A DESIGN ANALYSIS OF DYNAMIC LINKING AND LOADING OF MICROPROGRAMS

by

JOHN DOUGLAS WRIGHT

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1973

Signature of Author. . . . .
Department of Electrical Engineering, May 24, 1973

Certified by. . . . . . .
Thesis Supervisor

Accepted by. . . . .
Chairman, Departmental Committee on Theses

ABSTRACT

The ability to dynamically redefine the computational base of a processor is seen as a possible approach to improving the processor's performance. Provision of a facility to allow dynamic linking and loading of microprograms is considered as a practical means of accomplishing this modification. The feasibility and the desirability of this approach is evaluated in terms of the implementation overhead and the performance improvement possible with such a facility. It is concluded that this method provides a viable means of increasing processor performance and flexibility.

Table of Contents

Chapter	Page
1. Introduction and Purpose	1
2. The Conventional Microprogrammed Processor	5
3. Implementation of the Dynamic Linking and Loading Environment	19
4. Evaluation of the Dynamic Linking and Loading Environment	31
5. Justification of the Implementation of the Dynamic Linking and Loading Environment	68
6. Implications of Dynamic Linking and Loading of Microprograms	84
7. Conclusion	86
Bibliography	88
Appendix A. Microprogramming the Hypothetical Machine	89

List of Figures and Illustrations

		Page
Figure 1.	Overview of the Data Structure	8
Figure 2.	MAR-DBR Interrelationship	11
Figure 3.	Control Structure Organization	14
Figure 4.	Linking Mechanisms	22
Figure 5.	Overview of the Hypothetical Machine	29
Figure 6.	Missing Segment Fault Algorithm	37
Figure 7.	Microprogram Behavior	39
Figure 8.	Least-Frequently-Used Algorithm Including Use of Association Lists	45
Figure 9.	Least-Frequently-Used Algorithm Including Use of Removal Counts	47
Figure 10.	Least-Frequently-Used Algorithm Including Use of Both LFU-AL and LFU-MFR ..	49
Figure 11.	A Modification of the Combined Algorithms .	51
Figure 12.	A Variation of LFU-AL	56
Figure 13.	Garbage Collection Algorithm	61

Chapter 1. Introduction and Purpose

The control structure of microprogrammed digital computers has evolved in the direction of increased flexibility. Early microprogram control stores consisted of read-only memories; whose physical and development costs all but prohibited custom modification. Later, developments in high-speed memory technology lowered the cost of the control store, thereby allowing both longer microprograms and limited provision of special features. Early efforts in this area included additions to the IBM 360 instruction set[1] and numerous ventures aimed at developing machines for direct execution of higher level languages, notatbly APL[2]. These efforts have been extended to include operating systems[3] and combined machine level programming with microprogramming to provide significant performance gains[4]. Recently, the addition of read-write memories and field programmable read-only memories, to the conventional read-only memory, allowed the user limited ability to tailor the machine for specific applications. Dynamic loading capability has been announced by some manufacturers[5]. Extensive provision for user microprogramming has, however, been confined to small or medium scale computers[6]. Present trends indicate that decreasing costs of high-speed memories will result in increased provision, by manufacturers, of software and system support features at the microprogram level.

A conceptual extension of this trend would encompass a control structure which supported dynamic modification of the control store in response to the immediate needs of an active process. Such a facility would require a dynamic loading mechanism; to allow swapping of special microprograms as they are needed. A dynamic relocation or linking mechanism would also be required to allow arbitrary combinations of these programs to be simultaneously control store resident and allow linking among programs in the conventional subroutine or function manners. Combination of the linking and loading mechanisms could be utilized to provide a virtual control store or simply facilitae user control of the microprogram environment.

This concept differs significantly from the traditional (and continuing) emphasis on providing an essentially static set of microprogram features. A dynamic facility, however, provides not only significant reductions in the execution times of a large class of useful programs, but additionally allows microprograms of greater length than the physical control store. This avoids the otherwise crucial issue of which special features should be provided in a limited set. These advantages are partially offset by the overhead invovled in implementation of the dynamic linking and loading capabilities, especially if virtualization of the control store is desired. The essential issue therefore becomes an evaluation of the performance increases offered

by dynamic linking and loading, in comparison with the overhead involved in providing this facility.

Initially, a hypothetical machine is considered which is suitable for latter conversion, to allow dynamic linking and loading, but reflects the design of conventional microprogrammed processors. The design was constrained to include only those features which had previously been incorporated in processor design, or those features which were easily derivable from previous designs. This machine is described in Chapter 2.

Additional hardware is then added to allow the dynamic linking and loading capabilities. Modification of the existing control store address mechanism from absolute addressing to relocatable addressing is also required. A description of these features is provided in Chapter 3.

Evaluation of the completed design indicates the probable overhead involved in utilization of the dynamic linking and loading mechanisms. Direct user control of the linking and loading process is discussed as a prelude to a description of the difficulties involved in providing a virtual control store. Removal and memory management algorithms are considered in relation to the microprogram environment. Additional consideration is given to the effect of external factors, especially interupts, on the linking and loading process. In concluding Chapter 4, a summary of the overhead involved in linking and loading is provided.

Justification of the overhead involved in providing the mechanism results from an analysis of the potential performance gains achievable through use of the mechanism. A reduction process is described which utilizes the relationship between microprogramming and machine language programming to provide criteria for microprogramming a section of machine language code.

Continuing from Chapter 5, Chapter 6 provides a discussion of other considerations posed by dynamic linking and loading. Especially important are the issues of compatibility and manufacturer support of machines with varying instruction sets.

In concluding, it is felt that dynamic linking and loading mechanisms, at the microprogram level, provide a viable means of upgrading processor performance.

Chapter 2. The Conventional Microprogrammed Processor

Introduction

Microprogrammed digital computers may easily be described in terms of their data and control structures. The data structure consists of the available paths for the transfer of data between the storage elements (registers, buffers, memories, etc.) of the processor, and the computational units included in each path. These computational units provide the basic data manipulative functions of the machine, such as add, subtract, logical AND, etc. Various special purpose subprocessors may also be provided. Most typical is the availability of a floating-point processor for many machines. The control structure allows definition of the transfers and functions provided by the data structure during each microprocessor cycle (a microprocessor cycle is defined as the period between the execution of successive microinstructions). For microprogrammed computers, this control is apparent in the interpretation of various fields of the microinstructions.

Historically, a wide variety of data structures have been utilized by manufacturers. The IBM 360 and IBM 370 series computers exemplify the use of several special-purpose-computational units and multiple bus structure to allow maximal parrallelism in the processing of each machine language instruction. While supporting an almost identical instruction set, the RCA Spectra 70 series computers employ a much different architecture. The

Spectra 70 series computers utilize a three bus structure and single general purpose arithmetic and logical unit to provide the computational facility; and provides an additional data transfer bus of higher bandwidth.[1] Variations of this basic three-bus structure are in widespread use. This is especially true of smaller computers where the simplicity of the design (and lower cost) is more important than processor speed. More radical in design is the Nanodata QM-1 machine. This computer utilizes multiple busing and a single computational unit, but allows prolonged specification of bus interconnections (indefinitely long) by the microprogram, and thereby allows a limited degree of "rewireability".[2]

The variations in the data structures among computers are reflected in their respective control structures. Each microinstruction specifies some action(s) to be performed by the data structure prior to the execution of the next microinstruction. Additionally, each microinstruction must provide a means for determining the next microinstruction to be executed (hereafter referred to as the successor function). Again, wide variation may be found in both the successor functions available and the control signals which may be specified during each microprocessor cycle.[1,3]

To provide a uniform basis for discussion of the dynamic linking and loading concept, a selection of the features available in the data structures and control

structures of conventional computers has been incorporated in the design of a hypothetical conventional proccessor. A variation of the three-bus architecture was chosen as typical of many designs, and appropriate control fields were provided in each micrinstruction. The successor functions provided were chosen to reflect a variety of the existing strategys and additionally, to illustrate the concerns involved in providing a dynamic linking and loading facility for an arbitrary conventional machine.

Description of the Hypothetical Machine's Data Structure

The principle data paths of the hypothetical machine are summarized in Figure 1. The A, B and F buses provide transfer of data within the processor and supply operands to the arithmetic and logical unit (ALU). A main memory bus allows transfer of data to the machine's cache memory and to the operand registers of the floating-point processor (not shown). Each of the internal buses (A, B, and F) is chosen as eight bits wide to allow flexibility in byte manipulations, and to reflect similar choices in the design of other processors.[1,3] The main memory bus is chosen as 64 bits wide, thereby reflecting typical main memory bandwidths, floating-point operand length and the width of the cache memory.

The arithmetic and logic unit (ALU) provides a variety of simple functions on eight bits. These include incrementing or decrementing the A-operand, adding the B-operand and

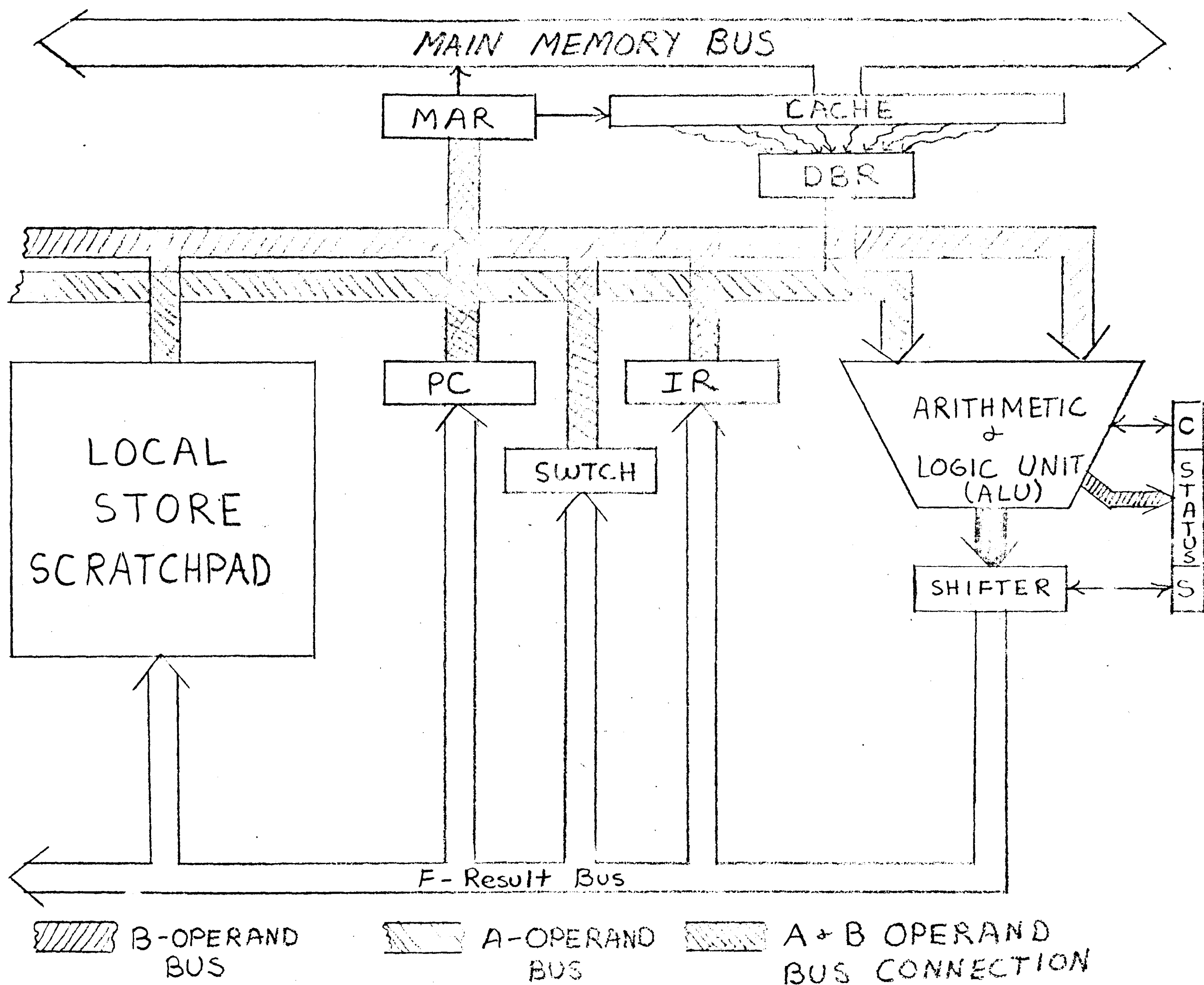


Figure 1. Overview of the Data Structure

A-operand, subtracting the B-operand from the A-operand, and sixteen logical functions of two variables (1s, 0s, A, $\overline{A}$, B, $\overline{B}$, AB, $\overline{A}B$, $A\overline{B}$, $\overline{AB}$, $A+B$, $\overline{A}+B$, $A+\overline{B}$, $\overline{A+B}$, $A \oplus B$, $\overline{A \oplus B}$). The arithmetic operations allow specification of a carry-in bit and result in a carry-out bit thereby allowing multiple precision operations. Both logical and arithmetic operations generate status information concerning the nature of the operands and result of the operation. Conditional information, such as zero result, negative result, overflow and underflow are provided. The result of the ALU operation is passed through the shifter. This unit shifts the result either left or right and substitutes a specified S-bit for the bit positions vacated by the shift. In the event of single bit shifts (either left or right) the S-bit may be used to allow extended precision shifts in a manner analogous to that used for the ALU. The result of the combined ALU and shifter operations is written from the [illegible] into the proper register at the end of each microprocessor cycle.

The various registers are selected by special control fields in each microinstruction to be gated to each of the three buses. Within the context of the data structure, each register may respond to more than one address as a result of its length or to commands encoded as different addresses. For example, the program counter (PC) responds to two addresses; one of which simply references the PC as data, the other additionally indicates that the PC

should be incremented as a result of the reference. The local store scratchpad (LSS) also responds to several addresses, but the effect here is to utilize the encoding to select an address within the scratchpad, rather than indicate an operation to perform. In this manner, the LSS serves to provide the numerous general registers normally asscociated with a machine. Various other special registers are provided. These include the status register, the instruction register (IR),the memory address register, the memory data buffer register and the switch register. Additional special purpose registers may be connected to any of the buses, but only 256 discrete addresses are allowed by the control structure.

Special consideration should be given to the main memory access port provided by the memory address register (MAR), the data buffer register (DBR) and the cache. A detailed summary of the interelation between these registers is provided in Figure 2. The DBR recognizes several addresses with the following possible results: first, that the reference is simply for data, second, that the contents of the MAR should be incremented and that either a read or write operation should occur following the current micro-processor cycle, or third that the MAR should be decremented and read or write operation should occur. Similar encoding of the MAR address lines allow specification of read-write or DBR modification operations. Note that as a result of these operations the contents of a register would remain

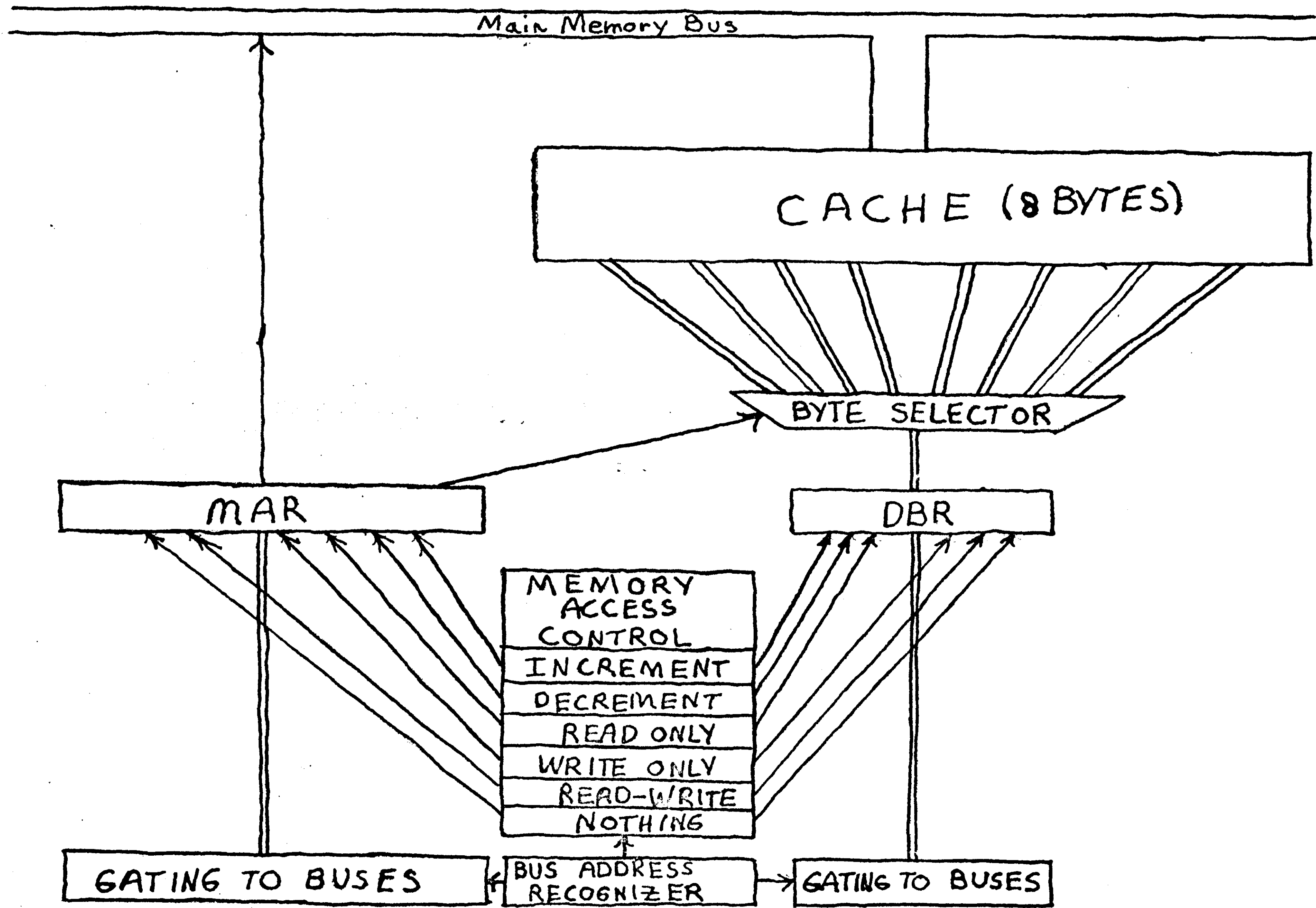


Figure 2.
MAR-DBR Interrelationship

invalid for some period after the operation is initiated. If the register is addressed by a subsequent microinstruction before it contains valid information, then the register responds by indicating, to the control structure, that the microprocessor cycle should be delayed. This allows a form of asynchronous operation by the computer, but the overall effort is towards providing fixed-cycle-synchronous operation.

