14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pennsylvania 15213

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

The Design and Construction of Flexible and Efficient Interactive Programming Systems

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)
James G. Mitchell

6. REPORT DATE
June, 1970

7a. TOTAL NO. OF PAGES
275

7b. NO. OF REFS
54

8a. CONTRACT OR GRANT NO.
F 44620-70-C-0107

b. PROJECT NO.
A0827-5

c.
61101D

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES
TECH, OTHER

12. SPONSORING MILITARY ACTIVITY
Air Force Office of Scientific Research
1400 Wilson Blvd. (SRMA)
Arlington, Va. 22209

13. ABSTRACT

This dissertation treats interactive programming systems at three levels:

(1) the design of an interactive programming system (IPS) from considerations of the psychological and intellectual needs of the user;
(2) the implementation of an IPS which is both flexible for the human and efficient with computer usage; and
(3) the construction of an IPS by bootstrapping.

The bulk of the work centers on the second point. In particular, a method is developed for building an interpreter for a language (to achieve flexibility) which also yields a compiler for that language as a (relatively cheap) byproduct. Programs "become" compiled in the process of being interpreted, but are able to respond to changes in the interactive environment as if they were being interpreted only.

With such an interpreter/compiler, bootstrapping becomes a visible means of building an IPS. This is especially valuable in making the interfaces between the human and the system easily malleable to suit the user's needs and abilities.

DD FORM 1 NOV 68 1473

Security Classification

[Sk 68] Sklansky, J., Finkelstein, M. and Russell, E. C. A Formalism for Program Translation, J.ACM 15,2 (Apr 1968) 165-175.

[St 67] Standish, T. A. A Data Definition Facility for Programming Languages, (thesis), Carnegie-Mellon Univ., Dept. of Computer Science, May, 1967.

[St 69] Standish, T. A. An Essay on APL, Carnegie-Mellon Univ., Dept. of Computer Science, March, 1969.

[SC 69] IC-9000 System Summary, Standard Computer Corporation, 1969.

[VW 69] Van Wijngaarden, A. (ed.), Mailloux, B. J., Peck, J. E. L. and Koster, C. H. A. Report on the Algorithmic Language ALGOL 68, Numerische Mathematik 14:79-218 (1969).

[VZ 69] Van Zoeren, H. R. LCC Reference Manual, Carnegie-Mellon Univ., Dept. of Computer Science, Nov., 1969.

[Wai 69] Waite, W. M. Base for a Mobile Programming System, C.ACM 12,9 (Sept., 1969).

[WM 69] Wulf, W. A. and Mitchell, J. G. Data, Structures and Declaration in WREC, Westinghouse Research Labs. Memo, Pittsburgh, Pa., 1969.

[Wu 70] Wulf, W. A. A Note on Parsing WREC-like Languages, Westinghouse Research Labs. memo, Pittsburgh, Pa., Jan., 1970.

[Le 69] Lett, A. S. and Konigsford, W. L. TSS/360: A Time-Shared Operating System, AFIPS 1968 FJCC Proc., Part 1:15-18.

[Lo 65] Lock, K. L. Structuring Programs for Multiprogramming Time-Sharing On-line Applications, AFIPS 1965 FJCC Proc., 457-472.

[LM 69] Lankford, A. and Mullins, W. Jr. An Introduction to LCC, Carnegie-Mellon Univ., Dept. of Computer Science, Sept., 1969.

[LS 70] Lampson, B. W. and Simonyi, C. Model I CPU Reference Manual, Berkeley Computer Corporation, Feb., 1970.

[Mi 68] Miller, R. B. Response Times in Man-Computer Conversational Transactions, AFIPS 1968 FJCC Proc., Part 1:267-277.

[Mit 69] Mitchell, J. G. Memo for Digital Equipment Corporation on some implementation problems and their solutions for a fast Fortran compiler with error-checking at run-time, July, 1969.

