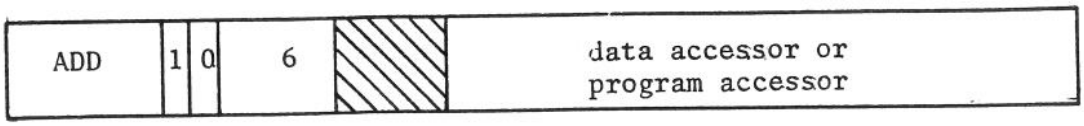
Case (ii): S is a data or program accessor.

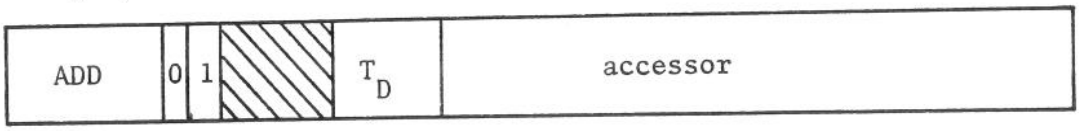


In this case, the value to be used is that referenced by the accessor, with Auto/Fetch or Auto/Execute employed to ensure a resulting N-value. Note that if S signifies a program accessor, the function which changes the operand type to one suitable for ADD (called a transfer-function) performs the action of executing the program addressed by S in order to obtain an N-value.

[01]: ADD ,D

This case is basically the same as [10], except that the operand specified by the top-of-stack element is added to the N-value specified by D. Auto/Fetch and Auto/Store (its counterpart for the purpose of storing) may be invoked for the stack operand and/or D.

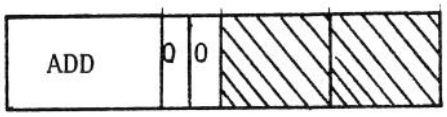
The format of the instruction and its operand is similar to the [10] case:



Note however, that in this case, D≠N since we require an accessor in order to know where to put the result of the ADD; the same is true of the destination operand in all these cases (except, see below concerning condition code setting).

As an example of the different possible forms of an instruction, we shall describe "ADD" with its various possible operands (the "label" in front of each diagram denotes the value of the presence-bits for that format):

[00]: ADD



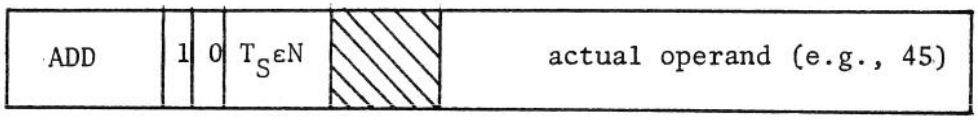
This format adds the operands specified by the top two stack elements, replacing them by a single result value which is their sum.

[10]: ADD S

This form of the instruction adds the operand specified by S to the operand specified on the top of the stack. The result replaces the item which was on the top of the stack.

There are two possible cases, depending on T_S, the type of S:

Case (i): $T_S \epsilon N$, where N={undefined, binary word, integer, real, byte word, field descriptor} .



This case adds the value immediately following the instruction to the top-of-stack item. It corresponds to a class of operations sometimes termed "immediate" on many computers, meaning that part of the data for the operation is contained in the instruction itself.

<OPR,op_1-description,op_2-description>

where op-description is a pair <presence-bit,operand-type>. The presence-bit indicates whether or not the corresponding operand is present or defaulted. The operand-type is related to the basic CBM types, with some exceptions; the basic types which are allowable as direct operands in instructions are listed in figure A2-1. Notice that this class includes types 0 through 8 and type 14. Now, if an accessor is to be used as a literal instead of the address of the actual operand which is to be used, then types 9, 10, 11 and 12 are used in place of types 6, 7, 8 and 14 respectively. This allows us to place a reference to a value on the stack using the same instruction as would be used to put the value referenced by that accessor onto the stack.

A "bare" CBM instruction, without operands, has the following format:

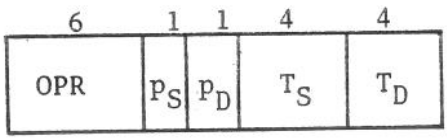


Figure A10-1: Format of a Bare CBM Instruction

p_S and p_D are the presence-bits for the source (first) and destination (second) operands; and T_S and T_D are the types of the operands, if present. The operands have been denoted as source and destination to indicate the fact that movement is to the destination, whether the instruction is arithmetic or logical in nature. Thus, ADD S,D adds the value denoted by S to that denoted by D, and places the result at the place denoted by D.

table themselves). A process also contains the name of its parent or creating process, except for the process called SYSTEM which has no parent. How this information relates to interprocess control and communication is discussed later.