<u>Description of the Hypothetical Machine's Control Structure</u>

Each microprocessor cycle consists of the execution of a microinstruction and determination of the next microinstruction to be executed. In practice, these operations are overlapped in a manner which allows continuous execution. This indicates that evaluation of the successor function during the execution of an instruction must use the conditional status information resultant from the previous instruction, and cannot use (wait for) status information generated by the current instruction. This effect may be summarized as:

```
              -- A,B gating + ALU + Shift + F writing* --
             /                                        /
*fetch ------ successor evaluation + next fetch ------
```

were the asterisks indicate the maximum microprocessor cycle. Defining T_{ex} as the time required for the upper branch, after T_{dec}, the time required to initially decode control structure signals; and T_{next} as successor evaluation time; and T_{csac} as the time then required to fetch the next microinstruction; yields the result that the microprocessor cycle time is:

$$T_{mpcy} = \max(T_{dec} + T_{ex} , T_{next} + T_{csac}).$$

Assuming that a given microprocessor cycle time is desired* and that the decode and execution times of the data structure have been specified, then it becomes possible to indicate control structure requirements of the processor. Basically, control signals must be generated in the form of A, B and F bus address fields, and for ALU and Shifter control. The control structure must also provide storage for the microprograms with sufficiently small access time to allow:

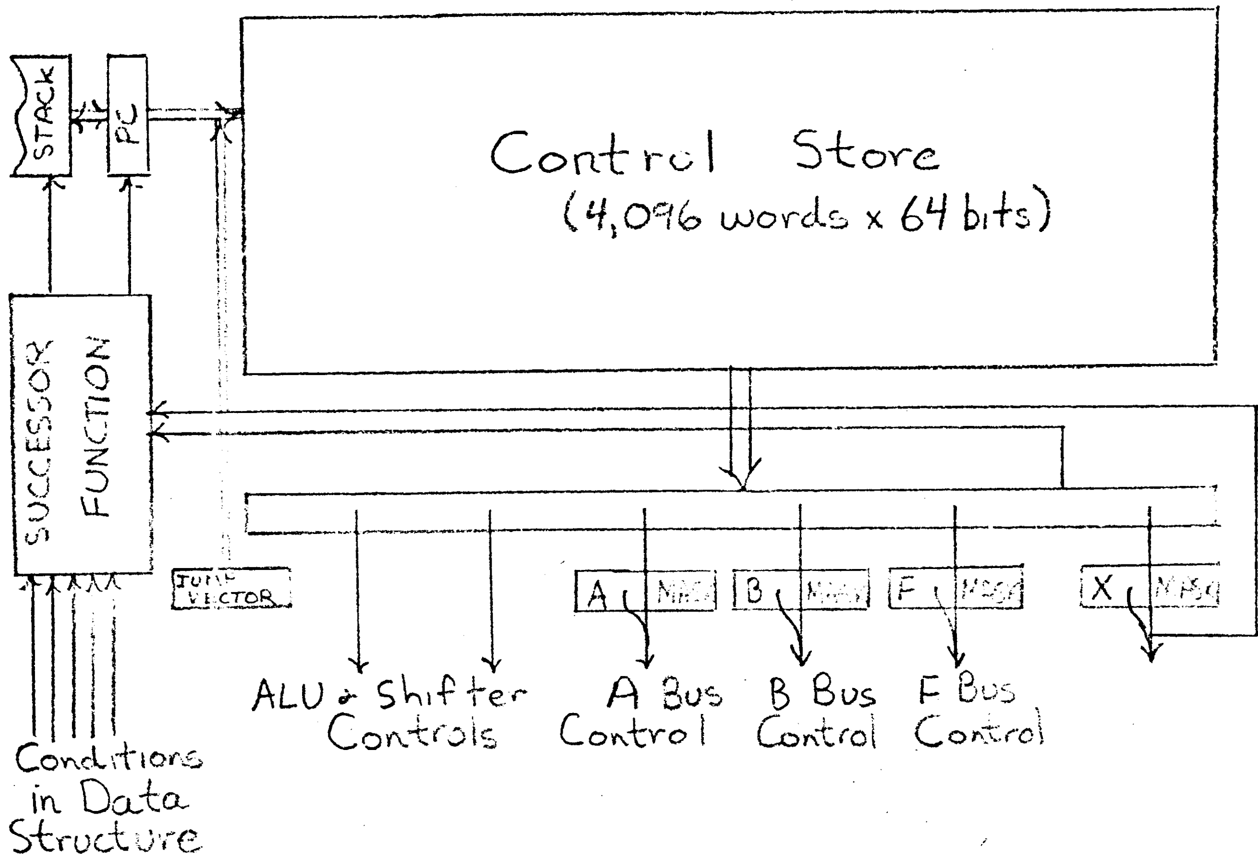
$$T_{mpcy} = T_{dec} + T_{ex} = T_{next} + T_{csac} .$$

This represents the optimal choice of control structure parameters because a larger value of $T_{next}+T_{csac}$ would result in wastage of the available data structure transfer speeds; wereas a smaller value would not improve T_{mpcy} and the cost involved in providing the smaller value would not be justified. We proceed on the assumption that proper choices of logic elements will allow the optimum condition, and describe the control structure and microinstruction format.

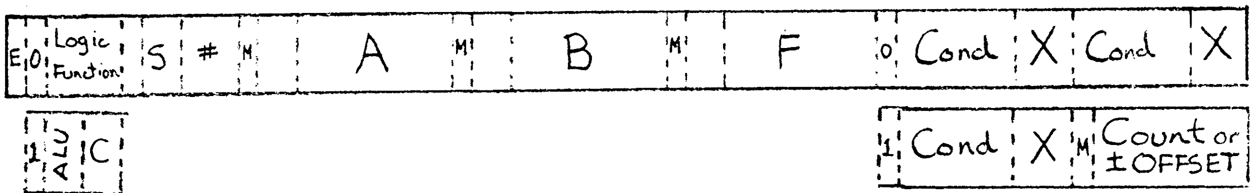
An overview of the control structure organization is shown in Figure 3. The control store is chosen as 4,096 words of 64 bits each; reflecting the microinstruction format and the approximate size of control stores in medium to large scale computers. The successor function provides

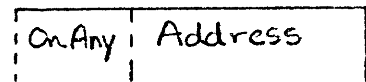
* T_{mpcy} for most processors is on the order of 100-300ns. This reflects tradeoffs between desired processor price, processor speed and available technology.

Figure 3.
Control Structure Organization



Microinstruction Format (64 bits)





Possible interpretations for successor X:

Step - increment PC by 1
Skip - add 2 to PC
Repeat - do not modify PC
Jump - replace PC with Jump Vector
Call - push PC+1 on stack, then replace PC with Jump Vector
Save&Step - push PC on stack, increment PC by 1
Return - replace PC with top of stack

the address of the next microinstruction based on the method, as indicated in the successor field of each microinstruction, and conditional testing required. The control fields include the A, B and F address fields, the ALU control field, the Shifter control field and associated bit-steering fields (bit-steering fields indicate how another field or fields are to be interpreted if more than one interpretation exists).

Each of the bus-address fields is identical and consists of ten bits. Eight constitute the address of a register on the bus, or an eight-bit constant to be substituted as an operand. The remaining two bits indicate the interpretation of the field as a constant or, in the case of extended precision operations, whether the address is to be incremented, decremented or remain the same in determining consequetive operands. This feature is a generalization of that presently used on many word oriented machines which process bytes at the microinstruction level (IBM 360 and RCA Spectra 70). An additional mask (M) bit is associated with each field and indicates whether the eight bit instruction field or the logical OR of the mask register associated with that field and the instruction field are to be used. The symbolic reprsentations for these fields have the following format:

X(Y,Z) or X(Z)

where X is the contents of the eight bit instruction field, Y is M if the field is masked, and Z is the field variation in the event that the instruction is immediately repeated.

The Z field is either +, -, /, or C depending on whether the modification is to be increment, decrement, remain the same or interpret as a constant, repectively. The total of the address fields constitute 33 bits of the micro-instruction.

The remainder of the control signals required to manipulate the data structure are provided by means of the ALU and Shifter control fields of each instruction. The ALU fields include the E-bit which designates an extended precision operation, the function to be performed and the initial value of the carry (C) bit in the event of arithmetic operations. The shifter field includes specification of right or left shift, the number of bit positions to shift, and the shift-in bit (S) to replace the bit positions freed by the shift. Either the C or S bits may be specified as 0, 1, X, or $\overline{X}$. The ALU functions have been previously described. The shifter control allows for shifting of from zero to seven bit positions (zero corresponds to no shift) in the direction specified. The symbolic representation for these fields is:

X(C), Y(S) or X(C) or LX, Y(S) or LX

where C and S designate the desired carry-in and shift-in bits, X indicates the ALU function (prefaced by L if the operation is logical), and Y indicates the direction and number of positions to shift (for example L3, R5 etc). Prefacing X with E indicates an extended precision operation. The ALU and Shifter control fields require 12 bits total.

The remaining nineteen bits of each microinstruction constitute the successor function. Four forms of this function are possible. The first results from the use of the E-bit in specifying an ALU operation. In this case; the first bit of the successor function is ignored; the second conditional field is interpreted as a mask-bit and eight bit count specifying the number of repitions, and the first successor conditional field is interpreted as "on condition do x", where x is specified as one of the seven successor possibilities. After its first execution, a statement will be repeated until either the condition is met (and x is therefore chosen) or the instruction has been repeated the designated number of times, in which case the default successor function is Step. Other basic forms of the successor function are specified by the first bit in the field. Either the interpretation: "on condition A do x, else on condition B do y, else Step", or the interpretation: "on condition do x, else M±OFFSET", may be specified. The latter alternative indicates that the X register contains an offset which should be added to the current value of the microprogram counter, and indicates whether or not the field is masked. An additional possibility results from specification in the first cond-ition field of an"on any condition", in which case the remaining bits of the instruction are interpreted as the absolute address of the next instruction. The possible values of "x" and field sizes are included on the Figure 3.

These choices of possible successor functions provide wider diversity than would normally be expected. This results from the attempt to represent many of the features provided by present machines. The extended precision function is present in many machines in varying forms. The "on condition x" form is a modification of the Burroughs Interpreter[3]. The ±OFFSET form is a generalization of the RCA Spectra 70 and IBM 360 equivalents[1]. The absolute form is present in nearly all machines.

Conclusion to Chapter 2

The hypothetical machine described will serve as a basis for discussion of the architectural considerations posed by dynamic linking and loading. This approach is considered valid in that the features present in its design reflect a variety of those presently implemented in various processors. In this sense, an analysis of this machine will indicate many problems inherent in the conversion of present processor designs to allow dynamic linking and loading, but not all of these considerations will necesarily pertain to a given design.

As an addenda to this chapter, Appendix A provides examples of microprogramming this machine to accept a modified subset of the IBM 360 machine language.

Chapter 3. Implementation of the Dynamic Linking and Loading Environment

Introduction

Implementation of a processor which supports dynamic linking and loading of microprograms, in addition to the features provided by the hypothetical machine, requires modification of the microprogram address mechanism and hardware support for the linking and loading process. The concept of dynamic linking and loading of microprograms is meant to include both the mechanism for microprogram definition and the apparent virtualization of the control store. Direct user contact with the facility is provided by means of machine language primitives, which allow the user to define, modify or delete segments. This is intended to allow the user to dynamically redefine the base provided by the microprocessor to support the expected needs of his program. Indirect user contact with the linking and loading facility is provided through the automatic linking and loading of programs as they are referenced. This allows presentation of a virtual control store to the user.

The hardware facilities required to provide this facility are the subject of this chapter. The additional facilities are discussed in Chapter 4.

Three areas are of principle concern. The first is the suitability of absolute addressing, as provided by the hypothetical machine, to a dynamic environment. The second and third areas involve implementation of the

linking and loading facilities, respectively.

Modification of the Address Mechanism

As defined, the concept of dynamic linking and loading requires both the relocatability of microprograms and the ability to address microprograms of total length greater than that of the control store. Neither of these goals may be achieved with an absolute address mechanism.

Examination of microprogram organization indicates that division of microprograms into discrete sections is easily accomplished. This results from the traditional construction of microprograms as special sections of code associated with each machine instruction, and linkages from these sections to a few common sections of code. The length of the special sections of code varies widely, but is generally very short (a few instructions). A fixed block organization would therefore seem inappropriate. A variable length segment organization, however, would seem appropriate to both the apparent organization of microprograms and the varying length of the "independent" special sections of code. Under this scheme, each segment will be identified with a unique label (binary number) and addressing within each segment is provided in the form of an offset relative to the start of the segment. Local addressing, addressing within a segment, therefore requires only stipulation of an offset; wereas global addressing, addressing between segments requires both the label and offset within the segment desired. Implementation of this

scheme requires that these segment-offset addresses be mapped into the absolute addresses of the physical control store. This effect is achieved through dynamic linking.

