RICE UNIVERSITY COMPUTER PROJECT

Elements of BLM

J. K. Iliffe

This note describes the essential features of a Basic Machine, and indicates the main choices which remain to be made in designing a practical system.

7 November 1968

Elements of BLM

J. K. Iliffe*
Rice University

1. The BLM experiment

"Basic Language Machine" is the name given to an experimental computer built in the research division of International Computers Limited in Stevenage. As the name indicates, the experiment is directed at both the "hard" and "soft" aspects of computer design, but more particularly with the way the system appears to its users. It is "Basic" in the sense that it is not concerned with those refinements which can be achieved entirely by "soft" translation of commands from one language to another, nor with equivalent hardware transformations which do not affect the machine's logical capabilities. In algebra, the "base" of a vector space is a set of independent vectors from which all elements can be derived by linear combination of operations; by a rough analogy, the "base" we have been looking for is a set of objects and operations from which all elements of programming "space" can be derived. The classic bases of computation, e.g. Turing machine or recursive function theory, are unsuitable in this respect, since they do not take account of the finite nature of machine registers, or of the task of program construction, to name only two important factors.

Most computers in use today are based on the von Neumann model of machine organisation, which can be described in terms of a memory consisting of an ordered sequence of words, in which instructions and numbers are stored. A sequencing rule is introduced to describe how control passes from one instruction to the

*On leave of absence from ICL, Stevenage, England.

next. The instructions are split into function and address fields, and an addressing rule is used to determine the numbers on which the functions will operate. The definition then proceeds to more detailed definition of each function, including the conditions under which it "fails", either because the addressing rule can't supply it with operands, or because it can't produce a meaningful result.

It is well known that the von Neumann model as it stands is an insufficient basis for programming. This is difficult to "prove", since most problems can be programmed, with a given function code and sufficient store. What is lacking is the generality of expression which alone makes the effort of writing a program worthwhile. In other words, the idea of words in memory must be replaced by a more general information structure, and the addressing rule must evolve into a more general means of identifying items of information and relating one to another. The conventional approach to this problem through "high level" languages provides at best a partial solution: it offers the programmer a useful set of working concepts, but it attaches restrictions, in the interest of run-time efficiency, which are not always easy to accept. Moreover, the original program is often so mutilated in the course of translation that there is no hope of changing it during execution, or of checking that the structure is being interpreted consistently.

In the BLM we are attempting to provide a "hard" programming base sufficient to meet operating system and user requirements in the most general way, i.e. by retaining structural information as an essential component of a program, distinct from numbers and instructions, and adapting the function code to operate on it. Such a plan of work obviously calls for much detailed study of current programming practice, trends in component costs, market requirements, and so on. However, the essential features of a Basic Machine are almost as simple as a von Neumann machine; it is the purpose of the next three sections to introduce them at the same level of detail as would be offered in a first lesson on programming, i.e. assuming

that the various exceptional conditions which are noted in passing are rare enough to be ignored, though we know from experience that they occur often in the human timescale. In section 5, we indicate very briefly the various lines of development leading to a properly engineered system, some of which have been outlined in ref. 1, and others are under investigation on the experimental BLM.

2. Process Base elements

We begin by describing the "store" of a Basic Machine, as it appears to its users. There are three types of stored elements, corresponding to the three categories of information in the machine, i.e.

Type NUMERIC - corresponding to numbers;
CONTROL - corresponding to instruction sequences;
ADDRESS - corresponding to structural information.

Each element occupies the same amount of physical storage - a word size - but it is not necessary to disclose the amount, or the detailed formats involved. The elements are truly "atomic", and are defined only by the effect they have in the system.

Type NUMERIC may include several different numerical representations: for present purposes, we assume zero or signed integer values, of limited magnitude, in two's complement form.

Type CONTROL points to the first of a finite sequence of stored instructions: we shall see that it is not possible to isolate a single instruction, and CONTROL elements will be used only to change the flow of calculation.

Type ADDRESS points to the first of a finite sequence of stored elements, each of which is itself either NUMERIC, CONTROL, or ADDRESS type; it is possible to select a single element, operate on it, and return it to the same or another position in store, and all calculations devolve eventually into such actions.

A process base is a finite group of the elements described above, each of which can only be accessed by using its name. A unique process base is associated with each concurrent activity in the machine, and the names used in that process relate directly to its base. For the present, we denote elements of the base by expressions of the form "$\underline{X}_i$", where i = 1,2,..., etc.