A10 CBM Instructions

Up till now we have given only sketchy examples of the form of instructions on the CBM, In this and the following section we will give a detailed description of most of the CBM operations, including instruction and operand formats, and the applicability of some operations to non-unit sets. The extension of operators to programmably extended data structures will be described as well as the set of operations which may act on accessors directly.

The basic instruction formats of the CBM allow zero, one or two operands. Thus, OPR S,D might indicate a two-operand instruction. But either or both of S or D can be defaulted (in most cases), in which case the corresponding operand is assumed to be the one on top of the stack (or may be the second item on the stack if both operands are defaulted). This scheme allows some ease for translators on the CBM: simple Polish postfix notation [RR 64] can be produced, as well as operations to and from memory. This philosophy has in fact been implemented on at least one commercially available computer, the Digital Equipment Corporation PDP-11 [DEC 69].

In order to allow such defaulting of operands, the format of an instruction must be a triple:

The lowest part of memory contains three different types of information:

(1) interrupt locations for "hardware" errors and exceptional conditions; these locations contain a program accessor and a status word, which will be loaded into the CBM registers PRP (PRogram Pointer) and PSI (Program Status Information) respectively, after "manufacturing" a call from the currently executing program so that the interrupt routine may be programmed much like a procedure;

(2) state information for the CBM, including a table naming the objects in the CBM, "ports" over which input and output may flow between the CBM and I/O devices or other machines, and status information for those ports;

(3) a single, special PAC which is the first free PAC in memory; it cannot be allocated and therefore acts as the head of the primary free storage list.

Now, the CBM is not just one of the "virtual" machines outlined above, but many. Each such "machine" is called a process and has a unique "name" (a 32-bit integer) by which it is known within the system. Data structures can be communicated between processes and one process can invoke another while suspending its own execution until later re-invoked by its invokee, thus mirroring a coroutine structure [Kn 68,Kr 69,DN 66]. The form of interprocess accessors will be given later.

Each process contains at least one name table describing the nameable items in that process (including the process and the name

A9 The Process Concept in the CBM

Thus far we have been speaking of the CBM as one machine with a stack and some other memory. Diagram A9-1 represents the layout of the CBM:

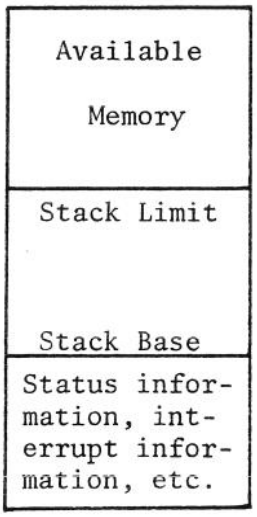


Figure A9-1: Layout of a CBM

The stack base is at some location β in memory (β is fixed in a given implementation, but left unspecified here), and the stack is allowed to grow upward so long as it does not exceed Stack Limit. The remainder of memory is initially one or more PACs which can be allocated for program and data structures.

The stack's allowable upper limit is indicated by a register called the SLR (Stack Limit Register), and an attempted PUSH which would overflow the stack will cause an interrupt. The CBM may then attempt to increase the allowable stack limit or link stack segments together. The TSR (Top of Stack Register) always points to the top element of the stack.

In a global stack accessor, the displacement field is made relative to the bottom of the stack instead of with respect to any display register. The difference between a stack accessor and a global stack accessor is that the DISPLAY INDEX of the global form is all 1-bits.

Note that such an accessor cannot be used to access an object to which is was not given access: for, if an attempt is made to pop a stack cell whose reference count is not zero, an interrupt occurs. Hence, even if a global stack accessor is copied into a location not in the stack, it will not be allowed to exist longer than the item to which it refers.

called a Mark Stack Control Word (or MSCW, type 13), and which also plays an important role in the procedure call sequence.

Associated with the stack is a set of Display Registers which allow procedures which have a common lexical scope to share values on the stack, even if the procedures are used recursively. A stack accessor consists of two parts: a display index and a displacement.

