A MICROPROGRAM CONTROL UNIT BASED ON A TREE MEMORY

by

N. Tokura

STAN-CS-75-514
AUGUST 1975

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

A Microprogram Control. Unit Based on a Tree Memory

Nobuki Tokura

Department of Enformation and Computer Sciences
Faculty of Engineering Science
Osaka University
Toyonaka, Japan

Abstract

A modularized control unit for microprocessors is proposed that implements ancestor tree programs. This leads to a reduction of storage required for address information. The basic architecture is extended to paged tree memory to enhance the memory space usage. Finally, the concept of an ancestor tree with shared subtrees is introduced, and the existence of an efficient algorithm to find sharable subtrees is shown.

The printing of this research was supported by IBM Corporation. Reproduction in whole or in part is permitted for any purpose of the United States Government.

1. Introduction.

The limitation of pin count makes the architectural design of microprocessors difficult [6]. It leads to a somewhat restricted instruction and system capability. The bit-sliced modularization has been successfully used to make high performance modules. This method is good for regularly structured units such as ALRU (Arithmetic - Logic -Register unit), stacks, and others. However, modularization of control units with less regularity has not yet been achieved.

The modularization of control units is the principal theme of this paper, and structured programming the subordinate one. There have been several attempts to realize some control primitives, e.g. DO WHILE, IF THEN ELSE , on conventional machines. This is done not by changing the machine itself, but by limiting the usage of the machine. The efficiency of this restricted code is one of the important problems to be solved. In this paper we take an opposite approach by examining a machine oriented to structured programming. However, there seems to be no general agreement on what structured programming is [9]. Also, structured programming has been the subject of criticism, especially for its inefficiency [5]. As one proposal, we choose the ancestor tree program to be the basis of structured programming. This selection leads to an efficient instruction set and a simply modularized control unit. Section 2 presents the basic notion and a possible implementation. In Section 3, a paged tree memory system is proposed to answer the problem of memory chip efficiency. Also, a new paged memory system is described which has a distributed address table entry on bit-sliced pagedmemory. In Section 4, the problem of the coding efficiency of an ancestor tree program is examined. The result is a broader type of structured programming, allowing ancestor tree programs with shared sub-trees. The implementation and the existence of an efficient algorithm to find sharable subtrees are briefly described.

2. Tree Memory.

Let us first recall some definitions relating to binary trees [4]. In Figure 1, an example of a binary tree is shown. Each node has an alphabetical label for reference. Node A is called the root of the tree. The root is the unique node to which no edge enters. A node Y connected by an edge from a node X and placed on the left side of X is called a left son of X and the edge (X,Y) is called a left edge. Right sons and right edges are defined similarly. In Figure 1, node C is the left son of node B and node G is the right son of node E . If Y is a left son or a right son of X , then X is called the father of Y . A node with no sons is called a leaf; e.g. nodes D , F , G , I , K , M and N are leaves in Figure 1. If there is a path from X to Y, then Y is called a descendant of X and X is called an ancestor of Y . A node X is considered to be an ancestor and a descendant of itself.

For a binary tree T and a node X in T , a subgraph with the root X consists of all the descendants of X in T and edges (Y,Z) in T with both Y and Z descendants of X .

The depth of a node in a tree is the length of the path from the root to the node. The height of a tree is the length of a longest path from the root to a leaf. In Figure 1, node D is of depth 3 and the root A is of depth 0 . The height of the tree is 4 .

An ancestor tree is a binary tree with a (possibly empty) set of back edges. Each back edge connects a leaf to one of its ancestors, and each leaf has at most one back edge leaving it. Figure 2 shows an example of an ancestor tree. Edges (D,C) , (G,A) , (K,H) , (M,J) and (N,H) are back edges. For any ancestor tree T , the binary tree T' obtained from T by removing all the back edges is called the basis tree of T . The tree of Figure 1 is the basis tree of the ancestor tree of Figure 2.

A flowchart is a labeled directed graph with the following properties:

(1) There is exactly one node with label START which no edges enter.

(2) There aye nodes with label STOP from which no edges leave.

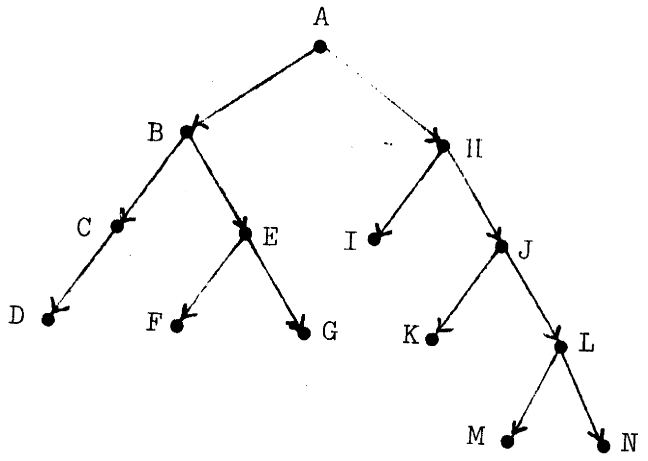


Figure 1. A binary tree

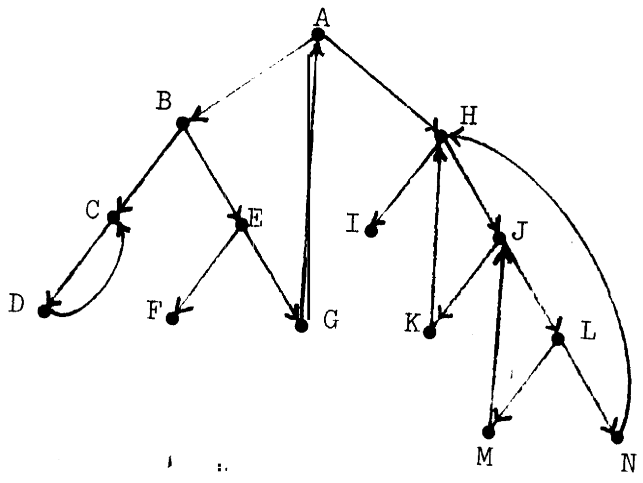


Figure 2. An ancestor tree.

(3) The other nodes are divided into two classes: one class is the set of nodes with functional labels and having only one outgoing edge, and the other class is the set of nodes labeled with test predicates and having two outgoing edges labeled 0 and 1 .

If a flowchart of a program is a labeled tree (or a labeled ancestor tree), then the program is called a tree program (or an ancestor tree program, respectively).

