Abstract Machine Modelling to Produce Portable Software—A Review and Evaluation

M. C. NEWEY
University of Stanford, Stanford, U.S.A.

P. C. POOLE
Culham Laboratory, United Kingdom Atomic Energy Authority, Abingdon, England

AND

W. M. WAITE
University of Colorado, Boulder, U.S.A.

SUMMARY

This paper discusses the use of abstract machine modelling as a technique for producing portable software, i.e. software which can be moved readily from one computer to another. An overview of the principles involved is presented and a critical examination made of three existing abstract machines which were used respectively to implement a macro processor, a text editor and a BASIC compiler.

KEY WORDS Abstract machines Portability Adaptability Macro processor STAGE2

INTRODUCTION

The purpose of this paper is to summarize our experience with abstract machine modelling as a technique for producing portable software. We shall present an overview of the principles involved, and then discuss three concrete examples. Our emphasis throughout will be on the design decisions: why they were made in a certain way, what performance resulted from them and how they should be altered to improve that performance.

Abstract machine modelling is based on the concept that the fundamental operations and data types required to solve a particular problem define a special purpose computer which is ideally suited to that problem. The algorithm for producing the solution can then be encoded as a program for this 'abstract machine model'. In order to obtain a running version, the abstract machine model is realized on an existing computer by implementing its basic operations and data types.1, 8

The key to the whole technique is the design of suitable abstract machines. Three points must be taken into consideration:

1. The convenience of the abstract machine language and its suitability for expressing the particular algorithm for which the machine is designed.
2. The relationship between the abstract machine language and the structure of available computers.
3. The limitations imposed by the tools used to convert the abstract machine language to a language for the real machine.

Received 24 September 1971
Revised 10 January 1972

© 1972 by John Wiley & Sons, Ltd.
In this paper we shall make a critical examination of three existing abstract machine designs in the light of these points. Before discussing any of the machines in detail, we shall present general implementation techniques for abstract machine models. Each of these techniques will be applied to the specific designs of subsequent sections.

IMPLEMENTATION TECHNIQUES

We favour a macro processor as the tool for realizing an abstract machine model on a real computer. The merits of this approach have been presented\textsuperscript{2-4} and we shall not argue them at length here. In our view, the most significant advantage of a macro processor is the ease with which changes can be made to the definitions relating the abstract machine to the real one, and consequently the degree of control which can be maintained over the generated code. We believe that this advantage outweighs the space and time penalties of the macro generation itself.

The problem to which we address ourselves is a special one: that of transferring working software to a new computer. We can assume that the source program is syntactically correct and therefore no extensive translation time diagnostics are required. Source statements can be in a fixed format if this will significantly aid the macro processor. The text can be organized to guarantee that declarations precede program text and that all attributes of a variable are defined before that variable is used. Thus none of the traditional arguments of compiler versus macro processor apply to the case in which we are interested.

There are two basic approaches to the transfer of software. The most commonly described, known as a half bootstrap, is to use tools currently available on one machine to implement the software on another. Our experience has been that in many cases this approach involves insurmountable communication problems. Peripheral incompatibility, differing character sets and distance between the machines, all conspire to put the implementation beyond the patience of mortal man. The second approach is called a full bootstrap. All work is done on the target computer, beginning from symbolic source text.\textsuperscript{1,2} A full bootstrap cannot be carried out if the resources of the target computer are not sufficient to run the macro processor with the necessary definitions.

We believe that all portable software should be designed for implementation by the full bootstrap process. Such a design in no way precludes the possibility of employing a half bootstrap. If, on the other hand, the system is designed for a half bootstrap, then a full bootstrap is usually impossible.

We aim to produce software which is not only portable but which is also economically viable. This means that we must consider the problems of optimization as well as simple code generation. In general, different criteria of optimality will be used for different programs. A module which is heavily used should be fast, while infrequently activated code should occupy as little space as possible. By changing the macro definitions, one can tailor the generated code to specific criteria without altering the source program. Later in this paper we shall present a detailed discussion of such tailoring which was carried out on the TEXED machine.

When an abstract machine is designed for a specific problem, the basic operations suitable to that problem become the instruction set of the abstract machine. Most problems will suggest a number of specialized operations which could possibly be implemented quite efficiently on certain hardware. The designer must balance the convenience and utility of these operations against the increased difficulty of implementing an abstract machine with a rich and varied instruction set. One way of balancing the requirements for portability and
for a close modelling of the problem is to employ a hierarchy of abstract machines. Instead of realizing the initial abstract machine ($A_1$) directly on a real computer, design a second abstract machine ($A_2$). The operations of $A_1$ are then defined in terms of $A_2$ operations. Such a definition is independent of the realization of $A_2$ itself and hence $A_1$ may be realized by realizing $A_2$. The hierarchy can be carried to any depth by defining $A_3$, $A_4$, and so on. The base machine of the hierarchy $A_k$ is then realized on a real computer.

