This paper, motivated by the classical work of Bush, discusses the possibilities of designing an information processing system based on intrinsic addressing techniques.

The primary design objective is to develop a system with increased capability for non-numerical information processing.

Suggestions for the physical and programming system logic are outlined from a macroscopic point of view and some applications of the system are indicated.

An intrinsically addressed processing system

by J. E. Griffith

In 1945, Vannevar Bush published a famous article[1] discussing his "memex" machine in which information would be selected by methods of association, rather than indexing. This paper[2] will describe in general outline an information processing system which is based on methods of association, and which is designed as an extension of the von Neumann[3] general purpose computer organization.

In particular, we will comment on addressing and computer design, consider organization of a machine employing an intrinsic address memory, discuss the associated programming, mention some applications, and conclude by noting certain problems related to implementation.

Addressing and computer design

types of addresses

Briefly, there are two ways of ordering information. One way, called *extrinsic*, is basically enumeration. For instance, if we can enumerate the words in a memory, we assign a value to each word to denote its place in the order. This is called an *address* in modern computers. The correspondence between the contents of any word and its memory address is usually specified by an indexing algorithm which is programmed to effect the retrieval of the desired information.

The other method of ordering information, called *intrinsic*, is to specify some attribute of the data itself. This method has the

advantage that a class of data may be specified by any of its intrinsic attributes. The first method allows data to be retrieved only by discovering its place in the order. The second method allows data to be retrieved by specifying the characteristics of the class to which it belongs. We will arbitrarily define an extrinsic address as one which is not contained in the memory storage fields, and an intrinsic address as one that is contained in the memory storage fields. This implies that additional bits of storage may be supplied to contain the usual extrinsic address, which makes it intrinsic, or that any of the data fields may be used as an address, or both. This arbitrary definition is made to separate the two address types in terms of currently envisioned hardware, and to serve the purpose of this paper. A more complete definition will probably be necessary when further progress has been made in this field.

It can be seen that extrinsic addresses are unique and intrinsic addresses are not necessarily unique. This property of intrinsic addresses can be very useful, as Bush realized. One can thus use a defining attribute whenever uniqueness is not important, which is true in many information processing problems. A discussion of this point occurs in Reference 4. Reference 5 defines these types of addresses in greater detail. Slightly different definitions are given in Reference 6.

The actual hardware implementation of intrinsic addresses may conceivably be accomplished either by a selection or a scanning process. In this paper, we will assume that the scanning process is used, and that its speed is commensurate with the technology used to fabricate the memory. However, such technological considerations are outside the scope of this paper. Two types of scanning processes will be assumed; serial scanning by bit or character will be denoted as *serial intrinsic addressing*, and serial or parallel scanning by word will be called *parallel intrinsic addressing*. These definitions do not agree in detail with those given by Falkoff[5].

The actual process of scanning and comparing fields to be retrieved may be reviewed as nothing more than a table lookup operation. There are two ways of implementing such operations. One method is exemplified by the CONVERT command of the IBM 7090. In this case, the table is ordered and indexing procedures are used to select the correct table entry. The other method, used on the IBM 650, searches the table, comparing the value of the argument in each entry until the correct entry is found. This form of table lookup must be used whenever the table cannot be ordered. Many information retrieval applications require this form of table lookup. Thus, the scanning and comparing operation, which we can call a table lookup operation, gives us the "selection by association" that Bush desired. We may, therefore, consider one form of Bush's "memex" machine to be an information processing system designed around the table lookup operation, and this will be our approach.

It is not surprising that the table lookup operation can be

used as the basis for an information processing system, for it is a very general operation. Any logical or arithmetic operation may be duplicated by means of table lookup procedures. If one searches the table, one is actually performing the table lookup operation by intrinsic addressing methods.

computer design philosophy

A philosophical point or two is in order. The von Neumann computer organization is one in which the arithmetic-logical unit is used as a focal point of the machine design. In such an organization, the memory is used in a passive manner, meaning that it is not generally given any logical power of its own. Like memory, I/O equipment is also commonly used as an adjunct to the arithmetic-logical unit, and is usually used in a passive manner. We may consider this phase of modern computer design as Phase I.

Phase II of modern computer design is the phase that is presently beginning. This phase has been evolving around the difficulties of processing non-numerical data on von Neumann organizations. In this phase, the focal point of the design philosophy may be the memory, not the arithmetic-logical unit. If so, memory will become an *active* element of the machine design and will contain internal logical ability of its own. An alternative viewpoint is that the memory and the arithmetic-logical unit, inasmuch as they are both active, will merge and become one element of the machine. This combination would be a processor with internal memory functions. It does not seem important to worry about whether such an element will be considered as a memory with an internal processing capability or a processing device with an internal memory. It seems possible that computers designed during this phase may have superior non-numerical processing capability when compared to current von Neumann organizations. Phase II computer designs would be built as an extension of the von Neumann organization, and thus will retain all of the present advantages of the latter. The system described in this paper may be considered as an early example of Phase II computer system design.

