria for assessing the quality of an implementation are given equal weights. This proposed scheme also demonstrates that it is possible to achieve the benefits of dynamically variable arrays at a lower cost than any of the other schemes.

Even though all the performance characteristics were obtained under the assumption of no paging, the proposed scheme will have some implications on the paging rate of the Reference Processor (RP). The RP, with the present scheme of implementing dynamically variable arrays, has one important problem with regard to handling large data arrays. When a particular array member is required, the RP searches through the array of variable length fields to reach that array member. If a page-out is encountered, the Central Processor (CP) shuts down on that terminal until the required page has been loaded into the core memory. When the CP, and subse-

quently the RP, is started up on the terminal once again, the RP begins the search process over again, instead of starting from where the page-out was encountered. Consequently, if the array is larger than the amount of core memory allocated to that terminal, no forward progress will be made.

This problem indicates that in order to reach a desired element located in a certain page, the present scheme requires a working set (21) consisting of that particular page and of all the pages containing the elements preceding the desired element. If this working set size is larger than the allocated core memory, then thrashing (22) occurs.

On the other hand, the proposed scheme needs only the page or pages containing the level vectors and the page containing the desired element. The proposed scheme can reach the desired component through the linking in the level vectors and so it does not need to search through the pages containing the preceding elements. Therefore, the proposed scheme requires a smaller working set in order to reach the array elements even for large arrays. The working set size can be minimized by concentrating the level vectors in as few pages as possible. In this scheme once the page containing the required element is loaded into core memory in addition to the page of level vectors, the RP can begin the searching process over again and find the desired element without generating a page-out. As a result, the use of the proposed

scheme will allow the BP to handle large data arrays even without having restart capability (this is the direction taken by the current effort to solve this problem) or software for breaking large arrays into manageable sizes.

The results of this investigation indicate that the next step in the work on the SYMBOL array handling capability is a software implementation of this proposed scheme. Hopefully, the performance of this software implementation will lead to a hardware implementation.

VII. ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr. Roy J. Zingg for suggesting the problem and for providing help and detailed guidance in the course of this investigation.

I am also grateful to Dr. Alvin A. Read for his advice and encouragement throughout my graduate education both at the University of the Philippines and at Iowa State University. I would also like to thank Perry Hutchison for reading the thesis and for making valuable comments and suggestions.

I would like to acknowledge that my graduate education was supported by a Ford Foundation fellowship awarded through the Educational Projects, Inc. and the University of the Philippines. The research itself was supported by the National Science Foundation under Grant No. GJ33097.

VIII. REFERENCES

1. Sitton, G. A. "Operations on Generalized Arrays with the Genie Compiler." Comm. ACM, 13 (May, 1970), 284-6.

2. Burks, Arthur W.; Goldstine, Herman H.; and von Neumann, John. "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument." Vol. 5 of Collected Works of John von Neumann. Edited by A. H. Taub. New York: The Macmillan Co., 1963.

3. Hellerman, H. "Addressing Multidimensional Arrays." Comm. ACM, 5 (April, 1962), 205-7.

4. Brillinger, Peter C., and Cohen, Doron J. Introduction to Data Structures and Non-Numeric Computation. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1972.

5. IBM System/360 Operating System PL/1(F) Language Reference Manual. Order No. GC 28-8201-39, IBM Corporation, White Plains, New York, June, 1970.

6. Iliffe, J. K. Basic Machine Principles. 2nd ed. New York: American Elsevier, Inc., 1972.

7. Organick, Elliot I. Computer System Organization: The B5700/B6700 Series. New York: Academic Press, 1973.

8. Hassitt, A., and Lyon, L. E. "Efficient Evaluation of Array Subscripts of Arrays." IBM Journal of Research and Development, 16 (January, 1972), 45-57.

9. Hassitt, A.; Lageschulte, J. W.; and Lyon, L.E. "Implementation of a High Level Language Machine." Comm. ACM, 16 (April, 1973), 199-212.

10. Barton, R. S. "Ideas for Computer System Organization: A Personal Survey," in Vol. I of Software Engineering Coins III. Proc. of the Third Symposium on Computer and Information Science 1969. Edited by J. T. Tou. New York: Academic Press, 1970.

11. Dennis, Jack B. The Current Challenge to Computer System Architects. Computation Structures Group Memo 92, Project MAC, M.I.T., Cambridge, Mass., October, 1973. Cambridge, Mass.: Project MAC, M.I.T., 1973.

12. Chesley, G. D., and Smith, W. R. "The Hardware - Implemented High Level Language for SYMBOL." AFIPS Spring Joint Computer Conference Proc., 38 (1971), 563-573.

13. Rice, R., and Smith, W. R. "SYMBOL - A Major Departure from Classic Software Dominated Computing Systems." AFIPS Spring Joint Computer Conference Proc., 38 (1971), 575-587.

14. Smith, W. R.; Rice, R.; Chesley, G. D.; Laliotis, T. A.; Lundstrom, S. F.; Calhoun, M. A.; Gerould, L. D.; and Cook, T. G. "SYMBOL: A Large Experimental System Exploring Major Hardware Replacement of Software." AFIPS Spring Joint Computer Conference Proc., 38 (1971), 601-616.

15. Richards, H., and Zingg, R. J. "The Logical Structure of the Memory Resource in the SYMBOL 2R Computer." Proc. of the Symposium on High Level Language Computer Architecture, College Park, Maryland, 1973.

16. Hutchison, P. C., and Ethington, K. "Program Execution in the SYMBOL 2R Computer." Proc. of the Symposium on High Level Language Computer Architecture, College Park, Maryland, 1973.

17. SYMBOL 2R Reference Processor--EM067. Unpublished document. Fairchild Semiconductor Research and Development Laboratory, Palo Alto, Calif., 1970.

18. SYMBOL 2R Reference Processor Flow Charts. Unpublished document. Fairchild Semiconductor Research and Development Laboratory, Palo Alto, Calif., 1970.

19. SYMBOL 2R Memory Specifications--SP028E. Unpublished document. Fairchild Semiconductor Research and Development Laboratory, Palo Alto, Calif., 1970.

20. SYMBOL 2R Memory Controller Flow Charts. Unpublished document. Fairchild Semiconductor Research and Development Laboratory, Palo Alto, Calif., 1970.

21. Denning, Peter J. "The Working Set Model for Program Behavior." Comm. ACM, 5 (May, 1968), 323-333.

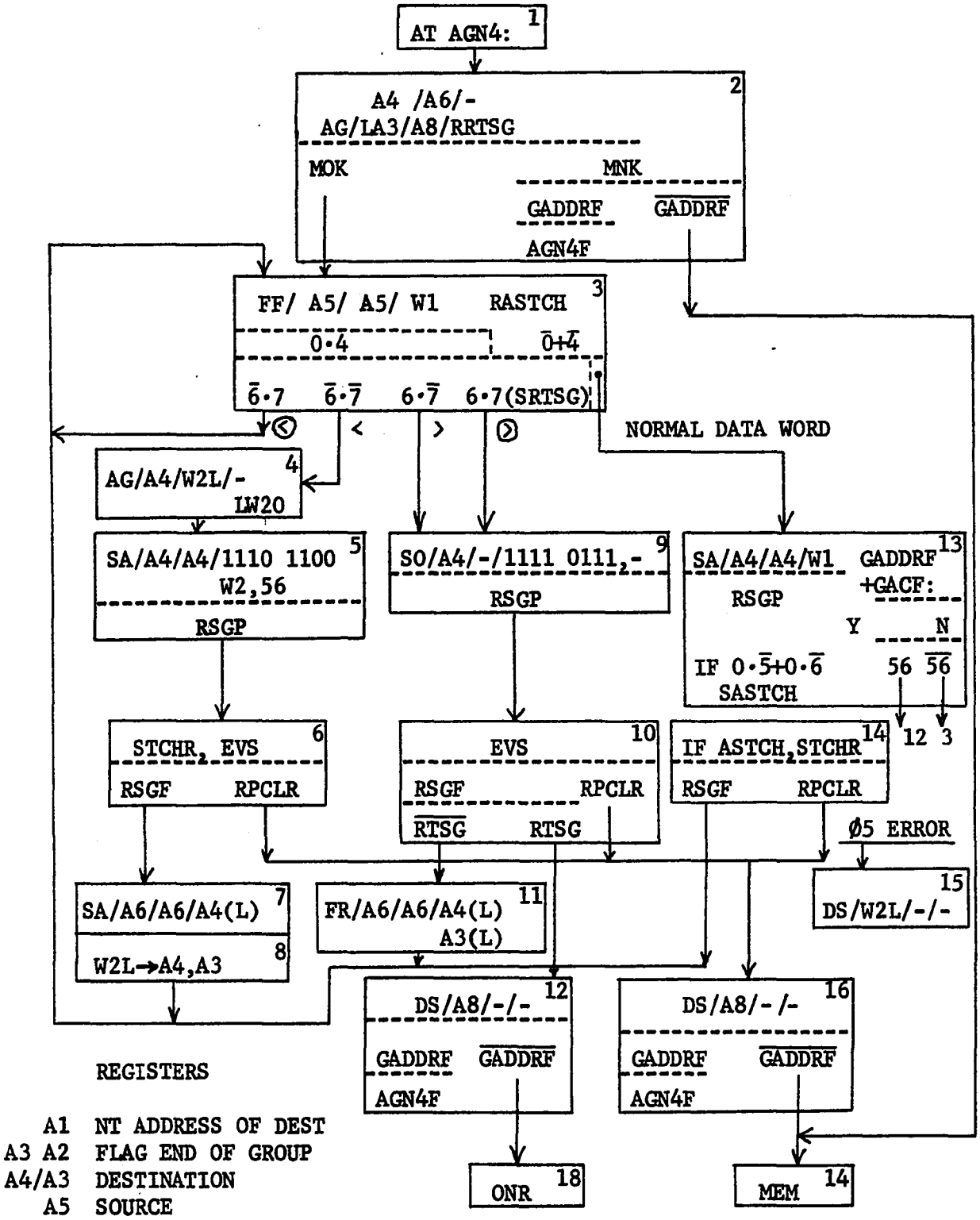
22. Denning, Peter J. "Thrashing: Its Causes and Prevention." AFIPS Fall Joint Computer Conference Proc., 33 (1968), 915-922.

IX. APPENDIX: FLOW CHARTS AND THEIR DESCRIPTIONS

A. Present Scheme

All the flow charts in this section on the present scheme are from the Reference Processor Flow Charts (18). The algorithm performed by the Structure Assign section will be described with the help of the flow chart shown in Figure A1. This and all other following flow charts in this appendix contain the abbreviations for the SYMBOL memory operations which are described in (19). An example of the format for memory operations in the flow charts is the following: FF/A1/A4/W1. This represents a Fetch and Follow memory operation using the A1 address register as the source address with the next address returned to A4 and the data to data register W1. The various registers used in the section are also noted in the figure.

The Structure Assign section is called when the signal AGN4 is turned on. Phase 2 is entered where an assign group is performed in order to create a temporary stack. This temporary stack will be used to maintain the links to the various structure points as the structure is passed over. Phase 3 fetches the source data from the IS stack and analyzes it to ascertain what it is. A left super group mark will mean reentering phase 3 to fetch the next source data. On the other hand, a left group mark implies entering phases 4 and 5 in succession. Phase 4 assigns a new area for that structure



REGISTERS

A1	NT ADDRESS OF DEST
A3 A2	FLAG END OF GROUP
A4/A3	DESTINATION
A5	SOURCE
A6	IS STACK-DOWN
A8	IS STACK-UP(TEMP)
A9	IS STACK-START
W1	TEMPORARY
W2L	NEXT DEST. WORD

Figure A1. Structure Assign section

level while phase 5 stores a link to substructure pointing to that data area. In phase 3, if a right group mark or a right super group mark is encountered, then phase 9 is entered and an end vector is stored in that structure level. Finding a right super group mark in phase 3 also sets the flip flop RTSG. If the source data in the stack is not any of the previous structure marks, then it must be a normal data word. This is stored into the present structure level in phase 13. In all these cases the rapid search generate process is initiated through phases 6, 10 and 14 respectively. These phases also set up the various specialized conditions for the rapid search generate routine. If a left group mark had previously been encountered and phases 4, 5 and 6 had been passed, then phases 7 and 8 are entered. These two phases cause the new structure level to be put on the stack of structures so that that point can be recovered upon finishing with this structure level. Phase 8 leads to a recycle back to phase 3. Similarly, encountering a right group mark or a right super group mark and then passing through phase 9 and then phase 10, will mean going to phase 11. This phase pops up the structure stack because the structure had been completed at that particular level. If a normal data word had been encountered and phases 13 and 14 had been passed, then recycle back to phase 3 because there has been no change in the structure levels. The final exit from this section is

started when the right super group mark is detected in phase 3 then leads down through phase 9 and 10. In phase 10, detecting the RTSG flip flop turned on in phase 3 will lead to phase 12. This phase performs the termination of the routine after deleting the temporary stack which had been used for the structuring process.