The purpose of a hierarchical realization is to ease the initial implementation of the abstract machine on a new computer by providing a simple base machine. With additional effort, optimizations of either speed or space can be carried out by skipping some steps of the hierarchy. For example, suppose that $A_1$ provides a string move instruction. $A_2$ might implement this operation in terms of full word fetch and store operations, plus shift and mask instructions to handle end conditions. On System/360, $A_2$ could be by-passed and the string move implemented directly. Thus it is possible to take advantage of specialized hardware instructions without altering the algorithm. This would not be possible if $A_1$ did not provide the string move operation.

Note that there may be other instructions of $A_1$ whose translation to $A_2$ is not by-passed. There is no compulsion to by-pass a level completely if this does not seem appropriate. Only those instructions which measurement shows to be critical need be considered. We shall discuss hierarchies in more detail later in this paper, in connection with the machine AIM1.

The macros which define an abstract machine are equivalent to the hardware of a real computer. Just as a manufacturer must test for faulty components and wiring errors, the implementor of an abstract machine must test for macro coding errors. To aid the implementor in this task, the designer must provide a series of test programs. These ‘engineering tests’ are indispensable. They must be designed to verify every macro, pinpointing errors as carefully as possible. Applications software for the abstract machine cannot provide the discrimination required to detect macro coding errors effectively.

Our test programs are similar to conventional hardware tests. We have no ‘customer engineer’s panel’, so we must rely on the normal I/O mechanisms to report any failures. The first test therefore simply reads and prints one line. Each subsequent test reads a line describing a failure and then checks to see whether that failure occurred. If so, the line is printed. Otherwise the next test is begun.

The designer must select a minimal set of operations, test these and then use them in the testing of other operations. Selection of the test sequence and overall strategy depends upon the organization of the abstract machine.

FLUB

Design rationale

FLUB (First Language Under Bootstrap) is an abstract machine designed specifically for the task of constructing STAGE2, the macro processor which we use to realize all other abstract machine models. STAGE2 deals with three types of data: strings, trees and integers. Thus the basic organization and operations of the FLUB machine must be suited to manipulating these data types.

The representation chosen for a tree was a slight modification of that presented by de la Briandais and Nishimura. This representation determined the composition of the FLUB word. Figure 1 shows a tree containing the strings CAT, COT and DOT set up in a
sequentially addressed store. Notice that each word is divided into three fields. The FLG (flag) field contains indicator bits, the VAL (value) stores one character and the PTR (pointer) is used as an address.

<table>
<thead>
<tr>
<th>Address</th>
<th>FLG</th>
<th>VAL</th>
<th>PTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>C</td>
<td>107</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>A</td>
<td>104</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>103</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>0</td>
<td>O</td>
<td>0</td>
</tr>
<tr>
<td>105</td>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>106</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>0</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
<td>O</td>
<td>0</td>
</tr>
<tr>
<td>109</td>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Continuation of CAT)  
(End of CAT)  
(Continuation of COT)  
(End of COT)  
(End of DOT)  

*Figure 1. Representation of a tree*

Given the structure of the FLAG word, a string is easily represented as a linked list of words. The VAL field of each word contains a character of the string, and the PTR field addresses the word containing the next character. Any substring of such a string may be specified by a word whose PTR field addresses the first character of the substring and whose VAL field contains the length of the substring. The VAL field must thus be long enough to hold either the largest character or the length of the longest substring.

There are several ways to represent an integer in a FLAG word: use the full word, use a combination of fields or use a single field. At the moment there is no clear reason for choosing one of these representations over the others. Let us therefore defer the question temporarily and consider the operations required on each field.

The FLG field is used as an indicator, and thus operations which test and set this field are important. Because the VAL field is used to hold a string length, addition and subtraction of VAL fields must be possible. A test for equality of two VAL fields is required to permit character matching. Addition and subtraction must be possible on the PTR field to provide for sequencing through a tree. Because the PTR field is long enough to contain an address, it can hold the return address for a subroutine call.

Considering the operations, it seems reasonable to use the PTR field to represent an integer. This field is already the largest, and addition and subtraction operations are defined for it. Thus only multiplication, division and a test for relative magnitude must be added.
The length of the pointer field determines the range of integers allowed in a particular implementation.