The base is a relatively small amount (usually less than 100 elements) of information, the remainder of the program being accessible indirectly, by using ADDRESS or CONTROL elements pointing to data and instructions respectively. We think of a program as a "base", directly accessed by naming its elements, and a "store" area, distinct from the base, composed of instructions, numerical data, and pointers, though strictly speaking the "store" may be shared by several different processes.

3. Machine functions

Each machine function is defined in terms of the types of its two arguments, i.e. NUMERIC, CONTROL, or ADDRESS. The arguments are always base elements or constants, and it is convenient to denote the type of $\underline{X}_i$ by "$\underline{T}_i$" and abbreviate the type identifiers to N, C, A respectively. Thus:

$$\underline{T}_2 = \underline{N}$$

implies that $\underline{X}_2$ is a NUMERIC element.

Functions fall into four groups:

Group A: Arithmetic and logical operations
B: Base manipulation
C: Control jumps
D: Addressing operations

Table 1 summarises the assigned functions, each of which is identified by a mnemonic code as in a conventional assembly program. In this instance, however, the binary representation is not disclosed, nor do

we assume that each symbolic instruction represents just one "machine" order.

The addressing rule for BLM

Certain functions cannot be applied directly to ADDRESS elements, e.g. it is not possible to perform arithmetic on, or direct control to, an ADDRESS. However, if the ADDRESS points to a sequence whose first element is NUMERIC or CONTROL, then it may be possible to interpret the function meaningfully. In theory, the search for a useful operand could go through several addresses, but we have found this rule difficult to apply in practice, and only one step is taken in the BLM. The resultant rule, which applies whenever a NUMERIC or CONTROL element is required, is termed the Auto Fetch (A/F) convention: it can be thought of as a rule for converting each "argument" which appears in an instruction into a useful operand.

A similar rule is applied to the storage of NUMERIC results: it simply states that the result of any Group A operation overwrites the first operand, which must be in the process base or one step removed, depending on whether a Fetch has not, or has, been applied to find it. This rule is termed the Auto Store (A/S) convention.

The use of A/F and A/S is illustrated in the following brief discussion of machine functions.

Group A: Arithmetic and logical operations

ADD,SUB,MPY,DIV,MV,SC,AND,OR,NEQ,NOT

Both operands must be NUMERIC, with the result that by A/F a single function such as ADD has four possible combinations of operand sources:

i.e.	1st Arg		2nd Arg
	N	base + base	N
	N	base + store	A
	A	store + base	N
	A	store + store	A

However, the function fails if either A argument points to anything but a defined NUMERIC element, or if either argument is C.

After applying A/F, the operation is performed conventionally, and the result is used to set the state of a pair of condition code indicators (similar to IBM System/360) which may be tested by control functions. The result then overwrites the operand selected by the first argument.

Group B: Base manipulation

COPY,LOAD,STO

These three functions are needed to circumvent the A/F and A/S conventions when moving elements of the process base. COPY simply copies from one base position to another. LOAD applies A/F to the second argument only, then uses the selected operand to overwrite the base element named by the first; similarly, STO applies A/S to the first argument only, and overwrites the selected operand with the base element named by the second argument. (Note that all Group A and B functions are defined so that the "information flow" is from the second argument to the first.)

Group C: Control jumps

J,JL,JNL,JLT,JGE

A destination is always specified by the second argument, as a CONTROL element, possibly by A/F. The condition under which the jump is taken is given by the first argument and the function. If the first argument is a condition mnemonic (Table 2), then the function must be "J" and the control sequence is changed if the current condition codes satisfy the selected condition; otherwise control passes to the instruction following the current one.

If the first argument is a base name, then the function may be J or one of the counting jumps JL,JNL,JLT,JGE. If J, then

the first argument simply indicates where to store a link*:

$\underline{X}_2$ J $\underline{X}_6$

sends control (unconditionally) to $\underline{X}_6$, leaving a CONTROL element in $\underline{X}_2$, pointing to the next instruction.

The counting jumps all depend on the value of the first argument. For JL, JNL it must be an ADDRESS, which is modified by unity on each JL or JNL until it points to a sequence of just one element, which is regarded as the "last"; in this way, and with the help of A/F, one would iterate through a sequence of elements in store. For JLT, JGE the first argument must select a NUMERIC element (by A/F) which is decremented by unity on each application of the test, and the result used to determine whether to take the jump or not.