What happens in the Get Address Subscripted section is shown in the flow chart in Figure A2. The names and functions of the registers used are also given. The Get Address Subscipted section waits in phase 1 for the proper operation code from the IS. What the RP will be encountering in the IS stack is the subscripted identifier address which is followed by a list of subscripts arranged in sequence with the highest level subscript first. The list is terminated by a right bracket.

When phase 2 is entered the subscripted identifier address word is accessed in order to ascertain the type of link to be worked on. The first 8 bits of the word are checked since these contain the link code. If it is a simple variable or a complete structure, then phase 4 is entered in order to access the name table entry for that item. At the same time, the proper flip flop is set to flag the type of datum being worked on. NVF indicates a name of a simple variable field and LKFLD indicates a link to a structure field. Back in phase 2, if a link to data in name table is detected,

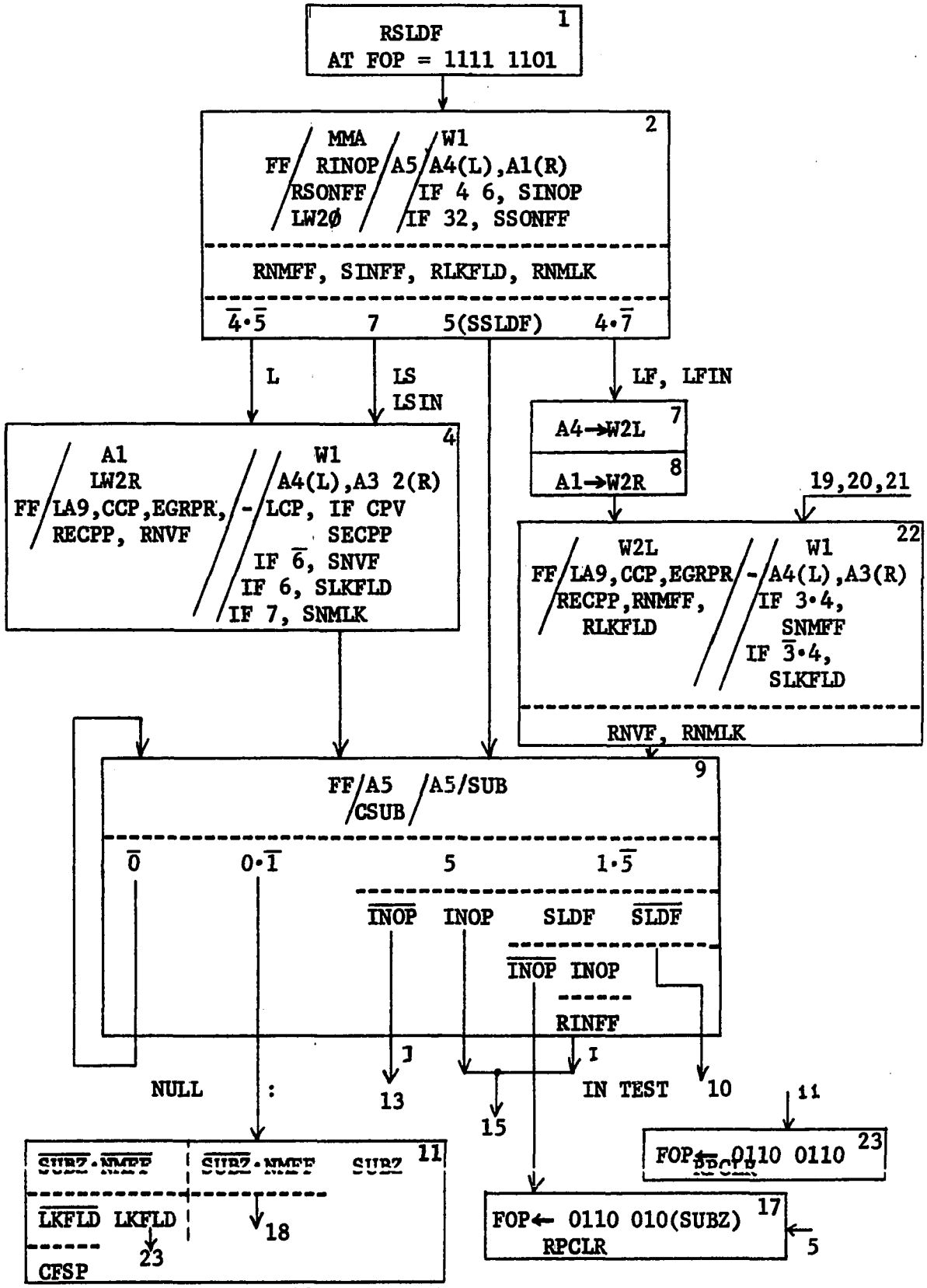
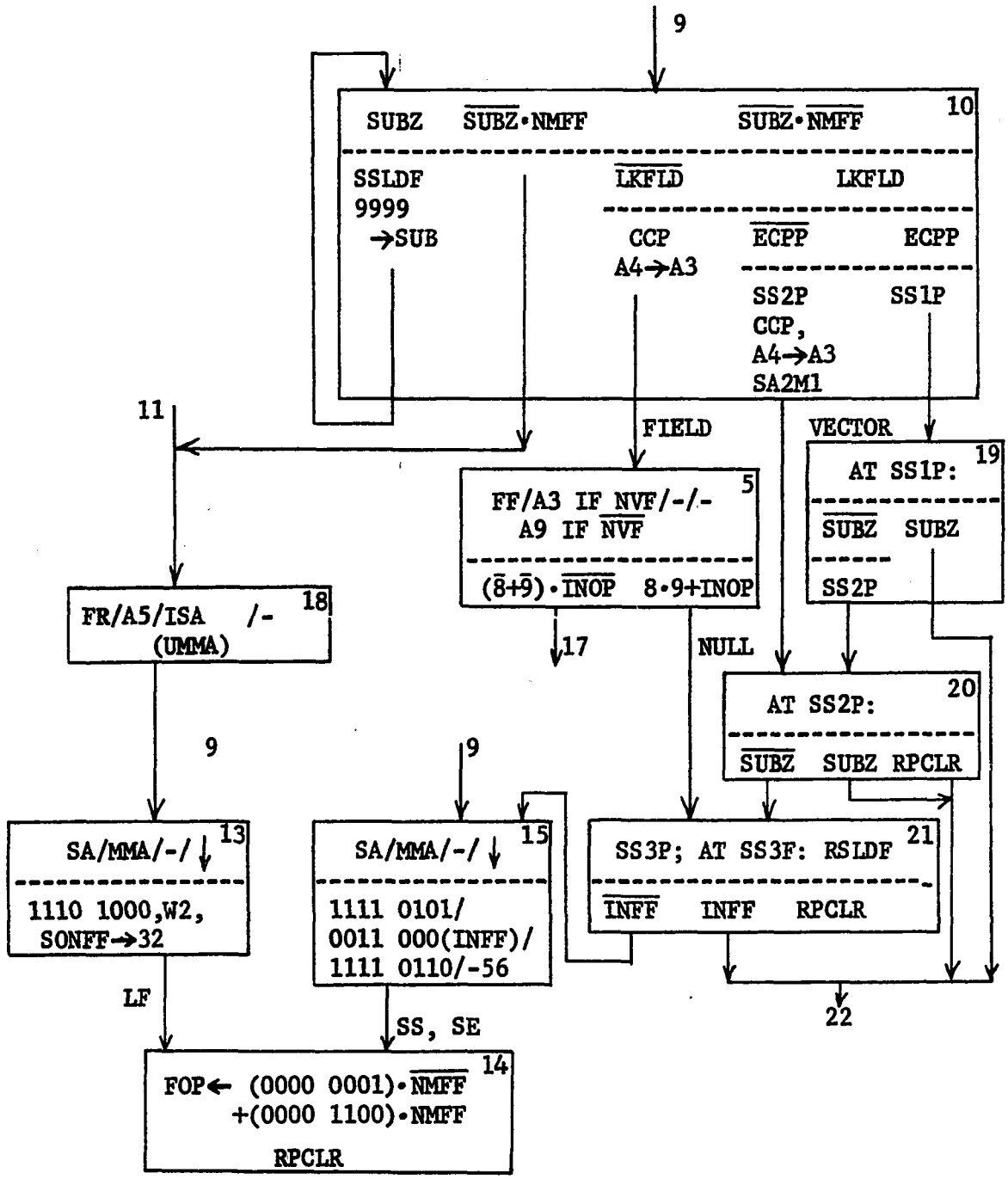


Figure A2. Get Address Subscripted section



REGISTERS

W1/A1	NT+PREVIOUS LINK
A2/A3	CP ADDRESS
A4	DATA START + NEXT GROUP
A5	CURRENT STACK POINTER
A9	ADDRESS FOR CP
SUB	SUBSCRIPT
CP	CURRENT POINTER
RS	RAPID SEARCH (8)

Figure A2. Get Address Subscripted section

then the SLDF flip flop is set and phase 9 is entered. If a link to a field of a structure is detected, then a substructure is being dealt with. In this case, registers are loaded with the proper addresses in phases 7 and 8. In phase 22 that substructure point is accessed in order to begin analysis from that point on down into the substructure. Depending on the condition, either the name flip flop NMFF or the LKFLD is turned on. This phase leads to phase 9.

Phase 9 is the basic fetch phase for each subscript from the IS stack. The subscript register is cleared and then loaded from the top of the stack. The contents of the flag field are then tested. A null indicates a need to repeat phase 9. A colon will cause a branch to the character fetch routine which will not be covered here. Detection of a right bracket indicates that subscripting is finished and starts the termination procedure.

If it is not any of the above, then it must be an integer code which indicates a valid subscript that must be processed. Two possible branches may be taken under this condition. If the SLDF flip flop is on, indicating the data was in the name table, then the branches may lead to phase 17 or phase 15. Phase 17 signals that data has been destroyed by a processing error. Phase 15 is entered if an IN test response is desired. The IN test is a check to see if a particular component is contained in a structure. This is

necessitated by the fact that accessing a nonexistent component creates that component. If the SLDF was off, phase 13 is immediately entered for checking of the various cases. A zero subscript will cause 9999 to be loaded into the subscript register and will set SLDF before going back through phase 13 again. These conditions will allow subscripting to continue on until the end of the vector.

Also in phase 10, detection of a nonlink or a simple field causes entry to phase 5 in order to check for a null field so that field creation can take place. Otherwise the exit taken is to phase 17 where the error is indicated. If a link field is detected, either phase 19 is entered to start the current pointer routine or phase 20 is entered to start the rapid search routine if no current pointer exists. After the current pointer finishes, the rapid search starts from the current pointer address unless the subscript became zero in the current pointer routine. If the subscript does not become zero in the rapid search routine, then phase 21 is entered to start the word scan routine.

The current pointer routine flow chart is shown in Figure A3. This routine is started by a SS1P signal from phase 19 of the main Get Address Subscripted routine. The current pointer is subtracted from the subscript register in phases 2 and 3 and then a test for underflow is performed in phase 4. Underflow indicates the current pointer is larger

than the subscript and hence cannot be used, so phase 5 restores the subscript to its old value before terminating.