[MPV 68] Mitchell, J. G., Perlis, A. J. and Van Zoeren, H. V. LC^2: A Language for Conversational Computing, in Interactive Systems for Experimental Applied Mathematics, 1969.

[NB 69] Naur, P. and Russell, B. (Eds.) Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, Oct., 1968.

[Pe 69] Perlis, A. J. Introduction to Extensible Languages, SIGPLAN Notices 4,8 (Aug 1969) 3-5.

[Ros 67] Ross, D. T. The AED Free Storage Package, C.ACM 10,8 (Aug 1967) 481-492.

[RR 64] Randell, B. and Russell, L. J. Algol 60 Implementation, Academic Press, 1964.

[Sa 66] Saltzer, J. H. Traffic Control in a Multiplexed Computer System, MAC-TR-30 (thesis), Mass. Institute of Technology, Cambridge, Mass., Nov., 1966.

[Sh 64] Shaw, J. C. JOSS: A Designer's View of an Experimental On-line Computing System, AFIPS 1964 FJCC Proc.

[Si 66] Simon, H. A. Reflections on Time-Sharing from a User's Point of View, in Computer Science Research Review, Carnegie-Mellon Univ., Dept. of Computer Science, 1966: 43-51.

[Si 69] Simon, H. A. Lecture given at Carnegie-Mellon Univ. on the Social Implications of Computers.

[EER 68] Engelbart, D. C., English, W. K., and Rulifson, J. F. Development of a Multidisplay, Time-Shared Computer-Facility and Computer-Augmented Management Research, Stanford Research Institute Report, April, 1968.

[Fe 64] Feldman, J. A. A Formal Semantics for Computer Oriented Languages, (thesis), Carnegie-Mellon Univ., May, 1964.

[Fi 70] Fisher, D. Control Structures for Programming Languages, (thesis) Carnegie-Mellon Univ., Dept. of Computer Science, May, 1970.

[Go 67] Gold, M. M. A Methodology for Evaluating Time-Shared Computer System Usage, Carnegie-Mellon Univ., Dept. of Computer Science, Aug., 1967.

[Ge 70] Gentleman, M. private discussion on the ALTRAN method of bootstrapping and exporting systems.

[Il 68] Iliffe, J. K. Basic Machine Principles, American Elsevier Publishing Co., New York, 1968.

[Il 69] Iliffe, J. K. Elements of BLM, The Computer Journal 12,3 (Aug 1969) 251-258.

[Ha 69] RC-4000 Software Multiprogramming System, Per Brinch Hansen (Ed.), A/S Regnecentralen, Copenhagen, Denmark, April, 1969.

[IBM 69] Command System User's Guide, Form C28-2001, IBM Corporation, TSS/360 Programming Publication, Yorktown Heights, N.Y., 1969.

[IF 68] Iverson, K. E. and Falkoff; A. D. APL\360: User's Manual, IBM Corporation, 1968.

[Ka 69] Kay, A. The Reactive Engine, (thesis) The Univ. of Utah, Computer Science Dept., 1969.

[Kn 69] Knuth, D. E. The Art of Computer Programming, Volume I: Fundamental Algorithms, Addison-Wesley, 1969.

[Kr 69] Krutar, R. A. To Catch a Thief, preliminary paper on coroutine-like processes, Carnegie-Mellon Univ., Dept. of Computer Science, 1969.

[KK 66] Kemeny, J. G. and Kurtz, T. E. BASIC - A Manual for BASIC, the Elementary Algebraic Language Designed for Use with the Dartmouth Time-Sharing System (third edition), Dartmouth College, Jan., 1966.

[La 69] Lampson, B. W. Dynamic Protection Structures, AFIPS 1969 FJCC Proc.: 27-38.

BIBLIOGRAPHY

[Ab 70] Abrams, P. S. An APL Machine, Stanford Linear Accelerator Center, Feb., 1970.

[Ba 66] Baker, C. L. JOSS: Introduction to a Helpful Assistant, Memo RM-5058-PR, The RAND Corporation, July, 1966.