Group D: Addressing operations

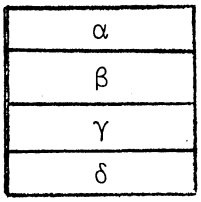
MOD,LIM,INDEX,MEM

An ADDRESS always points to the first of a sequence of stored elements, and indicates how many follow it: this number is termed the index of the sequence. Clearly, the index is non-negative; if it is zero the sequence consists of only one element, and the ADDRESS pointing to it is said to be "singular".

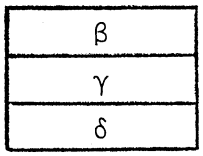
Given an ADDRESS, selected by the first argument, MOD and LIM are used to form an ADDRESS pointing to a sub-sequence of the original one. In each case, the second operand must be NUMERIC, non-negative, and not greater than the initial index value. Let its value be K. Then MOD has the effect of "removing" the first K element of the sequence; and LIM has the effect of replacing the initial index value by K, i.e. "removing" all but the first K + 1 elements. (By "removing" we mean that elements are no longer accessible by using this address: there may be other addresses still pointing

*The BLM programming convention is to place the function mnemonic between the first and second arguments, see page 9.

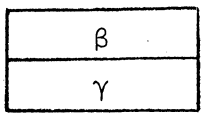
at them.) The effect of MOD and LIM on $\underline{X}_1$, assuming $\underline{T}_1 = \underline{A}$ and index($\underline{X}_1$) = 3 initially, is summarized in the following diagrams, with K = $\underline{X}_2$ = 1:



(a) initial sequence of four elements addressed by $\underline{X}_1$, index 3



(b) sequence of three elements addressed by $\underline{X}_1$, after:
$\underline{X}_1$ MOD $\underline{X}_2$



(c) sequence of two elements addressed by $\underline{X}_1$ after (b) then:
$\underline{X}_1$ LIM $\underline{X}_2$

The remaining two addressing functions are used to obtain the index of a sequence (INDEX) and to enquire whether the first element of one sequence (given by the first argument) occurs within another:

$\underline{X}_1$ MEM $\underline{X}_2$

If it does, its relative position is written into $\underline{X}_1$, and condition codes set accordingly (ZE,GT,IR).

4. Formal definitions

In the BLM, formal syntax takes the place of the binary formats which are usually presented as part of a machine definition, though the format of NUMERIC elements is known and used in logical operations. Besides NUMERIC, the input language must be capable of introducing ADDRESS and CONTROL elements, and giving a precise meaning

to the idea of a control sequence. The following set of definitions leaves the form of labels as well as base names undefined: it is required that they be distinguishable from each other, and from the numerals.

Definition of code segments:

 :: = <line>|<line>
<line> :: = <label>:<instruction>|<instruction>
<label> :: = $\underline{L}_1|\underline{L}_2|\underline{L}_3|\ldots$
<instruction> :: = <control jump>|<action>
<control jump> :: = <condition test>|<loop test>|<set link>
<condition test> :: = <condition mnemonic>J<destination>
<loop test> :: = <base><JL|JNL|JGE|JLT><destination>
<condition mnemonic>:: = GE|GT|IR|LE|LT|NZ|UN|VR|ZE
<set link> :: = <base>J<destination>
<destination> :: = <base>|<label>
<base> :: = $\underline{X}_1|\underline{X}_2|\underline{X}_3|\ldots$
<action> :: = <first argument><function><second argument>
<first argument> :: = <base>
<second argument> :: = <base>|<signed numeral>|<structure>
<signed numeral> :: = <numeral>|<sign><numeral>
<sign> :: = +|-
<numeral> :: = <digit>|<numeral><digit>
<digit> :: = 0|1|2|3|4|5|6|7|8|9
<function> :: = <group A>|<group B>|<group D>
<group A> :: = ADD|SUB|MPY|DIV|SC|AND|OR|NEQ|NOT|MV
<group B> :: = COPY|LOAD|STO
<group D> :: = MOD|LIM|MEM|INDEX

Examples: Before proceeding further, we give some examples of code segments for performing specific calculations.