A random access memory of size $2^m \times n$ consists of 2^m registers each of which is capable of holding a binary word of length n . Each register is associated with a distinct binary word of length m which is called the address of the register. Usually, the addresses are treated as binary numbers ranging from 0 to 2^m-1 . This leads naturally to the concept of linear memory. However, there are other concepts. For example, two (or more) dimensional memory can be conceived [11]. Another idea is that of a tree memory [2].

In [2], Berkling considered the register with address (0 . . . 0 1) as the root in a memory of size $2^m \times n$ and the register (a_{m-1} a_{m-2} ... a_1 0) (or the register (a_{m-1} a_{m-2} . . . a_1 1)) as the left son (or right son, respectively) of the register (0 a_{m-1} a_{m-2} . . . a_1) . Figure 3 shows a tree memory of size $m = 3$. The register (0 . . . 0) is left unused. Berkling has shown that a simple shift register suffices to traverse the tree (Figure 4). By shifting the register SR left one place supplying 0 (or 1) to the right end, we can visit the left son (or right son, respectively) of the node currently pointed to by the register SR . Conversely, by shifting SR right one place supplying 0 to the left end, we can visit the father of the current node. The one-bit register v can be used as an overflow indicator which is 1 when the register SR points outside of the tree.

Berkling discussed several aspects of a tree memory, but apparently did not intend to apply his scheme to instruction sequencing. This is quite natural because the basic scheme doesn't have enough sequence control capability for the task. This paper, however, will show that by adding several features, his scheme becomes applicable.

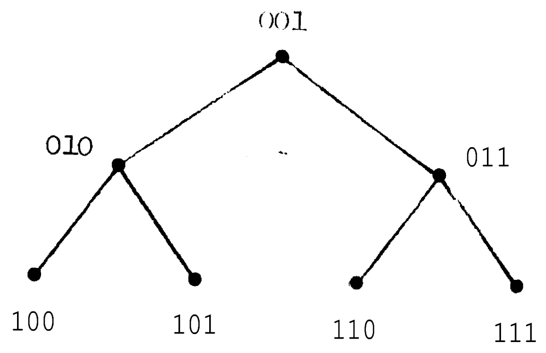


Figure 3. A tree memory.

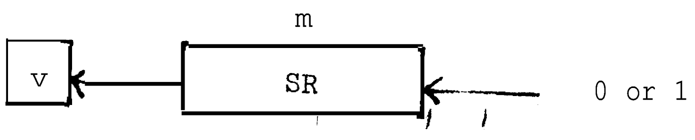


Figure 4. A shift register.

Engeler Normal Form.

It is known that every flowchart program can be transformed to an equivalent Engeler normal form [3]. The transformation is very simple:

> Try to make an equivalent tree program, but if a node already occurs in the path from the root to the currently scanned node, then make a back edge.

This is demonstrated by example in Figure 5. The normal form construction process must always terminate because the height of the tree obtained by removing the back edges cannot exceed the number of nodes in the original flowchart. Note that this transformation retains the equivalence of programs, but introduces some duplicated nodes, such as nodes G and STOP in our example. More precisely, this transformation preserves not only equivalence but also isomorphism, that is, the possible sequences of actions are identical. The ancestor tree defined here is slightly different from an Engeler normal form in that the latter may have a back edge from a non-leaf node. But this difference is not essential. We can easily obtain an equivalent ancestor tree program from an Engeler normal form by introducing a leaf for each back edge (if necessary). In Figure 5, G" is obtained from G' by introducing a new (no-operation) node A' .

Two Primitives.

Hence, we can concentrate on ancestor tree programs. To implement ancestor tree programs, the two sequence control operations GO DOWN and GO UP are sufficient.

The operation GO DOWN replaces a test instruction of a flowchart program. It has a selector field end is executed as follows:

(1) Select a signal specified by the selector field.

(2) If the signal is 0 (or 1), then visit the left son (or right son, respectively).

The operation GO UP is used to transfer the control along a back edge. It has a displacement field and is executed as follows:

(1) Set the number specified in the displacement field into a counter.
(2) While the counter is not zero, decrease the counter by 1 and visit the father of the current node.