It remains to be seen what the next phase of computer design will be; possibly I/O equipment will become a more dominant feature of the design.

We will next discuss the influence of active memories on the organization of an information processing system.

An intrinsically addressed processor

design principles

Although the system discussed in this paper uses intrinsic address techniques as the basic mode of operation, it is assumed that an actual operational system would have both extrinsic and intrinsic addresses in the memory. Many numerical problems will not be aided significantly with intrinsic addresses. However, this paper is confined to a discussion of intrinsic addresses in order to suggest their potential.

The example of Phase II design outlined in this section will

be based primarily on three principles. These are as follows:

- Memory as the focal point of the system design.
- Use of active memory rather than static memory.
- Table lookup operations executed in the memory.

We will now discuss the general organization of the processor.

general organization

Memory. The memory used in this system will have the following features:

- Fixed word length.
- Intrinsic addressing—no extrinsic addresses.
- All store and retrieval operations via table lookup.
- No word format.

Since the retrieval is associative, and not indexed, powers of two are no longer magic for word lengths. The word length, therefore, can be any that is convenient or economically desirable. We will assume a word length of 100 bits.

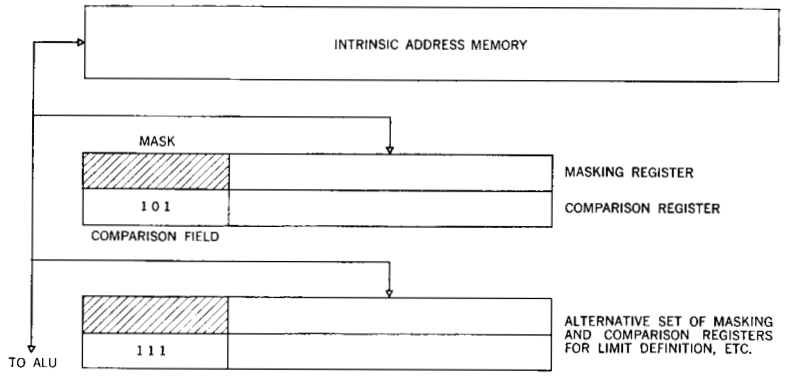
Arithmetic-logical unit (ALU). The ALU will be assumed to be similar to that of a typical modern computer (the IBM 7090, for instance). We will assume that it will execute the usual complement of operations.

Bulk storage. In this system, all memories will be addressed in the intrinsic mode only. Disk storage will use serial intrinsic addressing, drum storages will have both serial and parallel intrinsic addressing, and tapes will have serial intrinsic addressing only, as at present.

Input/output. The I/O equipment will consist of printers, card readers and punches, displays, etc., as at present.

Figure 1 gives a diagram of the system organization. This system is based on memory designs which have only intrinsic addressing. Extrinsic addressing schemes may be programmed as at present, and a technique for doing so will be discussed later. The standard mode of retrieval from the memories is by table lookup instructions. In order to implement these instructions, two

Figure 1 Machine organization



pairs of "mask" and "comparison" registers are included. The mask register of the first pair holds a bit mask which defines the field in which the argument of the table lookup operation is located. Any binary pattern may be placed in this register, and the pattern may be contiguous or non-contiguous. Its companion comparison register contains the value of the argument, if any, to be used in the table lookup operation.

The alternative pair of mask and comparison registers are used to supplement the first pair in some types of table lookup operations which require more information to be specified than can be contained in one pair of mask and comparison registers. Other pairs of these registers would probably be required for more complex forms of table lookup operations.

Once the appropriate registers are loaded, various table lookup operations may be used to retrieve information from the memories. The retrieved information may be sent to the ALU for processing, or to the I/O gear, or to another memory. The reader may assume any such units that he pleases. They will not be discussed here.

table lookup operations

Many varieties of table lookup operations are possible, but this paper will deal with a rudimentary set in order to suggest how they may be used. The set included here is arbitrarily chosen and almost certainly does not represent the best choice. We will assume that our memories will respond to the following set of table lookup operations. Notice that these imply parallel intrinsic addressing wherein all bits of one word are compared at once. Depending on the hardware, many or all of the words in such a memory may be simultaneously scanned and compared. The specific operations are now described.