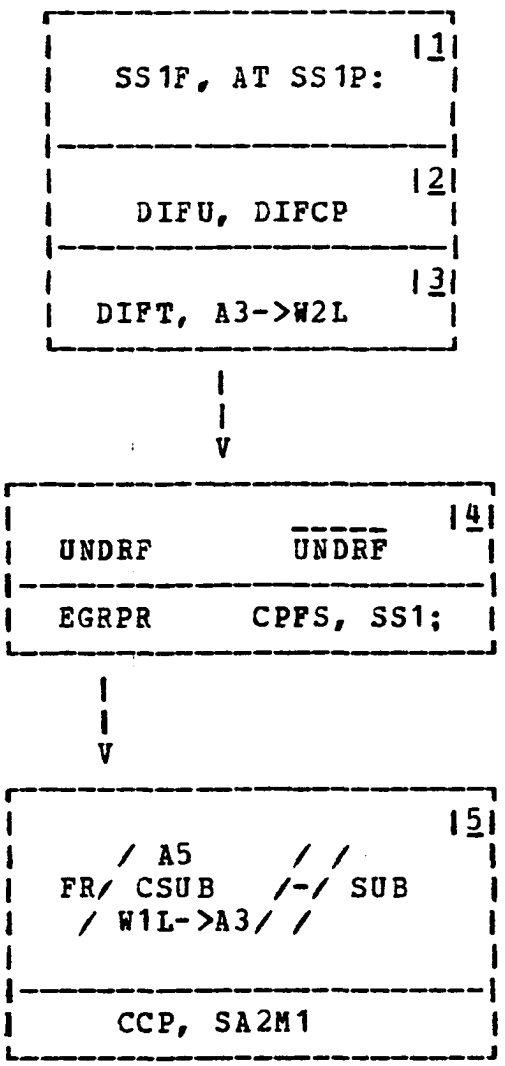


Figure A3. Current pointer routine

The rapid search routine, whose flow chart is shown in Figure A4, will now be considered. Phase 2 accesses the group link word and checks whether or not this is the last group of the vector. If not, then rapid search can go on and phase 3 is entered. In phase 3, if the subscript is not zero

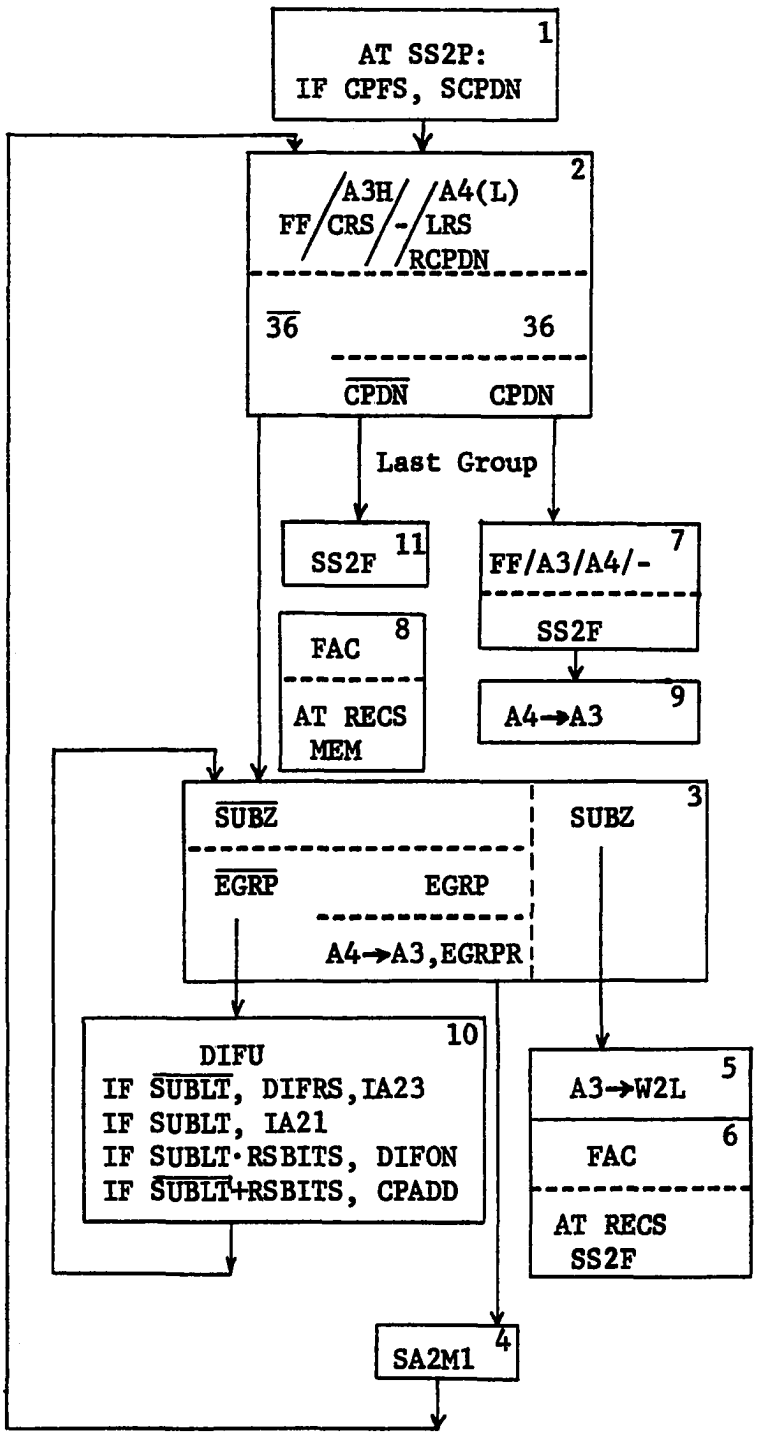


Figure A4. Rapid search routine

and the end of the group has not been reached, then phase 10 is entered. Here the rapid search field is checked and the subscript register is decremented while the address register and current pointer register are incremented. Phase 10 leads back to phase 3. If the subscript is not zero and the end of the group is reached, then the address register is loaded with the address of the next group in the storage string. Then phase 4 is entered to reset the A2 register to minus one and from here back to phase 2 where the cycle starts all over again. If the subscript register becomes zero in phase 3, then phases 5 and 6 are entered to load the proper data address and to call the current pointer generate routine to update current pointer and finally, to terminate the rapid search routine. As long as the subscript is nonzero, when the end of the group is encountered phase 2 is reentered to access the group link word of the next group to be rapid searched. This sequence repeats until either the subscript goes to zero or the last group in the vector is reached. The last group is always word scanned because of the possibility of oversubscripting which creates new structure points. If the last group is reached and CPDN is set, then the current pointer operation is valid and the current pointer must be updated with the latest value from phases 7 and 9. If CPDN is off, then a simple exit from the routine is taken via phase 11.

All the proper terminations lead back to phase 20 of the main Get Address Subscripted routine where the subscript register is checked to see if it is zero or not. If it is zero, then the search is complete. Otherwise, the word scan routine is initiated by the signal SS3P and then phase 21 is entered to wait for the word scan routine to finish. If neither the current pointer nor the rapid search routine could be used, then the word scan would have been entered directly.

In the word scan flow chart shown in Figure A5, phase 1 waits for the initiating signal SS3P and then tests for the various status conditions. The most basic case is that of a link field. This case leads to phase 2 where a word is fetched from the storage string making up the vector and then tested for component starts. Whenever a start is found, a jump to phase 5 is performed. Here the subscript register is decremented by one while the current pointer address register is incremented by one. Then phase 6 is entered and the subscript register is checked to see if it has gone to zero. If not, the sequence is repeated by going back to phase 2. This goes on until finally the subscript becomes zero and then the current pointer generate is called to update the current pointer. After this is done, the section is exited in order to return to the main Get Address Subscripted routine. If an end vector is detected in phase 2 before the subscript

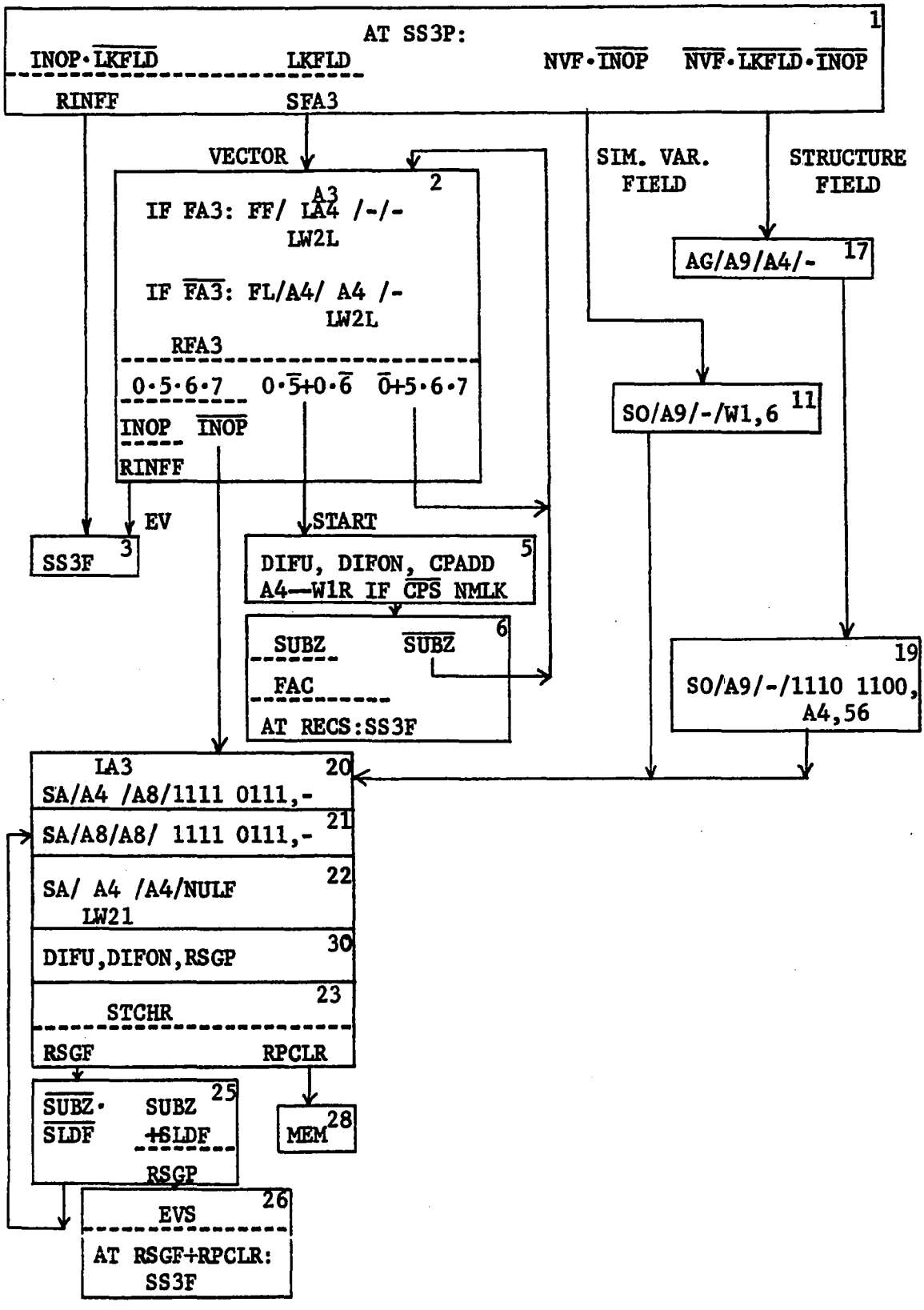


Figure A5. Word scan routine

becomes zero, then this means that subscripting has gone beyond the vector that actually exists and new vector points must be created. If the SLDF flip flop is on, then the subscript was initially zero and only one null field is created after the end of the vector. This happens from phase 20 and on if the access is not an IN test, otherwise what is desired is a negative answer to indicate that the element desired does not exist. This is done by exiting from phase 3 to phase 21 of the main routine with the IN flip flop off in order to cause a transfer to phase 15 where a zero is put on the stack to indicate that the component desired does not exist. The same process takes place from phase 1, if a simple variable field and not a link field is detected with the IN operation turned on.

Two other possible cases can be detected in phase 1. The first is not an IN test and a simple variable field. The other is not an IN test and not a simple variable field and not a link field. Both these cases lead to the creation of structure since the data space being subscripted into does not presently contain a structure. This means that a valid nonzero subscript is in the subscript register and yet the field pointed at is a simple variable field or a structure element which does not contain a link field. Consider the non-link field condition first. Phase 17 is entered in order to assign space for the new structure and the address of the

data space is loaded with a link to substructure in phase 19. Then on to phase 20 which starts the construction of the vector for the subscript. Now, if a simple variable was detected in phase 1, then phase 11 is entered where the structure bit is turned on in the Name Table Control Word. Phase 20 is entered after this. Exits from phase 2, 11, and 19 all indicate a need for a new structure to be created since the subscript is larger than the number of actual components in the storage string.