(i) Let $\underline{X}_1$ contain a number p, i.e. $\underline{T}_1 = \underline{N}$
and $\underline{X}_2$ address a sequence of numbers, i.e. $\underline{T}_2 = \underline{A}$ and each element of the sequence is type $\underline{N}$.
To form in $\underline{X}_3$ the sum of the elements addressed by $\underline{X}_2$:

```
        X3   LOAD   0
L1:     X3   ADD    X2
        X2   JNL    L1
```

Note: The "loop" consisting of the last two instructions is repeated until $\underline{X}_2$ is "last", i.e. cannot be modified by unity; on exit from the loop $\underline{X}_2$ is null* and the sum is in $\underline{X}_3$.

(ii) From the initial conditions (i), to find the first element of $\underline{X}_2$ equal to p:

```
        X3   COPY   X1
L1:     X3   SUB    X2
        ZE   J      L2
        X2   JNL    L1

L2:  ...
```

Note: At $\underline{L}_2$, if $\underline{T}_2$ is $\underline{A}$, then $\underline{X}_2$ points to a sequence beginning with the value p. Otherwise, $\underline{X}_2$ is null*, indicating that no element equal to p was found.

(iii) Suppose $\underline{X}_1$ addresses a table of numbers, the first element of which indicates how many elements are defined at any instant. The defined elements, if any, follow the first. It is required to write a subroutine, using a return link in $\underline{X}_3$, which will use the value in $\underline{X}_2$ to search the table, inserting the new value if it is not already present, and return with its index

*See page 12.

relative to $\underline{X}_1$ in $\underline{X}_2$.

```
        X4    COPY   X1
        X1    LIM    X1
L1:     X1    JL     L2
        X5    LOAD   X1
        X5    SUB    X2
        NZ    J      L1
        UN    J      L3
L2:     X1    COPY   X4
        X1    ADD    1
        X1    MOD    X1
        X1    MV     X2
L3:     X1    MEM    X4
        UN    J      X3
```

$\underline{X}_1$ addresses defined elements, if any

Test for equality

Insert new element

Find index of element

Exit via link

Interpretation of code segments

Meaning is assigned to a sequence of instructions by saying what each instruction does, and how its successor is determined. The result of applying any function can be inferred from the addressing rule and Table 1, with the provisions that a <numeral> represents a NUMERIC element with appropriate value, and a <label> represents a CONTROL element pointing to the instruction on the line, if it exists, with that label as first component. For example, the instruction:

$\underline{X}_2$ MOD 3

requires that $\underline{T}_2 = \underline{A}$, and automatically satisfies the requirement that the second operand have type $\underline{N}$. The action is to modify $\underline{X}_2$ by 3, and replace $\underline{X}_2$ by the resulting address.

Sequencing rule for BLM

(i) After any completed <action> instruction, the next to be obeyed is the one that follows, if it exists, in the definition;

(ii) After any <control jump>, the next instruction obeyed is selected by applying A/F to the second argument, if the jump is taken; otherwise, as for (i).

What happens when arguments are not of the correct type, or when an invalid result is produced, or when there is no "next" instruction, must be stated as part of the system definition. For present purposes, it is sufficient to recognise that such events will lead to a relatively slow interpretation of the instruction by stored routines.

Structural definitions

It remains to provide a means of "creating" the stored sequences on which a program operates, and attaching them to the process base by means of ADDRESS elements. The method of doing this is by means of a syntactic device which is turned into a request to the operating system to supply a certain amount of storage space. The syntactic form can also indicate the initial values of the stored elements: for instruction sequences, this is essential; for data, it is a convenient option.

<structure> :: = [<numeral>]|(<value-list>)|SEGEND

<value-list> :: = <value>|<value-list>,<value>

<value> :: = <structure>|<signed numeral>|Ω

where Ω is "undefined" or "null", and is represented by a type C element pointing to a monitoring routine.

Representation of structures

A structure is represented by an ADDRESS or a CONTROL element. If it is defined by [<numeral>] then the ADDRESS points to a sequence of null values, whose INDEX is given by <numeral>; if it is defined by (<value-list>), then the ADDRESS points to a sequence of values, as many as appear in the list, each represented by an ADDRESS of a set, a NUMERIC value, or null; if it is defined by , then it is represented by a CONTROL element pointing to the first instruction.

A <structure> definition can legitimately be used as second argument in any context where an ADDRESS element is meaningful, i.e.