1. *Write in first blank*—this is a table lookup operation which locates the "first" blank word (first is known only to the machine) and writes the word to be stored into the blank word. Notice that successive words to be written in memory will not, in general, be stored in adjacent locations.
2. *Read out exact match*—in which all words that exactly match a masked field in a comparison register are read out. One memory cycle may be required for each word read out.
3. *Read out highest (lowest)*—in which the words containing the numerically highest (lowest) field as specified by a mask in a register are read out. One memory cycle may be required for each word read out.
4. *Read out next highest (lowest)*—in which the words containing the numerically next highest (lowest) valued field relative to that specified by a masked comparison register will be read out in one memory cycle per word. Only one number will be the highest, though it may occur more than once.
5. *Read out highest (lowest) in a range*—in which the words containing the highest (lowest) values are read out as in Number 3, but the value of the field is limited to the range between two values specified by two pairs of masked comparison registers.

6. *Read out next highest (lowest) in a range*—in which the words containing the next highest (lowest) values are read out as in operation Number 4 except that the value of the field is limited to a range specified by two pairs of masked comparison registers.
7. *Read out nearest logical match*—in which the words containing fields for which the greatest total number of bits match the contents of a masked comparison register are read out. This is the logical equivalent of operation Number 4. Many bit arrangements may give the same number of logical matches.
8. *Store in field*—in which the contents of one masked comparison register are stored in all words which match another masked comparison register. If two other masked comparison registers are used, the contents of the first pair may be stored in all words bounded by the two values of the masked comparison registers. Note that the location of the bits to be stored may coincide with, overlap, or be disparate with the location of the bits defining the words to be modified. This is a generalized store address operation. This operation will take one or two memory cycles total for any number of words changed.
9. *Read out in order*—in which the words specified by a masked register field or limited to a range specified by two masked register fields are read out in order from high to low or low to high. This is a "sort" operation which allows the words selected to read out in sorted order. This operation will require one memory cycle per word read out.
10. *Chained lookups*—in which a masked portion of the word read out as a result of any of the above read-out operations is taken as the argument for another lookup. This operation can proceed automatically until no read out occurs, or it may stop after each read out awaiting a signal to proceed from the program or ALU.

If the bulk storage files (disks) in a system can operate in the serial by character mode as well as the parallel by word mode, we need to define one more table lookup operation for use only with memory devices that operate in the serial mode. Table lookup operations Numbers 1–10 hold for serial intrinsic devices as well as parallel intrinsic devices.

11. *Read out longest match*—in which the longest sequence of bits or characters that match the value of a masked comparison register are read out. The sequence may start from either end of the value field to be matched, depending on the mode of operation specified.

Table lookup operations Numbers 10 and 11 are related to operations contained in the AN/GSQ-16 Air Force Automatic Language Translator[7].

Generally speaking, it is preferable to have readout from all memories operate in both *destructive* and *non-destructive* modes.

This will facilitate certain storage allocation problems since the normal mode of storing is to write in "first" blank words.

The programming system

symbolic addressing

In a computer with a memory using parallel intrinsic addressing over the entire word length, the use of table lookup operations allows some interesting programming techniques. For example, suppose that a programmer should arbitrarily reserve the first three or four character positions (the memory is binary; therefore an equivalent number of bits would be reserved, 24 bits for four characters in this case) of each word as an address field. This particular formulation requires that each instruction have its own name whether or not it is used by the programmer. The addressing data obviously requires an inordinate amount of space but will serve here for tutorial purposes.

The address field would be defined by a pair of masks and comparison registers as in Figure 1, and the definition is thus entirely under control of the programmer. He may consider and use this field both as an extrinsic or an intrinsic address thus affording complete generality. In the *extrinsic mode*, he would fill the field with a sequence of numbers or characters. In the *intrinsic mode*, he would fill the field with symbols or characters of his own choosing, not necessarily sequential. A tag or name which distinguishes instructions from data may also be required in some cases.

In the intrinsic mode, any three or four symbols may serve as an address, and it is not necessary to compute or index one's way to an address. The programmer may ask for data or instructions in terms of these fields exclusively, if he pleases. Notice that:

- He does not have to know the extrinsic location of the data or instructions (table lookup operation Number 2).
- He may proceed by table lookup to the next highest or lowest address (table lookup operation Number 4) if the addresses are sequential.
- He may change the contents of any number of words simultaneously in one memory cycle (table lookup operation Number 8).

Therefore, it may be possible to write a program in symbolic notation of the programmer's own choosing and have the program executed in this notation. Since the program is always in symbolic notation, all debugging and modification will be performed in programmer's language.

List structures and push-down storage techniques are not hampered in any manner. If their use is desired, they may be implemented with ease. Since there are no absolute addresses in the memory, assembly routines may not be very useful. The function usually performed by heading cards in modern compilers is performed in this system by table lookup operation Number 8.

An interesting feature of intrinsic addressing is that an indication must be given whenever a table lookup operation fails to cause a word to be read out of the memory. This may be used to advantage when there is insufficient room in the memory for all subroutines to be used in the execution of a program. When one asks for the first instruction of a subroutine (transfers to its entry), and fails to retrieve it, an assumption may be made that it is not in the memory.