[Ba 67] Baker, C. L. JOSS: Rubrics, report P-3560, The RAND Corporation, March, 1967.

[Bal 69] Balzer, R. M. EXDAMS: Extendible Debugging and Monitoring System, AFIPS 1969 SJCC Proc.

[BCC 70] An Overview of the BCC Model 500 Time-Sharing System, Berkeley Computer Corporation, Feb., 1970.

[BL 68] Breed, L. M. and Lathwell, R. H. The Implementation of APL\360, in Interactive Systems for Experimental Applied Mathematics, M. Klerer and J. Reinfelds (Eds.), Academic Press, 1968: 390-399.

[BM 62] Brooker, R. A. and Morris, D. A General Translator Program for Phrase Structure Languages, J.ACM 9,1 (Jan 1962) p. 1.

[Ch 67] Cheatham, T. E. Jr. On the Basis for ELF - an Extensible Language Facility, Mass. Computer Associates Inc. report, 1967.

[Chr 68] Christensen, C. An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language, in Interactive Systems for Experimental Applied Mathematics.

[Cl 68] Clarke, Arthur C. 2001: A Space Odyssey, The New American Library, Inc., 1968.

[Co 63] Conway, M. E. Design of a Separable Transition-Diagram Compiler, C.ACM 6,3: 396 ff.

[Di 68] Dijkstra, E. W. The Structure of the "THE" Multiprogramming System, C.ACM 11,5 (May 1968) 341-346.

[DEC 69] PDP-11 Handbook, The Digital Equipment Corporation, 1969.

[DSC 69] The Digital Scientific META 4 Series 16 Computer System, preliminary system manual, Digital Scientific Corporation, Nov., 1969.

[Ea 69] Earley, J. C. VERS - An Extendible Language with an Implementation Facility (rough draft), Univ. of California (Berkeley), Computer Science Dept., April, 1969.

and a port name. The portnames are looked up in the designated processes' name tables and the ports then initialized so that they are linked together.

The initial activation of a new process can proceed by a CALL from the creating process, with parameters being passed to the new subprocess. Control goes to the main program in the called process. Copies of already existing processes can be simply activated using ports which they had previously.

This design is not meant as a final scheme for the CBM. In particular, there is no good reason why processes may not run in parallel if possible. The RC-4000 Multiprogramming System [Ha 69], for instance, has a number of features which are easy generalizations of the process concept which we have outlined here and earlier in chapter 2.

another, all uses of it will be updated since they access it only indirectly via some process table entry.

Coroutine control between processes is done with an ACTIVATE S,D instruction, where S is an accessor which refers to a primitive hetero-set called a port, after Krutar [Kr 69]. A port contains control and status information much like an MSCW, an accessor for D (put there by the ACTIVATE instruction) and a process accessor for a port in the process to which the S-port is connected.

The process activated then has access, through its port, to D and can later cause control to flow back across the port connection by itself using an ACTIVATE instruction. When an ACTIVATE occurs, the process in control is stopped; when later reACTIVATED, it restarts as if the suspension had not happened (except, of course, that D may have been changed).

Two actions are needed to establish a configuration of cooperating processes: CREATE and JOIN.

CREATE S,D creates a copy of process S and gives it the name specified by D. If S is defaulted, a special, virgin process is created with name D. An accessor for the created process is placed on the creator's stack; he can then access the name table of the new process, etc.

JOIN S,D is the means by which ports in separate processes are linked together. S and D are accessors for primitive hetero-sets called portnames, each consisting of two parts: a process identifier

A18 Miscellaneous Operations

MAKE S,D MAKE is the transfer-function operation and attempts to make the S-operand to be of type D (D is either a simple type number of an accessor for an HSD); a reference to the transformed S-value is placed on the stack as result. When S points to a PAC, this is part of the allocation procedure for a set of type D.