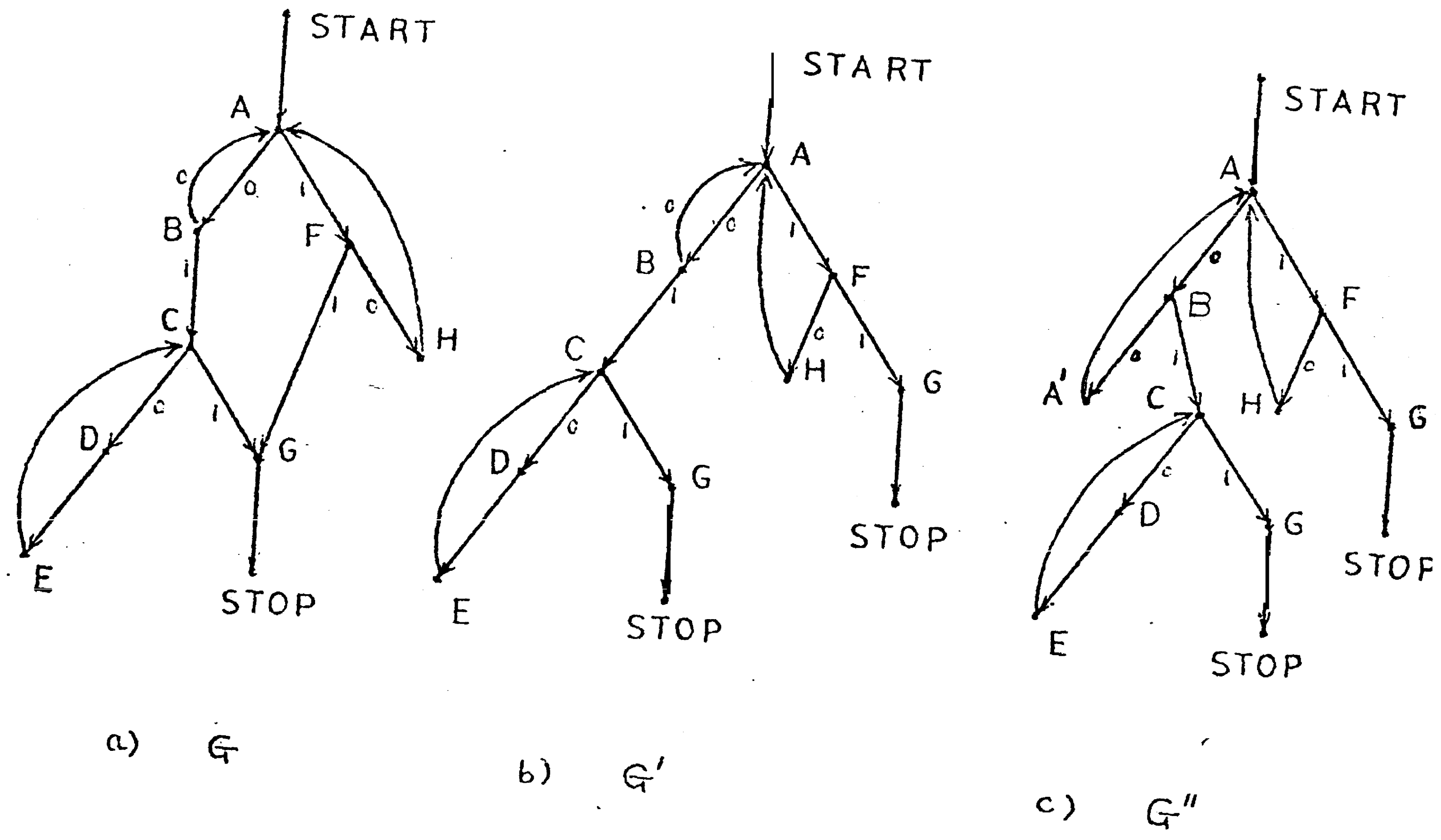


Fig. 5 a) A flowchart
b) its equivalent Engeler normal form
c) its equivalent ancester tree form

Microprogrammed Control Unit.

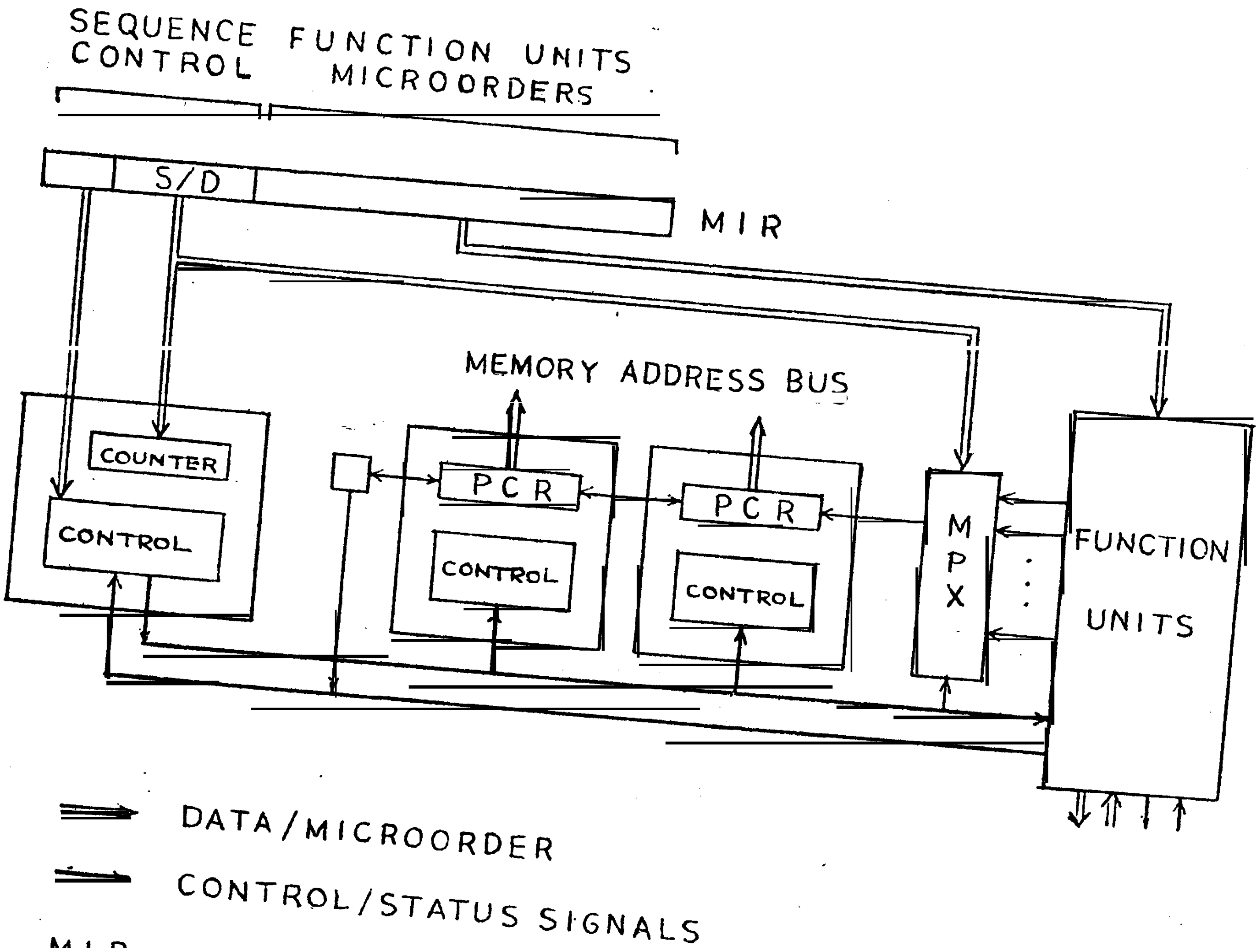
The microprogrammed control unit (for short, micro-control unit) is already modularized in comparison with its hard-wired random logic counterpart. The microprogram is stored in a microprogram memory. The form of micro-instruction is of broad range from vertical (highly encoded) instruction to horizontal (completely decoded) instruction. In the context of micro-processors, a medially encoded micro-instruction seems to be convenient. Highly integrated f'unction units such as ALRU will accept an encoded micro-order instead of a decoded micro-order. Because of the pin count limitation, communication between modules favors highly encoded information. Modularized function units can work in parallel. The sequence control of micro-instructions can be done by some modules. With this background, the micro-instruction is considered here to consist of one field for sequence control micro-order and several fields for f'unction micro-orders.

An Implementation.