Some form of input/output operations are required. Since a string is represented by a linked list of characters, character-by-character I/O operations seem reasonable. Characters are always stored in VAL fields, so that the operations read one character into a VAL field and write one character from a VAL field. The only characters which have some intrinsic significance are the decimal digits. In order to implement arithmetic expression evaluation, it must be possible to convert strings of digits into integers and vice versa. This can be done if the digits are represented by successive integers. By subtracting the representation of the character 0 from the representation of any digit, we obtain the integer value of the digit.

STAGE2 processes lines of input text to produce lines of output text. It must therefore have a means of detecting the end of a line on input and forcing one on output. This can be done by defining a special character, the carriage return, which does not occur in the normal character set. We therefore represent the carriage return character by -1 and restrict the representation of normal characters to non-negative integers.

The character operations discussed above were the only form of I/O provided in the initial design of FLUB. They are sufficient if STAGE2 uses a fixed set of I/O devices. After two years' experience with STAGE2, we realized that a more flexible I/O interface was necessary. The currently distributed version of STAGE2 permits the user, by suitable coding of his macros, to obtain input from or direct output to any one of nine I/O devices. The requirements which this feature placed on the FLUB machine should be examined in the light of the second criterion for abstract machine design: the relationship between the abstract machine language and the structure of available computers.

I/O hardware and/or operating systems of most computers transmit records to and from peripheral devices. This means that buffer packing and unpacking routines must be written to support character I/O on most computers. If characters are to be directed to a number of devices a number of buffers are necessary. Each character I/O operation must specify the device and this specification must be interpreted by the buffering routines each time a character is transmitted. Since STAGE2 does process lines of text and it is not possible to switch the I/O device during transmission of a line, this overhead can be avoided by defining record I/O operations as well as character transmission. The character I/O operations move information into and out of a line buffer; the record I/O operations transmit data between the line buffer and the peripheral devices.

Each record I/O operation must specify the peripheral device with which it communicates. Since STAGE2 only permits the use of nine peripherals, the VAL field of a FLUB word can be used to specify a peripheral number. The FLG field of the same word is set by the operation to reflect what happened (normal completion, end-of-file detected, illegal operation, permanent error).

In order to introduce record transmission and the line buffer, we must alter the character output operation. The problem is output of long lines. When a character output operation is transmitting information directly to a device, the user need not be concerned with the length of the line being written. If the peripheral device has a fixed record length, then its buffering program simply writes a record whenever enough information is available. When the line buffer is used, however, record I/O operations must be issued by the program whenever the line buffer fills. We did not wish to fix the length of the line buffer, so we made the character output operation set the FLG field of the word from which it extracts the character. If the character can be placed into the line buffer then the FLG field is set
to 0. Otherwise, it is set to 1. The user can thus test whether the character was accepted by the line buffer. In order to make this a general test for the end of a line we also set the FLG to 1 when a \(-1\) (carriage return) is output.

The other FLUB operations (integer arithmetic, conditional branching, etc.) are almost universally available on current computers. Therefore as far as operations are concerned the match between the abstract machine and the real one is good. It is quite uncommon, however, to find a computer whose words are partitioned into three fields of the type making up the FLUB word. Thus the match is poor for our data structure.

There are two ways in which our data structure can be implemented: we can pack the fields of a FLUB word into one or more words of the target computer, or allocate one target computer word to each field. The first choice will minimize the space required to store information but will result in large overheads to unpack and repack the fields for each operation. Opposite results (low overhead, maximum space) can be expected from the second choice.

We escaped this dilemma by providing the FLUB machine with a small set of 'registers', on which almost all operations take place. These registers can be implemented on the target computer with one word allocated to each field. Since the number of registers is small the space requirements are not prohibitive. The memory can then be packed to conserve space. Since memory is not accessed by most operations the overhead of packing will not be excessive.

A register organization requires memory/register transfer instructions and a way of specifying the memory address. Since the PTR field of a register can hold an address, a memory/register transfer instruction has two registers as operands. One participates in the transfer (either receiving or transmitting information) and the PTR field of the other specifies the memory location. Hence all access to the FLUB memory is indirect.

Each FLUB memory word may occupy several locations in the memory of the target computer. In the implementation of STAGE2 for System/360, for example, each FLUB word is eight bytes long; on the CDC 6400, each FLUB word is a single machine word. We have chosen to interpret a PTR field which addresses the FLUB memory as containing the \textit{target} computer address of the FLUB word. The addresses of successive words in the FLUB memory thus might not differ by 1. In order to increment the contents of a PTR field so that it addresses the next word in the FLUB memory, we must add the number of target computer memory locations per FLUB word. On System/360 this means that a PTR is advanced to point to the next word in FLUB memory by adding eight.