HDFLD S,D S is a number corresponding to a field in a set header (e.g., the MON-bit is field 0, the TYPE field is field 1, etc.). The S^{th} field from the set header of the set specified by D is placed on the stack as a binary word.

POP S The top S items are popped from the stack. MSCWs cannot be popped, and attempting to do so will cause an interrupt.

PUSH S,D S copies of D are pushed onto the stack. This is useful for initializing local variables for a routine, etc.

A19 Inter-Process Control and Communication

In order to allow independent processes to communicate data and control, a process accessor (type 14) is needed which will allow one process to access objects in another. A process accessor has two parts: a process identification part and a non-process accessor. The process identification part is left unspecified here; it may be a unique integer or a simple pointer into a process table containing more complex identification involving external storage addresses, and so on. Thus, if a process is moved from one type of storage to

the operands. Any transfer functions which may be needed to change the operands can be obtained using the MAKE operation (which is described below].

The procedure control, relative jump and sequencing instructions are handled by one more routine called the control routine. The storage allocation operations are handled by one routine, and the programmed extension for MAKE is handled by a single routine. Each of these routines is called with a parameter similar to those supplied to the routine which simulates the ALM class of instructions.

Altogether, then, each extended data structure requires four routines: one for arithmetic and data movement; one for procedure, jumping and sequencing control; one for storage allocation and reclamation; and one to handle transfer functions. A list of the operation classes, with this partitioning, follows:

CBM INSTRUCTIONS

ALM Class	Control Class	Storage Class
ADD	SQJL	SPLIT
SUB	BSQJL	UNMAKE
NEG	JMP	UNMAKEJ
DIV	JMPB	
MUL	CALL	
AND	RETURN	Transfer-Function
OR	EXIT	MAKE
XOR	EXITOR	
NOT	FCALL	
COPY		
MOVE		
HDFLD		
SETIX		
POP		
PUSH		

Figure A17-1: List of CBM Instructio

three types of jump, each of which may be conditional, depending on the setting of the CBM condition code. They are

(1) JMP α,β,κ Jump Forward (indirect): the address α is taken as a pointer to a code position pair [FIRST:LAST] and used for the jump by adding α.LAST to β and the current value of the CBM Program Pointer,PRP; κ is the condition code to be checked for a conditional jump;

(2) JMP ,β,κ Jump Forward: since α is defaulted, this instruction is a direct relative jump to PRP+β ; as before, β must be an integer literal, and κ is a condition setting;

(3) JMPB α,β,κ Jump Backward (Indirect): this is the same as (1) except that the point jumped to is PRP-α.FIRST+β.

Jump Backward (direct) is not necessary because it is the same as (2), since β may be positive or negative.

A17 Operators Applied to Extended Data Structures

In the HSD for a class of extended data structures is a set of program accessors. The procedures to which they refer act as programmed extensions of the CBM operations.

One routine simulates all the ALM class of instructions. It is activated with a set consisting of the skeleton of the instruction to be simulated and its operands as a parameter. The CBM will have replaced any implicit stack operands by explicit stack accessors for

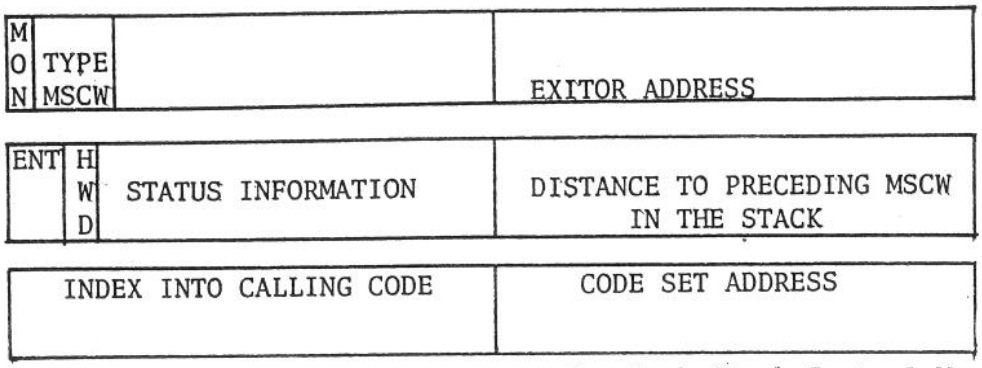

Figure A16-1: Format of a Mark Stack Control Word

There are three possible ways of invoking a routine:

(a) a function call can occur in the course of obtaining an operand value (Auto/Execute) or by a FCALL instruction;

(b) the CALL instruction can be used;

(c) an interrupt can cause routine activation, and the called routine can view its activation as resulting from a call; thus interrupt routines can be initiated by either hardware or software.

In the diagram of an MSCW (Figure A16-1 above), ENT describes the type of entry ((a), (b), or (c) above) and HWD tells which half of the word specified by the indexed program accessor contains the next instruction to be executed. Note that both the EXITOR address and the program address have no types associated with them since they can only be code sequences (because of the way they are created).

A17 Relative Jumps

One last simple control mechanism is the ability to perform the relative jumps required in the TFI mechanisms. Basically, there are

and are accessed relative to the MSCW, using a relative stack address. Thus, the parameter cell is item 1 above the MSCW and the first cell for a local variable is item 2, and so on. The allocation of local variables is simply a matter of using the PUSH operator to make space for as many cells as necessary.

The CALL operator, then, has two arguments: the first specifies the routine to be called; and the second specifies the parameter for the call. The form of the instruction is

CALL routine,parameter

If the routine address is omitted, the top of the stack is used. If the parameter is omitted, an "undefined" element is passed to the called routine.

Mark Stack Control Words

The main mechanism for implementing calls, returns, and exits is a data type called a Mark Stack Control Word (MSCW), an atomic element which can only exist on a stack. The MSCW must contain the following information:

(i) status information of the calling routine,

(ii) a link to the MSCW for the routine which called the current routine,

(iii) the program counter to be used to restart the current routine once the called routine returns control,

(iv) the address of the exitor routine (if any) for the current routine,

(v) some indication of the manner in which the "call" is being made.

user interrupts occurred. To summarize, the action which occurs when some routine relinquishes control via an EXIT α is the following:

(1) most of the action of a return (e.g., picking up the MSCW which describes the call) is performed, except that control is given directly to the exit routine for the routine which initially performed the call;

(2) the MSCW is unlinked from the stack just as if a normal return had occurred, and the exit-condition is placed on the stack for possible use by the exit routine; if the exit routine should itself execute an EXIT or RETURN, it is speaking for its host.

If no exitor has been appointed by the time a call is performed, then any exit will cause an automatic exit from that routine with the original exit-condition being passed back until some routine in the call hierarchy is encountered which has an EXITOR. If none such exists an interrupt will occur. Exitors can be deleted by executing an EXITOR command with no parameter.

One also needs to be able to pass parameters to a routine when calling it, and provide for local variables, residing on the stack, for the called routine. First, only one parameter is allowed to be passed with a call, and that parameter will always be put on the stack immediately above the MSCW for the call by the CBM. Of course, since that parameter may specify an entire set, there is no restriction inherent in the single parameter constraint. If no parameter is passed, that stack entry is made to be undefined (type 0).

Local variables sit on the stack above the parameter cell,

then an interrupt occurs; the IPS can then recover from such a problem by using the REVERT algorithm in section 4C3C and re-create interpretive control to handle the problem. In order to accomplish this, each code sequence (which is a primitive set in the CBM) must have a validity bit in its header for that purpose. But then in order to check for validity after a call, the CBM must be able to reference the header of the active code sequence as well as the next instruction to be executed in that sequence. Maintaining an indexed form of accessor as the program counter for the CBM solves this problem because indexed accessors always point at the head of the set referenced, using the index field to select one item in that set.