A possible implementation is shown in Figure 6. The micro-instruction register (MIR) holds one micro-instruction. Each micro-instruction consists of two parts: one part is a sequential control part which has a 2 bit field for sequence control micro-order codes and a k-bit selector/displacement (S/D) field and the other part is a function part which has micro-orders for the f'unction units. The program control register (PCR) is a shift register which holds the address of the next micro--instruction. This register can be implemented as bit-sliced modules, The PCR receives orders from the sequence control block (SCB) such as shift right, shift left, and clear. The SCB executes sequence control micro-orders. There are four kinds of micro-orders. Each micro-order is executed as follows:

Pre-test GO DOWN

(1) Shift the PCR left one place with the output of the multiplexer shifted in. The multiplexer selects one signal from the function units and other external sources according to the S/D field.



MIR	:	Microinstruction register
SCB	:	Sequence control block
PCR	:	Program control register
MPX	:	multiplexer

Fig. 6 An implementation of microprogram control unit

(2) Start the execution of function micro-orders.

(3) After the comletion of all the function micro-orders, the micro - instruction pointed to by the PCR is fetched.

Post-test GO DOWN

(1) Start the execution of function micro-orders.

(2) After the completion of all the function micro-orders, shift the PCR left one place with the output of the multiplexer shifted in.

(3) Fetch the next instruction.

Go UP

(1) Start the execution of function micro-orders.

(2) If the S/D field is zero, then go to 3 ; otherwise set the value of the S/D field in the counter of SCB. While the counter is not zero, decrease the count by 1 and shift the PCR right one place. After completing the shifts, fetch the next instruction.

(3) The case with zero displacement is interpreted as a return to the root. So the PCR is cleared and a "1" is shifted into from the right. This is easily done by selecting a signal " 1" with selector (0 . . . 0) .

STOP

(1) Clear the PCR and others.

By a START/RESTART/SINGLE STEP, the PCR will be set to 0 . . . 0 1 and the execution will resume.

The following control capabilities of the SCB will be minimally required to implement this system.

A. Control.

(1) Start the execution of function micro-orders.

(2) Shift left/right and clear.

B. Status monitoring.

(1) Detect the completion of the function micro-orders.

(2) Detect the overflow and zero of the shift register.

The readers can convince themselves of the realizability of these modules by current technology if they compare the modules with chips currently available.

Remark 1. Another type of GO UP operation can also be used. It requires a counter (D-counter) which keeps the depth of the current node. The operation is executed as follows:

While the D-counter is larger than the displacement field, decrease the counter by 1 and visit the father of the current node.

The displacement field in this case specifies the depth of destination, while the displacement field in the GO UP explained above specifies the relative difference of the depth to the destination. These two operations may be distinguished by calling the former a Go UP operation with absolute displacement (for short, aboslute GO UP) and by calling the latter a GO UP operation with relative displacement (for short, relative GO UP). The discussion with them are parallel. Therefore, in what follows, we consider only (relative) GO UP operations.

Amounts of Address Information.

The most prominent feature of the proposed system is that the address information for sequence control is reduced greatly. The displacement field of length $\lceil \log m \rceil$ */ is sufficient for the tree memory of height m . Let us compare the amount of address information necessary to implement an ancestor tree program under this system with the amount required using the conventional linear memory. Let b and p

*/ The base of logarithm is 2. $\lceil x \rceil$ stands for the least integer not less than x .

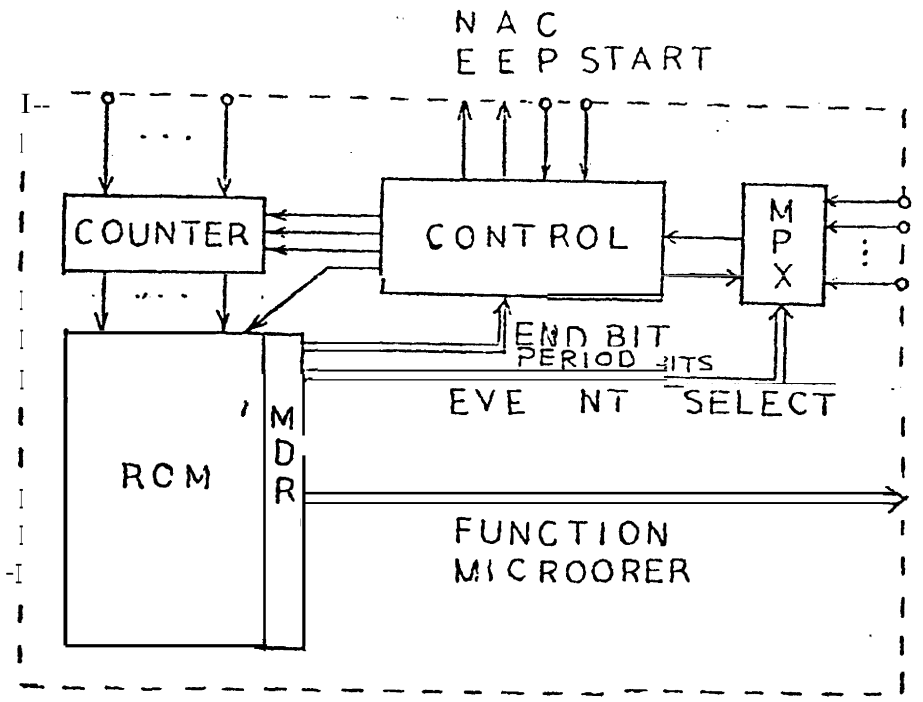
be the number of test nodes and back edges respectively. Then, in conventional method using conditional branches and/or jumps, $(b+p)m$ bits are used to specify the destination address, while only $p\lceil \log m\rceil$ bits are necessary in the present method.

The experience of trying to emulate the PDP-11 using a currently available high speed microprocessor has shown that the design of the instruction decoding section is tedious and the resulting emulator requires a great deal of time for decoding [7]. This inefficiency was attributed to the limited branch/jump capabilities of the microprocessor, although admittedly the decoder of a PDP-11 should be rather complex. This decoding problem may be easily solved in the proposed system. It is even possible to implement a multiple branch by shifting several bits into the PCR.

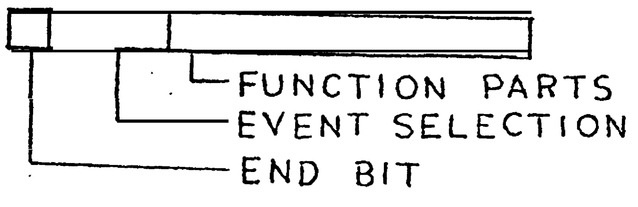
The Problem of Tree Height.

The problem in this method is the limit of the height of trees implementable in a limited memory. The rest of this paper is mostly devoted to this problem.