The phases from phase 20 on are devoted to constructing new null components which will account for the difference between the subscript and the actual number of existing components. Phases 20, 21, and 22 take care of storing null fields in the storage string. Phase 30 decrements the subscript register and calls the rapid search generate routine to create the rapid search field. When this procedure is finished in phase 23, then the subscript is checked in phase 25. A nonzero subscript means the whole sequence must be repeated. A zero subscript starts up the rapid search generate routine again and in phase 26 the end vector signal is set. This is utilized by the rapid search generate to indicate that it is finished. When the rapid search generate termination signal is received, then the word scan finish signal SS3F is sent to phase 21 of the main section.

In the main Get Address Subscripted section, exits from phase 19 or 20 or 21 with subscript zero mean that the particular component has been found. A recycle is then performed by going up to phase 22 to access the new structure point and then to phase 9 to get the new subscript to repeat the whole process. In phase 9 detection of a right bracket indicates there is no new subscript and that subscripting has been completed. An IN test leads to phase 15 which indicates a success since the structure component existed. If the IN operation was not on, then phase 13 is entered in order to return a link to a field with the proper address. The link to a field and address are placed on the top of the IS stack through the use of the MMA address register. Both phase 15 and phase 13 exit to phase 14 which is the completion phase.

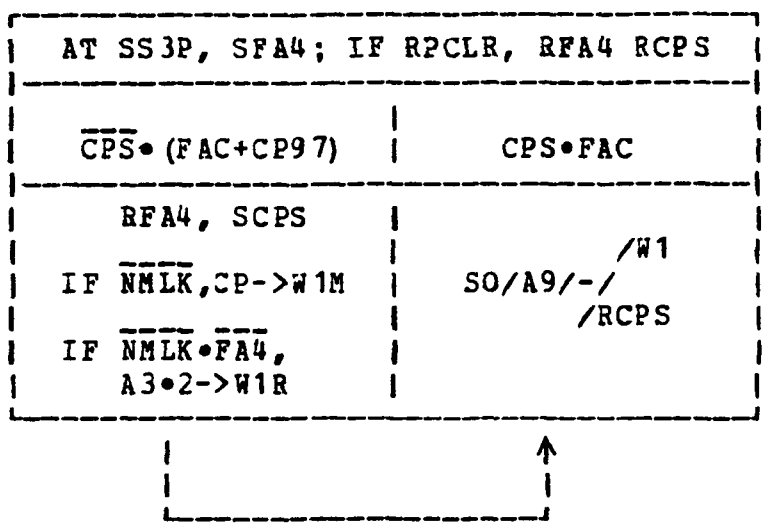


Figure A6. Current pointer generate routine

The current pointer generate routine shown in Figure A6 has only one phase. This is initiated by the signal FAC which causes the current pointer to be transferred to the proper bits of the W1 register and the address of that current pointer from the A3•2 register into the last 24 bits of the W1 register. Then the contents of the W1 register is stored in the current pointer area of the vector link word. A CP97 causes the CP and address to be transferred to W1 but these will not be stored until the signal FAC.

The rapid search generate routine shown in Figure A7 has the task of setting up the rapid search field of the group link word as new structure is created in a storage string. There are three important signals used in this routine. EVS indicates that the end of the vector has been reached, GEGRP says that the end of the present group has been reached, and finally STCHR indicates that a start character has been found.

Phase 1 waits for the initiating signal. Phase 2 fetches the group link word into the W1 register. The number of component starts already found in the group are used to set the right bits of the rapid search field register. The contents of the W1 register are then stored back into the group link word in phase 3. Phase 13 increments the current pointer address and, if a start character has been found, the proper bit of the RS register is set. Phase 4 checks whether

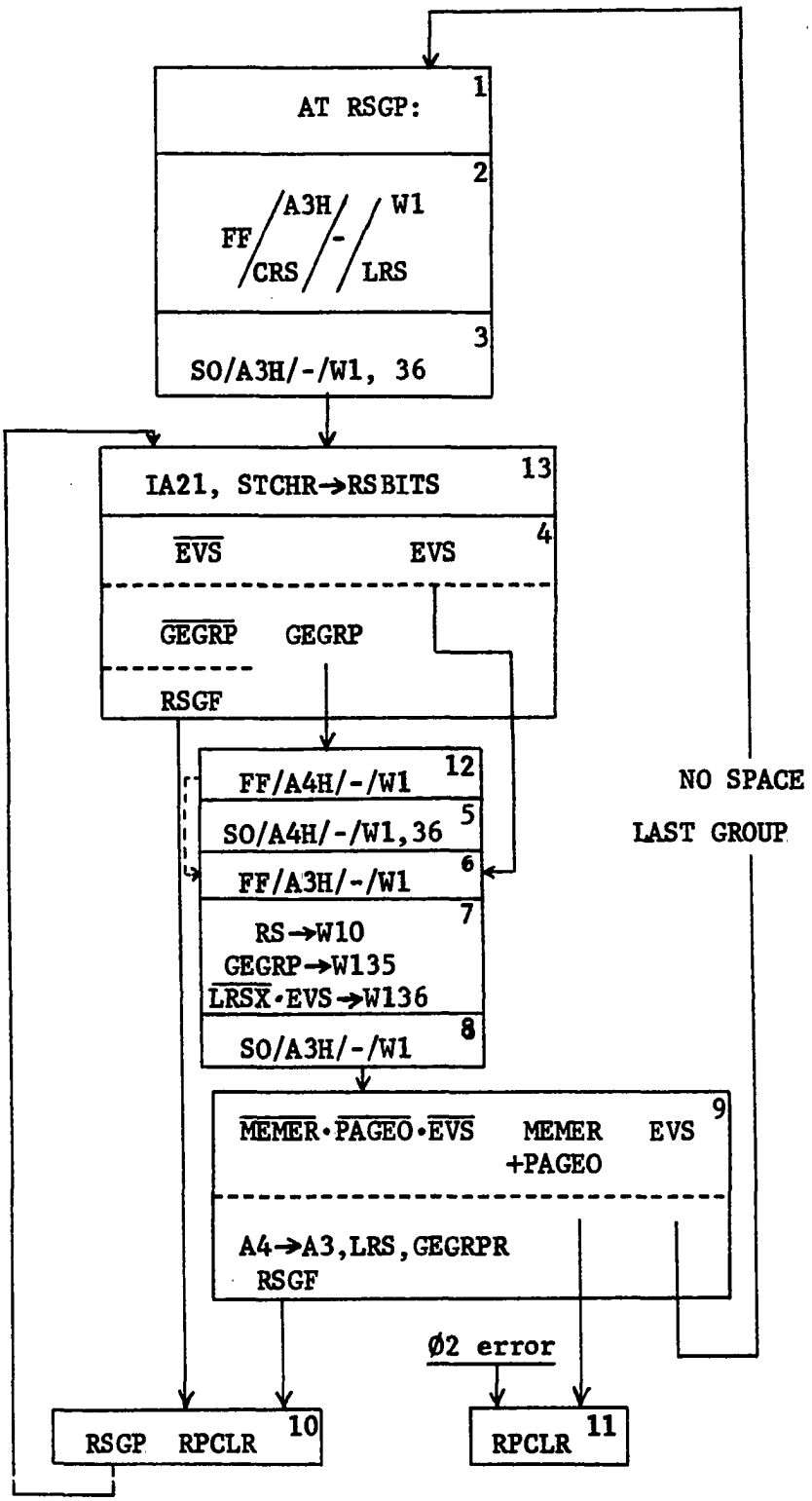


Figure A7. Rapid search generate

the end vector has been found. If the end vector has not found and the GEGRP flag is not set, then the routine is temporarily terminated by going into phase 10 which gives the end signal. On the next call from phase 10, the same thing is done except phase 13 is reentered from phase 10. Phase 13 increments current pointer and updates the rapid search register. When the end of the group is encountered, phase 12 is entered where the next group link word is fetched and loaded into W1. Then phases 6 and 7 are entered where the previous group link word is fetched, tagged with a no space available and a last group indicator if it is the last group. Phase 8 stores the old group link word back. Phase 9 checks whether the processing is finished or not. If not, the next group address is loaded into the proper register. An end vector detected in phase 4 indicates that processing is finished, so jump to phase 6, 7 and 8 to dress up the group link word and then exit in phase 9 with the end vector flag giving the termination signal.

B. Modulo 16 Scheme

The flow chart of the Structure Assign routine for the Modulo 16 scheme is shown in Figure A8. Phase 1 waits for the AGN4 signal which initiates the routine. A group is assigned in phase 2 and the RTSG flip flop is reset. The group assigned will be used as the temporary stack for maintaining the links to the various structure points as the structure to

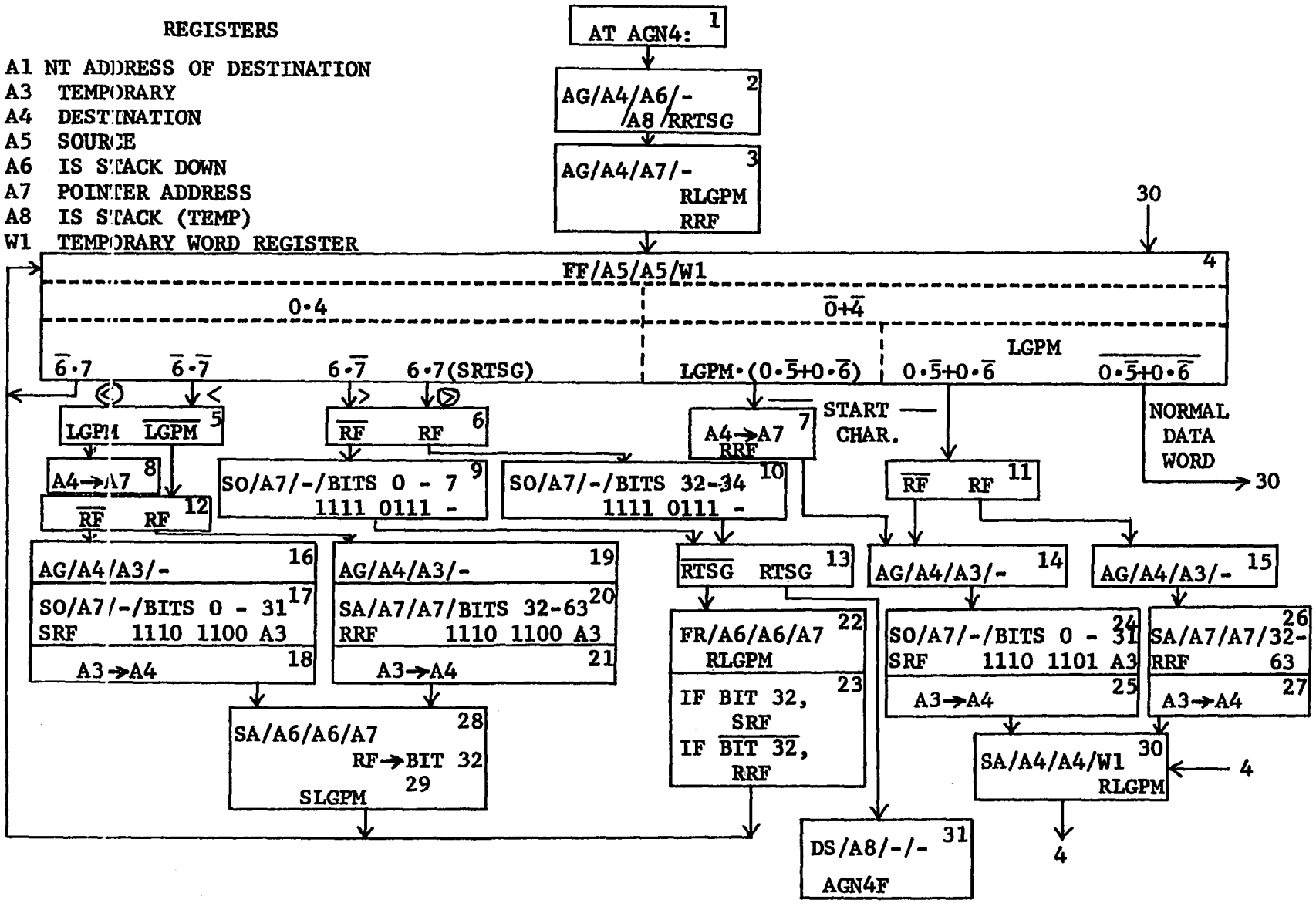


Figure A8. Structure Assign section of the Modulo 16 scheme

be stored is passed over. Phase 3 allocates a group which will be the start of the storage string containing the halfword links. Also the LGPM flip flop, which indicates a left group mark as the previous item on the stack, and the RF flip flop, which indicates which half of the pointer word is to be used, are reset. Then phase 4 is entered. This phase fetches the source data from the IS stack and analyzes it. A left super group mark means a recycle back to phase 4 to fetch the next word on top of the IS stack. A left group mark leads to phase 5 where the LGPM flip flop is checked. If it is on, then the data on the stack is a left group mark. Phase 8 then transfers the contents of the destination address register, A4, to the pointer address register, A3. The RF flip flop is also reset and then phase 12 is entered. If the LGPM is off, then the previous item on the stack was not a left group mark and so phase 12 should be entered. Here the RF flip flop is checked in order to determine whether to store the next pointer in the left or right half of the word whose address is on the pointer address register.