An I/O supervisory routine may then be instructed to write the subroutine into the memory. A single instruction (table lookup operation Number 8) may then be used to erase enough memory to make room for the subroutine. There are many ways in which this may be done; one way would be to indicate the priority of information by the first character of the symbolic address field. An AXXX address would be high priority information that cannot be erased, and would indicate erasable storage or information of the lowest priority. Addresses beginning with letters B through E would indicate an intermediate priority scaled according to the relative position of the letter in the alphabetical sequence.

When a subroutine, for instance, is to be read into memory, and there is insufficient blank storage to contain it, table lookup operation Number 8 may be used to erase enough low priority information to provide the necessary space. If one does not have individual addresses coded to indicate the exact table entries to be erased, there is danger that some information will be erased that should be retained. An address like FAXX or FBXX could be used to indicate individual blocks of FXXX storage. However, another approach is possible and will be discussed later.

A frequent problem in the control of I/O units is that of main- **input/output control**

Figure 2 Input/output control

I/O SERVICING TRIGGER

1

I/O UNITS 1, 4, 5, 7 REQUIRE SERVICING. PRIORITY ORDER: 4, 1, 7, 5, UNLESS OTHER I/O CONTROL INDICATORS ARE TURNED ON DURING SERVICING.

CONTROL INDICATOR BIT	3 BIT PRIORITY FIELD	CONTROL WORD FIELD
1	110	CONTROL WORD #1
0	111	CONTROL WORD #2
0	001	CONTROL WORD #3
1	111	CONTROL WORD #4
1	010	CONTROL WORD #5
0	101	CONTROL WORD #6
1	100	CONTROL WORD #7
0	101	CONTROL WORD #8

taining priority control over several units operating simultaneously. Usually, this requires that the highest priority units be serviced first and lower priority units be serviced in order of their priority. When several I/O units need servicing simultaneously, the solution to this problem becomes very complex to program.

With table lookup procedures, this problem is somewhat easier. Suppose that (for simplicity) we have a control word memory with a capacity of one control word per I/O unit. Each location in the control word memory will be assigned to a particular I/O unit, although this is not a necessary feature (see Figure 2). The high order three bits of each control word will be reserved for priority indication. The priority will be assigned by a supervisory routine and may be changed at any time under program control. Another bit of each word will be reserved for control indication. This bit will then be turned on whenever the unit requires servicing. An I/O servicing trigger X in the machine will be turned on when *any* I/O unit needs servicing and will be turned off only when no I/O units need servicing.

When the I/O service trigger is turned on, this will cause the program control to be transferred to the I/O supervisory routine. This routine will then perform a table lookup operation on the control bit plus the three-bit priority indicator field. Table lookup operation Number 3 (highest value) will thus cause the retrieval of the control word of the highest priority I/O unit that needs servicing. When servicing is complete, the control indicator bit is turned off. If the I/O service trigger X is still on, another table lookup operation Number 3 will retrieve the next highest priority control word. This operation, repeated as long as the I/O service trigger X is on, will cause the highest priority I/O unit at any given instant to be serviced first. When the I/O service trigger X is no longer on, control will be returned to the main program. Notice that no scanning routines are necessary; they are eliminated and replaced by a single table lookup operation that performs simultaneously the function of scanning and highest priority selection. Thus, I/O supervisory routines may be aided by the availability of memories that allow table lookup operations to be performed.

programming multiplexed computers

In this paper the term "multiplexed computers" refers to an installation of several computers, not necessarily all alike, which can operate concurrently on the same or different problems. For the simple case that will be described, all of these machines will use the same memory. Figure 3 shows a sample system using three computers. In order for each machine to separate its data and instructions from that of the other two machines, each machine will use a different initial letter in the symbolic addresses of its data and instructions. A different *range* of letters would be used for each machine in an actual case, and the reader can develop more sophisticated schemes for himself. Thus, one machine will use symbolic addresses beginning with A, such as AXXX; another machine will use symbolic addresses like BXXX; another machine will use CXXX. No confusion will result since each machine will

determine its own addresses within the set of symbols assigned to it. If certain letters are not used, gaps in the natural sequence will occur. If indexing procedures are used, and if one accidentally indexes his way beyond his assigned set of symbols, no instructions or data will occur with the address produced, thus no table entries will be read out—a warning of potential trouble.

An alternative method would be to place a specially coded word at the end of each block of data addresses. An extra pair of mask and comparison registers would automatically scan each memory word read out and cause machine interruptions or transfers of control if the appropriate match were obtained.

During the solution of a problem occupying the capacities of the three machines, one machine may desire to pass data to another machine. If so, all that is necessary is to convert the first letter of the data words to be passed to the letter used by the other machine. Converting an AXXX, say, in certain data-word-addresses to a BXXX will have the effect of passing data from machine *A* to machine *B*. Such conversion will take only *one* memory cycle for any number of data words. The speed of this operation is due to the fact that it is nothing more than another table lookup instruction, except in reverse. Table lookup operation Number 8 is the one referred to here.