Implementation of Dynamic Linking

Implementation schemes for the segment-offset address mapping into the control store addresses are shown in Figure 4. Translation of segment-offset addresses is provided in two ways. The segment table provides the necesary linkage information: the physical location of each segment. Addition of the required offset to the location of the segment desired results in the absolute address required. This form of mapping is used to address control store A (Fig. 4). Note that the segment table is implemented as a memory, where the contents of location SEG contain the linkage information related to the segment SEG. Alternatively, direct hardware recognition of the segment-offset address form may be provided through the use of relocation registers. This scheme, however requires a relocation register and physical memory block for each segment. Assuming a large number of segments (there will be), then the cost of providing these register would be prohibitive of the relocation register approach. Also, the control store configuration required (control stores B and C) is considerably more expensive than the single block configuration allowed by use of the segment table. Also, the segment table may be used to contain information concerning segments not currently control store resident

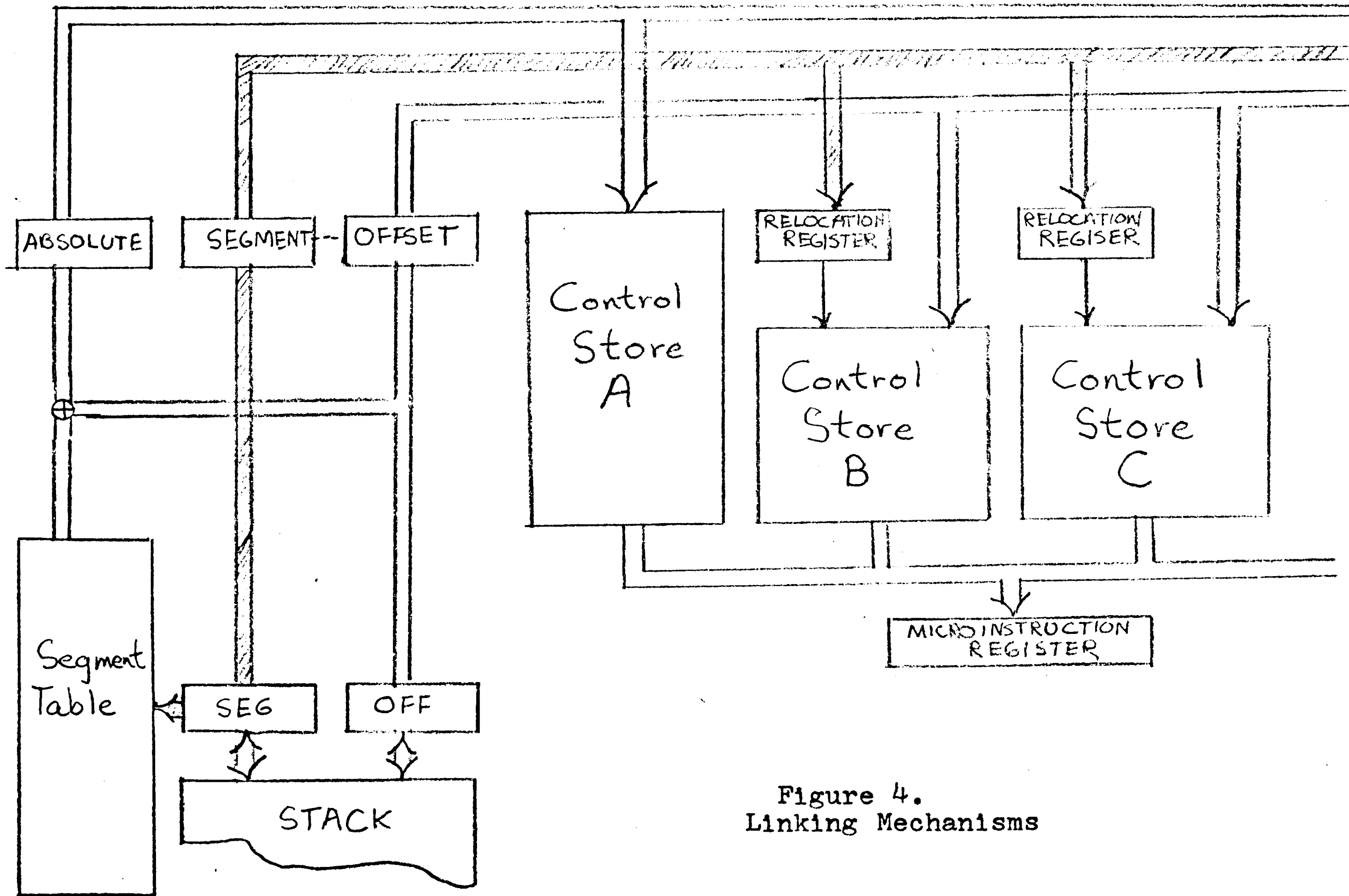


Figure 4.
Linking Mechanisms

when virtualization of the control store is desired. The segment table approach will therefore be used in conversion of the hypothetical machine to the segment-offset addressing form and its inherent linking capability.

Modification of the Successor Function

The successor function of the hypothetical machine was designed to accept data relevant to the calculation of absolute addresses from absolute addresses. This must now be modified to allow calculation of absolute addresses from segment-offset input. Note that the local functions Step, Repeat, Skip and $\pm$OFFSET are program counter relative. If the program counter is mainatained in both absolute and segment-offset forms, then the only modification required of the successor function in these cases is to update both forms. The Save&Step and Call functions must be altered to push the segment-offset form of the program counter onto the stack, rather than the absolute form. This is required in case the segment in which the successor function was evaluated is relocated before the value is popped by the execution of a Return. The unconditional branch to an absolute address form of the successor function must be modified to a useful segment-offset form. Preliminary to consideration of this problem,we note that the Jump and Call functions required the introduction of a twelve bit address into the jump-vector prior to execution. Practically, however, this required two transfers of eight bits each, despite

the fact that only twelve were required. This leads to the choice of allowing up to 256 segments of maximum length 256 words, and converting the jump-vector to a segment-offset interpretation. Now, difficulty in converting the absolute address form to a useful segment-offset form is apparent: sixteen bits are required but only twelve are available. The unconditional branch form will therefore be converted to two forms: Jseg which causes transfer to the first word of SEG as specified by the X register (may be masked), and Joff, which causes transfer to the segment in the SEG register at the offset specified by the X register (again may be masked). The Call, Jump, Return, Jseg and Joff successor functions all require use of the segment table to determine the appropriate address. Recalling the optimum choice of the times required for control store access and successor evaluation,

$$T_{mpcy} = T_{csac} + T_{next} ,$$

indicates that linkage between segments as a result of these global functions might result in adverse effects on T_{next} and therefore either T_{mpcy} and T_{csac}. If however, introduction of a value into the SEG register causes that location to be automatically fetched, then the Jump, Call and Joff succssor functions will cause no overhead if the segment table access time is equivalent to the local store access time; because these functions require prior introduction of a constant into SEG. The Return and Jseg

functions, however, provide no prior information on the segment referenced and will incurr overhead equal to the segment table access time plus the address generation time (Return only). This indicates that either the time required to evaluate the successor function must be reduced, or the access time of the control store must be reduced, if continuous linkage overhead is to be eliminated during microprogram execution. In proceeding, it will be assumed that the long-run overhead involved in these linkages has been effectively transferred to hardware overhead through appropriate modification of the original successor evaluation hardware, or if necesary, reducing the access time of the control store.

The Successor Function and Segment Faults

Another consideration in conversion to the segment-offset form is the possibility of generating addresses which have no control store equivalent. This will occur when either the segment referenced is not defined or the segment is defined but is not control store resident. In either case, a linkage or missing segment fault has occurred. The successor function must detect this condition and automatically take appropriate action. This action is defined as evaluation to the address of a microprogram specifically designed to handle this problem (the segment fault handler). An arbitrary implementation of this scheme is to define the missing segment fault handler as segment zero, and provide a special register to allow rapid determination of the absolute address

of segment zero. In the event of a segment fault, the successor functions value is contained in this register and transfer of program control occurs automatically.

Implementation of Dynamic Loading

The loading and relocation requirements of the dynamic linking and loading mechanism are most easily satisfied through the use of data block transfer channels. Use of such direct memory access (DMA) channels allows such transfers to continue without program intervention; and therefore allows continued execution during the transfer.

Implementation of the DMA channel is staigthforward in that it simply represents conversion of an already implemented concept to this application. Basically, a length register (L) is used to contain the number of words to transfer starting at the location specified by CSADR1 to the starting address CSADR2. Either address register may specify a main memory location, but not both. This therefore allows relocation of blocks within the control store, loading of blocks from main memory or storing blocks into main memory.

Transfer to (or from) main memory from the control store proceeds by"stealing" control store memory cycles once every T_{cy} (main memory cycle time) and the total time to transfer a block of length m is therefore $m \cdot T_{cy}$. This overhead may be considered considerably reduced if useful computation can procede in parrallel with the transfer. If the main memory cycle time is longer than

the microproccessor cycle time by a significant amount (at least a factor of two) then the following analysis of the overhead involved in such transfers is true. Assume:

1. Exactly m control store cycles are required for the transfer (ie block length is m)
2. The optimal choice of $T_{mpcy} = T_{next} + T_{csac}$ holds
3. The control store is organized as k equal length blocks (analysis with unequal lengths similar)
4. The main memory is organized as t equal length blocks of equal cycle times
5. Useful computation is available during the transfer

Then noting that the interim between successive references to the control store by the microprogram is T_{next} then the maximum time delay resulting from conflicting references to the control store is $T_{csac}-T_{next}$ and these conflicts can be expected with frequency m/t. Therefore the expected value for the time loss during the transfer is:

$$\frac{m}{k}(T_{csac} - T_{next}),$$

as a result of conflicts occurring in reference to the control store. Additionally, there is the probability of time loss at main memory in the event that the useful computation involves main memory references. In this case, the time loss due to conflict is T_{cy} and the maximum number of attempts to reference data from main memory (resulting in fetches--attempts which are satisfied by the cache don't count) is m. Assume n references are attempted (n less than m) then the time loss at main memory can be expected to be; nT_{cy} / t .

Actually, the probable value of n is low, especially

in comparison with t, and this value may probably be ignored. However, the expected value for the overhead involved in actually loading or storing is:

$$\frac{m}{k}(T_{csac} - T_{next}) + \frac{n}{t}T_{cy} \quad .$$

This value will be of later use in calculating the total performance gain achievable with dynamic linking and loading.

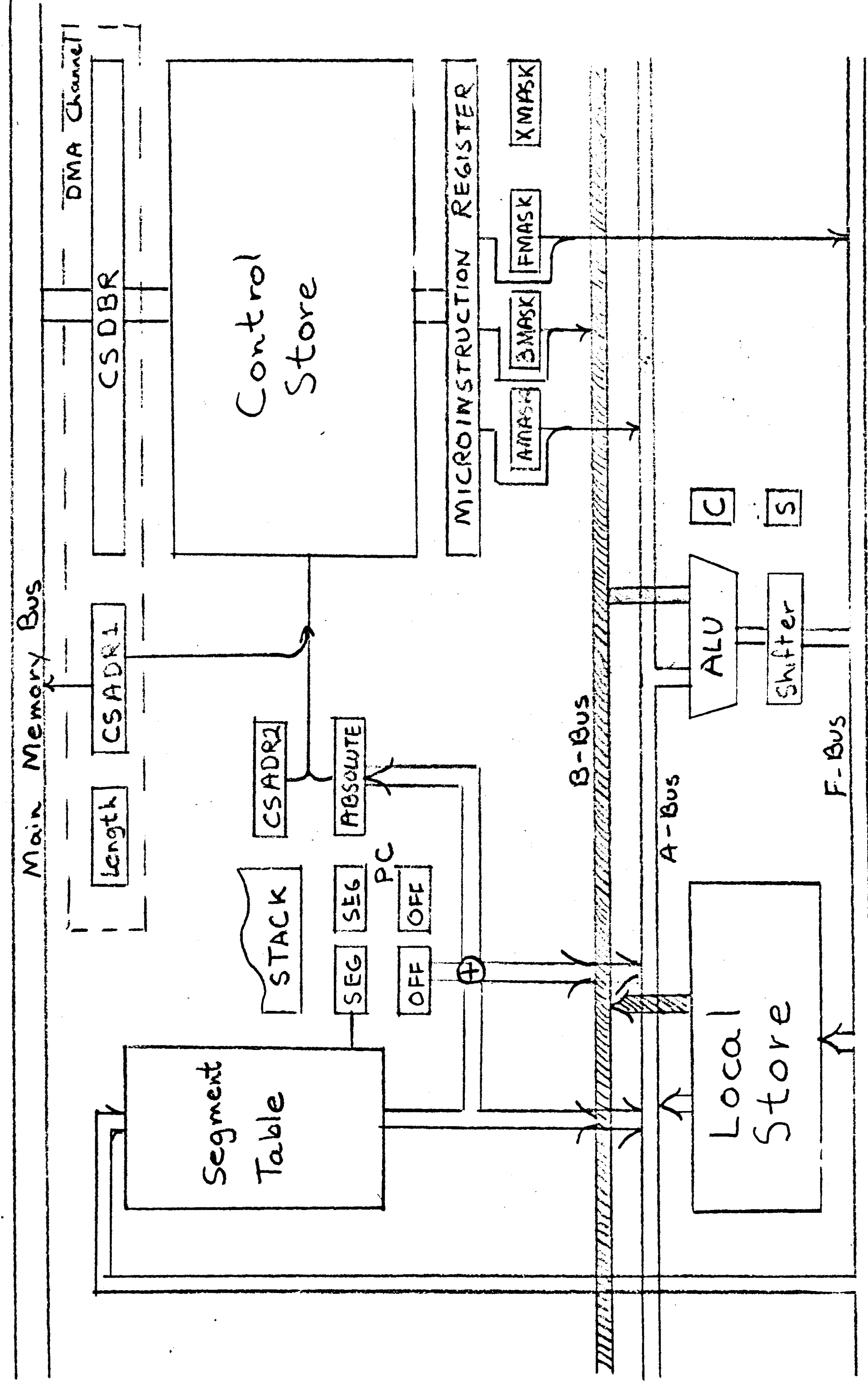
A similar analysis may be used to compute the time loss expected during relocation of a block of length m. The total time required is simply $2mT_{csac}$ if the transfer proceeds uninterupted within a single block, or about $(m+2)T_{csac}$ if uninterupted within two separate physical blocks.* Note that either one or the other should apply as references are made to consequetive addresses. If useful computation again exists, then the overhead is again expected to depend on the probability of conflicting references, the total number of references and the time loss involved in each conflict. The value for relocating a block of length m would therefore be:

$$\frac{2m}{k}(T_{csac} - T_{next}) \quad .$$

Again, this value will be of later use in establishing the performance gain acievable with dynamic linking and loading.

*This value also requires two points of access to the control store, otherwise the first value still holds.

Figure 5.
Overview of the Hypothetical Machine



Conclusion to Chapter 3

The overhead involved in the implementation of the dynamic linking and loading hardware has been examined. The requirements include conversion of the read-only control store to read-write capability, the provision of a segment table and modification of the successor function to allow segment-offset addressing and segment fault handling. Additionally, the overhead involved in provision of the loading mechanism has been described, and the overhead involved in loading computed. A summary of the implementations of these capabilities is provided in Figure 5. Chapter 4 will provide the necesary algorithms to allow these facilities to be used to provide an effective dynamic linking and loading environment.

Chapter 4. Evaluation of the Dynamic Linking and Loading Environment

Introduction

Evaluation of the linking and loading environment, provided in Chapter 3, requires an analysis of the use of that environment to provide a useful linking and loading facility. This evaluation involves essentially two areas of interest. The first, the initial overhead involved in establishing the environment, was discussed in Chapter 3. The second, involves the overhead involved each time the facility is utilized.

Direct utilization is provided by means of machine language primitives designed to allow program manipulation of the control store and linkage table. While this allows the user the priviledge of dynamically reconfiguring the microprogram environment, it also presents the user with problems of memory management and requires a rather definite knowledge of his programs' behavior.

Indirect utilization of the facility is provided to free the user from these responsibilities. This requires an automatic mechanism to cope with the problems of memory management and segment faults. The manual "virtualization" provided through the use of machine instructions is therefor replaced with virtualization in the conventional sense. This transition, however, requires a discussion of the problems involved in segment fault handling. Especially difficult are the problems posed by memory management and

the segment removal algorithm. The combination of the overhead involved in implementing suitable solutions to these problems may be combined with the estimates of the initial overhead to determine the minimal degree of effectiveness required to justify implementation of the dynamic linking and loading facility.

Program Controlled Linking and Loading - Description

Machine language instructions, sufficient to allow program control of the linking and loading processes, are:

> define Seg,Link , which replaces the old segment table entry for the segment Seg with the contents of the location pointed to by Link,
>
> move Seg,Loc , which moves the segment Seg to the location indicated by Loc and updates the segment table entry for Seg to indicate its new location (the move may not be specified as main memory to main memory),
>
> status Seg,Loc , which stores the segment table entry for the segment Seg at the location indicated by Loc.

The define primitive allows the initial creation of a segment table entry, thereby defining a segment. The segment table entry must contain the main memory location of the segment and the length of the segment; other information concerning the segment can also be provided, but will not be needed in this case. Subsequent use of the move primitive allows the segment to be loaded, relocated or stored. Relocation and storage of segments may be required to allow the loading of another segment; this determination is made on the program's knowledge of the location and length of all control store resident segments, and the length of the segment to be loaded.

Concievably, the program would not have sufficient information to make this determination, in which case the use of the status primitive allows examination of the location and length of any segment. This situation is liable to occur when calls to subroutines or functions are executed, and the function or subroutine is not aware of the exact microprogram environment at the time of the call. Note that this difficulty results from the necesity of loading all segments,before they are referenced, in the absence of an automatic segment fault handling facility. Therefore, the programmer must be careful to provide the expected microprogram base when machine language program behavior is not stricity sequential. As the complexity of program behavior increases, this could pose a formidable problem for the programmer.

Program Controlled Linking and Loading - Overhead

In addition to providing potentially difficult problems to the programmer, program controlled linking and loading is less efficient than the hypothetical optimal algorithm. The optimal algorithm uses advance knowledge of the program's behavior to provide optimal decisions when the need for linking or loading occirs. This therefore provides a measure of the absolute minimum of overhead involved in dynamic linking and loading. Program controlled linking and loading is less efficient than the optimal algorithm because it may require the linking and loading of segments to avoid potential segment faults which do not exist. Similarly, it may also

store segments unecesarily to allow the loading of other segments. A measure of the overhead involved is obtained by considering the conditional probability that a segment will be referenced given that another segment has been referenced. This measure is applicable to the case where it is known that the interpretation of a machine instruction will result in execution of a particular microprogram segment, but this segment potentially links to n other segments with probability p_i, for the ith segment. This should be combined with the probability of executing the relevant machine instruction, m, and the overhead involved in providing the ith segment, t_i, to give the expected value of the overhead incurred by the optimal algorithm:

$$m \sum_{i=1}^{n} p_i t_i \, .$$

This should be compared with the overhead incurred by the program controlled algorithm:

$$\sum_{i=1}^{n} t_i$$

where it is assumed that the program does not know if the instruction will be executed. In practice, the values for the overhead involved, t_i, may be calculated from the occurrence of the primitives and the probability that the relevant instruction sequence will be followed; and the overhead involved in the execution of each primitive. For the <u>define</u> and <u>status</u> primitives, this value is determinable from the microcode required in their interpretation; but for

the <u>move</u> primitive, the overhead is that calculated in the previous chapter for the desired operation. As the optimal algorithm is impossible to compute, the use of primitives provides an effective approximation of the maximum overhead required to support dynamic linking and loading in specific instances. This does not indicate that a different algorithm cannot perform worse in some instances and better in others, but simply indicates the maximum overhead which need be allowed in each case. In this sense, it provides a basis for measuring the "excess" overhead involved in the implementation of the automatically controlled linking and loading algorithms.