If the RF flip flop is off, then phase 16 is entered. Here a group is assigned in order to provide space for the lower level pointers. In phase 17 the address of this group and a link to substructure are stored in the left half of the word whose address is in the pointer address register. The flip flop RF is also set. Then in phase 18 the address of

the group assigned is put in the destination address register A4. If the RF flip flop is on, then phases 19, 20 and 21 are entered in succession. Phase 19 does the group assignment that would have been done by phase 16 in the other branch. Phase 20 stores the address of the group and the link to substructure in the right half of the pointer word. Phase 21 updates the destination address register. Both phase 18 and phase 21 lead to phase 28 where the contents of the pointer address register is put on the top of the stack together with the contents of the flip flop RF. Then phase 29 sets the LGPM flip flop. After all this, phase 4 is reentered for a look at the next item on the IS stack.

Detecting a right group mark in phase 4 will cause an entry to phase 6. A right super group mark will also lead to phase 6 in addition to setting the RTSG flip flop back in phase 4. In phase 6 the RF flip flop is checked to determine which half of the present pointer word the end vector mark, F7, goes into. If RF is off, then the end vector is put in the left halfword in phase 9. Otherwise it is put in the right halfword in phase 10. Both of these alternative phases go to phase 13 where the RTSG flip flop is checked.

If the RTSG flip flop is off, then phase 22 is entered. Phase 22 pops the top of the structure stack because the structure at this level has been analyzed and stored. The LGPM flip flop is also reset in this phase before proceeding

to phase 23 which sets or resets the RF flip flop according to what was stored in bit 32 of the word on top of the stack. Phase 4 is reentered once again after this.

Finding a set RTSG in phase 13 means that the last item in the stack has been found and so the termination process is initiated. The temporary stack used is deleted and returned for reclamation in phase 31. Then the termination is signalled by AGN4F.

Now consider what happens if data words are detected by phase 4 on top of the stack. If the LGPM is on and a start character is detected, then the contents of the destination address register A4 is transferred to the pointer address register A7. This is done to provide space for the pointers for the new level. The RF flip flop is also reset before proceeding to phase 14. On the other hand, if LGPM is off and a start character is detected, then this means the start of a new data element at the same structure level. Phase 11 then checks the RF flip flop. If it is off, then it means the link to data and the address of the data must be stored in the left half of the pointer word. Phase 14 assigns the group for the data. Phase 24 stores the link and updates the destination address register. If RF is on, then similar actions are performed in phases 15, 26, and 27 to store the link in the right halfword. The two branches both converge to phase 30 where the data word is stored in the space

allocated. The LGPM flip flop is also reset before going back to phase 4. In phase 4, if LGPM is off and the word on top of the stack does not contain a start character, then the word must be a normal data word which is directly stored in the proper location by phase 30.

The flow chart of the main Get Address Subscripted section is shown in Figure A9. This is essentially the same as the main section of the present Get Address Subscripted routine. There is a minor change in phase 22 to take care of deciding which half of the pointer word should be loaded into the destination register. There is also a change in phase 13 where the MMA is loaded from the left half of the W2 register. The biggest change is the removal of the current pointer, rapid search and word scan sections. These sections are replaced by a single search section, which is shown in Figure A10.

The organization of the search section is similar to the present word scan section since the search section will trace out the path denoted by the subscripts and create new elements if necessary. Phase 30 waits for the signal to initiate operations. Upon receipt of this signal the various possible conditions are checked. If it is an IN operation and the LKFLD is off, then the INFF is reset and phase 44 is entered where the termination signal is transmitted. With the LKFLD on, signifying a link field, the most basic case has

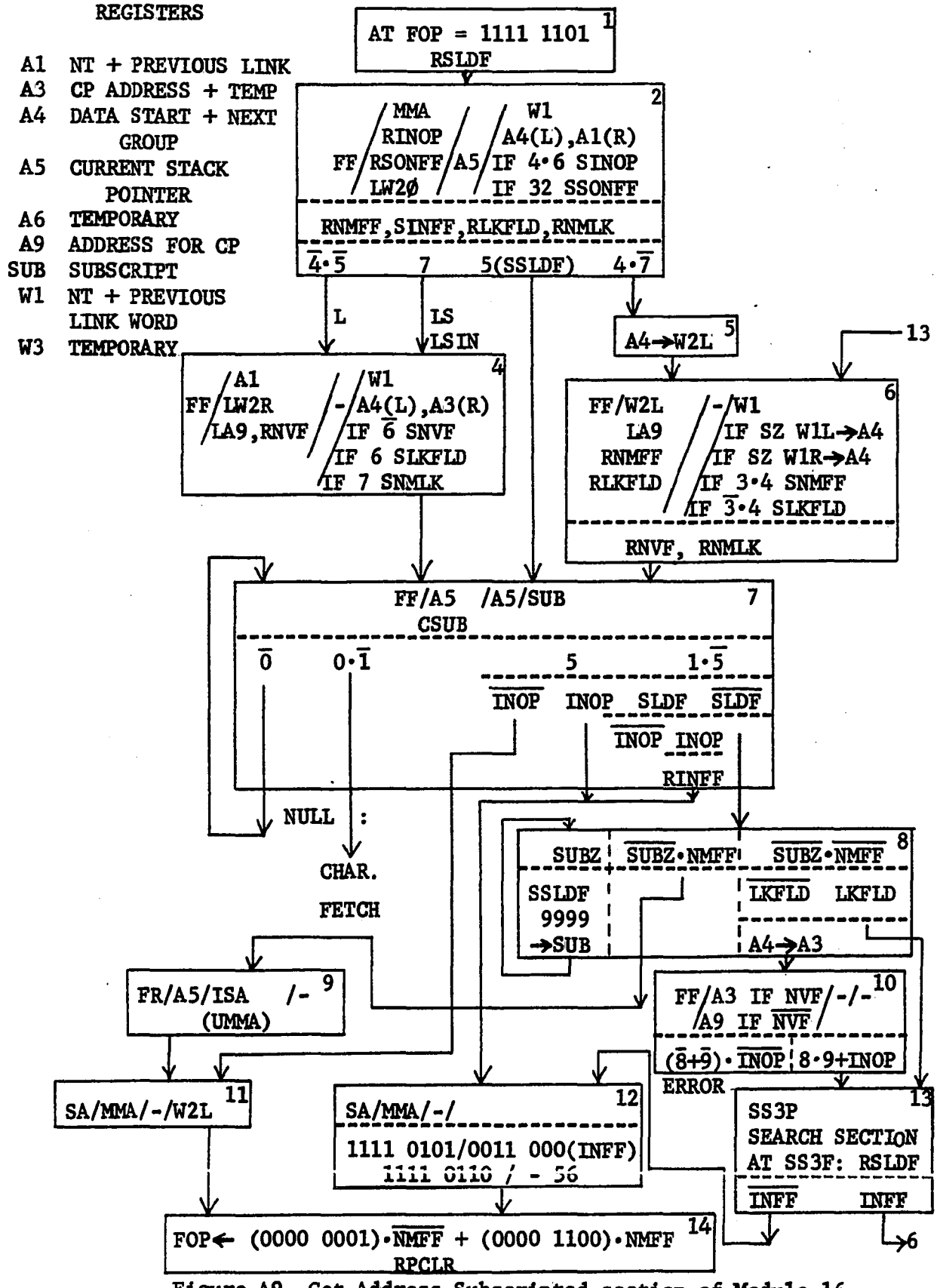


Figure A9. Get Address Subscripted section of Modulo 16

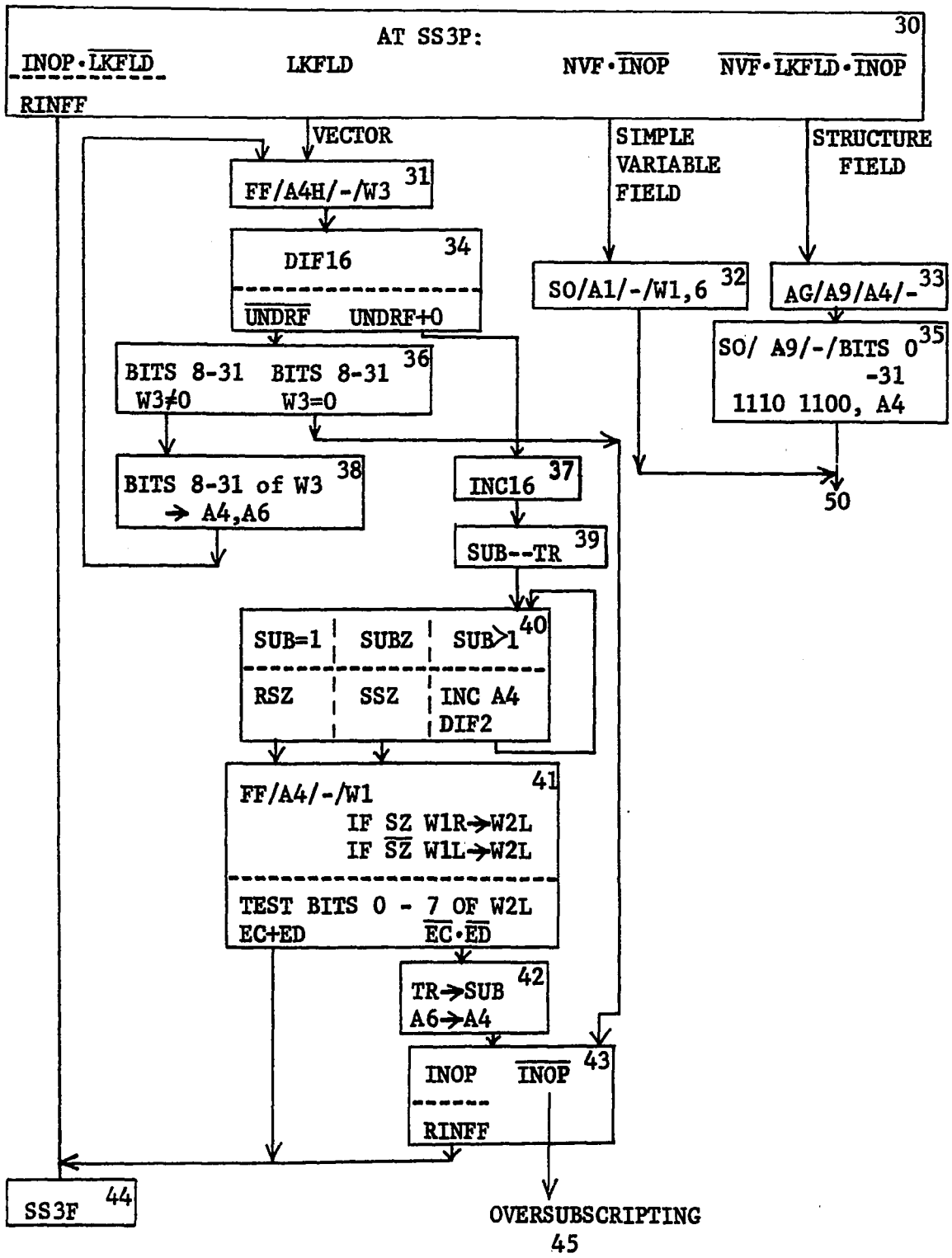


Figure A10. Search section of the Modulo 16 scheme (Part I)