As outlined above, we also wish to handle the counterpart of a subroutine call, namely an EXIT with an abnormal condition. To accomplish this there is a command

EXITOR α

which supplies a program address α to be used whenever an exit condition arises. An EXITOR can be specified for every call, but can also hold for some longer time such as the entire activation of the particular routine which is its "host". When a routine is called, there is no exitor for it until one is specified by this command. Whenever an exit-condition is given as the result of a call, the exitor is given control; it can then inspect the exit-condition; perform cleanup actions for its host routine; and either propagate the exit-condition backward, by itself executing an EXIT, or choose to continue execution of its host routine. This method was heavily used in the LC^2 system [MPV 68] and proved of great value in "unwinding" control when the user requested it or when errors or

An MSCW contains enough relevant information to implement CALL, RETURN and EXIT; the exact form of an MSCW will be discussed after the CALL mechanism is described.

In order to implement the TFI, the act of returning from a subroutine must check that the code to which control is being returned is valid. If it is not,

the three because of the number of storage accesses needed. The second alternative is closely related in type to the third, but has possible savings in virtual memory systems because of the relative closeness of the cells in the HSD. The most efficient of the group is the first, since it allows the use of a program which can do much more powerful things -- such as automatic extension of a set -- and can be more efficient.

A16 Procedure Call and Return Mechanisms

Any sequence of CBM code may be treated as a subroutine. Since instruction sequences also have limits associated with them, attempting to execute past the last instruction of such a set will cause a return to the calling routine. A code sequence does or does not return a value (when terminated in this way) depending on whether or not it leaves a value on the stack when it terminates.

Returns can also be done explicitly, using the RETURN operation:

RETURN A

If A is present, it is the value to be returned, with Auto/Fetch being applied if necessary. To return with a return-code, indicating an exceptional condition, the operation

EXIT A

is available. A is the return-code; if omitted, it is taken to be the item on the top of the stack.

A CALL pushes an item called a Mark Stack Control Word (MSCW, type 13) onto the stack, which MSCW is linked to the previous one.

references can be used primarily for linking between sets. While not a necessary constraint, this division of uses for accessors seem right, in that one ought to index into indexable structures and link or point into non-indexable structures.

Indexing into a hetero-set results in an indexed accessor of the same form as those for homo-sets; the action associated with the indexing is not, however, as simply described. More than one method is available in order that there be some flexibility in the amount of information needed to index. Indexing can always be done by the CBM's inspecting the hetero-set itself; or it can be accomplished somewhat more efficiently using the template in the HSD describing the structure class; or finally, it can be done by one of the routines, called the indexor, whose address is in the HSD. This range of possibilities compares favorably with the implementation facility discussed by Earley in his paper on VERS [Ea 69]. The order of attempted application of these methods (the first which is found applicable is used) is

(1) if the HSD contains an address for an indexor routine, then call that routine with the stack in the configuration for a normal call with the index and reference items as parameters;

(2) if the HSD contains a template for the structure, then use that to calculate the position of the item specified by the index part of the accessor; or

(3) if neither (1) or (2) holds, then sequence through the hetero-set, using the limit values of the individual elements of the set to move from inner set to inner set.

The last alternative will always work but is the least desirable of

When the index for an access is known at creation time of a CBM instruction sequence, such an indexed accessor can be placed into the code in order to gain some efficiency by binding the accessor then, instead of re-creating it on each execution of that code. One can change an accessor from the set to the indexed form simply by changing the S/I bit from 0 to 1,and conversely. If a set accessor is thus changed to an indexed accessor for the last element of the set, because the index field is in the same place as the limit field is normally.

The index field can be set by the SETIX operation which uses the source as the index value and the destination operand as an accessor whose index value is to be set. If the accessor is not an indexed accessor, this operation will make it one by setting the S/I bit to 1. The destination is taken either as a reference to the accessor to be changed, or as a literal if so used. If the index value is defaulted, however, the accessor on the stack is changed, and not what what it refers to.