Each display register points to an MSCW; an accessor, therefore, represents an item with a certain displacement from that MSCW. The format of a stack accessor which accesses items using the display registers is the following:

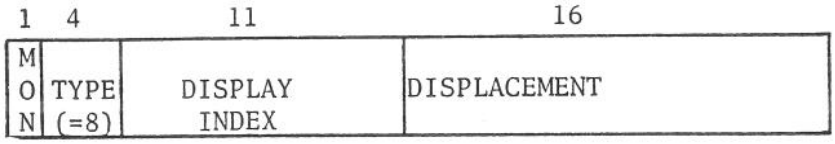


Figure A8-2: Simple Stack Accessor

Each display register contains a simple address of an MSCW in the stack. Although we do not specify how many display registers there should be, display 0 is always a copy of the current display, and is identical with the LBR mentioned earlier. This means that using DISPLAY INDEX = 0 will always access only local variables.

When an accessor is needed which is to access an object which, while in the stack, is outside of the current display, a different form of stack accessor is used. We will call such an accessor a global stack accessor. This situation occurs when an accessor is passed as a parameter with a call from a non-local procedure; therefore the display registers are not held in common with the called procedure.

can only be a unit set, this is sufficient for most purposes. Values which are to be monitored require one more bit, and a reference count is needed.

One advantage of this implementation is that, knowing the size of all possible primitive values (the maximum is 3), we can determine a fixed amount which can be added or subtracted from the stack pointer to accomplish the actions of "pushing" and "popping" the stack. Since these operations are performed as CBM primitives, the stack is safe in the same sense as discussed for the general memory. For our CBM, this fixed amount is 4 words: one for header information, and three for values.

The stack items can be formatted as in figure A8-1; notice that only one word of type information is needed for each item.

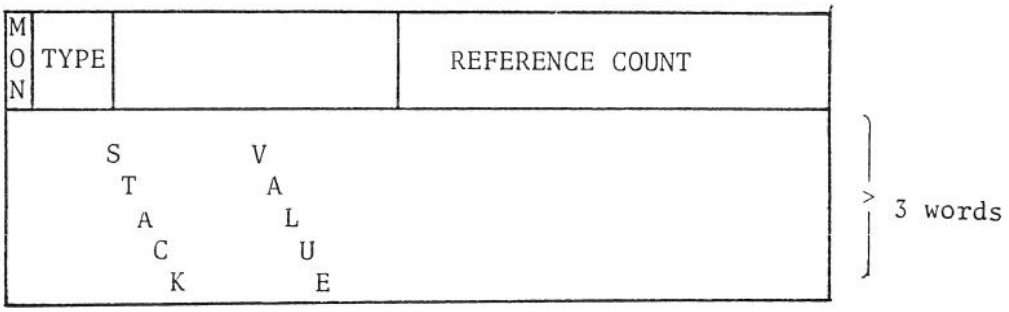


Figure A8-1: Format of CBM Stack Elements

Variables which are local to a routine can reside in the stack. Space is allocated for them at procedure entry time using the PUSH operation, and they are accessed relative to a register called the LBR (Local Base Register) which points to a special stack element

together for searching purposes. It occupies space normally used for an HSD address and is unnecessary here since PACs are primitive. The ADDRESS OF PRECEDING FREE PAC field can be used to implement a storage management system in which adjacent free PACs can be coalesced in order to retard the fragmentation of storage, a common phenomenon in such systems [Kn 68].

A8 The CBM Stack

The CBM stack is a primitive data structure which can be used to hold sets of certain forms and control information for procedures, functions, interrupts, etc. The stack can also be used for intermediate values of expressions, for local variables, and for parameters passed during procedure calls.

The form of atomic elements on the stack is somewhat different than for atomic elements in general memory. This is done for two reasons: (1) the stack, being the primary control mechanism, needs to be implemented efficiently; and (2) if only primitive values are allowed on the stack, then no size field or HSD address is needed to maintain stack integrity.

One piece of information which does remain constant for sets, whether in memory or on the stack, is the type field; therefore, each item in the stack is tagged with its type. Since the set described

on the list headed by A, then the B-PAC is coalesced with those into one large PAC.

It is left to the reader to assure himself that most allocation systems can be implemented using these primitives.

In the case of storage allocation, SPLIT always yields a PAC as result and the act of making a PAC into any other type of structure is done simply by changing the type and possibly the size fields of the set header. Normally, the contents of the set itself are left untouched, except when the new set is to contain accessors. In that case, each entry is initialized to zero, indicating an invalid address.