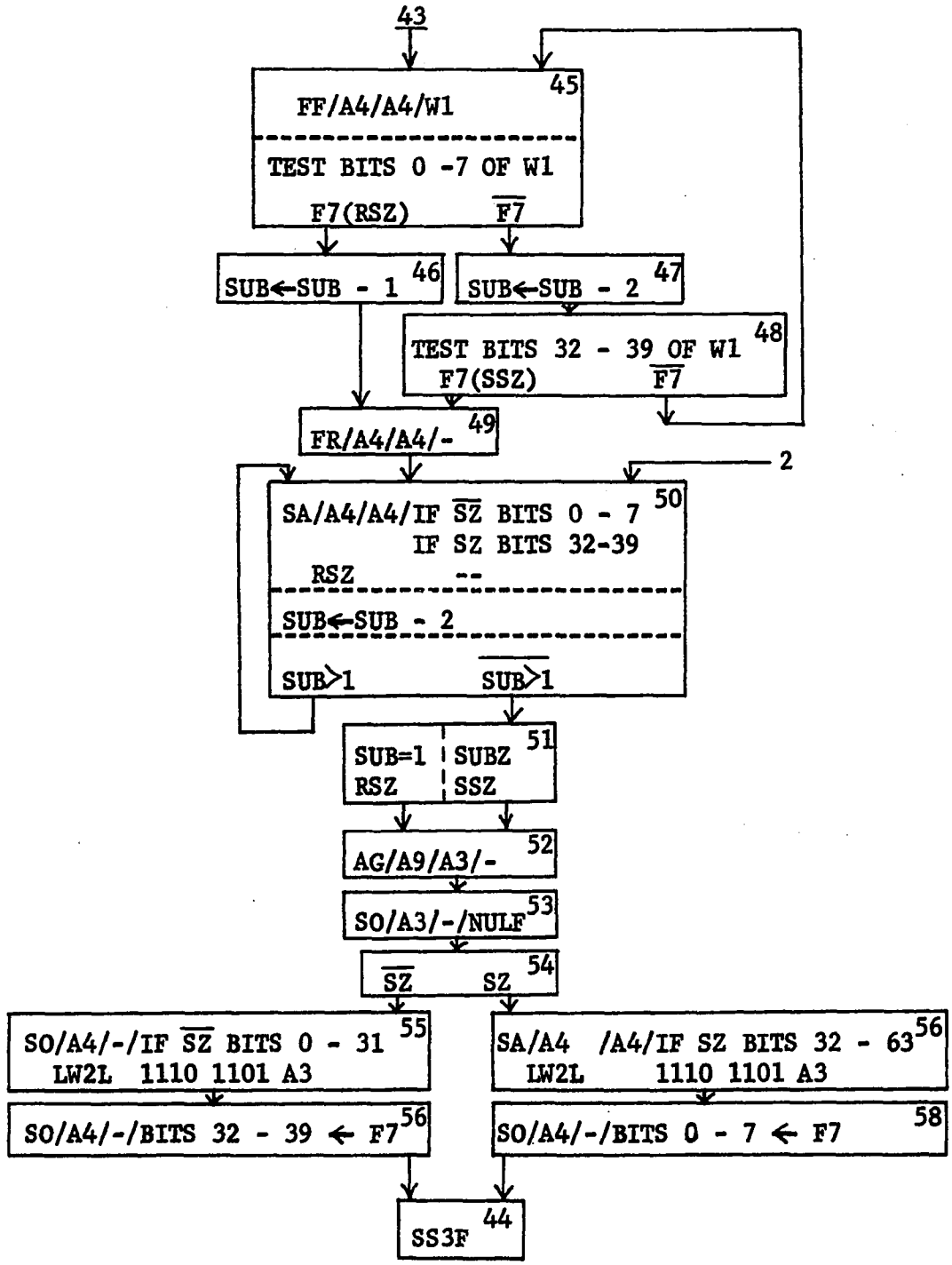


Figure A10. Search section of the Modulo 16 scheme (Part II)

been found.

In this case phase 31 is entered. Here the group link word of the address in the destination register is fetched and loaded into a word register W3. The subscript register is decremented by 16 in phase 34. A check for underflow is then performed. If there is no underflow, then the forward link of the group link word is checked. If it is zero, then the subscript has gone beyond the actual number of data elements in the structure and so phase 42 is entered. A nonzero forward link address means there are still some data elements available. The forward link address is loaded into the next group address register and then phase 31 is reentered to get the group link word. The subscript register is again decremented by 16. This cycle repeats as long as the subscript is greater than 16.

If the subscript is less than or equal to 16, then an underflow or a zero will be detected in phase 34. This leads to phase 37 where the 16 is added back to the subscript register. Phase 39 copies the contents of the subscript register into a temporary register. The destination address register will also be saved so that when oversubscripting is detected these values could be recovered. In phase 40 the value of the subscript is checked. If the subscript is greater than 1, then the destination address register is incremented by 1 and the subscript is decremented by 2.

Phase 40 is reentered and the whole cycle is repeated as long as the subscript is greater than 1. Exit from phase 40 is possible only if the subscript is not greater than 1. If the subscript is zero, then the SZ flip flop is set. Otherwise it is reset if the subscript is 1. Both of these two cases lead to phase 41 which fetches the contents of the word pointed at by the destination address register into a word register W1. The left half of word register W2 is loaded from either the left or right half of W1, depending on the setting of flip flop SZ. Bits 0 to 7, the link bits, of W2L are then examined. Finding a link to substructure or a link to data leads to phase 44 and the termination of the routine. Not finding either kind of link indicates oversubscripting and so phase 42 is entered. This phase reloads the old value of the subscript into the subscript register and the starting address of the group is also reloaded into the destination address register. Then, in phase 43 there is a check for an IN operation. An IN operation results in resetting the INFF flip flop and then terminating normally, while no IN operation means the oversubscripting phases must be entered.

Phase 45 is the start of the section which performs the creation of new elements to correspond to the remainder of the subscript. Here each halfword in the last group is checked for the end vector mark so it can be changed to accommodate new elements in the vector. The left halfword is

checked in phase 45. If it is found in the left halfword, then the subscript is decremented by 1 in phase 46. Phase 49 is then entered. If the left halfword does not contain the end vector mark, the subscript is decremented by 2 and then the right halfword is checked in phase 49. If the end vector mark is not there, then back to phase 45 to check the next word in the group. This goes on until the end vector mark is found. Once it is located, the address register A4 is pointed to the word containing the end vector mark. In phase 50 the mark is changed to all zeros to signify a null element. After this, the subscript register is decremented by 2 and then its value is checked. If it is greater than 1, then back to phase 50 to create null elements and decrement the subscript by 2. This keeps going until the subscript is not greater than 1. In this case phase 51 is entered to determine if the value is zero or one and the SZ flip flop is reset or set according to this value. Next, phase 52 assigns a new group where a null field is stored in the first word by phase 53. The null field will consist of a string start and a string end. Phase 54 determines which halfword in the pointer is to be used. Phases 55 and 56 store the link to data code and the data address in the proper halfword. An end vector is then stored in the next halfword. After all this, the termination signal is given by phase 44.

The section which creates new elements to correspond to the remainder of the subscript is also entered from two other places. These two points correspond to the two remaining cases which can be detected in phase 30. The first is the case when there is no IN test and a simple variable field is found. This leads to the oversubscripting section after putting the proper link code in the NTCW. The second is the case when no link field is encountered. Again, the oversubscripting section is entered to create a new substructure after space has been assigned and linked with a link to substructure from the original structure field.

C. Pseudo Level Vectors Scheme

The Structure Assign routine for the Pseudo Level Vectors scheme is shown in Figure A11. The additional address registers and word registers required should be noted. Two counters are also used. LVL is a two bit binary counter used to keep track of the level of the vector currently in use. While PTR is a 4 bit binary counter keeping track of the halfword links in the vector (group).

The Structure Assign section is initiated by the signal AGN4 which is detected in phase 1. The second phase allocates a group for the temporary stack which is used to maintain links to the various structure points as the structure is passed over. In the same phase, the right super group flip flop RTSG and the end of group flip flop EGRP are reset.

REGISTERS

A1 NT ADDRESS OF DESTINATION
A4 DESTINATION
A5 SOURCE
A6 STACK DOWN
A8 STACK UP (TEMP)
A2 LEVEL 0 ADDRESS
A3 NEW LEVEL ADDRESS
A7 TEMPORARY
A9 GROUP LINK ADDRESS
W1 TEMPORARY
W2 NEXT DESTINATION WORD
W3 STACK WORD

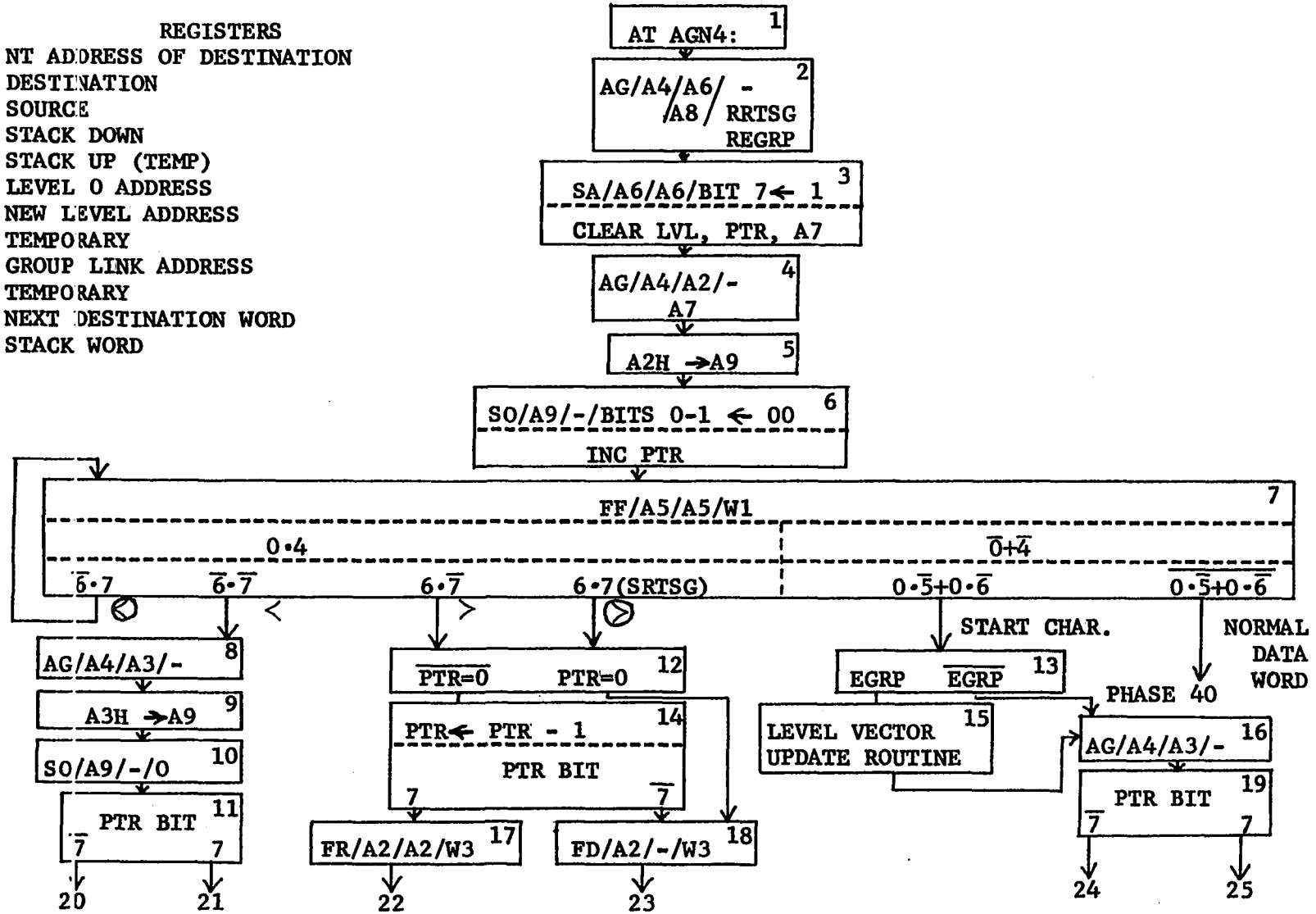


Figure A11. Structure Assign section of the Pseudo Level Vectors scheme (Part I)