There may be situations in which protection is desired against access to one machine's data or instructions by another machine. If so, the address bits may be randomly spread throughout the data word format by insertion of the appropriate mask. This pattern will be difficult for another machine to recover but the technique will not protect the data from being written over by the other machine. Generally, the latter type of protection is best built into the machine hardware.

Since the address field of each symbolic instruction is symbolic and may not be sequential with respect to an order of execution of program steps, there is a question as to how the order of execution may be specified. In this system, we will use table lookup operation Number 10 (chained TLU). A sequence of instructions may be listed as in Table 1.

The right-hand table shows the chained TLU formation. This is akin to a technique used in computers like the IBM 650, wherein each instruction specifies the location of the next instruction. The difference here is that the location of the next instruction may be in symbolic notation. The left-hand part of each pair is the present instruction to be executed, and the right-hand part is the next instruction to be executed. The A+4 entry shows two possible next instructions, A+5 and B+1. In other words, A+4 is a conditional branch and A+6 is the return to the start of the loop. If the address symbols were as shown here, the order is evident, and could be indexed. However, the symbolic notation could also be as shown in Table 2.

It will be noted that the chained table lookup form does not require sequential (extrinsic) addresses and will operate correctly

Figure 3 Multiplexed computers

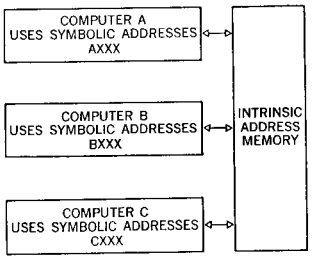


Table 1

Sequential	
A + 0	
A + 1	
A + 2	
A + 3	
A + 4; B + 1	
A + 5	
A + 6	
A + 0	
Chained	
A + 0	A + 1
A + 1	A + 2
A + 2	A + 3
A + 3	A + 4
A + 4	A + 5; B + 1
A + 5	A + 6
A + 6	A + 0

order of execution

Table 2

Sequential	
GHQI	
RVAD	
KBEM	
TYWM	
MDFL, NPKY	
QRXA	
IFMB	
GHQI	
Chained	
GHQI	RVAD
RVAD	KBEM
KBEM	TYWM
TYWM	MDFL, NPKY
MDFL	QRXA
QRXA	IFMB
IFMB	GHQI

with either intrinsic or extrinsic addresses. Since this method of specifying order is independent of the actual address, some restrictions inherent in indexing schemes are no longer present. In the above example, any entry may be modified without concern for the effect on other addresses in the chain. Also, modifications to the logical structure of the chain may be accomplished by modifying or adding entries to the table.

The chaining of table lookup operations may thus be seen as a technique for specifying structure that is independent of the arguments (address fields) of the individual table lookup operations. Thus, it is an address invariant scheme which includes indexing as a special case. Each table entry will generally denote the two end points of one segment of the mesh represented by the table, but several segments joined to a common end point (a branch) may also be denoted by a single table entry. This example illustrates how list structures may be implemented using intrinsic addressing techniques. The use of intrinsic addresses is not necessary; extrinsic addressing methods with indexing will sometimes be more suitable than fully symbolic intrinsic addressing techniques. If the use of extrinsic addresses is desired, they may be simulated as discussed above, or they may be built into the computer hardware as at present.

Application of table lookup chains to problems in information retrieval may also be possible. For instance, a chain may be used to represent the mesh of bibliographic references on a given subject. The structure of the bibliography would be denoted by the chain structure, possibly allowing conclusions to be drawn about the degree of relation between two different items in the bibliography. For instance, suppose that it is desired to order a bibliography by date. Use of table lookup operation Number 4 (next highest value) will allow the chain of bibliographic items to be retrieved in chronological order. Each item retrieved would then have its symbolic address (or other distinguishing attribute) added to a new table lookup chain. The new chain thus built up gives the structure of the bibliography in chronological order. This particular solution could also be obtained by sorting, using table lookup operation Number 9 (read out in order).

If the bibliography were to be arranged in a structure such as that given by Roget's Thesaurus, sorting may not work. In this case, each Roget classification would be made up into a chain. The main classification, or beginning, of each chain would be used to make another chain. The words in the titles of each bibliographic entry could then be compared with each entry in each chain and type of relevance indicated by the main classification of each sub-chain that correlated with the words of the titles.

This technique might possibly be extended to provide a scheme for analyzing articles for subjective content. For instance, political concepts are sometimes indicated by conjunctions of words that often appear in other contexts. A chain of words may be set up to indicate a possible format of context and word order.

editing

Editing may be an easier problem in this system. The usual case of rearranging several items into a line print format may be solved by attaching symbolic addresses to each field. The order of the symbols should be the order in the print format. Then, the use of table lookup operation Number 9 (read out in order) will cause the fields to be read out in the desired order. Alternate formats of the same set of fields may be represented by alternate tags.