<u>Virtualization of the Control Store - Automatic Linking and Loading</u>

Except for the original definition of segments, the "virtualization" of the control store through direct program control could be replaced with an automatic mechanism. Such a mechanism frees the programmer from the difficulties involved in maintaining the appropriate microprogram environment, and retains the advantages of control store virtualization provided by the programmer. Implementation of the automatic mechanism however, requires that the control store bookeeping operation,provided by the programmer,be replaced with suitable algorithms. Overhead will then be incurred by the execution of these algorithms. In comparison with program control, this may or may not exceed the savings achieved through the elimination of unecesary operations involved in the direct

control algorithm.

An overvew of the automatic segment fault handling algorithm is shown in Figure 6. The procedure is simple, provided sufficient free space always exists to allow immediate loading of missing segments. In this case, the overhead is approximately equal to the overhead involved in the loading process. Practically however, if the total length of the microprograms exceeds the length of the physical control store, then segment removal and garbage collection must preceed the loading process. This situation is analogous to that of conventional virtual memories; but, significant differences exist between the considerations involved at the microprogram and machine program levels. Consideration of appropriate segment removal and garbage collection algorithms is therefore necesary.

Microprogram Behavior - A Basis for the Algorithms

The choice of segment removal and garbage collection algorithms must be made on the basis of expected microprogram behavior. The lack of data in this area, especially for processors which allow user definition of microprograms, requires the choice of reasonable algorithms without proof of their optimality. Traditional microprogram organization and behavior will serve as the basis for the choice of these algorithms.

Microprograms are traditionally oriented toward the interpretation of a predetermined machine language. The

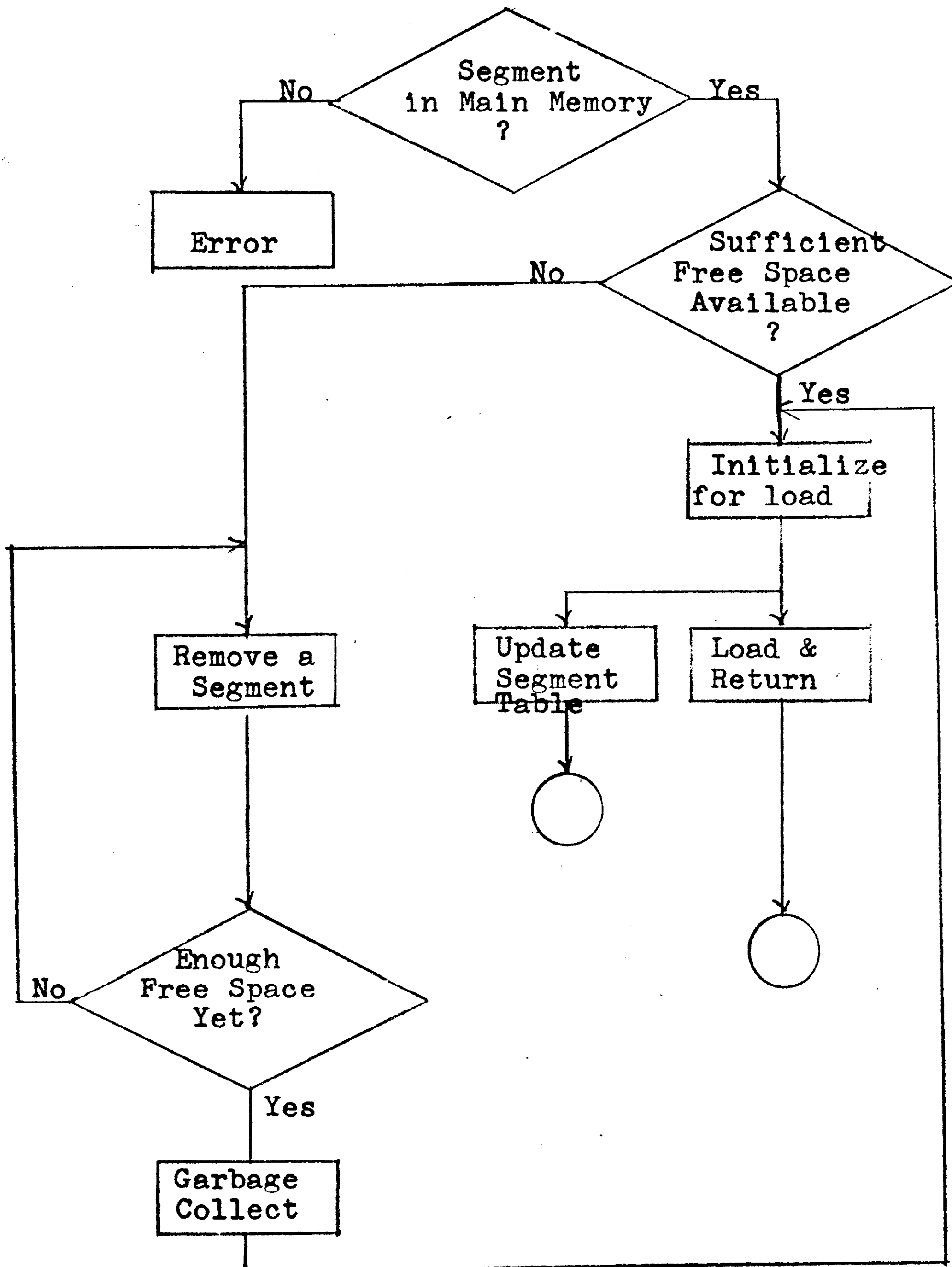


Figure 6.
Missing Segment Fault Algorithm

usual method of implementing this interpretation is to code special sections of microcode for the interpretation of each instruction. These sections are linked to the machine instruction fetch and decode section(s) of the program, and also to the special segments associated with the interupt mechanism of the processor. A typical example is the interpretation of the IBM 360 machine language in the Model 50. The first phase involves the initial instruction fetch and program counter updating. Two successive phases result in the decoding of the instruction. The instruction is then executed. This phase may entail additional succesive phases, as in the possible case of microprogrammed floating-point operations. After execution, the microprogram branches to the interupt checking phase and then succesive branches eventually return to the initial instruction fetch phase.[4] This behavior is summarized in Figure 7a. The exact behavioral path differs from machine instruction to machine instruction; therefore, if the occurence of a particular type of machine instruction is treated as a random event, then this behavioral pattern will be reflected in the execution of various microprograms associated with the particular instruction. An additional characteristic of this pattern is the relatively short duration of many of the phases, especially the decode and execute phase for the more basic machine instructions. If each phase is implemented as a separate segment (this is virtually required to allow

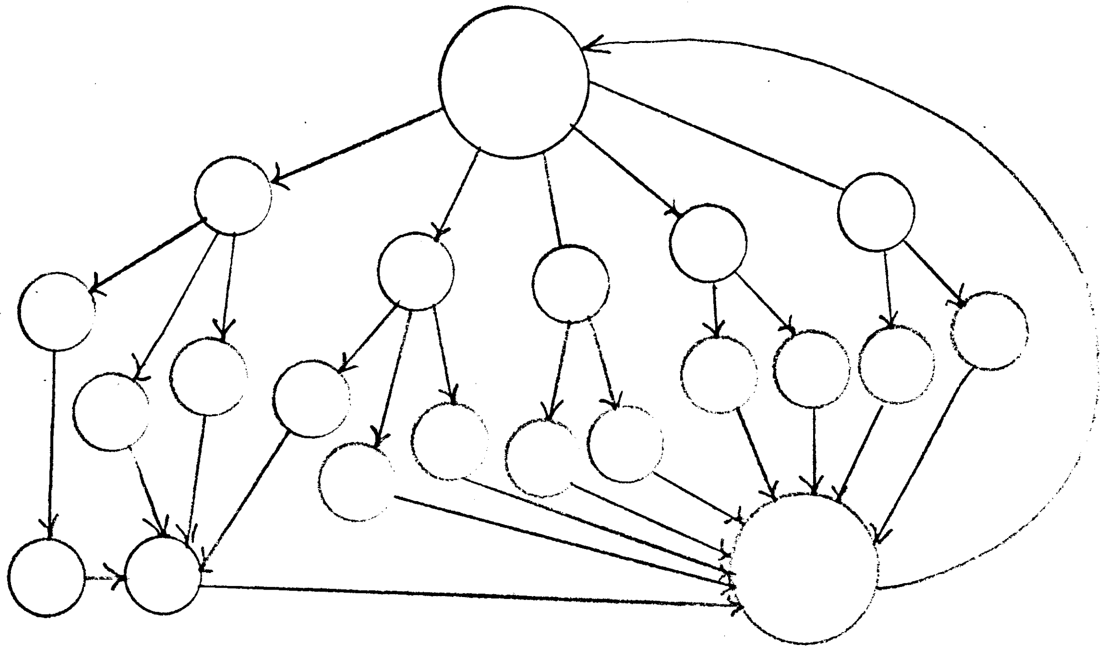


Figure 7a
Conventional Microprogram Behavior

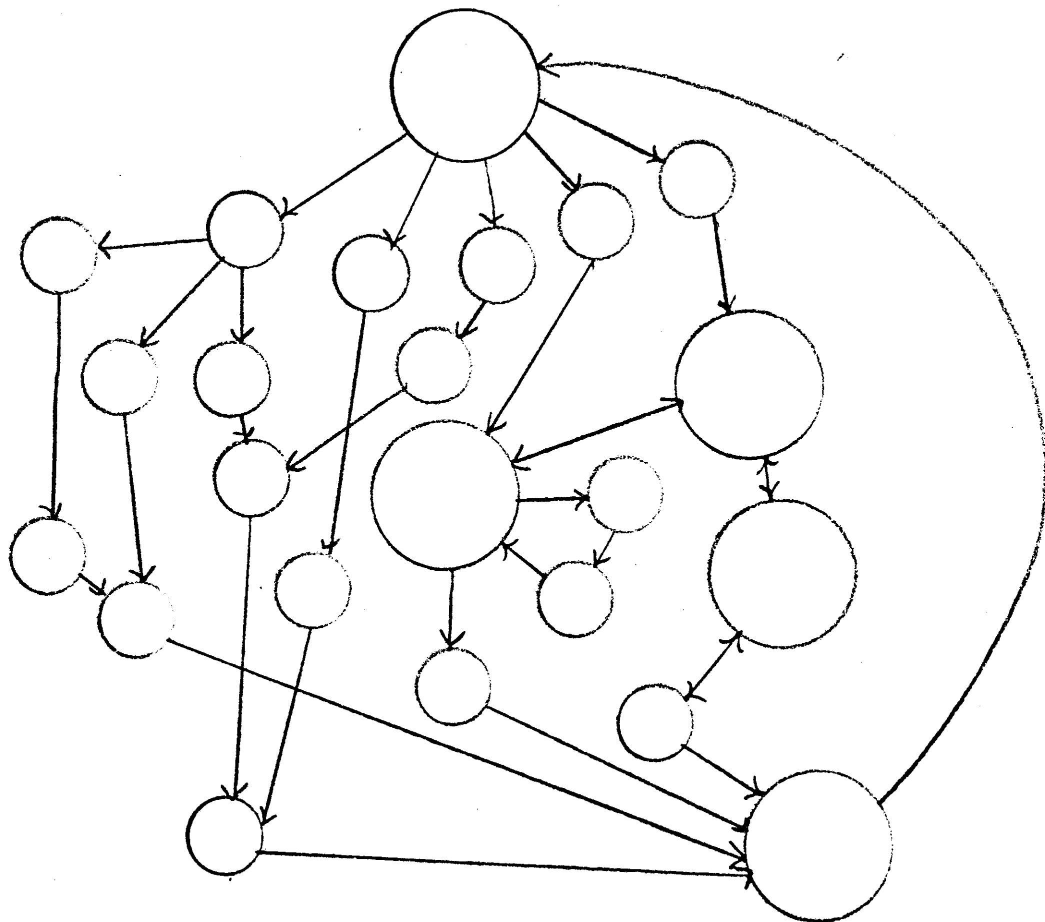


Figure 7b
Expected Microprogram Behavior

the appropriate linkages) then the following summarizes the behavior of conventional microprograms:

1. The use of a distinct segment for the interpretation of each machine-level instruction.
2. The random occurence* of execution of these distinct segments as a result of machine-level program construction and execution.
3. Relatively short intervals of local execution.
4. Relatively frequent linkages between segments.
5. Microprogram overall organization as a very large number of short segments (with exceptions).
6. The execution of certain segments regardless of the instruction being executed.

Allowing user definition of microprograms, or the definition of microprograms which compute more complex algorithms than those typically chosen for machine-level instructions, is expected to change the overall behavior of the microprogram very little. The major difference is expected to be the use of longer segments (with longer spans of execution time) which link to several other shared segments. This would result, for example, from the provision of floating-point operations such as add and subtract which would be used by instructions computing trigonometric functions. Another example would be the use of common string manipulative functions in the construction of microprogrammed editors or parsers. Figure 7b represents the expected behavior of microprograms when the dynamic linking and loading facility is implemented.

<u>The Removal Algorithm - Discussion of Conventional Methods</u>

Analysis of this behavior pattern indicates that the

*By "random occurence" we mean that the occurence of a particular instruction is independent of the instructions preceeding it. The frequency of occurence of different machine level instructions is expected to vary.

conventional segment removal (or page removal) algorithms may not be suitable for use at the microprogram level. Algorithms, such as least-recently-used* (LRU), are dependent upon the assumed local behavior of machine programs. In other words, the machine language program is expected to proceed in a basically sequential pattern or execute in relatively small portions of the program as a result of looping, subroutines, functions or recursion. This type of behavioral pattern allows the assumption that if a segment is referenced once, then it can be expected to be referenced again within a short time interval. This also allows the assumption, that if a segment has not been referenced for a long period of time, then it probably will not be referenced within a short time interval. These observations result in the use of LRU (or modifications thereof) algorithms at the machine-language level.

The behavioral assumptions for machine language programs are clearly different from those established for microprograms. The random nature of microprogram behavior and the execution of certain segments regardless of the previous execution pattern are inconsistant with the intent of the least-recently-used algorithms. Additionally, these algorithms require the real time maintenance of segment reference data; and the relatively frequent occurence of

*This algorithm is interpreted as follows: assume the available memory space is filled and a segment fault occurs, then remove the segment which has not been referenced for the longest period of time and place the code for the segment requested on the top of a stack. Subsequent references to segments cause their codes to be pushed onto the stack. The least-recently used segment is therefore always on the

linkages between segments,at the microprogram level, creates significant overhead in maintaining such data. Also, the number of segments which will probably be control store resident greatly exceeds the number of segments or pages normally used in the machine-language environment. Therefore, the use of conventional segment removal algorithms,at the microprogram level, is considered invalid; and construction of an algorithm, suitable for use at the microprogram level, is required.

The Removal Algorithm - Least Frequently Used, A Possible Solution

The behavioral characteristics of microprograms would suggest the use of a least-frequently-used (LFU) algorithm for segment removal. This algorithm stipulates that the segment which has recieved the fewest references is the segment which should be removed. Full implementation of this algorithm would require that a data base be continually updated to reflect the number of references to each segment. This is not useful for two reasons: first, the overhead involved in maintaining this data base is prohibitive, and second, after a significant period of execution the relative priorities among most segments would remain essentially constant. The second effect results from the fact that frequently used segments would rapidly develop priorities of such greater magnitude than other segments, that short-

bottom of the stack. Note that when the code for a segment is pushed onto the stack, any other reference to that segment is removed from the stack; therefore each segment is referenced by at most one stack location.

anomilies in the behavior of segments would not be corrected. Further, after prolonged periods of execution essentially all of the segments are expected to develop relatively constant priorities. This does not, however, preclude the possibility of changes in the relationship between segments of extremely low or extremely close priorities.

This discussion indicates that a static assignment of priorities could be made which reflected the observed long-run frequencies of execution of the various segments. This method eliminates the overhead involved in maintaining a data base, while maintaining the essential features of the LFU algorithm.