The format of a PAC is given below:

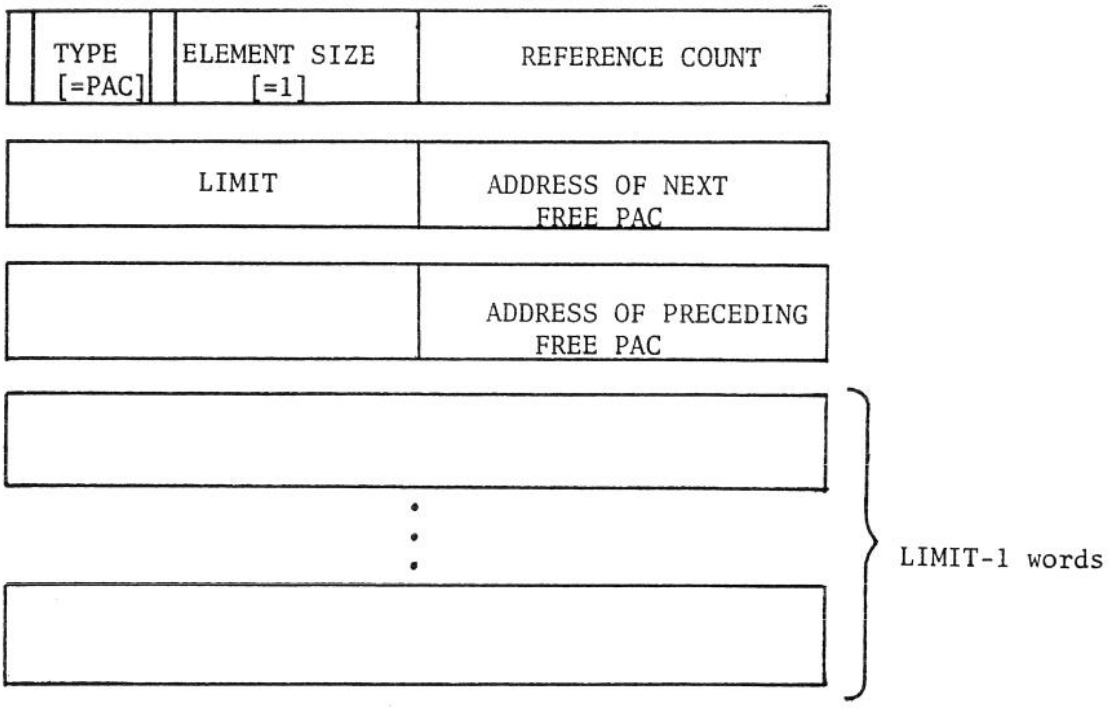


Figure A7-1: Format of Primitive Allocation Cell

The field labelled ADDRESS OF NEXT FREE PAC is used to link PACs

by programs written for the CBM. Steps (b) and (e), however, require manipulations of sets which must be primitive in order to maintain storage integrity. Those primitives and a brief description of their functions follows:

SPLIT A,B — causes the PAC referenced by B to be split into two PACs, the second of which is of size A; the address of the second PAC is returned on the CBM's stack.

MAKE A,B — causes the set at B to be changed by the CBM into a set of type A. If this operation were not primitive, we would need some way of altering the header of a PAC to make it look like a set of type A -- this cannot be done without abandoning the integrity of storage and, hence, the operation is made a primitive. If a PAC is linked in a free list then MAKE also unlinks it from the free list; therefore, a free storage list is prevented from containing anything but PACs.

UNMAKE A — converts the set referenced by A to a PAC; it is up to the user's program to link the resultant PAC into a list of free storage if he so desires. If he does not, the PAC is automatically linked to the primary free list maintained by the CBM.

UNMAKEJ A,B — A is interpreted as the head of a free list; B is the address of a set to be free as in UNMAKE. An UNMAKE is done and the PAC is linked into the free list headed by the PAC to which A refers. If, in this process, the B-PAC is found to be adjacent to one or more free PACs

freeing memory in the conventional sense. Consideration (1) is simply a constraint on this process which says that no orphans are allowed: each word of memory must belong to some set. This philosophy toward memory management is part of the AED Free-Storage Package [Ros 67], a simplified version of which was used in the LC^2 system.

For the purposes of the CBM design we will divide allocation into four steps:

(a) finding a "piece of storage" P large enough for the set to which it is to be allocated;

(b) dividing P into (possibly) two sets, one of which will be allocated, and the other of which will remain as a "left over" fragment;

(c) correcting any linkage of the sets in free storage which may be necessary;

(d) returning a reference to the allocated set as the value of the allocation function; and

(e) making the allocated set have the desired type.