<second argument> :: = <base>|<signed numeral>|<structure>

Examples

```
        X2    COPY   [99]
```

creates a set of 100 elements, addressed by X_2;

```
        X4    STO    ([3],[3],(0,0,1))
```

places the address of a sequence of three addresses in the position addressed by X_4, assuming T_4 = A.

```
        X6    COPY   SEG
L1:     X1    ADD    X2
        X2    JNL    L1
        UN    J      X3
        END
```

places the CONTROL element equivalent to L_1 into X_6.

5. System Development

We have reached the point of coding useful calculations for the BLM, on the assumption that the program and its intermediate results lie within the capacity of the machine. Apart from this raw capability, one of the essential requirements of a computer is to be able to use existing <structure>'s as components of new programs. Such is entailed by all permanently resident "system" programs and, indeed, in the assembly and use of any complex information structure.

Dennis (ref. 2) has summarised the requirements of program modules (i.e. segments) if a system exhibits programming generality, namely:

(i) to create information structures of arbitrary extent;
(ii) to call on procedures with unknown requirements for storage;
(iii) to transmit information structures of arbitrary complexity to a called procedure;
and (iv) to share information structures among computations.

It is easy to see that the BLM has these properties, since (i) all structures are defined dynamically; (ii) no assumptions are made concerning future space requests; (iii) ADDRESS elements can be used to transmit precisely delimited structures between one stage of computation and another; and (iv) we admit the possibility of elements in different bases pointing to the same stored sequences.

It is equally easy to see that a von Neumann machine does not have them. However, for certain rigidly defined problems it does provide a standard of performance which cannot be ignored. Ideally, one would like to treat "generality" as a system component, like storage of one form or another, and buy just as much as suits a particular mode of operation. In practice, that is difficult to achieve, but it also turns out that all generality does not have to be forfeited to compete with a conventional machine, even in a narrow context. Dennis' requirements are met by all BLM program segments, in conjunction with specializations in other structures which remove the inefficiencies of the model so far presented.

In the following subsections, some of the alternatives in key design areas are outlined, and their effect on the elementary picture of the machine is noted.

5.1 Refinements in type coding

Each stored element admits subcategories of its type, aimed at increasing the density of information, the rate of processing, and the sensitivity of the machine to certain control states. Thus, NUMERIC elements may be represented as integer or floating point, and possibly complex, Boolean, or multiple precision. In each case the element carries its full type code, which is used to modulate the functions applied to it so that, for example, sub-type conversion is handled automatically between integer and floating point representations.

ADDRESS elements are developed by:

(i) recognising that in practice most data sequences are homogeneous, i.e. identical in type, so that the type code can be removed from the elements and stored in the addresses which refer to them;

(ii) inferring the size of an element from its type, thus allowing packed, homogeneous "character" sequences, and low-precision integers;

and (iii) introducing a "protection" bit, which prevents an address from being used to change the sequence it describes.

The above developments have no effect on access to information, though the <structure> definition must be extended to allow the coder to specify homogeneous sets of given type and protection status. However, they do complicate the Auto Store convention to the extent that a result, although valid, might not fit into its intended destination, and a flexible method of detecting such situations is needed. Detection of a "protection" bit by A/S leads unconditionally to function monitoring.

The CONTROL subtypes select various independent modes of execution, i.e. monitor on invalid result, A/S overflow, external interrupt, or change of control sequence. The definition of <label> and can be adapted to allow preset control modes; a MODE function must be introduced (in group C) to allow dynamic control; in other respects, treatment of modes is conventional. (Condition codes are regarded as a control subtype: they normally change when a jump is taken).

5.2 Program structure

System economics commonly dictate that more than one process be "in execution" at a time, though some will be of a special form associated with peripheral transfers, and others will be queuing for shared resources. Associated with each process is a base, which in turn delimits its access to the rest of the store. In order to run two concurrent processes independently it is sufficient that the areas they access should overlap only in code segments, which are invariant, or protected data structures. This condition is not satisfied by interacting processes, in particular by I/O operations, and it is then necessary to provide explicitly for the exchange of structures between processes.

The crucial characteristic of a multiprocessing machine is thus the ease with which structures can be detached from one region of store and attached to another. In the BLM it has been assumed that this is an important quality of the system, and points of attachment have been minimised by building information structures in tree-like form. It is in this context that codeword sets assume their importance as protected sequences of ADDRESS or CONTROL elements at the branch points of the tree. A "subtree" can be detached by altering its codeword and any other addresses which point into it. It is thus desirable to minimise the number of unprotected tagged elements in a program; it happens that most existing high level

languages are already restrained in this respect, relying heavily on the tree representation.