The first effective method to reduce tree height is to reduce the straight line parts of an ancestor tree program. One general method is to utilize parallel execution as much as possible. After that there will remain some straight line parts which should be done sequentially but have neither branches nor loops. These sections can be implemented by a programmable sequencer on a chip. Figure 7 shows one possible realization. This is essentially a microprogram controller without any branches. Several straight line programs can be stored. The end of each program is designated by a " 1 " in the end bit position (Figure 7 (b)). The entry points to the program can be set by setting the counter value with the START signal. Therefore, a straight line program can be entered at arbitrary points. This allows a program or its parts to be shared. Each word in ROM has the function part which is used to control the function units. The event selection bits select one signal from the function units which signals the completion of an operation. Counting clock pulses, the control waits for the selected signal to become asserted within a time period τ specified by the



a) MDR : memory data register



b)

Fig.. 7 a) a programmable sequencer
b) its word format

period-select bits. In most cases, no wait will be selected. If the control gets the response in τ , then it directs the next instruction fetch and initializes its inner timer unless the end bit is 1 . If there is no response in time τ , then it responds to the master with the abnormal end signal AE on-; If the end bit is 1 and the last response is received in time τ , then the control responds with the normal end signal NE on, and returns to its initial state.

This sequencer replaces "loops for wait" in a program by "waits" in the control. The time-out check also serves to detect hardware malfunctions. It is possible to use these chips in a hierarchical way, in analogy to a nested macro.

This sequencer can be used in several places. For example, micro-instruction fetch , read-in of a page on a missing-page fault (cf. Section 3), and register-saving in case of micro-level error are possible applications.

The proposed system with this sequencer seems to be sufficient to implement ancestor tree programs with many branches but of rather limited height, such as microprograms of minicomputer emulation. But for broader applications, other modifications are necessary and are discussed in the following sections.

3. Paged Tree Memory.

For a larger program, the memory space efficiency will become an important problem. This efficiency is defined as a ratio of the used memory area to actually implemented memory area. The mapping used in the last section is best for complete trees, where a tree of height m is called a complete tree if every leaf is of depth m . The mapping, however, leaves much unused memory area for incomplete trees. The worst case is a straight-line program. If the length of it is m , then the mapping will leave 2^m-m unused words. A missing subtree with the root of depth d in a complete tree of height m amounts to 2^{m-d} words loss.

Our problem is to find a good mapping for incomplete trees which preserves the simple sequencing capability of the proposed system, while it reduces the unused memory area. The following method is proposed.

Assume that we have memory chips with 2^p words and we want to store an ancestor tree program of height 2p-1 . Each memory chip can store an ancestor tree of height p-1 and this is considered a **page.** The following mapping will be used.

	program address		page address	line address	
	p	p	$p+1$	p	
	$0\ldots00$	$0\ldots00$	$00\ldots0$	$0\ldots01$	
	$0\ldots00$	$0\ldots1a_1$	$00\ldots0$	$0\ldots1a_1$	
	$\vdots$	$\vdots$		$\vdots$	
*	$0\ldots00$	$1a_1\ldots a_{p-1}$	$00\ldots0$	$1a_1\ldots a_{p-1}$	*
**	$0\ldots01$	$a_1a_2\ldots a_p$	$1a_1\ldots a_p$	$00\ldots01$	**
	$0\ldots1a_1$	$a_2a_3\ldots a_{p+1}$	$1a_1\ldots a_p$	$00\ldots1a_{p+1}$	
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
	$1a_1\ldots a_{p-1}$	$a_p\ldots a_{2p-1}$	$1a_1\ldots a_p$	$1a_{p+1}\ldots a_{2p-1}$	

This is equivalent to considering 2^p+1 trees of height p-1 instead of a single tree of height 2p-1 . Each small tree is implemented on a memory chip and is identified by its page address. If a page is never used, then the corresponding chip need not be equipped.

By the new mapping, a missing subtree with the root of depth p-i $(0 \leq i < p)$ amounts to 2^i-1 words loss, while it amounts to 2^p-1 words loss by the original mapping.

An Implementation.

The implementation of the PCR will change slightly as suggested in Figure 8. In the mapping shown above, the transition between the state * and the state ** should be treated differently from other transitions. There are several methods of accomplishing this, but it may be simplest to use a counter in the SCB which maintains the current depth. The counter will be incremented by 1 for each GO DOWN and decremented by 1 for each step of GO UP , and changes from p-1 to p (and vice versa) are detected.

This is but one possible implementation. More sophisticated schemes for the total system are conceivable.

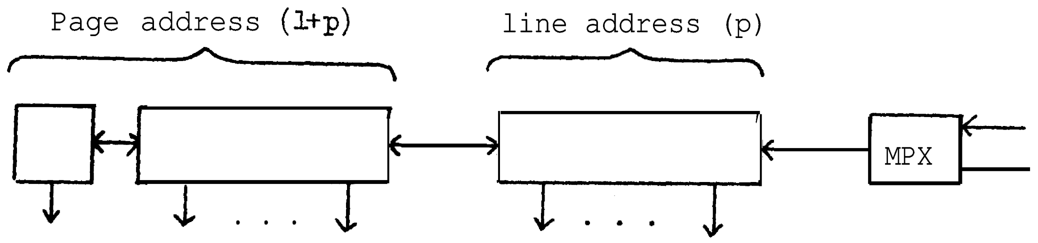


Figure 8

Paged Memory System.

The mapping defined above maps all the nodes in a subtree with the root of depth p-1 into a consecutive memory area with the same prefix (page address). This paged tree memory is similar in many respects to the usual paged memory. It is possible to place some of the pages in the secondary storage, to read pages into a rewritable memory (RAM) and to execute the instructions from that memory instead of implementing all the pages as ROMs. It seems convenient to specify important and frequently used pages as ROMs, and store others in the secondary storage. This approach provides us flexibility to change dynamically microporograms not in ROMs. To implement this paging system, a new simple method may be more effective than the sophisticated paging systems used in current machines. Most of the conventional paging systems use a page table in an associative memory and an address set-up mechanism. The method proposed here will use a different approach.

The simplest idea is as follows: for simplicity, each RAM chip is assumed to hold a page (later, this assumption is removed). Each ROM and RAM chip contains chip select logic. Each ROM has a fixed chip number (page number). But each RAM must change its page number according to its current contents. Each page has a one word with address (0 . . . 0) left unused. This word can be used to store the page address (or part of this address, as explained later in conjunction with Figure 9). The page address on the page address bus is checked against the content of the word. If they coincide, the chip responds. If there are no responses from any memory chips -- this is detectable by time-out logic -- then the missing page fault procedure will be started, and a new page will be brought into a RAM chip. This can be done by a programmable sequencer described in an earlier section.