A basic objection to the LFU algorithm is that the occurence of nonprobablistic behavior of one segment will also indicate nonprobablistic behavior of the segments associated with the first. This could lead to the case of two (or more) segments which alternately reference each other, but both are of extremely low priority and therefore replace each other. Severe system performance degradation could result if these segments performed a significant number of cross-linkages. Two variations of the LFU algorithm are now considered which attempt to reduce or eliminate occurrences of this sort.

Least Frequently Used - Modified with Association Lists

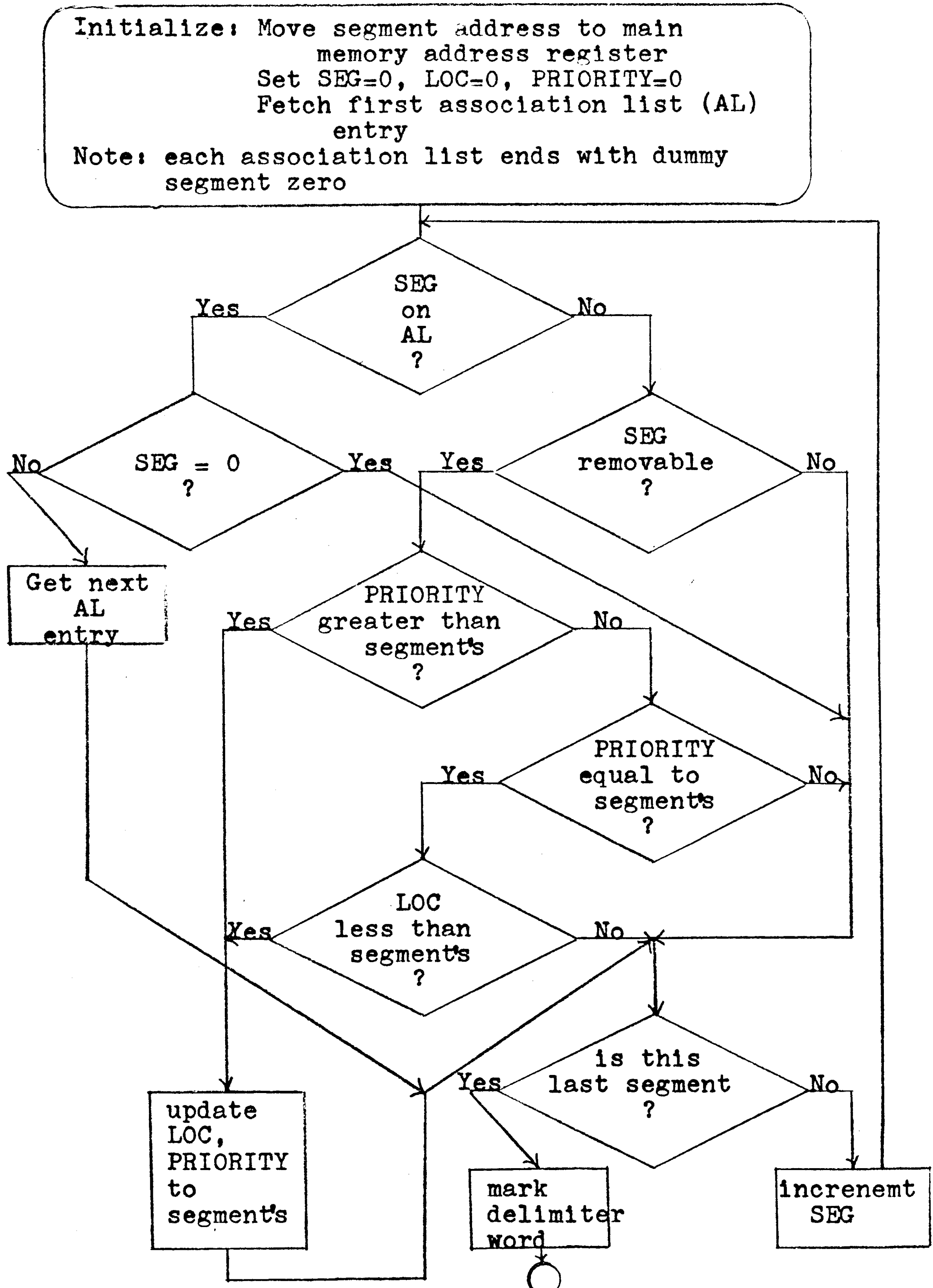
A direct approach to solving this problem is the specification of the segments associated with another segment. Included in the definition of each segment would

be a list of the segments to which linkages could be expected from that segment. The removal algorithm is then modified to stipulate that the least-frequently-used segment, not on the association list of the segment being loaded, should be removed. Implementation of this algorithm would therefore involve searching the segment table for the first segment which meets the criteria. This is effective if the association list is provided as an ordered list of the segments required; in which case the algorithm shown in Figure 8 requires at most one reference to each segment table entry and at most one reference to each member of the association list. The objection to this algorithm is that it never requires significantly less overhead,regardless of the segment loaded, because no look-ahead is possible. This is meant to indicate that despite the static definition of relative segment priorities, it is impossible to accurately predict which segment should be the next to be removed.

Least Frequently Used - Modified with Most Frequently Removed

An alternative approach to association of segments utilizes a most-frequently-removed algorithm and special control store search to determine which segment to be removed. Basically, this algorithm attempts to eliminate infrequent anomolous behavior by causing the least-recently removed segments of a given priority to be removed first; and attempts to adjust long-run anomolous behavior by increasing the priority of segments which show a high

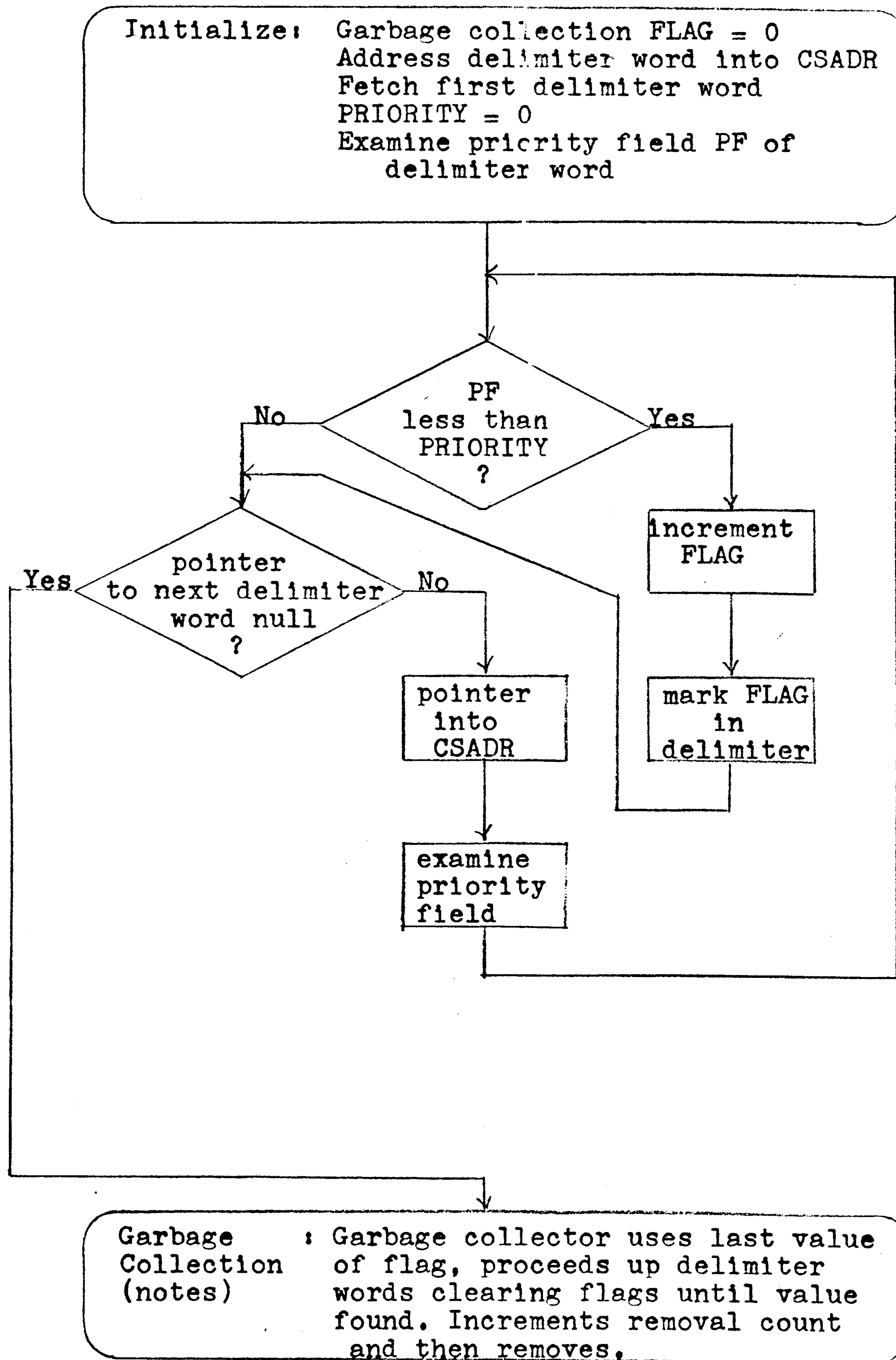
Figure 8.
Least-Frequently Used Algorithm Including
Use of Association Lists (LFU-AL)



frequency of return after being removed. This algorithm is implemented as follows. The delimiter word (see section on garbage collection) of each segment is modified to include a count of the number of times the segment has been removed in addition to the other data. Searching from the lower addresses of the control store (initially higher priority) the algorithm proceeds up the chain of delimiter words until the lowest priority segment is located. This segment is then removed; its count of the number of times it has been removed is incremented, and if the count overflows the priority of the segment is incremented and the count reset to zero; the garbage collection algorithm relocates all segments,located at higher addresses,downward; and the new segment is loaded at the top of the segment addresses. This algorithm is flowcharted in Figure 9. The result of this algorithm is to examine the most recently loaded segment last (in some sense this reflects LRU), and if a segment is removed several times (indicating it is used several times), then its priority is increased to reuce the frequency of its removal. The method of search and relocation also results in the accumulation of the most frequently used segments in the low end of the control store and thereby eventually decreases the overhead involved in garbage collection.

The difficulties with this algorithm are essentially the same as those of the LFU, except that they are reduced in overall effect. It remains possible to remove a segment which will be immediately requested, but frequent occurences

Figure 9.
Least-Frequently-Used Algorithm Including
Use of Removal Counts (LFU-MFR)

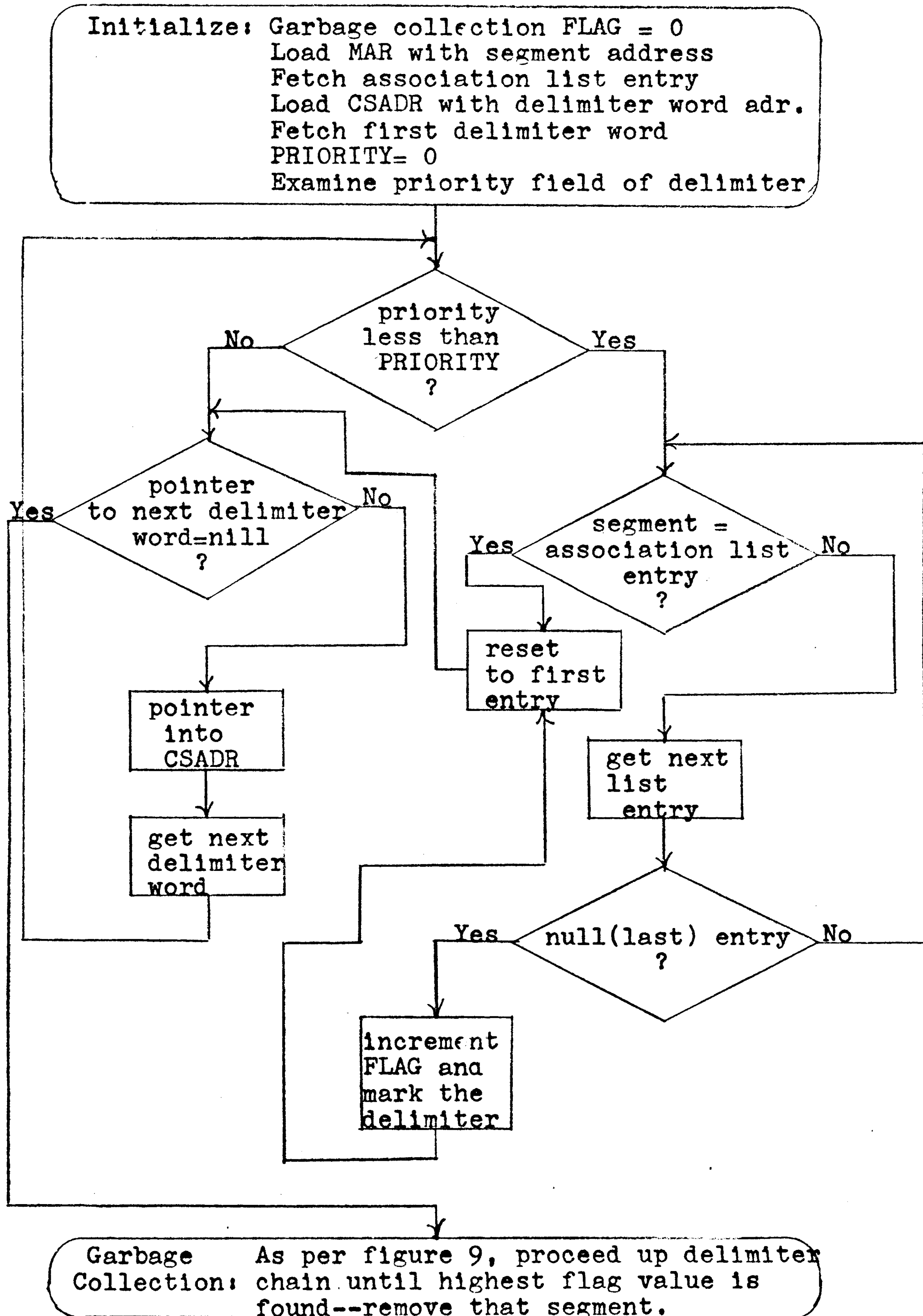


of such behavior will result in feedback which would eventually reduce the problem. Similarly, long-run anomilies in particular programs will be adjusted to improve performance. None-the-less, the immediate response of the algorithm to anomilous behavior is not as good as the use of association lists.