5.3 Interconnection techniques

Communication between procedures, and between concurrent processes, depends on the interpretation of base identifiers $\underline{X}_i$ in code segments (without loss of generality the labels $\underline{L}_i$ can be restricted to cross-referencing within the segment).

Each process base is represented by a sequence of tagged elements. A relation is established between identifiers $\underline{X}_i$ and indices x_i in the base; we write:

$$x_i = \psi(\underline{X}_i)$$

where ψ is normally determined by a symbol table for that process. In practice, certain $\underline{X}_i$ have the same indices for all processes: they include the "registers", information about the process status, and the address of the symbol table itself. We can replace such $\underline{X}_i$ by $\psi(\underline{X}_i)$ wherever they occur in the representation. In general, there will be a residue of identifiers whose corresponding indices vary from base to base, for which the substitution cannot be made without violating Dennis' requirement (iv), and which require dynamic interpretation of the identifiers, i.e. evaluation of $\psi(\underline{X}_i)$ for each argument.

At first sight, this seems not only time-consuming but also to imply a complex instruction format to allow both fixed and symbolic base references. However, it is possible to copy a symbolic element to a fixed position for processing, so with the aid of functions of the form:

$$x_i \quad \text{COPY} \quad \underline{X}_j$$

and

$$\underline{X}_j \quad \text{COPY} \quad x_i$$

it is necessary only to represent the other operations with fixed base elements as arguments. The time factor remains, but it should be noted that references of this sort are relatively rare. Typical examples are entries to diagnostic routines, or access to files:

most identifiers used in programming depend only on fixed base elements.

It remains to examine how reference is made to other bases. The first time an identifier is encountered in a process, a "free" base position is assigned to it. At the same time, it is possible to search symbol tables in certain other process bases for that identifier, and to copy its value if it is found. In the experimental BLM, a parent process can supply initial values to its subprocesses by this means. It does not prevent a subprocess from writing in its "own" value for a base element at any time. In theory, the search for an initial value could be carried through several process bases, but it can be seen that the task of detachment (section 5.2) becomes more burdensome as the potential for interconnection increases.

5.4 Use of registers

We have already seen that certain base elements have fixed index values. If we choose small indices, then the code operating on them can be compressed; if, in addition, we retain those elements in fast storage, then two of the most common properties of central registers have been achieved, and many programming regimes can use them effectively to increase system performance.

The third reason for having registers is to provide special logical or arithmetic functions which are not explicitly revealed in the programming language. They include, for example:

(i) the ADDRESS of the process base;

(ii) the sequence CONTROL element;

(iii) a stack ADDRESS, permitting temporary evacuation of fixed base positions;

and (iv) residual NUMERIC information after arithmetic functions, allowing for the recovery of remainder after division, and for coding multilength operations and shifts.

In general it is preferred to keep a uniform set of base elements, at the expense of introducing new function or type codes to take advantage of special registers. Thus, stacking operations can be implied by DUMP and UNDUMP orders, and by elaborating the syntax of <instruction>'s to allow arguments to be specified by recursively defined expressions. Similarly, multilength operands can be specified by one of the NUMERIC subtypes. The advantage of doing this is that there remains a wider choice of implementation options than if a particular register structure had been assumed.

Apart from the above considerations, ADDRESS elements can be designed so that storage may be remapped directly in a variety of ways to increase system performance, one of which is discussed in ref. 1, p. 69.

5.5 Escape actions

In the preceding discussion we have assumed the presence of monitoring routines to recognise undefined elements, to resolve illegal type combinations, and to take various actions conditional on the state of a computation and its control mode. In a sense, the need to supplement a calculation by interpretive routines suggests that the machine is incompletely specified; but this is a deliberate choice in the design, since the complete definition of a function may depend on circumstances local to its point of application. For the BLM, the main requirement is to offer a clean breakpoint between any incomplete instruction and its interpretive continuation.

Some insight into the Basic Machine can be gained by asking what would constitute a "complete" set of escape paths. In other words, how restrictive are the elementary definitions of sections 2, 3 and 4? For NUMERIC elements, we have already indicated that further discrimination can be introduced by refining the type coding: the degree of hardware interpretation is left open. The main constraint is on size, because all tagged sequences must allow for the maximum size of element, and this is likely to be uneconomical over

a varied class of problems. There is a basic choice to be made between introducing some form of 'size escape' which allows standard coding to be applied to outsize elements, and falling back on explicit coding for multilength work. The former alternative is more attractive, because it keeps the input language free from type declarations. There are other NUMERIC elements, such as Ω, which defy any explicit representation, and are handled by CONTROL elements pointing to interpretive code: they are detected in group A functions after failure on the primary type code (C).