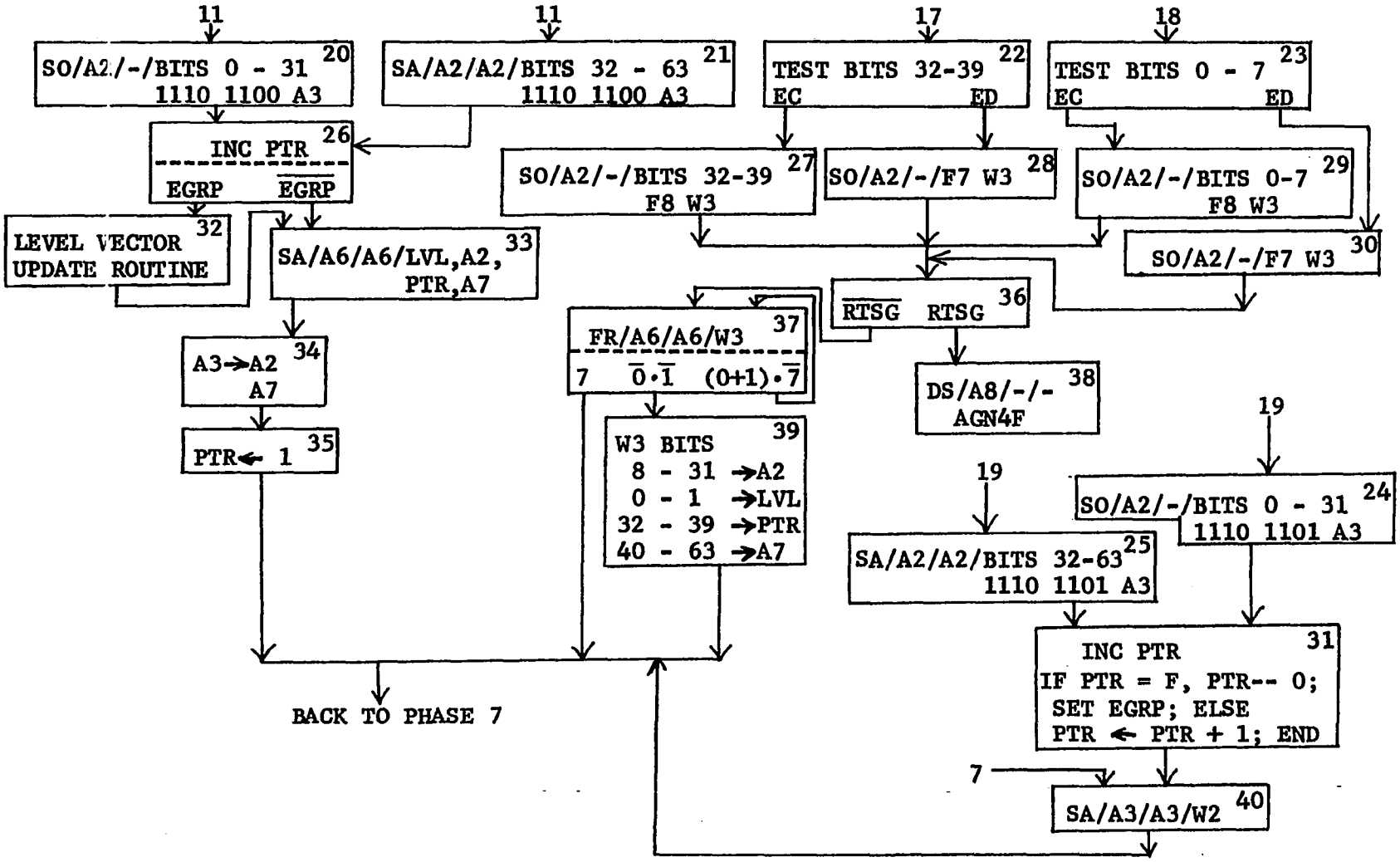


Figure A11. Structure Assign section of the Pseudo Level Vectors scheme (Part II)

In phase 3 the top of the stack is flagged by setting bit 7 to one. Then the two counters, LVL and PTR, together with address register A7 are cleared. Phase 4 assigns a group which will serve as the first level 0 vector. Notice that A7 keeps track of the address of the first word in the vector. Phases 5 and 6 load the LVL bits of the group link word of that vector with the level of that vector, which is zero. The PTR is also incremented so that the first element of the array is linked from halfword 1 instead of halfword 0 since in the original system elements are stored starting with subscript 1 and not 0.

Phase 7 fetches the item on top of the IS stack and checks it against the different cases possible. A left super group mark causes a repetition of phase 7 so that the next item on the IS stack can be examined. If a left group mark is found, then a new group is assigned to the new level address register A3 by phase 8. Phases 9 and 10 take care of placing a 0 in the LVL bits of the group link word. In the next phase, bit 7 of the counter PTR is examined. A zero or one in this bit determines whether the left or the right halfword of the present level 0 word is to be used. Phase 20 or phase 21 takes care of storing a link to substructure and the address of the new level group in the proper halfword. Both of these two phases lead to phase 26 which performs the INCPTR routine. What this routine does is to check if the

PTR is equal to F (hexadecimal) and if it is, then the PTR is cleared and the EGRP flip flop is set. If PTR is not equal to F, then PTR is incremented by 1. Phase 26 also checks the EGRP flip flop at the end of INCPTR. The EGRP being set means that the whole level 0 vector has been used up so that a new one has to be assigned and all the necessary linking performed. All this is done by the Level Vector Update routine which will be described in detail later. Phase 33 is entered after the termination of the updating routine. If the EGRP flip flop had not been set in phase 26, then phase 33 would have been entered directly. Phase 33 saves several values on the stack so that they may be recovered later when the particular structure level being processed is finished.

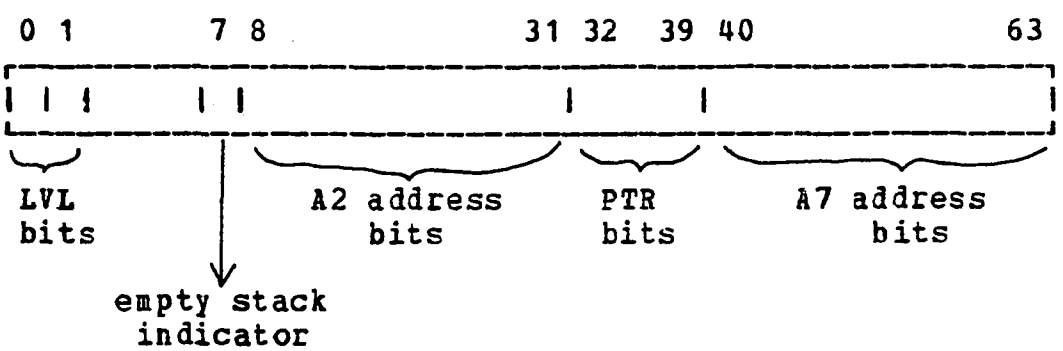


Figure A12. Bit assignment of the stack word

The bit assignment of the stack word is shown above in Figure A12. The present value of LVL and the contents of the level 0 address register occupy the left half of the stack word, while the contents of PTR and address register A7 use the right half of the stack word. After saving these values,

the contents of the new level address register is copied into registers A2 and A7 in phase 34. Then PTR is set to 1 before going back to phase 7 for a look at the next item on the stack.

If a right group mark or a right super group mark is found, then an end mark has to be stored in the previous link. To know what kind of end marker should be used, the link code of the previous link must be examined. An ED has to be replaced by an F7 while an EC has to be replaced by an F8. Detecting a right group mark or a right super group mark in phase 7 will lead to phase 12. In addition a right super group mark turns on the RTSG flip flop in phase 7. The contents of PTR are checked in phase 12. Phase 14 decrements PTR by 1 and then checks the 7th bit of PTR. A zero in bit 7 of PTR means the previous word has to be fetched in order to look at its second half. If bit 7 is one then the first half of the word pointed at by the level 0 address register will be examined. The word desired is fetched into a word register, W3, by either phase 17 or 18. Phase 22 checks bits 32 to 39 of the word fetched by phase 17. Depending on the link code found in either phase 27 or 28 replaces the link code with the proper end mark code. On the other hand phase 23 checks bits 0 to 7 of the word fetched by phase 18, while phase 29 or phase 39 will replace the link code with the appropriate end code. All the possible cases lead to phase 36

where the RTSG flip flop is checked. If it is on, then the whole structure has been passed and the temporary stack is deleted and the termination signal AGN4F is sent by phase 38. If RTSG is not on, then the top of the stack is checked since we want to go back to the previous level if there was one. If the top of the stack contains a level 0 address, then proceed to phase 39 where the saved values in the stack are loaded into the proper registers and counters. Otherwise, the stack is popped repeatedly until a LVL equal 0 is found or the bottom of the stack is detected in which case phase 7 is entered. Phase 39 leads back to phase 7 for an examination of the item on top of the IS stack.

Back in phase 7, if a data word with a start character is detected, then it must be linked properly before it is stored. Otherwise an ordinary data word, which does not contain a start character, is stored in the allocated space directly since the starting address has been linked already. The storing of the data is done in phase 40. After this, phase 7 is reentered to look at the next item on the stack.

If a starting data word is detected, then the EGRP flip flop is checked in phase 13 in order to find out whether there is an available link in the level 0 vector. An EGRP flip flop that is set means that a new level 0 vector has to be assigned and so the Level Vector Update routine is called in phase 15. After the updating routine, phase 16 is en-

tered. This phase is entered directly from phase 16 if the EGRP flip flop was not set. In phase 16, a new group is assigned as the space for the data whose starting point has been detected. Phase 19 determines in which halfword the link will be stored. Either phase 24 or 25 will store the link code and the starting address of the data in the proper halfword of the available level 0 word. Both of these phases lead to phase 31 which starts the INCPTR routine before going into phase 40 to store the data in the space assigned. Phase 7 is again reentered after this phase.

The Level Vector Update routine, shown in Figure A13, is called by the Structure Assign section whenever all 16 halfwords in the present level 0 vector have been used for linking. The Level Vector Update routine has the task of assigning a new group which will serve as the new level 0 vector. But before the group can start serving as the level 0 vector, it must first be linked with the level 1 vector. Whenever the higher level vector gets used up, the updating routine also takes care of allocating a new group to serve as the higher level vector a and of linking the group with next higher level. This routine also uses the temporary stack, whose primary function is to maintain structure links, to serve as the stack for maintaining links to the addresses of vectors of level greater than 0.