The operation of expansion and contraction of the print format may be accomplished by using multiple symbol tags to denote the level in the format that alteration is to take place.

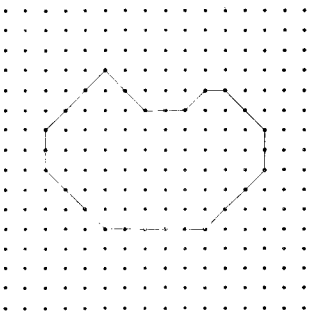
debugging

Many new techniques for debugging are made available by table lookup procedures. For instance, table lookup operation Number 2 (read out exact match) may be used as a trap for spurious addresses. The suspected or known spurious address can be specified as an argument and all words with that address will be retrieved. Since it is not necessary to know where they are, this operation eases the problem of locating spurious addresses. Alternatively, this operation may also be used to locate all instructions that refer to a given location or subroutine.

Table lookup operation Number 8 (store in field) is very convenient for changing addresses of instructions. One pair of masked comparison registers may specify the value of the address that is desired. This operation will convert all addresses of the given value to the new specified value without specifying their location in memory, or specifying how many such addresses exist in the memory. This suggests the elegance of intrinsic addressing techniques.

Table lookup operation Number 9 (read out in order) may be used to read out all instructions in order by address value. This may be used to examine chains and other routines for consistency of addressing. Reading out symbolic locations in order may help to establish a measure of correspondence between the addresses used and the location specified. For cases where the symbolic address is modified in manner analogous to indexing, this technique may not be useful.

Figure 4 Pattern problem



Applications to information processing

When Bush proposed his "memex" machine, he sensed that retrieval by association would be better than retrieval by indexing. Having roughly outlined the features of a computer system based on intrinsic addressing methods, or retrieval by association, it is now appropriate to examine in more detail the implications of Bush's suggestion. In order to do this, several application areas will be discussed with suggested directions of solution indicated.

pattern matching

Suppose that a simple closed contour lies on a two-dimensional mesh as shown in Figure 4. The mesh may have irregular or random spacing. Starting at any point on the contour, nearest neighbors are found by using table lookup operation Number 6 (read out

next highest, lowest in a range) and one is selected that lies nearest to or on the contour. "Next highest" and "next lowest" here refer to the coordinates of the point on the mesh. With a given point whose x and y coordinates are known, it is desired to find the nearest neighboring contour points by asking for the contour point with next highest and next lowest x and y coordinates. Squaring the differences in coordinates and comparing the values may be necessary in some cases.

Repeating this action until closure is obtained defines the contour. Doing the same for another unknown contour defines it also.

To compare the two contours, select any point on one contour and calculate a few vectors to other points on the same contour, to every tenth point on the contour, for instance. Repeat this procedure with each point as the starting point. On the other contour, do the same for a point chosen at random. If this set of vectors is not nearly alike any of the sets calculated for the first contour, repeat the procedure starting with the next point on the second contour. Table lookup operations Numbers 5, 6, and 9 should aid in the comparison. If there is a sufficiently good correspondence, more refined calculations will determine how alike the two contours are.

Notice that only data on or near the contour is examined. All other data is unnecessary. This procedure will not determine the *shape* of a contour, only the degree of likeness with another.

More sophisticated methods may be used to compare the contours, and there are probably better ways of picking the starting points of the calculation. More complex contours will require more elaborate tables for solution, but the basic table lookup method should remain unchanged.

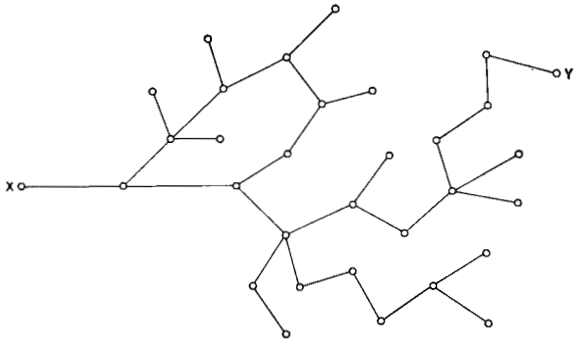
maze solving

An alternative version of the preceding problem is found in the recovery of multiple branched trees from unknown data. In this problem, closure is of secondary interest. One form of this problem is often termed *maze solving*; the object is to find a path from X to Y through a maze or labyrinth of data. It is interesting to note that a favored technique of solving such problems on von Neumann type machines is that of table lookup. List structures are often used in order to provide an approximate degree of intrinsic addressing not actually found in the machine hardware.