The main restriction on ADDRESSes is in assuming they point to finite, ordered sequences with, in effect, static indices, i.e. the position of any element relative to the first stays constant. An upper limit of 2^{16} elements (of any size) has been imposed without difficulty. However, one can envisage sets of elements ordered by magnitude, frequency of use, or any other rule of enumeration, which could be valuable programming aids. Some work has been done on such "implicit addresses" in the experimental BLM. They have been used to address devices, files, and processes, all of which require control by special system functions. The more general mechanism which is, in effect, a disguised form of function call, has not yet been applied.

Diagnostic Reports

The standard system action on most escapes is to abandon the control sequence and return to the routine which initiated it. If the programmer has other plans, he can supply a routine which will take corrective action and continue the calculation. One of the advantages of dynamic type checking is that many programming faults are brought to light quickly. At the same time, diagnostic information can be printed out in terms of the current process structure and base names X_i (another reason for keeping to a 'tree' as closely as possible), thus allowing the coder to pinpoint his error easily. In designing the BLM system, a fairly rapid interaction

between each user and his process, in terms of its current structure, has been assumed to be the normal mode of operation.

6. References

1. J. K. Iliffe — "Basic Machine Principles" American Elsevier (1968)

2. J. B. Dennis — "Programming Generality, Parallelism, and Computer Architecture" Project MAC Computation Structures Group Memo No. 32 (see also IFIP Congress Proceedings 1968).

Table 1: BLM functions

Function		Operand Types		Result	Notes
Mnemonic	Group	1st	2nd		
ADD	A	N	N	x+y→x	a
AND	A	$\overline{N}$	$\overline{N}$	x∧y→x	a
COPY	B	any	any	Y→X	
DIV	A	N	N	x/y→x	a
INDEX	D	any	$\overline{A}$	index(Y)→X	a
J	C	any	$\overline{C}$	link→X; y→control	
J	C	<condition>	$\overline{C}$	if <condition>, y→control	
JGE	C	N	$\overline{C}$	x-1→x; if x≥0, y→control	a
JL	C	$\overline{A}$	$\overline{C}$	X MOD 1→X; if $T_x \neq \underline{A}$, y→control	a
JLT	C	$\overline{N}$	$\overline{C}$	x-1→x; if x<0, y→control	a
JNL	C	$\overline{A}$	$\overline{C}$	X MOD 1→X; if $T_x = \underline{A}$, y→control	a
LIM	D	$\overline{A}$	$\overline{N}$	y→index(X)	b
LOAD	B	any	any	y→X	c
MEM	D	$\overline{A}$	$\overline{A}$	Index of X in Y→X	a
MOD	D	$\overline{A}$	$\overline{N}$	Move pointer (X) through y positions, and decrement index (X) by y	b
MPY	A	N	N	x*y→x	a
MV	A	$\overline{N}$	$\overline{N}$	y→x	a
NEQ	A	$\overline{N}$	$\overline{N}$	x≠y→x	a
NOT	A	$\overline{N}$	$\overline{N}$	⌐y→x	a
OR	A	$\overline{N}$	$\overline{N}$	x∨y→x	a
SC	A	$\overline{N}$	$\overline{N}$	$x*2^y \rightarrow x$	a
STO	B	any	any	Y→x	d
SUB	A	N	N	x-y→x	a

Notation:
- X : First argument (base position)
- Y : Second argument (base position)
- x : First operand (using A/F)
- y : Second operand (using A/F)
- T_x : Type of X
- T_y : Type of Y

Notes:
- a : Condition codes set by result
- b : Invalid result if y>index(X)
- c : Fails if $T_y = \underline{C}$
- d : Fails if $T_x = \underline{C}$

"control" refers to the instruction sequence pointer

"link" is the $\underline{C}$ element pointing to the next instruction in sequence

"→" indicates "replaces the element at"

Table 2 : Condition Mnemonics	
GE	Result≥0
GT	Result>0
IR	Invalid result
LE	Result≤0
LT	Result<0
NZ	Result≠0
UN	No condition
VR	Valid result
ZE	Result=0