Least Frequently Used - Modified with a Combination of Association Lists and Most Frequently Removed

A combination of the two modifications of the LFU algorithm is a useful means of providing both long-run and short-run adjustment of anomolous behavior. This algorithm may be implemented as in Figure 10. The basic procedure is to follow the procedure outlined for the most-frequently-removed algorithm with the addition of comparing each segment,which otherwise would have been chosen for removal, against the association list of the segment being loaded. Inefficiency results from the fact that the search is no longer ordered, and therefore the association list may be searched several times before the algorithm completes. This problem could be reduced by restricting the length of the association list, or by providing a small associative memory to allow parrallel comparison with all the association list entries simultaneously. Neither alternative is especially desirable: the length restriction must be severe to allow overhead comperable to that obtained solely with the association list method (reduction to about 2 or 3 segments); and the second requires a potentially significant hardware invest-

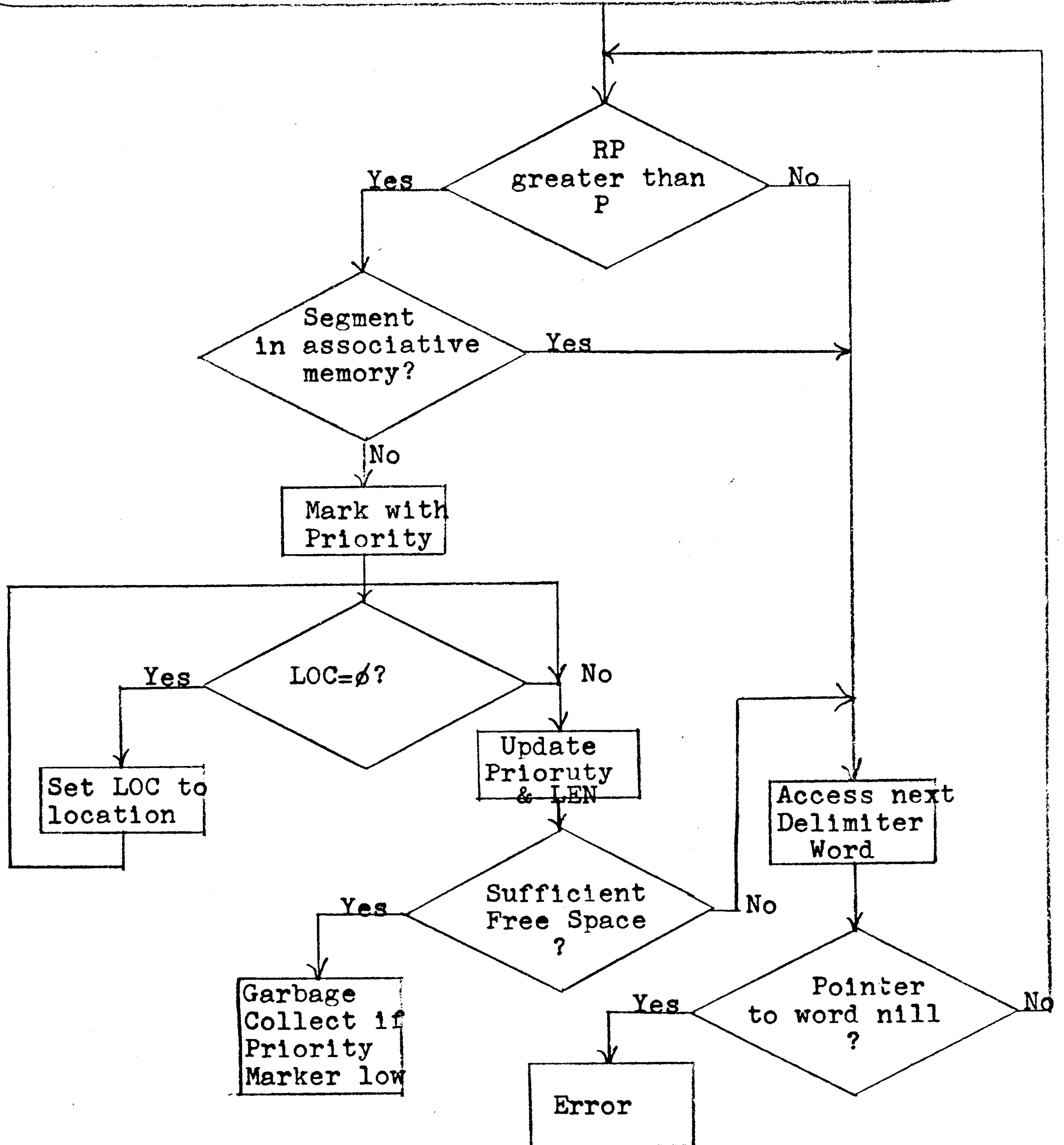
Figure 10.
Least-Frequently-Used Algorithm Including
Use of Both LFU-AL and LFU-AL-MFR



ment. Again, combination of these methods might prove effective, especially if the original lengths of the association lists were relatively short. Limited experience in the construction of microprograms has indicated that association lists on the order of eight to sixteen segments long are sufficient to eliminate first-order effects of segment faults. Higher-order effects, resultant from segments which link to segments which link to other segments (all linkages of low priority), may require significantly longer lists; but these could be treated as special cases in construction of the association list. Therefore, a limit of about sixteen members per segment association list will be imposed, and an associative memory of this length will be provided (for this size memory, the addition constitutes negligible cost). The revised implementation of the LFU algorithm is shown in Figure 11.

The Least-Frequently-Used Algorithms - Examples

An intuitive comparison of the variations of the least-frequently-used algorithm may be achieved by considering examples of the behavior of these algorithms in special cases. As a prelude to this discussion, it is necesary to establish the number of priorities and the length of the allowable removal count, the number of possible segments and similar factors. Arbitrarily establish the maximum number of priority levels as sixteen, the maximun number of segments as 256, the maximum length of each segment as 256, the overflow value for the removal count as 256 and



A Modification of the Combined Algorithms
Figure 11.

the average value of the length for each segment as 20 words. The last assumption indicates that about four-fifths of the available segments will be control store resident. Also, establish 16 as the highest priority level for removable segments, and establish 0 as the priority for segments which cannot be removed (the segment fault handler for example). Initially, let most of the segments have priority 0 and allow the following configuration of high address control store hold:

```
g(2,0)
f(1,0)          k(1,0) not control store
e(3,0)                 resident
d(4,0)
c(6,0)
b(7,0)
a(11,0).
```

The notation x(y,z) indicates"the segment x whose priority is y which has been removed z times at that priority level". Now consider the following examples.

Example 1. - Normal Behavior

In the absence of segment faults, all of the algorithms perform equivalently and create no overhead.

Example 2. - First-Order Effects

In this case we assume that segments f, c and b are linked in an iterative loop and that segment k is also part of the loop. In the use of the unmodified LFU algorithm, the first reference to k will create a segment fault and segment f will be removed. The next reference to f will then result in a segment fault. Subsequent references to these segments will continue to yield segment faults.

In use of the LFU algorithm with association lists (LFU-AL),

the first reference to k will again generate a segment fault. However, if k's association list is f,c,b, then segment g will be removed and the loop will continue without creating further segment faults. In the use of the LFU algorithm with the most-frequently-removed variation (LFU-MFR), the first reference to k and subsequent references to f and k will cause segment faults for the first 256 references causing these faults. At this point, f and k will have accumulated priority greater than g, and segment g will then be removed. Note that the responsiveness of LFU-MFR is highly dependent upon the overflow value for the removal count. Use of lower value improves responsiveness, but the speed with which maximum priority is reached is increased; at which point, the priorities must be reinitialized. The use of the LFU algorithm with both modifications (LFU-AL-MFR) results in behavior identical to that of LFU-AL. In summary, LFU-AL and LFU-AL-MFR performed well in eliminating first-order effects in linkages between low priority segments.

Example 3. - Higher-Order Effects

In this case, we assume that segment k references segments g or e, and that e occasionally references f as a called function. Again assume looping behavior, also assume the following association lists: for k, g,e,f; for g, k,e,f; for e, g,f,k; and for f, g,e,k. The use of LFU will again result in segment faults for each alternate reference to segments k or f. The use of LFU-MFR may eventually result in priority increases for g,f and e such that segment d will

be removed; but this effect will require considerable time. If normal behavior follows this loop, then segment d may not be of sufficiently high priority and further difficulty could result from subsequent segment faults. The use of LFU-AL or LFU-AL-MFR results in the removal of d immediately upon the first segment fault (reference to k), and normal behavior thereafter. For more complicated examples involving longer association lists, the behavior of LFU-AL and LFU-AL-MFR will begin to differ. This results because the specification of a long enough association list eliminates all order effects in the LFU-AL, wereas the limited length of the LFU-AL-MFR association list will allow manifestations of higher-order effects.

In concluding the disscussion of the examples, two points are evident. The first is the superiority of the association list method over probablistic approaches to segment removal. The second is the inflexibility of this approach in adapting to "long-range" effects of looping at the machine instruction level. Another less obvious point is that the use of too long an association list will make segment removal impossible. This problem may be eliminated by providing, in addition to the association list, a removal list which indicates which order the segments on the association list should be removed. The problem of long-range inflexibility is more difficult.

Another Variation of Least-Frequently-Used

This situation would indicate that a slightly dif-

ferent form of the LFU-AL-MFR should be employed. A possible approach is the use of the LFU-AL algorithm to determine which segment should be removed, and the use of a fault count (this is essentially equivalent to the use of a removal count) to indicate the number of times a segment must be loaded. As in the case of the MFR variant, this count would be used to adjust the priority of the segment. Because LFU-AL is used, the fault count will reflect the machine-level behavior of the program, while the use of association lists will provide knowledge of the microprogram behavior in the processing of each machine instruction. An example of this application would be a mchine-language instruction loop which included a seldom used instruction; in which case, only microprogram behavior is important and the use of LFU-AL is sufficient. If however, the use of this instruction becomes frequent due to a large number of programs (as would be the case in a special applications computer facibty), then the fault count correction would adjust the priority to a more optimal level. Hence, this algorithm provides against short-run anomolous behavior and eventually corrects anomolies in the long-run application of the linking and loading facility.

The Least-Frequently-Used Algorithms- Execution Overhead

Besides considering the behavioral aspects of each of the removal algorithms, the overhead involved in the implementation of each indicates the desirability of that

Figure 12.

Figure 12 is identical to Figure 8, except that the initialization stage first involves incrementing the fault count for the segment. Also, the interpretation of priority now includes the fault count.

algorithm. Basically, the overhead involved in the implementation of the unmodified LFU and the LFU-MFR algorithms is the lowest because these algorithms allow look-ahead to determine a list of segments to remove. Both the LFU-AL algorithm and its variant (Figure 12) require essentially the same overhead; although the latter also requires a longer segment table entry. The LFU-AL-MFR algorithm requires the most overhead in that both more time and hardware are required to execute the algorithm.

With reference to Figures 8 and 12, the overhead involved in implementation is nearly constant. Most of the overhead results from the need to search the entire segment table (256 entries), and the approximate total of microprocessor cycles required to execute this algorithm is about 550 plus the length of the association list. Note that in general it requires only one microprocessor cycle to determine that the segments priority is unsuitable and to obtain the next entry, but that two or three are required if this possibility exists.

An initial consideration of the searching algorithm involved in either of the LFU variants involving use of the delimiter word chain indicates that about five or six microprocessor cycles will be required to determine the suitability of each segment. As at least half of the 256 segments are expected to be control store resident, the overhead involved in this area alone is nearly twice

that of the LFU-AL algorithms just discussed. In the case of the simple LFU-MFR this overhead could be significantly reduced by maintaining an ordered list of the segments to be removed. The overhead would then be reduced to a few instructions. This however, requires the addition of a separate table for this purpose, and also requires that this table be updated each time a segment is loaded. The overhead then requires slightly less overhead than the LFU-AL algorithms while not providing equivalent performance.

In summary, the variant of the LFU-AL algorithm shown in Figure 12 is expected to perform reasonably well, and also involves significantly less overhead than the acceptable alternatives. A conservative estimate of the expected overhead involved in the implementation of this algorithm is considered to be about 600 microprocessor cycles. This value will therefore be used in estimating the total overhead involved in linking and loading.

The Garbage Collection Algorithm

Having determined the segment or segments to be removed (or simply dropped) it becomes necesary to provide sufficient consequetive free addresses within the control store to allow loading of the needed segment. This may be accomplished in several ways, each of which involves a different degree or type of overhead. The principle overhead incurred by any approach will be the relocation of segments and the associated need to update the segment

table entry. Basically two approaches to garbage collection will be considered. The first is simply compacting the free space,obtained through segment removal, at the high address end of the control store. This involves relocating all the segments located at higher control store addresses than the segment removed. Regardless of the number of segments removed, each segment need only be relocated once. The second approach involves the chaining of segments and free space such that actual garbage collection occurs only when a certain proportion of the control store is free. This reflects an attempt to tradeoff control store for less frequent relocation of segments.

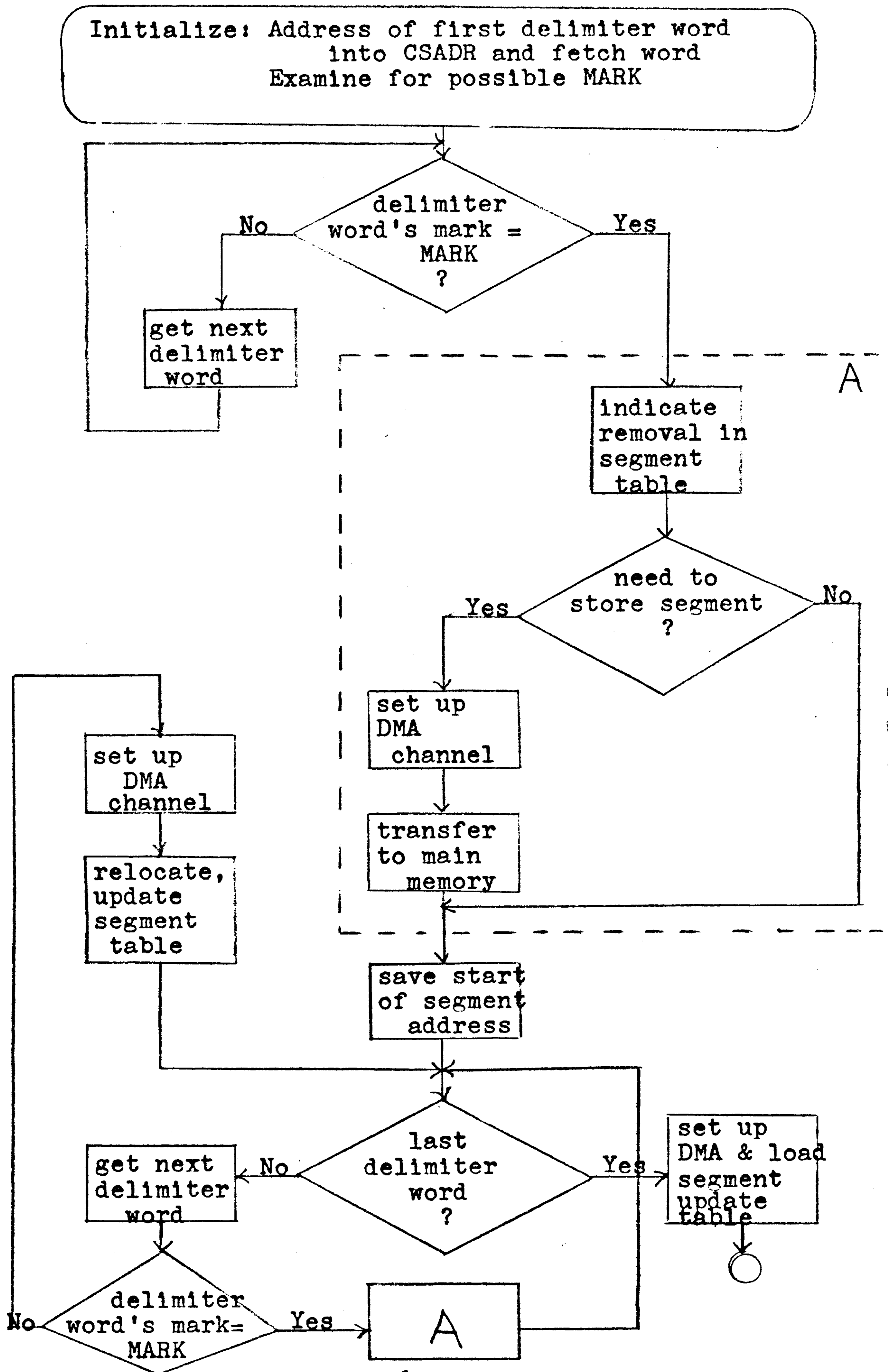
Garbage Collection Algorithms - Immediate Compaction

The immediate compaction of free space after segment removal always incurrs a significant amount of overhead in the relocation process, but also guarantees that the maximal usage of the control store is achieved. It also has the advantage, if used in conjunction with the LFU-AL method, of continually relocating the most frequently used segments towards the low end of core. This results in lower overhead from successive relocations because fewer segments will be relocated. This algorithm is implemented with the use of a segment delimiter word chain. A segment delimiter word is associated with each segment. This word is located in the word immediately preceeding the segment and contains the following information: the number of the segment, the length of the segment, the address of the next segment's

delimiter word, a flag (used by removal algorithm) which indicates whether or not the segment should be removed, and any additional information (such as special data for the segment in performance measurements) desired. The algorithm (Figure 13) proceeds by following the delimiter chain until the first flag is located. This address then represents the starting address for relocation. The algorithm stores this address and continues up the chain until the first unflagged word is found. The segment number is determined, the segment table is updated to show the new address, and the segment is relocated. The address to which the next segment should be relocated is automatically generated by this process. The algorithm continues up the chain, ignoring flagged segments, until a null pointer indicates the last segment. This pointer is replaced by the address of the first word of free space and the new segment is loaded at that point.

The overhead involved in this process is dependent upon the total number of segments in the control store and upon the number and length of those relocated. Assume that the control store typically contains q segments and that the length of each is m words. Also assume that control store and main memory are organized as k and t equal length blocks respectively. Also assume that about n words of control store need not be relocated, and that the desired segment is of length m. This allows calculation of the overhead involved in garbage collection

Figure 13.



and loading as:

T_a	$\frac{2m+2}{k}(T_{csac} - T_{next})$	, to relocate and relink each segment
a	$\frac{4,096 - n}{m} - 1$	, segments to relocate
T_b	$4(T_{mpcy})$	, to examine each delimiter word
b	$4,096/m$	, delimiter words to examine
T_c	$4(T_{mpcy})$	, when flagged word is found
T_d	mT_{cy}	, is time to load new segment in absence of reference conflicts

hence total time is: $aT_a + bT_b + T_c + T_d$.
For reasonable assumptions on the parameters involved this total value is between about 500 and 4000 T_{mpcy}, and can be assumed near 2000 T_{mpcy}. These values reflect an m of about thirty, a T_{cy}/T_{mpcy} of about four, a k of 1 and varying differentials of control store access and successor evaluation times. The major portion of the overhead results, as expected, from the relocation overhead in aT_a. The advantage of using this method, however, can be seen in the expected increasing value of n as appropriate priorities and respective control store locations are established for each segment. Nonetheless, the overhead involved is substantial and an attempt to reduce it should be made.