We will now outline a method of attacking these problems with an intrinsically addressed computer. The particular technique used is much like those now programmed on extrinsically addressed computers, but adapted for purposes of exposition. Figure 5 shows a typical branched tree. The coordinates of each end of each branch is given. The question arises as to whether or not there is a path from point X to point Y and its identity. Reference 8 discusses a better method in more detail.

To start, we will assume that we have a list of all possible segments with coordinates of both ends given. Label all segments via table lookup operation Number 8 (store in field) with a "U" in the symbolic address to indicate that each segment is "un-

Figure 5 Maze solving



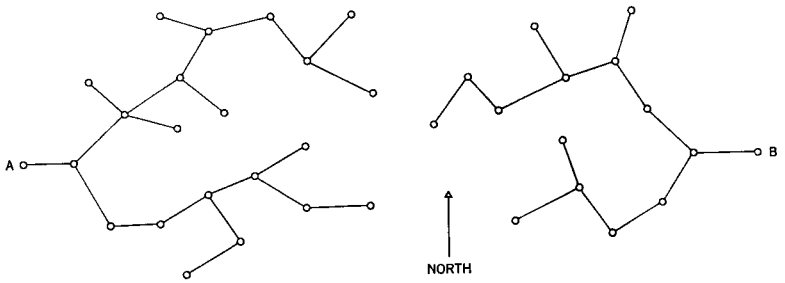
examined". Now examine the list of segments for any "U" labeled segment. Test to see if it joins another segment at both ends or not. Table lookup operation Number 2 (read out exact match) would suffice here. If the coordinates of the far end of the segment do not match the coordinates of any other segment end point, we may say that the segment comes to a dead end. Label the segment "D" (for dead end) in the symbolic address field. If it connects with one or more other segments, label it "E" for "examined". At any time, the list of segments may be operated upon by table lookup operation Number 2 to see if any "U" labels remain in the list. When all "U" labels are gone, each segment will have either an "E" or a "D" label.

Starting at X, retrieve the segment or segments with the coordinates of X at one end. If any of the segments have a "D" label, ignore them. Label any branch point "B" (an additional label) if it has two or more "E" segments. Picking an "E" segment at random, from the coordinates of the far end, retrieve the connecting segments. If more than one "E" segment is found, label the point with a "B" as before. Proceed until point Y is found or until a point is reached where only "D" segments occur. If a branch point is reached that has only "D" segments as successors, label the branch itself "D" instead of "E" and return to the preceding branch point, converting all "E" labels to "D" for any intervening segments. At the preceding branch point, label the examined "E" leg "D" and pick a new "E" leg. Doing this will label long legs that eventually end up in a dead end.

If by chance, a closed loop is discovered where the path returns to itself, it may be labeled "D" by the preceding routine if no other "E" exits are connected to it. The preceding routine will discover these if they exist.

When point Y is reached, the list of "E" branches must be examined to make sure that all long dead end branches have been eliminated. When this has been done, one may start at X and, picking "E" branches only, end up at Y. Other calculations may be performed to find the shortest path from X to Y or any other selection that is desired among the possible paths from X to Y.

Figure 6 Double maze



A practical version of this problem is represented by Figure 6. Suppose that Figure 6 represents the known information about a railroad network. Is there a path from A to B? If not, the eastern-most dead ends of the A network may be compared with the western-most dead ends of network B. The nearest dead ends give one clues as to possible connections that are missing from the data. Special efforts may then be guided to obtain the missing data. There are many versions of this problem in intelligence work. Sometimes the most important information in a file is the "missing link" type of data that is not actually there.

In the solution of this problem, note that the bookkeeping is accomplished by manipulating the symbolic addresses. This illustrates an advantage of table lookup procedures. In actual fact, the symbolic address fields constitute a set of table entries. These entries are then manipulated by table lookup operations. Bookkeeping is thus accomplished in a natural manner during the solution of a problem. Extrinsic addressing techniques sometimes require more superstructure in list techniques, push-down storages, etc.

When one considers the solutions of maze problems that are much more complex than those described here, he will often note that it is desirable at times to construct routines which are somewhat self-adjusting. One situation in which this is desirable is denoted by a maze whose structure changes in time. It is then necessary to find a new path through the maze each time the structure is changed. We will now outline a procedure for aiding this type of routine.

self-adjusting routines

Suppose that we have a set of subroutines each of which can be applied to some data representing a problem. Some of the subroutines will produce a more desirable output from the data than others. By trial, we may determine the success of each subroutine. We would then label the symbolic address of each subroutine with a letter that indicates its degree of success. An address AXXX would be a very successful subroutine and an address FXXX would be a very unsuccessful subroutine. If we wished to produce the most desirable output from a set of data, we would tend to use the most successful subroutines.

However, if the data changes from time to time, the degree of success obtained by any given subroutine will also likely change. In this case, the symbolic address will also change. This change would normally be executed with table lookup operation Number 8.