Now, we would like to build fast allocation systems for data structures with a high rate of birth and death, so as to minimize that overhead. Lists of fixed-size cells are such a class of structures. In the AED Free-Storage Package, Ross describes such a storage management system. While it is not necessary that the CBM implement such a system directly, it nevertheless behooves us to provide allocation/de-allocation primitives which aid such methods.

Steps (a), (c) and (d) in the above outline can be accomplished

occupies the same field as used for a reference count in the set header, the reference count field must be large enough to contain an address.

When a set header is a hetero-header, the HSD address field is used to "define" the class of extended structures to which that set belongs by pointing to the HSD for the class.

A7 Allocation of Sets

Storage allocation in the CBM raises a number of interesting questions. Initially the CBM has a special accessor and header for the real memory of the machine, which memory is viewed as a primitive hetero-set called a PAC (Primitive Allocation Cell). It is this structure from which storage is allocated for all sets, and to which storage is ultimately given back when no longer required by systems in the CBM. Thus, all the structures of the machine will receive (although possibly indirectly) their storage from this initial set.

Two immediate problems present themselves:
(1) to keep track of all the sets allocated from the primary memory, and (2) to determine how storage is "passed" from set to set. The latter question is the primary problem in storage allocation.

In the CBM, the storage used by any set is said to belong to that set. Since it previously belonged to some other set, which we can think of as the parent of the set to which it now belongs, the act of passing that storage, or more properly, responsibility for that storage, from set to set is the mechanism for allocating or

by the requirements of the other fields in word 0, including the "unused" bit (which is used when such a two-word block acts as an accessor value). Finally, the MON bit is used to indicate monitoring of a set.

The value of an accessor is also two words in length and is an almost exact copy of the header of the set to which it refers. The value is shown in figure A6-2 below:

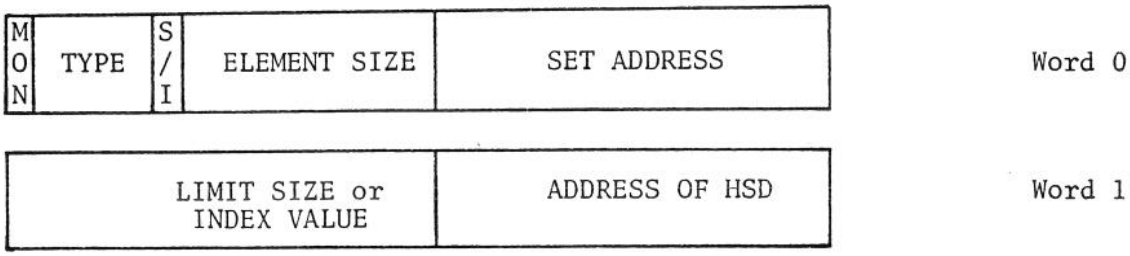


Figure A6-2: Format of Accessor Value

This format is applicable only to program and data addresses (types 6 and 7) and not to stack accessors (type 8).

Only two fields differ from those in a set header: the S/I field, and the SET ADDRESS field. The S/I bit indicates whether the accessor is a reference to the set as a whole or to an individual element of the set. In the latter case, the limit field is interpreted as an index (in terms of words) of a single element of a set. When used in this way, the index field can be made to sequence from element to element in the set. This is not allowed to go past the end of the set; the CBM checks each index change against the limit field in the set's header.

The SET ADDRESS field points to the referenced set. Since it

a homo-header is reasonably clear: an instance of a structure is initially created as a homo-set of binary words, plus a header, and finally, one special machine function is executed which can change the header to be a hetero-header.

A6 Headers and Accessors

Accessors are closely connected with set headers since they act as "remote" set headers for accessing purposes. The format of a set header (both for homogeneous and heterogeneous sets) is the following:

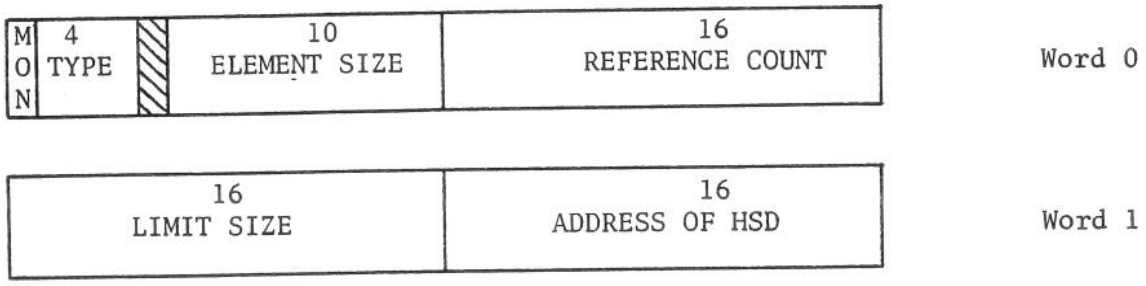


Figure A6-1: Format of Set Headers

Some of the fields in the header require explanation. First, it must be stated that the decision to use a 32-bit word for this CBM is completely arbitrary. Wherever this choice has influenced the design we will attempt to state that fact.

The size of the limit field must be the same as an address field in order to allow at least one set to be as large as the entire address space. This is a nice feature when starting up a virgin machine since all of memory can then be contained in that initial set.

The size of the element-size field has mainly been dictated

descriptors (HSD) and figure A5-2 is an example of one such. It is the HSD for the hetero-set given in figure A5-1 above.

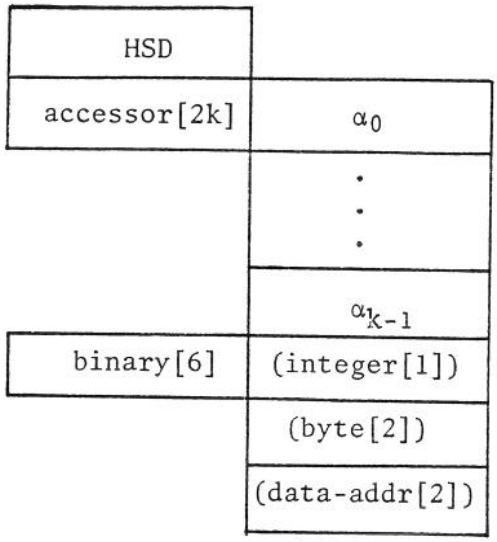


Figure A5-2: Example Hetero-Set Descriptor

The elements of the homo-set of k accessors refer to the routines which are to act as the programmatic extensions of the CBM primitive operators for homo-sets. Among these operators are a creator, destroyer, sequencer, indexer, etc. Some of the routine addresses may be left undefined in the case that certain functions are not defined for a given structure. For example, one does not normally index into a queue, but simply detaches items from the front end and attaches new items to the tail of the queue, so an indexer might not be required for such a class of structures.

The second homo-set in the HSD (binary [6]) is a set of binary words which are a "template" for instances of structures of the class defined by that HSD: they are skeleton headers for homogeneous sets in the guise of binary words. The manner in which such a word becomes

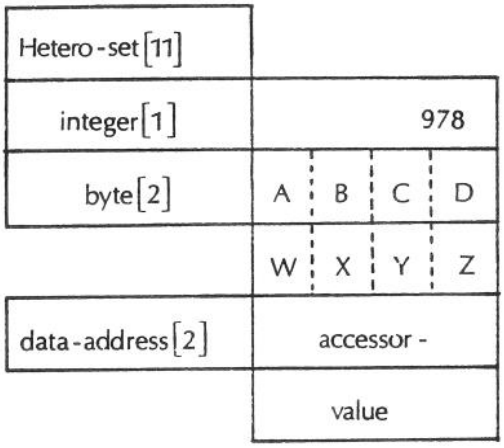


Figure A5-1: Example of a Hetero-set

Note that the limit value for the hetero-header is 11 (eleven) words, and includes the words occupied by the headers of the sets comprising the hetero-set.

The header for a hetero-set contains information about the set similar in nature to that given by a homo-header. There are some major differences, however. Since it may not be feasible to assign a unique "type" to each defined structure (especially since we intend to allow the user to define his own), and since we must keep some form of structural template for describing classes of structures, we will use an actual machine address as the "extended type" for structures. This address points to an instance of a special class of hetero-set which is used to describe structures. It is special in that it is considered primitive by the CBM and therefore needs no other data structure to describe it. Such "meta-structures" are called hetero-set

an accessor for it: however, storage reclamation operations always use the actual header on a set, and not any accessor for it in order to insure correct storage reclamation.