In this method, the page table is distributed among the chips and the associative search is replaced by a coincidence check against the content of the address 0 in each RAM. This idea can be used in systems other than tree memory system by adding an extra word to hold the page address.

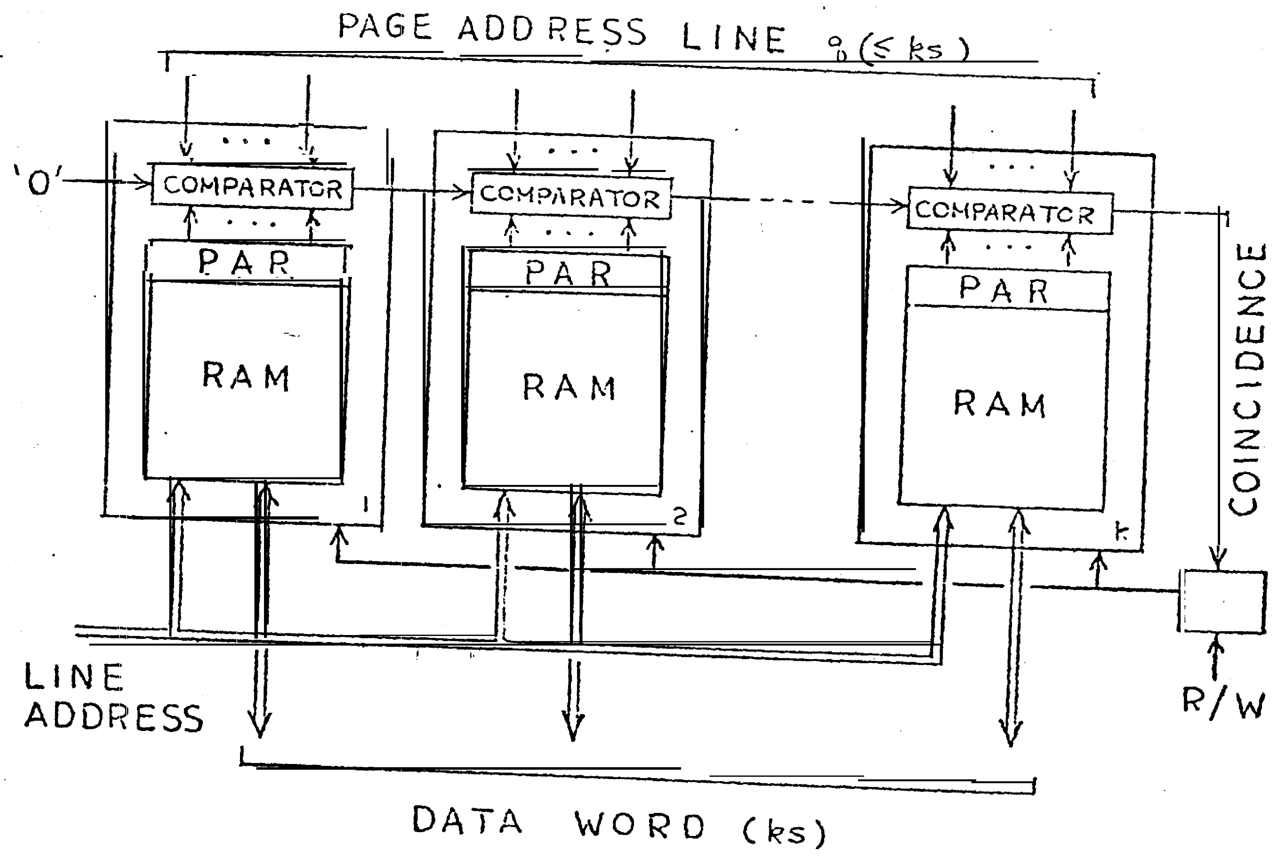


Fig. 9 A possible chip design and arrangement

To realize this system, the current design of memory chips must be slightly changed. The problem is the appropriate selection of parameters such as page address bits, line address bits, and word length under limited pin count.-. To fit a broad range of applications, the chip is desired to be adaptable to various parameter selection. Figure 9 shows a solution to this problem.

Let s be the word length of RAM words. Assume that the length of micro-instruction is ks and that page address is of q bits width with $q < ks$. Then k chips are used as a page and the coincidence signal will be obtained through k serially connected comparators which compare the s bit page address with the page address register (PAP). The PAR may be the word of address 0 or an extra register. The read/write control signal is applied to each chip ANDed with the coincidence signal. In a sense, this is a bit-sliced page memory.

We note, finally, that a secondary storage device which transfers a page selected by a page address to a suitably selected RAM chip should be available; e.g. an electronic disc with key-retrieval and block transfer capability.

What replacement algorithm is suitable in the paged tree memory system? One principle may be to replace the farthest page in terms of kinship to the current page. The dynamic behavior of a structured program reflects its structure (or should). This will make it easier to devise a replacement algorithm.

4. Ancestor Tree with Shared Sub-trees.

Now let us consider another possible objection. As mentioned earlier, every flowchart can be transformed into an ancestor tree up to isomorphism. But this transformation requires node-splitting, that is, some parts of the program must be copied. It does not change the execution time of the program but it may require greatly increased space. As an example, let us consider an acyclic graph isomorphic to Pascal's triangle truncated to height n . Let us call this graph T_n (Figure 10). The graph T_n has $\binom{n+2}{2}$ nodes and the transformed binary tree has 2^n-1 nodes. This is an example of exponential space explosion in the structured counterpart of an unstructured or go-to program. This example is an extreme one. But the feature that a lower node is shared by many ancestor nodes is not rare. Here is another example [10]. A *very* short module in PDP-11 DOS Monitor, containing 67 instructions, expands to an equivalent ancestor tree program of 212 instructions. One common design goal is to minimize program size -- especially in the monitors of minicomputers. Therefore GO TO's (jumps/branches in this case) are used freely in real programming despite Dijkstra's warning [5].

As a compromise, yet another instruction called SHARE is introduced. This is essentially GO TO but it is used with clear awareness that a sub-tree is shared with other control paths. Thus, the usage is limited. In this sense, we may be said to be yet in the realm of structured programming. For example, T_z (Figure 10) can be realized as a program of Figure 11. The label/' X' means an instruction " SHARE X ". This program has 3 more nodes than the original program. In general, a triangle program T_n of height n is realized as a binary tree program with $\binom{n+1}{2}$ more nodes than T_n with $\binom{n+2}{2}$ nodes. Thus, the exponential explode is excluded with the cost of speed (that is, the extra instruction SHARE must be executed). This instruction can be used with ancestor trees also. In this case, we must be careful due to possible back edges returning to ancestor nodes over the root of the shared subtree.