At any time, when one desires to apply the most successful subroutines to the data first, then the next most successful, etc., table lookup operations Numbers 3 and 4 will allow this selection to be made with ease. The symbolic addresses are, as in maze solving, nothing more than a table of functions whose appropriateness varies with the data presented. The data presented can be viewed as a set of arguments for which the best function that occurs in the table is desired. It can now be seen that the table of functions will adjust itself to the data (or arguments) presented. This is not learning, but the table of functions may adapt itself to situations never before experienced or foreseen.

If the set of available subroutines also changes in time, the use of table lookup operation Number 8 will allow the FXXX routines to be erased so as to make room for new, untried routines in the table. At any time the unsuccessful entries may thus be purged from the table to make room for new ones.

The reader can see that the table need not represent a set of subroutines. It could represent various paths through a maze, and the question occurs as to whether self-adapting maze solving routines can be developed.

A previous section referred to the storage allocation problem wherein there was insufficient storage space for subroutines. The symbolic address of each subroutine could be changed (e.g., from FXXX to EXXX) to indicate the frequency of use. When it becomes necessary to erase storage to make room for another subroutine, the FXXX subroutine will be erased first, then EXXX, etc., until enough storage has been vacated to make room for the new subroutine. In this manner, the least used subroutines will be erased first.

This technique may also have application in multiprogramming, where it is desired to delay low priority programs in favor of high priority programs.

filtering

If an intrinsic address memory is used as a storage for tables of indicators, it may be operated as a filtering mechanism for data. For example, suppose that it is desired to extract all sentences from text that have certain words in them. Let us suppose that it is desired to extract from a large source of text, such as a library, all sentences that have any of the following words included: rain, precipitation, clouds, cloud cover, rainfall, temperature. If we were to store this list of words in an intrinsic address memory and then pass the running text through a register system which performed table lookup operation Number 2 (read out exact match) on every word of the running text, a read out would occur only when a word in the running text matched one of the words in the stored list. The read out indication could be used to cause a copy of the sentence to be prepared on a separate tape unit.

In some message switching centers, it might be desired to switch selected messages to other addresses. This arrangement thus becomes a dynamic message switching center in which the message heading as well as the text would be examined and messages switched in real time. The analogy to a filter is clear in that the intrinsic address memory allows only that data which meets the stored criteria to pass over the selected lines.

Comments on implementation

A remaining consideration is that of the hardware implementation. There are several questions related to this point which require thorough study and discussion. One question is: can the major technological and engineering problems be solved in order to build intrinsic address memories with features such as those described? Will the improved capability be worth the difference in cost relative to the cost of extrinsic address memories? Will the future availability of very large economical storages and more sophisticated programming techniques weaken the advantages of intrinsic address memories?

Questions such as these will have to be considered in detail before we can balance the cost of producing intrinsic address processors against the gains to be expected from their use.

This paper has briefly described an information processing system based on intrinsic addressing techniques. The system illustrates some of the concepts which may be used if intrinsic address memories become available as production devices. Some of the implications of Bush's paper are pointed out. It is felt that intrinsic addressing techniques will permit computers to be designed that have increased non-numerical processing capability, and that once such machines are built, computer designers may be in a better position to tackle "intelligent" machine design. The characteristics of advanced information processing systems should be improved by the availability of intrinsic address processors.

CITED REFERENCES AND FOOTNOTES

1. Vannevar Bush, "As We May Think," *Atlantic Monthly*, **176**, 101, July, 1945.
2. This is an altered version of a paper originally prepared for the First Congress on Information Systems Sciences, November 20, 1962, Hot Springs, Va. (sponsored by the MITRE Corp.). Parts of this paper were also presented orally to the Washington, D. C. Chapter of the IEEE PGEC, March 14, 1962.
3. A. W. Burks, H. H. Goldstine, J. H. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Report prepared for the Research and Development Service, Ordnance Department, U. S. Army, June, 1946.
4. Bertrand Russell, "Introduction to Mathematical Philosophy," Second Edition (London: Allen and Unwin, 1920) pp. 12–13. A referee has pointed out that Russell uses these terms for identifying classes without respect

to order. He does not discuss computers or addresses. The terminology has been adapted for the purposes of this paper.
5. A. D. Falkoff, "Algorithms for Parallel-Search Memories," *Journal of the Assn. for Computing Machinery*, 9, 488, October, 1962.
6. G. W. King, "Table Lookup Procedure in Language Processing, Part I: The Raw Text," *IBM Journal of Research and Development*, 5, 86, April, 1961.
7. J. L. Craft et al, "A Table Lookup Machine for Processing Natural Languages," *IBM Journal of Research and Development*, 5, 192, July, 1961.
8. T. Singer, P. Schupp, "Associative Memory Computers from the Programming Point of View," (MITRE Corp.). AD 416 301.