Throughout such sequencing, the address field of an indexed accessor points steadfastly at the header for the set accessed. Hence, no direct reference is allowed to exist which points into the middle of a homo-set, except possibly momentarily in the "hardware" of the CBM.

This same scheme is used for indexing into hetero-sets, and although a direct reference may be used for accessing an element of a hetero-set, in many cases only indexing is required, and direct

This definition of field descriptors allows them to be used as operands in any of the ALM class of instructions without any special instruction notation being needed to say that field extraction is being done.

A15 Sequencing Through Sets

Given an accessor which points to an element of a structure, we can sequence to the next element using the operation SQJL ,B . This operator works only on accessors. After the sequencing part of the operation is done, the second operand B is used to effect a change of control depending on whether there was or was not an element to which to sequence.

SQJL jumps to B if there was no "next" element, or allows control to continue normally if the element was not the last of a set. The first operand, if present, tells by how much to index forward or backwards (in terms of value items, not words); it may be any one of the allowable operand forms except a field descriptor.

Indexing in the CBM is done using an indexed accessor; such accessors differ from the standard only in that they contain an index value in the field normally used for a limit-value for bounds testing. Only the S/I bit differentiates between the two breeds of accessor.

The software operations which allow one to sequence through a set (whether homo- or heterogeneous) are the SQJL operation given above, and its counterpart BSQJL (Backward Sequence and Jump if Last). Both BSQJL and SQJL check for out-of-bounds accessing.

copies it exactly as it is.

Field descriptors (type 5) are used in conjunction with accessors in order to gain easy access to subfields of basic data structures. These basic data structures include bit strings from binary words, byte fields from byte sets and the fields in a set header.

If a field descriptor is used as an operand in an instruction of the ALM class, the address with which it is associated is the position on the stack which would have been used if normal accessing were being used and that operand had been defaulted. We will use the following notation for field descriptors: $(\alpha,\beta:\gamma)$ means index into the set specified by some accessor, using α as the index value, and then take bits β to γ inclusive from the designated item. Thus, (3,0:0) would access the high-order bit (bit 0) of the fourth word (since numbering is from zero) in a set referenced by some accessor. If the item from which a field is being extracted is a byte set, then β and γ are considered byte positions starting at the word specified by α. In that case, neither α nor β is restricted to be less than 4, even though our version of the CBM has only four bytes per word. The byte string thus extracted is treated as a normal byte set by the CBM and any spare bytes in the last word of that set will be set to the value of the "ignore" character which was mentioned in the definition of byte sets.

When a field descriptor is used as an operand and the associated stack item is not an accessor, but is of some other type allowable for the instruction being executed, extraction is applied to it.

an accessor involves an extra memory access to the set header in order to increment its reference count. Similarly, destroying an accessor, even by simply overwriting it, must decrement the reference count of the set to which it refers. Decrementing a reference count to zero will cause an interrupt in order to reclaim the storage for the set which is no longer referenceable. Reclaiming the storage for extended data structures is accomplished by invoking a routine,called the destroyer, for the structure; it is one of the routines which are invoked via an accessor in the HSD for a given class of data structure.

The operations available for accessors are restricted in order (1) to guarantee knowledge of set ownership; (2) to provide memory protection across processes and procedures; and (3) to protect undebugged programs from themselves by catching invalid accessors when they are made (i.e., as soon as the program attempts to create or modify an accessor in an invalid manner). One may copy an accessor; sequence from item to item in a set; index, using an accessor, for purposes of accessing a particular item of a set; diminish or increase the limit-value in an accessor -- provided it remains within the bounds of the referenced set; extract the limit-value for inspection; and create an accessor (by creating a new set).