A5 Heterogeneous Sets

Thus far we have defined what could be termed a hardware base for the data of the CBM. Since it is viewed as a base, we place on it the constraint that programmed extensions of these data structures are to be handled in the same manner by the CBM as are homo-sets. The only requirement for this is that all necessary operations such as sequencing and accessing be provided by the program which "defines" such extensions. The main generalization which we wish to make involves heterogeneous or mixed sets: i.e., sets whose component elements may be dissimilar, unlike homo-sets in which all the elements are of the same type. We will show later that the elements of a homo-set can themselves be heterogeneous sets (hereafter called hetero-sets) so long as they are all instances of the same hetero-set class, and are of equal length.

An example of a hetero-set in which the first element is an integer, the second a homo-set of two byte words, and the third an accessor for another set of the same form is diagrammed in figure A5-1. The semi-isolated boxes to the left in the diagram represent the individual homo- and hetero-headers; the number of words in each set is indicated by the number in brackets following the type of the set written in the headers. The headers, of course, are contiguous with the elements to the right of them; the separation is made here only for clarity.

An accessor in the CBM corresponds to the concept of address-value in most computers. Accessors may refer only to the header of a homo-set, and may not point directly to an element of the set. This restriction is primarily for purposes of storage reclamation and is not as severe as it appears, since the elements of heterogeneous sets (extended data structures) can be homogeneous (or heterogeneous) sets which may be directly referenced. It is true, of course, that while accessing an element of a set there must be a simple address generated, but it is only temporary and, in a sense, exists solely in the bowels of the machine, inaccessible to programs running on the CBM.

Since it is impossible to do arithmetic on accessors - this is our major departure from a classical Von Neumann machine - we can guarantee that accessors behave as we have described. An accessor looks very much like the descriptor section of a set header (and, indeed, is created by copying parts of the header) but also contains an address pointing at the beginning of the set. There are two reasons for this redundancy of information in the accessor and the set header:

(a) if accessors are created and modified only in "correct" ways, which are under control of the CBM, then we can save an extra memory access to a set header each time that we need to access an element of the set;

(b) by allowing some "safe" modifications to be made to an accessor, we can get the effect of accessing subsets of the original set; also, if we wish, the type of a set can be changed in context by simply changing the type field in

Using the limit field, we can check for out-of-bounds accesses. Memory protection is provided in large part by the manner in which sets can be accessed; i.e., by such bounds checking. Also, checks on accessors being incorrectly modified will prohibit the changing of memory elements which, while not in some set, have the misfortune of lying "just beyond" it in the CBM's memory.

The reference count in a set header is used for reclaiming the storage occupied by a set when it is no longer needed. As long as there is an accessor referring to the set, the reference count will not be zero. When the reference count becomes zero as the result of the last remaining accessor for it being destroyed, then the set will also be destroyed, and its storage reclaimed for future use. This method does not provide a guarantee against sets becoming accessible, but does automate the common case of a set which is not part of a ring structure.

A4 Accessing Homogeneous Sets

The actual accessing of an item in a homo-set is accomplished via an accessor for the set. The accessor may contain a field which is considered as an index value. The actual element is then accessed by computing the element's displacement from the header for the set as

(index-value)×(element size) + address-of-header + header-size

where "address-of-header" is part of the accessor value, and "header-size" is the value 2 (since headers, as will be seen, are two words long).

A3 Atomic Elements

Primitive values, as described above, do not exist without structure as in most computers, but have a structure imposed on them by appending a header to each set of elements (recall that we stated that a primitive set was assumed to contain elements all of the same type). All data manipluation operations in the CBM then have access to the header of a set and may use that information to aid them in correctly performing their specific transformations or for checking for error conditions in the use of data.

Sets may be unitary (one element), or be a contiguous, indexable, homogeneous set. The empty set is also allowed, but its type has little meaning. A homogeneous set (which we shall call a homo-set) always has two parts, unless it is the empty set: the set header, which is two words containing information describing the elements of the set (this structures the memory of the CBM), and a value set, which is a contiguous group of primitive values, all of the same type.

Each set has fields in the header with at least the following information:

(1) the number of words in the set, called the limit of the set,

(2) the type of the primitive values in the set,

(3) a monitor bit for the set — if it is 1, then any normal access into the set will cause an interrupt,

(4) a reference count of the number of accessors which refer to the set at a given moment — used for storage reclamation purposes.

There is one special character consisting of all zero bits which is considered as an "ignore" character. This is done since it is unreasonable to expect all byte strings to be a multiple of four bytes in length. As well, ignore characters may lie anywhere within byte strings, thus making operations such as concatenation of strings or deletion of a substring within a string much easier.