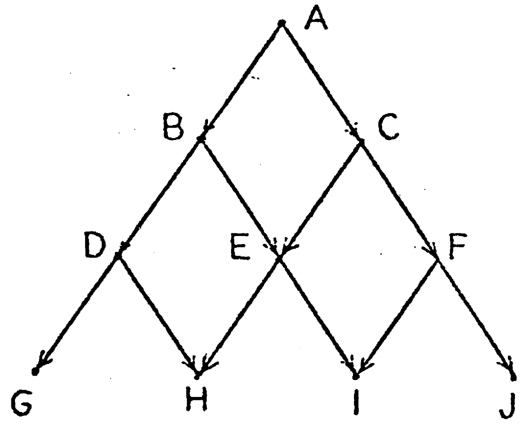


Fig. 10 T_3

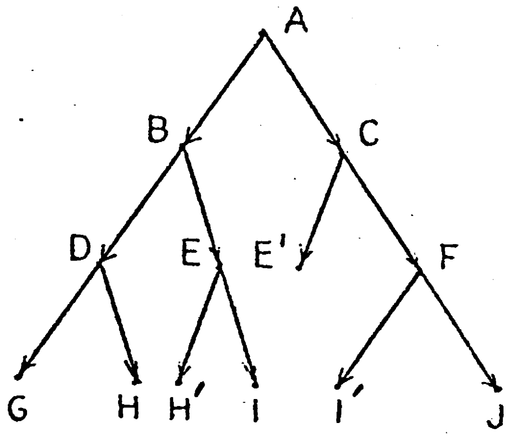


Fig. 11 T_3 realized by using SHARE

An Implementation.

The instruction SHARE requires a full address as an operand (page address and line address, in case of paged. tree memory). Each instruction es executed as follows (see Figure 12):

SHARE (address)

(1) Push down the FCR into the address stack and transfer the address part to the FCR.

(2) Push down the counter II into the counter stack and clear the counter II.

(3) Fetch the instruction.

GO DOWN (selector)

Same as the explanation in Section 2 except that counter II will be incremented by 1 at the same time the FCR is shifted left one place.

GO UP (displacement)

```
(1) Start the execution of function micro-orders.
(2) If displacement field is zero, then go to 3 ; otherwise
    set the value in the counter I.
    While counter I is not zero, do the following:
     (2.1) Decrement counter I by 1.
     (2.2) If counter II is not zero, then decrement it by 1
           and shift the FCR right one place; otherwise pop-up
           both the stacks to counter II and FCR and go to
           (2.2) again.
    If counter I is zero, fetch the next instruction.
(3) (Return to the root.) Clear counter II, counter stack,
    address stack, and FCR, then shift in a "1" from the
    right and fetch the instruction.
```

Remark. The use of instruction SHARE is not limited to sharing a subtree. When storing tall tree programs in the memory, SHAREs are used to join the sectioned trunks, This usage is nothing other than GO TO (jump).

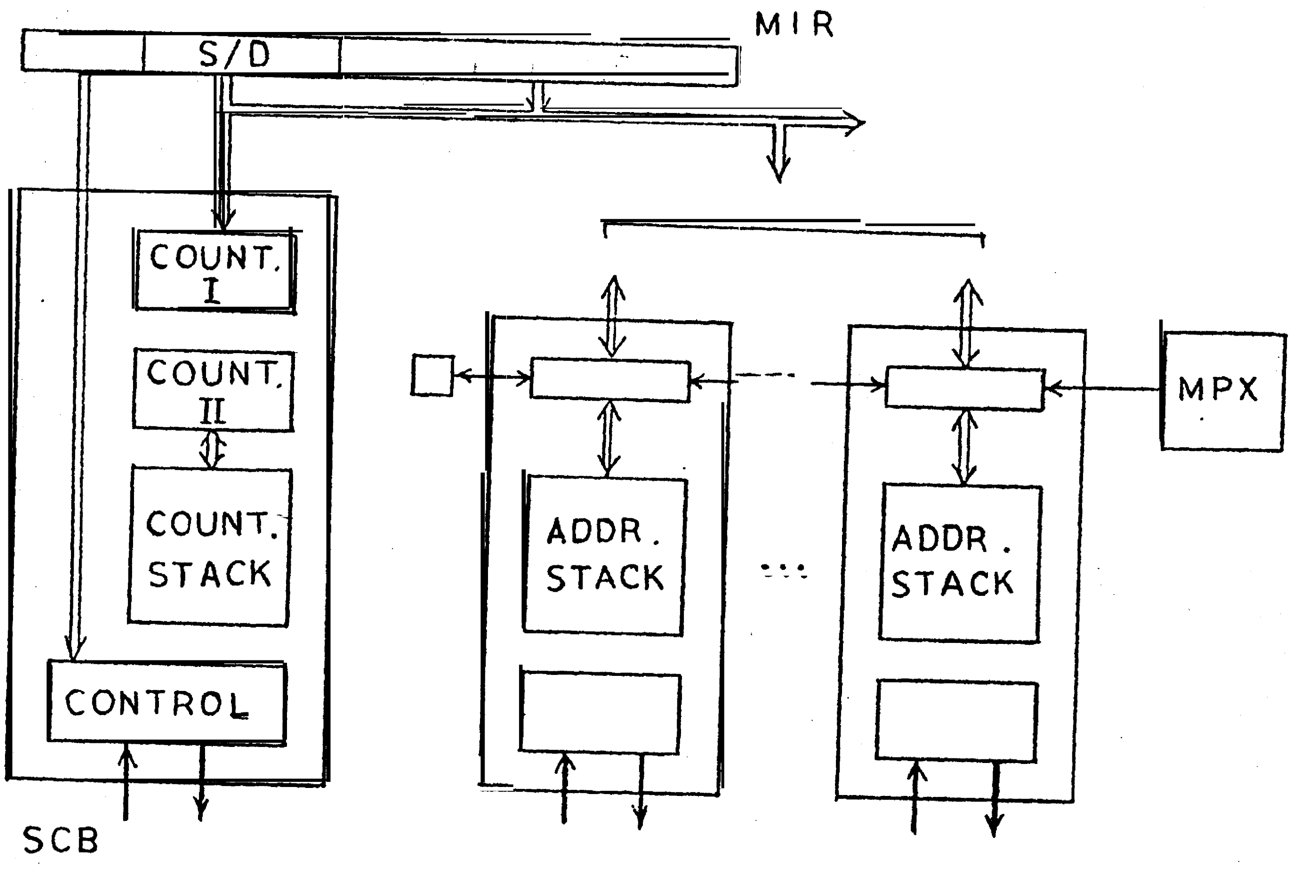


Fig. 12 An implementation of SHARE (Partial structure)

Search for Sharable Subtrees.