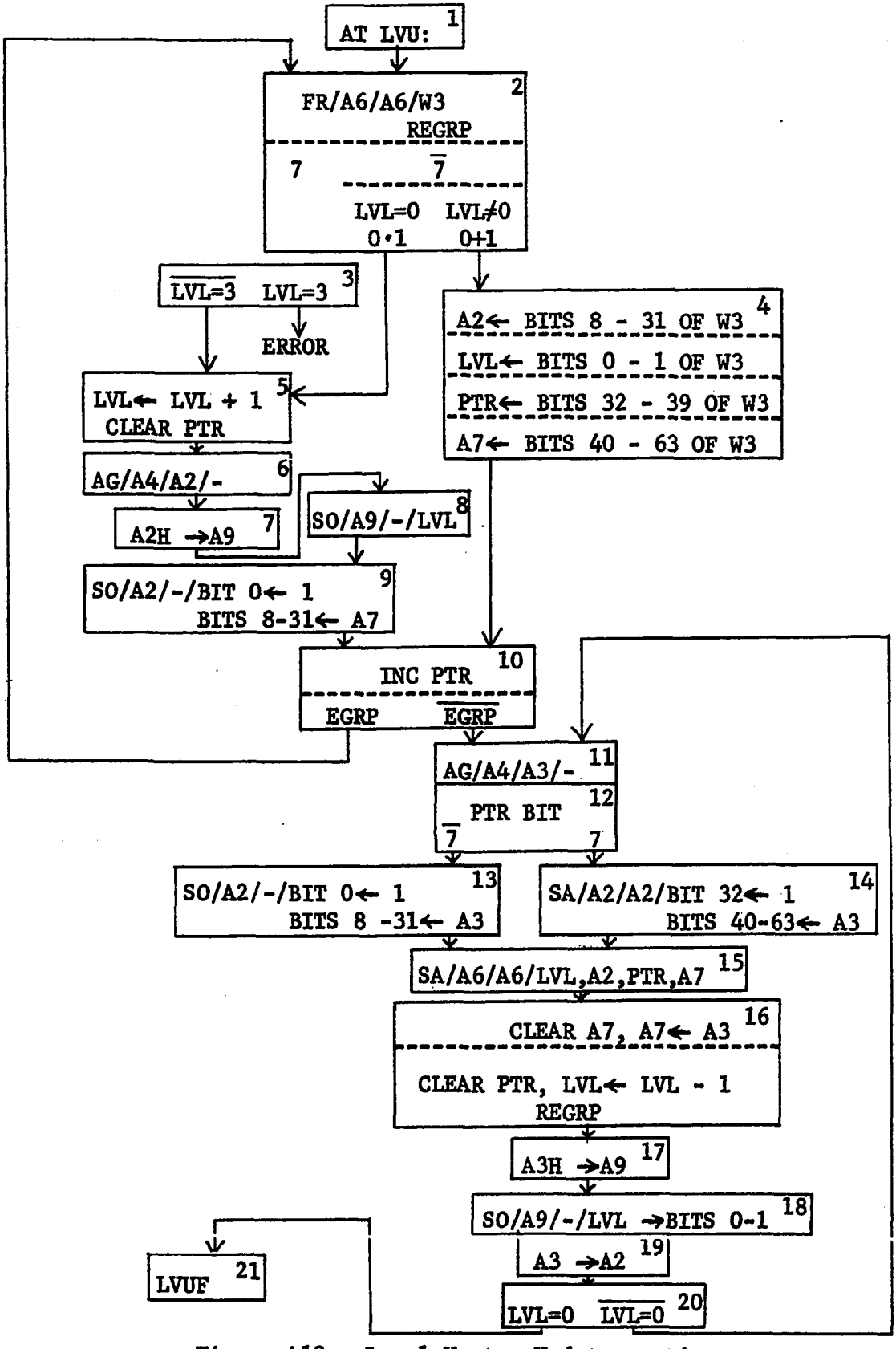


Figure A13. Level Vector Update routine

The updating routine waits in phase 1 for the initiating signal. Then the top of the stack is fetched into word register W3 where bit 7 is checked. If it is on, then there is nothing on the stack and the contents of LVL are checked to make sure that no level higher than 3 is created. The EGRP flip flop is also reset in phase 2. If the next level is not going to be greater than 3, then the branch allocating a level vector is taken. Otherwise an error signal is generated. The LVL is incremented by 1 and PTR is cleared in phase 5. The new level vector is assigned to the level 0 address register A2 by phase 6. Phases 7 and 8 then take care of loading the contents of LVL into the proper bits of the group link word of the allocated group. Then in phase 9, halfword 0 of the new vector is loaded with the starting address of the used up level vector and the activity bit of the link code field is set. Phase 10 is then entered to call INCPTR.

Back in phase 2, a zero bit 7 means the stack is not empty and so the level of the addresses on the stack is checked by looking at bits 0 and 1 of the word on top of the stack. If the level is 0, then the address belongs to a previous structure point and so the branch allocating a new level vector is entered. If the level is nonzero, then the necessary address registers and counters are loaded from the proper bits of the word from the top of the stack. This checking of the stack is necessary in order to prevent vec-

tors from being linked to vectors belonging to another structure level. Checking for a LVL equal to 0 is a sufficient check since whenever a left group mark is encountered, the level 0 address is stacked and another level 0 group is allocated. Once the address registers and counters are loaded, then the INCPTR routine is called from phase 10. After this is done, then the EGRP flip flop is checked. If it is set, then a new level vector must be allocated and so phase 2 is reentered. Otherwise phase 11 is entered.

Phase 11 takes care of allocating a new lower level group to the new level address register A3. Bit 7 of PTR is checked in phase 12 to determine from which halfword the new lower level group is going to be linked. Then phase 13 or phase 14 takes care of storing the address of the new group in the proper halfword at the same time setting the activity bit. Both of these phases lead to phase 16 which stacks the contents of LVL, A2, PTR and A7 since the new group has been properly linked. Then in phase 16 the address of the new group is copied into address register A7. After this, PTR is cleared, EGRP is reset and LVL is decremented by 1. The level of the new group is stored in the group link word by phases 17 and 18. The address of the new group is loaded into the level 0 address register in phase 19. In phase 20 the value of LVL is checked. If it is not zero, then back to phase 11 to assign another lower level group. The creation

of lower level groups goes on until level 0 is reached. Once the level reaches zero, the termination signal is sent in phase 21.

The other section that will have to be modified in order to accommodate the Pseudo Level Vectors scheme is the Get Address Subscripted section. This scheme will use the same flow chart of the main Get Address subscripted routine as that of the Modulo 16 scheme shown in Figure A9. Again the main difference between this flow chart and that of the present scheme is the removal of the current pointer, rapid search and word scan sections. They are replaced by a single section called the search section, which is shown in Figure A14.

The function of the search section is to find the desired component by using each hexdigit of the hex subscript to trace out the path to the desired element. The additional registers and counters are also given in the figure. Phase 19 waits for the initiating signal and once the signal is received then the various cases are checked out. If there is no link field and the IN operation is on, then the INFF is reset and the termination signal is sent. If the LKFLD flip flop was on then the most basic case is at hand. This means a vector has been detected.

Once a vector is detected, phase 30 is entered. Here LVL is loaded with a 3, DKNT is cleared and flip flop FRST is

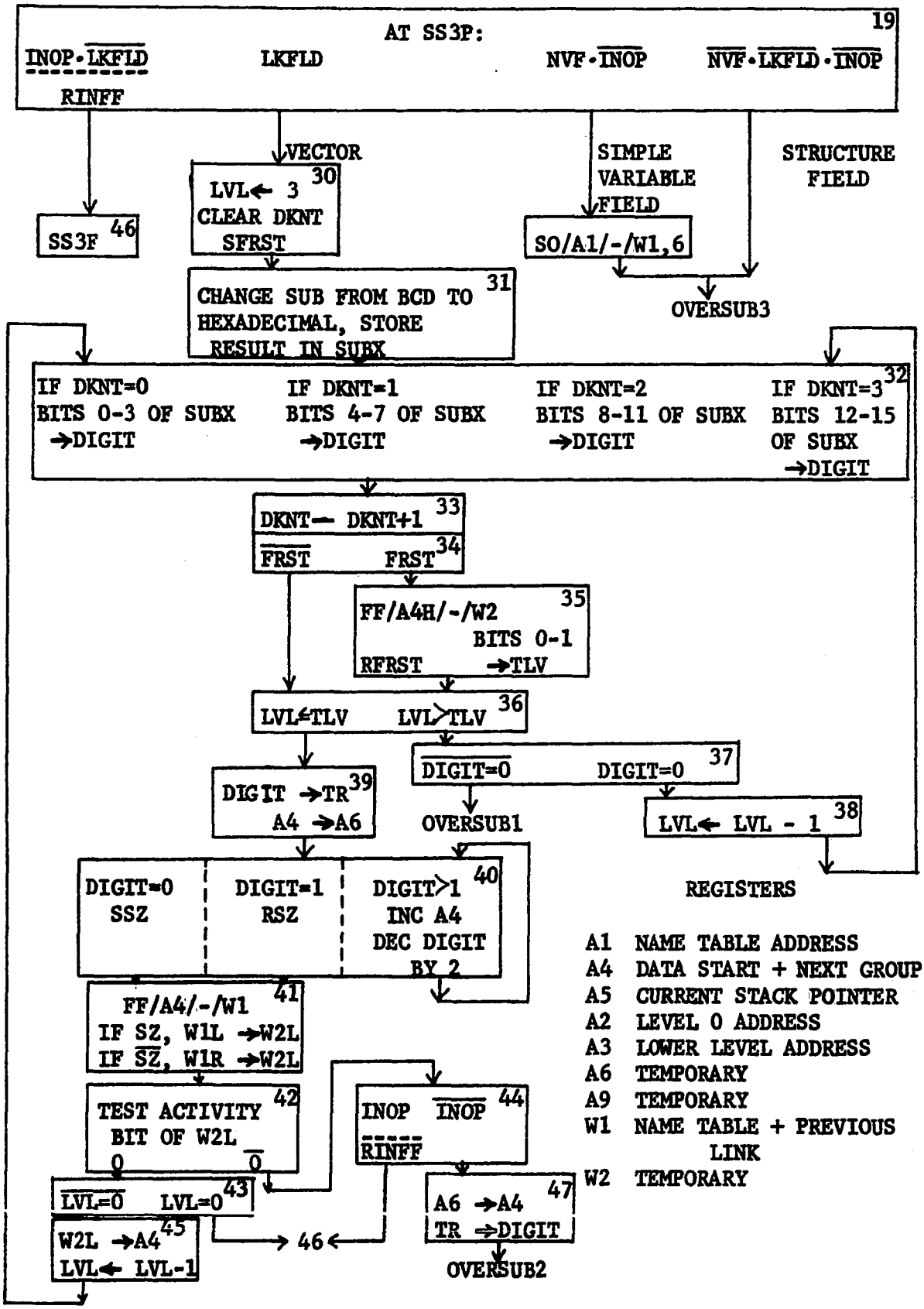


Figure A14. Search section of the Pseudo Level Vectors scheme

set. In phase 31 a routine to transform the subscript to hexadecimal is called. The subscript register SUB contains the subscript in BCD. The routine transforms this subscript to hexadecimal and stores the result in SUBX. This is a 16 bit register where the first 4 bits contain the third power hexdigit, the second 4 bits contain the second power hexdigit and so on.

In phase 32 the contents of counter DKNT is used to select which bits of of SUBX are copied into the DIGIT register. DKNT is incremented by 1 in the next phase and after this flip flop FRST is checked to see if this is the first time through phase 34. If it is, then fetch the group link word of the word in the destination register and load TLV from the level bits of the group link word. FRST is also reset. Phase 36 is entered from phase 35 and from phase 34 directly after the first pass. The function of phase 36 is to compare the contents of TLV and LVL. If LVL is greater than TLV, then proceed to check for oversubscripting. If LVL is greater than TLV and DIGIT is nonzero, then oversubscripting branch 1 is entered since the most significant digit of the subscript is of higher power of 16 than the most significant digit of the number of actual elements in the array. If DIGIT contained a zero, then just disregard it and decrement LVL by 1 before going back to phase 32. If LVL is not greater than TLV, then the contents of DIGIT and A4 are

copied over to TR and A1 respectively.

After this the value of DIGIT is checked in phase 40 in order to determine which halfword in the level vector is to be selected. If DIGIT is greater than 1, then increment A4 by 1 and decrement DIGIT by 2 and then repeat the check in phase 40. This is done repeatedly until DIGIT is either 0 or 1 in which case the SZ flip flop is set or reset. Phase 41 is then entered in order to fetch the word addressed by A4. The flip flop SZ is used to select which half of the word fetched is to be loaded into the left half of word register W2. In phase 42 the activity bit (first bit) of the link in the left half of W2 is checked. If it is active, then check if level 0 has been reached. If it has been reached, then send the termination signal. If not, then load A4 with the link in the left half of W2 and decrement LVL by 1. After this, go back to phase 32 to get the next hexdigit of the subscript. If the activity bit was not on in phase 42, then a check for the IN operation is done. An IN operation leads to termination after INFF is reset. No IN operation means another case of oversubscripting, so phase 47 is entered to load back the original contents of A4 and DIGIT before going into the second subscripting routine.

Back in phase 19, where the various possible cases are checked, two cases could be detected which can lead to creation of new structure. The first case is when a simple

variable is detected and there is no IN operation. This case leads to a third oversubscripting section after the proper link code has been put in the NTCW. The other case is when the field is not a link field. This means that subscripting is desired in a structure field which does not point to a substructure. This case also leads to the third oversubscripting section.

Three oversubscripting sections have been mentioned. The first one creates new structure starting from the topmost level since the subscript has been found to have hexdigits of higher power than the actual number of elements. This section creates links from the top level down to level 0. The second oversubscripting section is entered when the least significant hexdigit is more than the number of level 0 elements that exist in the path traversed by the search routine. This section then creates new null structure elements at the bottom level, level 0, after changing the end mark that has been encountered. The third and last oversubscripting section creates structure elements from scratch. This section uses each hexdigit, starting with the most significant, to trace the path where links have to be created until the actual data point referenced by the subscript has been reached.