COPY can be used to manipulate accessors as data and has the side effect of incrementing the reference count of the set which the accessor points to. Linking from set to set, as is often done, is also accomplished by the COPY operation, since it does not coerce the item of which a copy is to be made to be any special type, but simply

In this class of instructions we have included COPY and MOVE, the two primary operations used for moving data in the CBM. The rationale behind this decision is that all the operations in this class (which we shall call the ALM class) can also be executed with the "?" option, in order to set the condition code without actually affecting the destination operand. The code is set by the COPY and MOVE commands depending solely on properties of the source operand; these properties are outlined later with those instructions. Note also that c_0D denotes Auto/Fetch applied to D; c_1D denotes the address of the value specified by c_0D and cD means D, if D is a literal, or the item directly referenced by D, if D is a non-literal reference.

All of ADD,SUB,NEG,MUL,DIV, AND, OR, NOT, XOR have their usual interpretations with the operands as given in the ADD example earlier. COPY and MOVE both manipulate data. The only difference between them is that MOVE involves Auto/Fetch and Auto/Store coercions on its operands, whereas COPY does not. Thus, COPY could be used to replace an accessor by some other value (possibly also an accessor), but if MOVE were used for the same purpose, Auto/Fetch and Auto/Store would coerce the arguments so that only non-accessors were used in the actual operation.

A14 Operations on Accessors

An accessor contains a base address and a limit field, defining a bound on the set referenced in terms of a number of words. For storage reclamation purposes, the reference count associated with a set must include all references to it; thus, the act of simply copying

the condition code to indicate a zero result, although no result is actually produced at the destination. This was the previously mentioned special case in which it was not necessary that D∈A.

A12 Sets as Operands

In the CBM, most data manipulation instructions apply equally well to unit sets, general homo-sets, and to extended data structures. In particular, arithmetic operations can be used on entire homo-sets. Thus, ADD S,D where S and D both refer to 5-element sets, say, would add corresponding elements of the two sets, just as in Iverson's APL [IF 68]. For a more thorough discussion of this, see Section 5E3, CBM Instructions.

A13 Arithmetic, Logical and Data Movement Operations

Table A13-1 is an outline of the arithmetic, logical and data movement operations of the CBM:

OPERATION		EFFECT OF OPERATION
ADD	S,D	$c_1D \leftarrow c_0D + c_0S$
SUB	S,D	$c_1D \leftarrow c_0D - c_0S$
NEG	S,D	$c_1D \leftarrow 0 - c_0S$
DIV	S,D	$c_1D \leftarrow c_0D / c_0S$
MUL	S,D	$c_1D \leftarrow c_0D \times c_0S$
AND	S,D	$c_1D \leftarrow c_0D \wedge c_0S$
OR	S,D	$c_1D \leftarrow c_0D \vee c_0S$
XOR	S,D	$c_1D \leftarrow c_0D \veebar c_0S$
NOT	S,D	$c_1D \leftarrow \sim c_0S$
COPY	S,D	$D \leftarrow cS$
MOVE	S,D	$c_1D \leftarrow c_0S$

Table A13-1

[11]: ADD S,D

This form adds the operand specified by S to that specified by D.

The result is stored into the place specified by applying Auto/Store to D. The complete instruction format is

ADD	1	1	T_S	T_D	S	D

Because of these four cases ([00], [01], [10], and [11]), the ADD operation in our CBM can be one of .5, 1, 1.5, 2.5, 3.5, 4 or 4.5 words long, depending on the presence or absence and form of S and D.

A11 Condition Testing and Setting

Sometimes the restriction that DεA is unnecessary. Certain operations, which include at least the arithmetic, logical and data movement operations, set the condition code of the CBM, depending on the value of the result of the specified action. This condition code can then be used to determine branching within a CBM program. It is useful to be able to set the condition code as needed by some operation without altering the items involved in the operation. For this reason the above mentioned class of operations may be performed without changing any values, only setting the CBM condition code. This is denoted by appending a question-mark ("?") to the end of an operation mnemonic, a device similar to one used by Iliffe [I1 68]. Thus, ADD? -2,2 is valid and sets