Garbage Collection Algorithms - Delayed Compaction

An alternative to immediate compaction is the usual approach of delaying the process until a certain portion

of the control store becomes effectively useless space without compaction. The difficulty with this algorithm at the microprogram level is that the holes created by delaying compaction can be expected to become numerous with great rapidity. This results from the wide disparity in segment lengths and the fact that the agorithm does not consider length an important factor in segment removal. Further, this method involves considerable bookeeping to maintain a list of the holes. Also, the possibility exists that a large segment will be removed but only a small segment will be placed at its location. Therefore, the free list much be searched each time a segment fault occurs to determine if the segment will "fit" in one of the existing holes. This creates overhead in the case of a large enough hole's existence, and adds to the overhead involved in the removal algorithm in any event. If this overhead is not to be incurred, then a relatively large proportion of the control store would be wasted space at a given time or relocation would occur with greater frequency. This is important because references to control store resident segments will be much more likely than segment faults if even slightly more segments can be made control store resident. In general, this approach adds undue complexity and overhead to the garbage collection process without quaranteeing significantly less overhead than the immediate compaction. This method also does not have the desired effect of eventually locating very frequently used segments where they will be less likely to be relocated.

Combination of the Algorithms

The combination of the removal and garbage collection algorithms allows the implementation of the virtualized control store. To the overhead involved in executing these algorithms should be added the overhead involved in loading the faulted segment. The total overhead involved in this process will therefore be on the order of 2500-3000 microprocessor cycles. This is considerably greater than the overhead achieved through use of the primitives. However, the use of the primitives is designed for the loading of all of the segments appearing on the association list of a segment rather than just those segments actually referenced. Therefore, if a large number of possible linkages exist, but few are expected to be used, then the overhead involved in the program controlled control store manipulation may be higher.

Interaction between the program controlled algorithm and the automatic algorithm would appear desirable in cases where the overhead involved in use of the automatic mechanism was excessive, but use of the segment was important. Providing this mechanism is not especially easy. The difficulties involve the fact that the machine program cannot know the control store configuration at the instant of interaction; therefore repeated use of the status primitive would be required previous to the program controlled manipulation. This approach also results in considerable overhead and an alternative must be sought if machine language control is to be effective. A possible approach is the use of the

define primitive to vary the priority of the desired segment. Applying a bias to the removal algorithm in this manner can be used to prevent the removal of a given segment for a fixed period of time. The program may then reestablish the segment's old priority, and normal bias for the segment will result. It is therefore possible to use the direct control primitives to reduce the otherwise potentially serious overhead involved in some instances of purely automatic control.

An Additional Consideration - The Time Factor

An additional consideration in the linking and loading process is the effect of external factors, especially real-time interupts, on the execution of the removal, garbage collection and loading algorithms. Usually, it is necesary to test for interupt conditions after the execution of machine instructions. In the event that the duration of execution is indeterminably long, then breakpoints are provided to allow testing for interupts at the required frequency.[7] A similar situation exists in the use of the linking and loading algorithms. Thus, breakpoints must be periodically provided throughout the algorithms. This could be implemented as an index whose overflow would indicate that the testing procedure should begin, or implemented with a special counter which would be periodically examined by any microprogram to determine the nesecity of temporarily suspending normal execution.

A question then arises as to the state of the control

store when the need to interupt does exist. The algorithms described make no modification of the control store except during the marking (during searching procedures) of segments to be removed, during the loading process, or during the relocation process. Unless the frequency with which the interupt condition must be tested is lower than the time required to completely move and relink a given segment, then the possibility exists that this segment will be effectively unavailable during the interupt handling process. If this segment is required as part of the interupt handler, then serious complications result. A direct solution to this problem is to require that interupt service routines use only a specific subset of the machine language instruction set, and require that the appropriate microprograms remain control store resident. These segments would be located in the low end of the control store address space and would therefore not be relocated. Modification of these segments would require specific controls to insure that interupt handling would not be necesary until such modification was complete. While this effectively limits the application of dynamic linking and loading to non-critical areas of the system, it does not limit its general applications to software and system support.

Conclusion to Chapter 4

The potential overhead and difficulties involved in providing a useful dynamic linking and loading facility have been discussed. While the overhead may appear

considerable, it should be noted that 3,000 microprocessor cycles are only about 300 microseconds or about the execution time of several floating-point instructions. An additional consideration is the potential performance increase acievable despite this overhead. An analysis of this possibility is presented in the next chapter.

Chapter 5. Justification of the Implementation of the Dynamic Linking and Loading Environment

Introduction

When an algorithm is expressed in machine language for execution on a microprogrammed computer, several inefficiencies result from the interpretation of each machine instruction by the microprocessor. Directly microprogramming the algorithm eliminates many of these inefficiencies and thereby reduces the execution time of the algorithm in proportion to the number of machine instructions executed. While this would indicate that all programs should be microprogrammed, the cost of sufficient control store memory is prohibitive of such an approach. Design of conventional microprogrammed processors therefore involves the selection of a limited set of general purpose algorithms which correspond to the machine language instructions. The general purpose nature of these algorithms is dictated by the simultaneous requirements for a general purpose computational facility and for a small control store. A general computational base may not always prove most efficient, and historically, additional provision of highly specialized algorithms, for software or system support, has realized considerable improvements in processor performance. Provision of the dynamic linking and loading mechanism at the microprocessor level, allows the specification of an extremely large number of these specialized algorithms due to the effective virtualization of the control store. The linking and

loading process involves overhead which must be compared with the expected performance increase,resultant from microprogramming a given algorithm, to determine the desirability of microprogramming (rather than machine language programming) a particular algorithm. As the overhead involved in the linking and loading processes has already been considered, it remains to establish a method for estimating the performance increase to be expected from microprogramming a given algorithm,

An Introductory Example

As a basis for discussion, the following IBM 360 assembly language program (actually its machine language equivalent) provides insight into the possibilities for increased performance through microprogramming. The program computes the address of the first occurrence of a reference byte (contained in the low order byte of RØ) in a byte string of length m (contained in R3) pointed to by R1. If a match occurs, then the address of the byte is returned in R1, otherwise -1 is returned.

```
(1)   LOOP      IC R2,Ø(R1)
(2)             CR RØ,R2
(3)             BE DONE
(4)             A R1,ONE
(5)             S R3,ONE
(6)             BNE LOOP
(7)             L R1,NEGONE
(8)   DONE     continue program
                  .
                  .
(9)   ONE       DC  X'ØØØØØØØ1'
(10)  NEGONE    DC  X'FFFFFFFF'
```

Interpretation of this program requires several phases for each instruction which are not relevant to the algorithmic

intent of the program. Specifically, each instruction requires the following stages of interpretation:

1. Initial instruction fetch and program counter updating to reflect first fetch
2. Opcode decoding and linkage to next phase
3. Instruction execution, including:
   a) Operand recognition
   b) Additional instruction fetch (possible)
   c) Operand location determination
   d) Operation execution and result storage
   e) Condition code modification
   f) Linkage to next phase
4. Test for special conditions (interupts)
5. Linkage to either step 1, or the interupt mechanism as required by step 4.

Appendix A provides an example of an emulator for a modified subset of the IBM 360 machinecode which indicates the magnitude of the overhead involved in applying these phases to the execution of specific machine instructions. Microprogramming the equivalent algorithm is considerably more efficient:

```
(1)  EA, R1(+), D(/), MMAR(+,RW), 4
(2)  EA, X'FF'(C), D(/), R1(+), 4
(3)  EDEC(0), R3(+), D(/), R3(+), 4
(4)  SUB(0), RØ(/), MDBR(/,+RW), D(/),
                         OnNeg Step, Else Return
(5)     NoOp, OnEq Step, -2
(6)  DEC(0), MMAR(+), D(/), R1(+),
                         OnCarry Repeat, Else Return
```

where this routine is expected to be called, and then return. After the initial processing required to interpret the machine instruction invoking this program, phases 1, 2, 3a, 3b, 3c, 3e, and 5 may be eliminated from the microprogrammed algorithm. Further, the frequency of occurence of phases 3d and 4 may be reduced. In this example, the constants NEGONE and ONE need not be addressed nor fetched, thereby eliminating repetivive address calculations. The equivalent

of the machine language branch instruction is either incorporated into the successor function of another microinstruction or is reduced to one or two microinstructions. The address of the character string is retained in the memory address register, thereby eliminating operand address calculations and effectively using the cache memory. Also, the superior byte manipulative capabilities of the microprocessor and the use of hardware incrementing facilities (memory address register), and the use of decrement instead of subtract (one) allow greater operational efficiency. In summary, the advantages of microprogramming in this example include:

1. Reduction of the lexical phase of the interpretive process
2. Superior constant handling ability
3. More powerful instruction sequencing
4. Superior byte handling capability
5. Availability of special operations
6. Use of internal registers (MMAR)
7. More efficient addressing and fetch of operands
8. Elimination of instruction fetches

The availability of both the machine language programmed and microprogrammed equivalents would now allow direct calculation of the savings achieved, through microprogramming, in the execution of the comparison loop, Note, that the algorithm is dominated by the six instruction loop 1-6 in the machine language program, and by the three instruction loop 3-5 in the microprogram. The microprogram equivalent of the machine language loop, ignoring the effects of testing for special conditions, is nearly 100 microinstructions. This represents several hundred

microprocessor cycles, wereas the microprogram loop generates only six microprocessor cycles. Assuming the overhead estimate of about 3000 microprocessor cycles to link and load this program as the result of a segment fault, then the loop must be executed about 10-15 times before a performance gain is achieved. This indicates that there is a strong correlation between program behavior and the desireability of microprogramming the algorithm it represents. In this case, the assumption that the byte string is a random string of text characters, and that the byte string is of length greater than 10-15 characters would indicate that the program should probably be microprogrammed. An additional consideration is the frequency with which this program is to be used within the machine language program. If this algorithm recieves frequent use, or if it is assigned an initially high priority, then it is more likely to be control store resident and consequently can be effective for even adverse circumstances of program behavior. These circumstances would, of course, result from certain input strings and reference characters. In general, the gain achievable from microprogramming is dependent upon the reduction of inefficiencies in the interpretive process, the behavior of the algorithm, and the input to the algorithm. Proceeding on the assumption that the range of inputs, and therefore the behavioral patterns, is known for the algorithm, a partial generalization of

the criteria for microprogramming may be obtained.

A Sufficient Criteria - The Reduction Algorithm

An initial estimate of the performance increase available by microprogramming an algorithm is obtainable by reducing the machine language equivalent of the algorith. This may be achieved by consideration of the microprogram generated by the machine language. The reduction algorithm eliminates the interpretive aspects of this microprogram while retaining the algorithmic intent. The initial and reduced microprograms are then compared, the program behavior is determined, and an estimate of the performance gain is obtained. Practically, the algorithm is machine language dependent and may function in either of two equivalent manners. First, it could generate the microprogram and compare it with the initial program to determine the savings. Or second, it could establish a relationship between the type of machine instruction and the savings achieved by microprogramming the algorithmic equivalent of the instruction; summing these values for the machine language program would result in a measure of the savings. Again, this savings will also depend on the program's behavior.

The Reduction Algorithm Applied to IBM 360 Machinecode

Analysis of the reduction algorithm requires a specific machine language and its microprogrammed interpreter. The modified subset of the IBM 360 machine language presented in Appendix A will serve as the basis for discussion.

Briefly, this language includes nearly all of the basic IBM 360 machine language in their original form. Special instructions, storage-to-storage instructions, decimal feature instructions and floating-point instructions are omitted. Also, multiply and divide are omitted to allow their use as examples. The remaining instructions may be classified as register-to-register (RR), register-to-storage (RX and RS) and immediate-to-storage (SI) instructions.

The RR instructions allow the smallest savings, The interpretive process involves one instruction fetch phase, one instruction decode phase, two field isolations (R1 and R2) and a condition code result. Removal of these phases saves five microprocessor cycles. Alternatively, a RR instruction may be expressed as a single instruction of the form "operation, R1, R2, R1, step", instead of the interpretive code generated by Appendix A's interpreter.

The RX instructions allow slightly greater savings in that more fields must be isolated and an additional instruction fetch phase is required. Five field isolations are required: R1, X2, B2, and two for D2. Combining with the extra instruction fetch phase for the second halfword gives a savings of nine microprocessor cycles. The same savings are true of RS instructions.

The SI instructions require only four field isolations and can use the isolation of the immediate field to allow the equivalent of the extra instruction fetch phase. This

indicates a net savings of seven microprocessor cycles.

The analysis thus far has not considered that some of these instruction forms are not fully utilized. For example, the RS shift instructions reduce to a single microinstruction of the form "shift, R1, D, R1, n times". The savings in these cases are therefore less than would be indicated by the instruction format. In particular, the shifts require the isolation of only two fields and therefore save only six microprocessor cycles. In general however, the savings indicated are correct.

<u>Application of the Reduction Algorithm to Examples</u>

Knowledge of the savings attributable to each instruction now allows a conservative estimate of the overall performance increase available. Consider the previous example. The savings involved in the loop is 9 + 5 + 9 + 9 + 9 + 9 (statements 1-6 respectively) for a total of 50 microprocessor cycles. Again, if the overhead involved in linking and loading this segment is 3000 microprocessor cycles, then the loop must be executed at least sixty times. This is a factor of ten greater than the known achievable savings because the reduction algorithm does not account for many of the potential areas for saving presented with this example (pg 5-2 to 5-4). Another example is afforded by the attempt to provide a multiply instruction. The following program provides an unsigned logical multiply of the contents of R2 and R5 with the result in the even/odd

register pair R2/R3.

```
(1)             STM  R6,R8,SAVE
(2)             L    R8,THIRTWO
(3)             LR   R3,R2
(4)             XR   R2,R2
(5)             L    R7,ONE
(6)  LOOP       SRDL R2,1
(7)             SR   R8,R7
(8)             BM   DONE
(9)             LR   R6,R3
(10)            NR   R6,R7
(11)            BZ   LOOP
(12)            AR   R2,R5
(13)            B    LOOP
(14) DONE       LM   R6,R8,SAVE
                 .
                 .
(15) SAVE       DS   3F
(16) ONE        DC   X'00000001'
(17) THIRTWO    DC   X'00000010'
```

This program has two potential loops: statements 6-11 are always executed 32 times for a net savings of:

$$32(6 + 5 + 9 + 5 + 5 + 9) = 1248,$$

additionally, the loop may include statements 12 and 13 for a gain of 14. The latter is expected sixteen times for a random distribution of numbers and hence the total savings in implementing the loop (6-13) is 1472 microinstructions. As approximately 3000 microprocessor cycles may be required to link and load this segment, it should not be micro-programmed unless its frequency of use would indicate that it will be used twice each time it is used. An attempt to microprogram the above indicates that the situation is much better than this for the reasons listed previously. In this example; all of the branch instructions can be eliminated, the bit test (instructions 9-10) can be reduced to one microinstruction, etc. Modification of the reduction

algorithm to include these factors would allow the analysis of machine language algorithms on an automatic basis. The examples however, indicate the possible magnitude of error when insufficient optimization of the microprograms occurs. Actually microprogramming the equivalent multiply results in the savings of several hundreds of microprocessor cycles per loop; and is therefore effectively microprogrammed. Until suitably precise algorithms are developed for the reduction of machine language programs to microprograms the most accurate estimates of performance increases will be achieved by actual comparison of programmer generated microprograms and programs.

Variations of the Reduction Algorithm

The reduction algorithm as stated ignored several obvious benefits of the microprogrammed expression of algorithms. An initial variation of the algorithm would attempt to fully utilize the sequencing power of the successor function to eliminate machine level branching and testing statements from the reduction process. In general, algorithms involve one or more terminal conditions and a set of initial conditions. The initial conditions may be established through the introduction of suitable constants; thereby eliminating the time required to address and fetch these operands. Similarly, the testing for terminal conditions can be incorporated in micro-instructions which also provide algorithmic computation. Additionally, multiple branch vectors are possible from

the same instruction, a facility seldom provided at the machine instruction level. In the case of the multiply example previously presented, the double right shift would cause the S-bit to be set if addition were to be required; the next instruction would perform the index on the number of shifts and test for the necesity of addition; and the subsequent instruction could either add or shift (depending on previous successor evaluation) and simultaneously test for the terminal condition. This overlap of condition testing, branching and algorithmic execution is an important reason microprogramming is so much more effective than the purely sequential operations allowed by typical machine languages.

Another factor which should be included in the reduction algorithm is the elimination of a large number of operand fetches and operand address calculations. This also includes the fact that instruction fetches have been eliminated. Elimination of instruction fetches allows at least two areas of performance gain. First, the actual time lost in waiting for the instruction to be fetched is regained. Second, multiple references to the same address or to consequetive addresses are no longer interspersed with instruction fetches. This allows more effective use of the cache memory because main memory accesses are not as frequently widespread. Note that addressing consequetive locations within the cache allows effectively zero access time, wereas alternately referencing instructions and their

operands results in access times approaching those of main memory modules. In the case of microprogrammed parsers or similar string manipulative algorithms, the time saved by elimination of instruction fetches is greater than the effect of reducing the algorithm to the microprogram level. If the entirety of this type of algorithm is microprogrammed, then the process of byte by byte treatment of the string also results in much decreased operand access time. Note further that the design of a special system of cache memories to attempt to provide a similar effect would be extremely difficult, except for a limited number of special cases. Extension of the reduction algorithm by consideration of the effects of referencing consequetive locations or the storage of frequently used pointers and operands could greatly improve its accuracy.