Field Descriptor (type 5)

The value part of a field descriptor consists of three numbers α, β, and γ packed into one word. It is used in conjunction with an accessor (type 6, 7, or 8) to access the field in the α^{th} element of the set referenced by the accessor, and consisting of bits β to γ inclusive. If $\beta>\gamma$ then the value "fetched" is zero. This type's inclusion is simply in recognition of a most common operation in most large programs which attempt to concerve memory space by packing a number of different pieces of information into one word.

Accessors (types 6, 7, and 8)

Address values in the CBM are not as simple as on most machines; for this reason, we choose to refer to them as accessors or references. An accessor contains the address of the memory location of the object being referenced, a limit-value, which is the number of words in the referenced object, and the type of the referenced object. Stack accessors (type 8) are somewhat more restricted than this, but must meet many of the same other constraints as program and data accessors. We will discuss accessors in much more detail later in the appendix.

occupied by an indefined element may be used to contain codes indicating different forms of undefinedness, or to trigger special actions. The length is variable only so that the storage for an undefined element can be overlaid by any of the other basic primitive values, some of which are two words in length.

Binary Word (type 1)

This is simply 32 bits, the standard representation on most machines. The individual bits are manipulable by means of various shift operations, as well as by instructions using the field descriptor (type 5).

Integer (type 2)

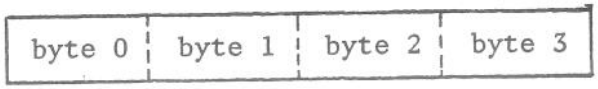
We will choose simply to leave unspecified the exact format for integers — whether uncomplemented with a sign bit, or one's or two's complement notation is not germane to this discussion.

Real (type 3)

The exact form of floating-point numbers, or reals, is also left unspecified. Note, however, that it is possible for two or more sizes of reals to coexist on the same machine, simply by allowing different lengths, as for type 0.

Byte Word (type 4)

A byte word (or character word) is simply a 32-bit word which has conceptual boundaries dividing it into four characters of eight bits each, thusly:



primitive data types follows.

Of the data types in figure A2-1, we will mention only the first eight (types 0 through 7) here. The remainder are described throughout the rest of the appendix.

TYPE	CODE	LENGTH OF VALUE	ALLOWABLE AS OPERAND
undefined element	0	1 or 2	yes
binary word	1	1	yes
integer	2	1	yes
real (floating-point)	3	2	yes
byte word (characters)	4	1	yes
field descriptor	5	1	yes
data address	6	2	yes
program address	7	2	yes
stack address	8	1	yes
extended data structure	9	unknown	yes (by extension)
hetero-set descriptor	10	H + K *	no
Primitive Allocation Cell	11	≥3	no
instruction sequence	12	≥0	no
mark-stack control word	13	2	no
inter-process accessor	14	3	yes
unassigned	15		

Figure A2-1: Table of Basic Types in the CBM

* H is the number of words necessary to provide addresses of the routines to be used as program extensions of CBM operations on homogeneous sets; K is the number of words necessary to specify a packed template for the hetero-set.

Undefined Element (type 0)

This "value" is closely associated with much of the philosophy behind the CBM. Accessing (as opposed to storing into) an undefined element will normally result in an interrupt. In some cases, when manipulated by certain instructions, no interrupt occurs, but all arithmetic instructions, for instance, on undefined elements will cause interrupts. The "value" of the one or two words of storage

efficiency or comprehiveness, for instance, should not require fundamental changes in data structures or the programs which operate on them.

Another function which shall be represented in the data structures is monitoring. Whenever a "monitored" set is accessed (or, alternatively written into), an interrupt sequence is initiated, and may result in the activation of a routine, due to some expression (called "continuously evaluating" by Fisher [Fi 70])assuming the value true. Thus, the routine so activated might be used to trace all changes to a variable, uses of a given function, and so on.

There are two basic types of sets in the CBM: homogeneous, in which each element is of the same type, and heterogeneous, in which the elements may be of different types. The latter case raises a number of difficult issues, whereas the former is fairly simple and causes no large problems.

A2 Primitive Values

Primitive values are those which are directly representable and manipulable by a hardware machine. Since this group may vary from machine to machine, we only give a representative sample here.

For our purposes we will assume the hardware memory of the CBM to be organized into words and characters (bytes), with bytes being k bits each, and words being some multiple of k bits wide, but at least 2k bits. In the discussion to follow, we have arbitrarily chosen k to be 8, with words of 32 bits. A description of the