The last problem in this paper is how we can find such sharable subtrees in an ancestor tree. If this is too hard, then we can not utilize the capability of SHARE sufficiently. Fortunately, we have a good algorithm to find all the sharable subtrees in an ancestor tree in time nearly proportional to the number of nodes in the tree.

There will be no loss of necessary information if a given ancestor tree T is replaced by a labeled tree with all back edges removed because the displacement field of the instruction retained as a label will enable us to recover back edges. Then the problem can be restated as finding all the identical subtrees up to their labels. The subtrees in the ancestor tree which correspond to the identical subtrees up to label in the corresponding labled tree can be shared. Here, an implicit assumption should be explained. That is, every instruction is treated as one label even if it contains the displacement field which is of order log log n , where n is the number of nodes in the *tree.* In the usual situation, this factor may be considered not to contribute to the efficiency measure. But if the asymptotical efficiency is of concern, the present algorithm can be safely said to be an O(n log log n) algorithm where O(f(n)) stands for " order of f(n) ".

Lemma. The list of identical subtrees up to labels in a labeled tree can be obtained in linear time.

See the Appendix *for* a sketch of the proof.

Thus the existence of an almost linear algorithm to find sharable sub-trees is shown, though the actual algorithm should be simplified further.

Conclusion.

The concept of tree memory is re-examined and applied to micro-program memory. It leads to a program with the address information greatly reduced, and a simple modularized control unit. With the programmable sequencer, ancestor tree programs of low height but with many branches seem to be realizable effectively; e.g. an emulator of minicomputers and possibly a core part of small monitor programs could be implemented. If the program is far from the complete tree, then the paged memory system will save the unused memory chips. A simple paged memory system is also proposed in which ROMs and RAMs can co-exist. This technique will be useful for broader applications such as micro-code replacement of some software routines and interpreters of high level languages.

Lastly we have reached a new class of structured programming by examining the efficiency issue. It was easy to implement a CO TO in the proposed system, but we stop at SHARE. This compromise to introduce SHARE does not harm the merits of structured programming, because an ancestor tree program with SHAREs can be directly expandable to an equivalent ancestor tree program.

Each user wants a structured programming system most similar to his problem structure [9]. The ancestor tree programs (with shared sub-trees) seem to be very near to many problems which people want to implement by microprogram, although it might be too early to say so when there is little data on this issue. What can be said at present is that users have obtained more freedom of choice. The presentation is intended to be suggestive and the readers are invited to develop their own ideas on this proposal.

Acknowledgment.

The author would like to thank Prof. I. Lee of' the University of California at Berkeley for his talk which inspired the present work. Thanks also go to Prof. I. Shirakawa of Osake University for his help. This work was done when the author stayed at the Department of Information and Computer Sciences at the University of Hawaii and at the Computer Science Department at Stanford University. He would like to thank all the people at the departments, Professor W. W. Peterson and Professor D. E. Knuth, especially.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms. Reading: Addison-Wesley, 1974.

[2] K. J. Berkling, "A Computing Machine Based on Tree Structures," IEEE Transactions on Computers, C-20 (4), 404-418, April 1971.

[3] E. Engeler, "Structure and meanings of elementary programs," Symposium on Semantics of Algorithmic Languages, 1971.

[4] D. E. Knuth, Fundamental Algorithms: The Art of Computer Programming, vol. 1, Reading: Addison-Wesley, 1968.

[5] D. E. Knuth, "Structured Prog amming with GO TO Statements," Comput. Surveys, Dec. 1974, 261-301.

[6] I. Lee, "LSI Microprocessors and Microprograms for User-Oriented Machines," Proceedings of ACM Micro-7, s1-s13, Sept. 1974.

[7] I. Lee, private communication, 1975.

[8] R. J. Lipton, S. C. Eisenstat, and R. A. DeMillo, "The Complexity of Control Structures and Data Structures," Proceedings of Seventh Annual ACM Symp. on Theory of Computing, May 1975, 186-193.

[9] C. McFarland, "Structured Microprogramming," Proceedings of ACM Micro-7, s28-s32, Sept. 1974.

[10] J. Okui, N. Tokura, and T. Kasami, "Analysis of a Disk Operating System," Kyoto University, Institute of Math. Analysis, 189, Oct. 1973, 101-116.

[11] N. Tokura, "A Multi-dimensional Addressing System," Trans. IECE Japan, 53-C, Nov. 1970, 855-862, in Japanese. (English translation: Systems, Computers, Controls 1, Nov.-Dec. 1970, 26-33.

[12] P. Weiner, "Linear Pattern Matching Algorithms " Proceedings of 14th Annual Symposium on Switching and Automat: Theory, Oct. 1973, 1-11.

Appendix

A Sketch of Proof of Lemma.

The form of the algorithm claimed in the Lemma is briefly described.

For a labeled tree T , let P(T) be a string of tree labels spelled out when T is traversed in preorder [4]. For example, the preorder traversal trace P(T) of a tree in Figure 11 is

A B D G H E H I C E ' F I J .

For a given tree T , P(T) can be obtained in linear time simply by traversing the tree. The length of P(T) equals the number of nodes in T .

P1. If subtrees T_1 and T_2 are isomorphic in a tree T , then there are two identical substrings $P(T_1)$ and $P(T_2)$ in P(T) . Conversely, if there are identical substrings w_1 and w_2 in P(T) for a tree T and if there is a tree T' such that $w_1 = w_2 = P(T')$, then there are isomorphic subtrees T_1 and T_2 with $w_1 = P(T_1)$ and $w_2 = P(T_2)$.

P2. It is decidable at most in time proportionalto the length of P(T) whether a substring w of P(T) is a preorder traversal trace of a subtree of T or not.

Proof. Simply by traversing.

The problem to find all the repeated substrings in a string is shown by Weiner [12,1] to be solvable in linear time. Therefore, with P1 and P2, we can conclude that the problem to list the root nodes of isomorphic subtrees in a tree is solvable in linear time. □