An example of the above effect is provided by the SUBSTR function which attempts to locate the first occurence of a string within another string, all such occurences, or the number of such occurences. Microprogrammed, this algorithm can be expressed consicely and efficiently by storing the substring in local store and consequetively referencing the bytes of the string. At the machine level, the number of references for bytes is the same but each is alternated with the fetching and execution of several machine instructions. Therefore, the machine language program proceeds at main memory

speeds, while the microprogrammed algorith proceeds at control store speeds. Additionally, the microprogram algorithmic expression is more efficient because the data structure is designed to enable rapid manipulation of byte operands. The frequency of use of functional variants of SUBSTR in parsers, editors, compilers and similar algorithms also indicates that the desired microprogram will be frequently control store resident; in which case no linking and loading overhead will be required and substantial performance gain will be achieved.

The reduction algorithm may also be modified to consider the relation between the microprocessor cycle time, the effective main memory access time and the resultant overlap between instruction fetch and execution. For cases in which main memory access time is significantly greater than the microprocessor cycle time the microprogram can be expected to obtain nearly one-hundred percent overlap between main memory references and instruction execution. In these cases, the microprograming of the algorithm will provide performance increases based on the memory access time and total number of nonsequential references, rather than on the savings in microproccessor cycles.

Another Reason for Dynamic Linking and Loading

The preceeding analysis indicates that microprogramming allows substantial performance increases but does not fully indicate the savings achieved through the dynamic linking

and loading. Conventional systems require either that the machine program control the loading of a microprogram or provide no facility for dynamically altering the control store. The lack of any such facility limits the number of microprogrammed algorithms to either the size allowed by the control store addressing mechanism or the size of control store the user can financially afford. A dynamic facility of equal size control store can perform at least as well; and additionally allows the definition of a large number of microprograms which could not be incorporated in the fixed control store, thus providing additional performance increases in these cases. In comparison to systems which allow limited machine language control of the loading process the dynamic facility is superior for at least two reasons. First, the automatic mechanism frees the user from this requirement and allows both program and microprogram revision independently. Second, conventional loading schemes are based on absolute addressing and provide little if any means of linking to already existing programs. This results in severe problems when the user desires two microprograms which happen to have been originally located at the same location. Also, the inability to share a variety of functions requires that many of these functions be duplicated for use by the special microcode associated with the interpretation of a particular machine level instruction. An easily concieved application of the

sharing mechanism would be the use of several elementary arithmetic functions by a more complicated algorithm. The number of such functions which could require duplication would probably result in conventional implementations of a few extra machine instructions, rather than microprogramming the entire algorithm. In general, the dynamic mechanism allows not only more convient modification of the control store but also provides a more efficient means for doing so. In this respect, it is generally more useful than currently available systems.

Conclusion to Chapter 5.

The advantages provided by a dynamic linking and loading facility are considerable. Not only are the performance increases normally associated with microprogramming possible, but also the number and flexibility with which these algorithms may be microprogrammed is enhanced. An attempt has been made to indicate that these advantages generally outweigh the overhead involved in implementation of the dynamic linking and loading mechanism. This is especially true of microprograms which are either inherently repetitive (iterative or recursive) or are executed with great frequency. In the latter case, the microprograms are more likely to remain continuously control store resident, while lesser used programs will be removed (see Chapter 4) thereby resulting in even greater performance gains. As most commonly used algorithms satisfy one of these alternatives, the

implementation of a dynamic linking and loading facility would seem fully justified.

Chapter 6. Implications of Dynamic Linking and Loading of Microprograms

The success of dynamic modification of the computational base of the processor in improving system performance is partially upset by the specificity of its application. An essential attack has been made against the trend towards higher-level languages and the general compatibility of software and systems. Microprogramming creates a tradeoff between this general compatibility and possible improvements in the performance of the system.

Compatibility of Systems

Provision of a specialized instruction set at each computer facility creates severe problems for the system designer. Further allowing this instruction set to vary in a dynamic fashion may cause unsurmountable problems. Either system software must have an established base which is supported by all systems; in which case the system does not fully utilize the available performance increases. Or, the perogative for use of the dynamic linking and loading facility must be largely restricted to the system. The user then benefits from the system provided special functions while he is restricted from providing his own.

Another consideration is that the dynamic linking and loading facility in fact promotes compatibility by allowing the use of a large variety of emulators, or allowing sharing of special instruction sets between computer installations. Compatibility is enhanced because a large variety of different systems can now be simultaneously

provided, wereas previously the control store size required would have been prohibitive.

Compilers, Interpreters, and Higher-Level Languages

Similar tradeoffs are apparent in the use of the dynamic linking and loading facility for the support of higher level languages or even assemblers. A significant question is posed by the availability of two levels of programming, and the possibility that one of these levels provides a varying instruction set. Compilers would need some knowledge of the instruction set capabilities in order to function, and modification of this instruction set would require the redesign of traditionally structured compilers. Another problem is posed by the need to establish the proper level of programming an interpreter; which code should pass through two levels of interpretation and which code should be directly executed. Development of an algorithm, similar in intent to the reduction algorithm, which could automatically establish the proper level of programming is essential if problems of this type are to be solved.

Conclusion to Chapter 6

In general, the implementation of a dynamic linking and loading mechanism will give rise to many issues which are not directly related to the performance of the system. The issues of who, what and how much should be carefully considered before extending the microprogram facility.

Chapter 7. Conclusion

It has been shown that the implementation of a dynamic linking and loading facility is both feasible and useful means of increasing processor performance. The general usefulness of microprogramming has been established by many authors. Extension of the control store, in a virtual sense, has also been shown useful for the variety of cases in which the possible performance increase exceeds the overhead involved in use of the facility. Specifically, the facility allows the microprogramming of long iterative or recursive functions which otherwise could not have been incorporated in the control store. Provision of this mechanism also creates several new difficulties in the areas of compatibility and support of higher level languages. These difficulties would imply the need for intelligent systems which could adapt to a varying environment. Another area of concern is the means of providing interpreters or compilers with both the knowledge of the existence of new instructions and the usefulness of these instructions in compilation or interpretation. These problems are largely analogous to those involved in providing the same software for two different machines.

The eventual usefulness of dynamic linking and loading will depend upon further study of the behavioral aspects of microprograms in a dynamic environment and the validity of the various algorithms presented. A rigorous means of

establishing the expected performance increases resultant from microprogramming a given algorithm must also be determined. Further studies in these areas could allow automatic computation of desired algorithms.

Another potentially critical area is the decreasing cost of extremely high speed memory and the availability of inexpensive processors on a single integrated circuit. Future developments in these areas could make dynamic microprogramming obsolete in the sense that no microprogramming would be required; or that custom design of a processor would be cheaper than microprogram development. In any event, dynamic microprogramming provides a degree of efficient flexibility in processor specification.

BIBLIOGRAPHY

List of Works Cited:

1. Husson, Samir S. Microprogramming Principles and Practices. Englewood Cliffs, N.J.: Prentice-Hall, 1970.

2. Hassit, A., et al. "Implementation of a High Level Language Machine," Communications of the ACM, April 1973, 199-212.

3. Liskov, B. H., "The Design of the Venus Operating System," Communications of the ACM, March 1972, 144-149.

4. Ramamoorthy, C. V. & Tsuchiya, M. "A Study of User-Microprogrammable Computers," AFIPS Conference Proceedings (SJCC 1970), 165-181.

5. Bienhoff, Milton G. editor. "New Products", Computer (IEEE Computer Society), May 1973, 32-33.

6. Reigel, E.W. et al. "The Interpreter-A Microprogrammable Building Block System," AFIPS Conference Proceedings (SJCC 1972), 705-723.

7. Rosin, Robert F. et al. "An Environment for Research in Microprogramming and Emulation," Communications of the ACM, August 1972, 748-760.

8. Hewlett-Packard Co. Microprogramming Guide for Hewlett-Packard Model 2100 Computer. Feb. 1972.

9. IBM Systems Reference Library. IBM/360 Principles of Operation. Sept. 1968.

List of Works Consulted:

10. Chu, Yaohan. Computer Organization and Microprogramming Englewood Cliffs, N.J.: Prentice-Hall, 1972.

11. Cook, Robert W. & Flynn, Michael J. "System Design of a Dynamic Microprocessor," IEEE Transactions on Computers, March 1970, 213-222.

12. SIG Micro Newsletter (ACM publication) July 1972. [Includes extensive bibliography of field.]

Appendix A. Microprogramming the Hypothetical Machine

Symbolic Notation

The binary representation of the microinstruction format (Figure 3, pg. 14) is inconvient for expression of microprograms. A more readable approach is the choice of symbolic equivalents for the respective fields of each microinstruction. While the scheme to be used was partially discussed in Chapter 2, a complete summary of possible formats and abreviations for registers follows. The basic format of expression is:

ALU, Shift, A-field, B-field, F-field, Successor;.

The possible symbolic representations for the ALU field are:

INC(C), add C to A-operand on first execution of instruction, if instruction is repeated add resultant Carry out instead. Initial C designation may be 0, 1, C, $\overline{C}$.

DEC(C), as in INC, only subtract Carry.

ADD(C), add C to sum of A and B operands on first execution of instruction, if repeated add resultant Carry out instead. Initial C designation may be 0, 1, C, $\overline{C}$.

SUB(C), as in ADD, only function is A-operand minus B-operand minus Carry

A, $\overline{A}$, B, $\overline{B}$, 0, 1, A+B, $\overline{A}$+B, A+$\overline{B}$, $\overline{A+B}$, A⊕B, $\overline{A \oplus B}$, AB, $\overline{A}$B, A$\overline{B}$, $\overline{AB}$, performs specified boolean function on the A and B operands. The notation $\overline{X}$ means not X.

Additionally prefacing any of the above with an E indicates an extended precision operation and consequently the interpretation of the successor function is modified. If no ALU function is specified then the logical function A is assumed.

The possible symbolic representations for the shift field are of the form X(S,Z). The "X" indicates the direction of shift: LS indicates left-shift, RS indicates right shift. The S field indicates the bit designation to be shifted into the bit positions vacated by the shift. If the instruction is immediately repeated, then the Shift out bit is used for successive shifts if and only if the Z-field indicates a shift of one position. Otherwise, the intial shift designation is reapplied. Initial values for S are 0, 1, S, $\overline{S}$. The Z-field indicates the number of bit positions to shift; a number between 0 and 7. Note that shifting 0 positions is a "no-shift" and may be indicated by either replacing the entire shift field representation by the code NS, or by omitting the field entirely.

The symbolic notations for each of the A, B and F operand fields are equivalent. Either the form X(Y,Z) or X(Z) are permitted. The X field specifies the contents of the eight bit field, the Y field (either M or omitted) indicates masking of the eight bit field, and the Z field specifies modification of the field in the event the instruction is immediately repeated. Possible Z field designations are:

- /, no modification (These indicate
- +, increment the field interpretation
- -, decrement the field as a bus address)

- C, do not modify the field and interpret it as an eight bit constant.

These designations are primarily established to allow

either extended precision or byte string manipulative algorithms to be expressed in a single microinstruction.

The possible interpretations of the successor field have been described in detail in Chapters 2 and 3; therefore, only a brief summary will be presented. The form "on condition c1, do x, else on condition c2, do y, else Step", has several abbreviations. Any unconditional specification is interpreted simply as that specification. For example, if c2 were unconditional, then rather than that "on any condition do y" simply state "do y". The abreviations used for conditions are usually readily interpreted, but for special cases the meaning will be clarified. In particular however, the notation $\not{y}$ indicates "not on condition y". The possible variations for the "do" fields are summarized in Figure 3, page 14.

Specification of the E bit in the ALU field designation indicates that the successor field should be expressed as "on condition c1, do x, else n". The interpretation of the condition field is as described above; the "n" field is either simply a number indicating the number of times to repeat the instruction (maximum number) before Stepping. The "n" field may also be expressed as Mn where the prefix indicates masking of the eight bit field with the X-register. The interpretation of the successor function designation "on condition c1, do x, else n" without the use of the E-bit is similar, except that the n field is now a two's

complement integer which indicates the desired offset from the present value of the program location counter. The offset form is easily recognized by the symbolic convention that +n or -n are offsets, wereas n is a count (E-bit set) or an absolute address (in the use of the linking mechanism, an absolute offset from the indicated segment's starting address).

The absolute designation of the successor function, "go to n", is specified simply as the address n. When using the linkage mechanism, use of a number in this fashion implies the absolute offset form wereas use of a symbolic label indicates the absolute segment form (the label is the segment's designation).

To clarify the preceeding discussion, consider the following examples.

Example 1. EINC(1), R5(+), D(/), R5(+), 4

This instruction indicates that the contents of the 4 consequetive bytes starting at location R5 are to be incremented as a single operand. As no conditional branch is specified, the default successor will be to Step after the fourth repetition of the instruction. The mask registers are not used. Explanations for the R5 and D specifications are given in Table 1. Here, the D indicates a dummy or null field (not used) wereas the R5 indicates a general location in the local store scratchpad. No control lines are encoded in any of the bus address fields.

Example 2. EINC(1), R5(+), D(/), R5(+), On C=0 Step, 4

This instruction will be equivalent to that of the previous example in that the same operation will be performed on the same operand. This form however is much more efficient. If the carry resultant from execution of the instruction is zero, then the operand has been effectively incremented and the instruction need not be repeated. This feature is provided by use of the conditional specification "On C=0 Step". Note that if C never becomes zero, then the instruction will be repeated at most four times as indicated by the count designation.

Example 3. $A\overline{B}$, MDBR(/,+RW), X'45'(C), D(/), On Zero Step, Repeat

This example indicates several features of the notation. The overall intent of the instruction is to examine the byte string pointed to by the memory address register until the first occurence of the byte X'45' is located (X indicates hexidecimal representation of the constant). Examination of the instruction indicates that the "equals" condition will be determined by the logic function $A\overline{B}$ on the memory data buffer register and the constant field specified in the instruction. The code +RW indicates that the control specification for memory reference to the next byte location is encoded in the MDBR address (see Table 1). The D specification in the F-field indicates that no result is stored; therefore, the only effects of this instruction will be to modify the memory address register and the condition codes.

```
Example 4.  EA, PC(+,+2), D(/), MMAR(+, RW), 4
            A, MDBR(/,+RW), D(/), XM(/)
            AB, MDBR(/,), X'0F'(C), BM(/)
            AB, RS(0,4), MDBR(/,), X'F0'(C), AM(/)
            A, AM(/), MMAR(/,+RW), FM(/), +M
```

As a final example, the above short program serves as introduction to the programming of a machine language processor. Briefly assume that the machine language consists solely of IBM 360 RR-type instructions. The first line of code moves the program counter into the memory address register, starts the instruction fetch, and encodes the program counter control field to indicate that the program counter should be incremented by two. The next instruction moves the first byte fetched into the X-register (i.e. the instruction opcode is now in the X-register). The third instruction takes the next byte fetched, ANDs it with a constant to isolate the R2 field of the instruction, and stores the result in the B-register. The next instruction uses the same byte from the data buffer register, isolates the R1 field by ANDing with a constant and shifting the result right four positions (zeroing the four leftmost bits of the result), and stores the result in the A-register. The final instruction causes the next byte to be fetched from main memory, moves the A-register contents into the F-register and does an offset branch masked by the X-register. This type of branch theregore allows rapid decoding of the opcode field. Note that for several instructions the successor function is not specified

and the default Step successor is chosen. Note also, the use of the B-field in the last instruction despite the fact that no B-operand is used by the operation. In this case, the field is used solely to specify a particular control signal. While this same operation could have been encoded in the prior instruction with greater efficiency, the intent of this example is more illustrative than practically suggestive.

Microprogramming a Machine Instruction Interpreter

Perhaps the best way to gain insight into the relationship between microprograms and machine-level programs is to construct a basic microprogram interpreter for the machine language. To avoid the difficulties involved in specifying a new machine language, a subset of the IBM 360 machine language was chosen. The interpretation of RR, RX, RS and SI instructions is presented in a simplified fashion, but illustrates the basic principles involved.

To provide program readability, the symbolic format for microinstructions is used. Additionally, the machine language instructions will be expressed in terms of their assembly language equivalents and the segments associated with a specific machine instruction will be labeled with the symbolic opcode for that instruction. Other conventions will be discussed as they arise. For a description of the IBM 360 machine code, the reader is referred to the IBM Systems Reference Library publication, IBM System/360 Principles of Operation.