When a subroutine jump is made the return address must be saved. Examination of real computers reveals two basic methods:

1. Place the return address in a register (IBM System/360, 7040/7090, 1401, 1410, ICL KDF9, 1900).

2. Place the return address in the called subroutine (IBM 1620, 7040, CDC 3000, 6000).

The first method is preferable for the FLUB machine because it does not require a store operation into the program area, and is therefore applicable to a wider class of real computers. It also avoids the problem of storage reservation at the head of the subroutine and allows multiple entry points with no special coding. The subroutine may, of course, store the register containing the return address and then recall it before the return is taken.

The final consideration in the design of an abstract machine is the limitation imposed by the tools used to realize it. FLUB is unique in this regard, because it implements STAGE2. STAGE2 is therefore not available for the realization of FLUB. Instead we use a trivial
macro processor called SIMCMP. SIMCMP is only capable of handling simple substitution macros whose parameters are single characters. Unlike STAGE2, it has no internal memory or conditional expansion facilities and can only make one pass over the input text.

Because of the limitations of SIMCMP we restrict the operands of FLUB statements to single characters or fixed-length strings of characters. Two types of operand are required: register names and program labels. We have therefore given the FLUB machine thirty-six registers and named them A–Z and 0–9. All program labels consist of two digits. Constants cannot be used as operands in FLUB. Instead, the registers named 0–9 are initialized by an external process to the values shown in Table I. Thus the instruction

$$\text{PTR A} = \text{A} + 1$$

increments the PTR field of register A by 1.

<table>
<thead>
<tr>
<th>Register</th>
<th>FLG</th>
<th>VAL</th>
<th>PTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table I. Initialization of digit registers

The lack of constant operands makes the production of error messages difficult. We have had to define special operations to write these messages. They behave like record output operations, but transmit the message rather than the contents of the line buffer. As with record output, the VAL field of a register specifies the device and the FLG field of the same register is set to reflect the completion condition.

A complete list of FLUB statements is given in Figure 2. A single apostrophe represents a register name and may be replaced by any letter or digit. Two successive apostrophes represent a program label and may be replaced by any two digits. The four apostrophes in the message output operation represent the message name. They may be replaced by CONV, EXPR, FULL or IOCH.

**Implementation**

It is difficult to discuss the implementation of an abstract machine without considering some real computer as well. We shall therefore use a System/360 implementation to provide a concrete frame of reference. Modifications for other hardware will be pointed out as necessary.

The first question which must be answered is how much storage to allocate for each field of the FLUB word. Table II summarizes the information which each field must contain. Most of the quantities in Table II are determined by the choice of a target computer (this is not strictly true as will become apparent below). The maximum string length
Data transfer operations

Register—Register
  FLG ='
  VAL = PTR
  PTR = VAL

Register—Memory
  GET ='
  STO ='

Integer arithmetic operations

VAL field
  VAL = +'
  VAL = -'

PTR field
  PTR = +'
  PTR = -'
  PTR = *
  PTR = /'

Control operations

Unconditional
  STOP
  TO '
  TO ' BY'
  RETURN BY'

Conditional
  FLG field
    TO ' IF FLG ='
    TO ' IF FLG NE'

  VAL field
    TO ' IF VAL ='
    TO ' IF VAL NE'

  PTR field
    TO ' IF PTR ='
    TO ' IF PTR NE'
    TO ' IF PTR GE'

I/O operations

Character transfers
  VAL = CHAR
  CHAR = VAL

Record transfers
  READ NEXT'
  WRITE NEXT'
  REWIND'
  MESSAGE ' TO'

Pseudo operations

Program label definition
  LOC '

End of text
  END PROGRAM

Figure 2. FLUB statements

and the range of integer expressions constructed by the STAGE2 user, however, depend upon the applications envisaged. Specifying the field sizes will place limits upon these quantities and it is important that the implementation does not collapse when the programmer inadvertently exceeds these limits.

The internal representation of a character can be chosen freely by the implementor, subject to the constraint that the characters 0–9 are represented by successive integers.
It is generally simplest to use the normal internal representation provided by the target computer. On System/360, this choice means that each character is represented by a non-negative integer less than 256. (The digits 0–9 are represented by the integers 240 to 249 and hence the constraint is satisfied.) Eight bits are required to hold a single character, so that the VAL field must be at least this long.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLG</td>
<td>Integers 0, 1, 2 or 3</td>
</tr>
<tr>
<td>VAL</td>
<td>Non-negative integer representing a character</td>
</tr>
<tr>
<td></td>
<td>Non-negative integer representing the length of a string</td>
</tr>
<tr>
<td></td>
<td>—1</td>
</tr>
<tr>
<td>PTR</td>
<td>Address in the FLUB data memory</td>
</tr>
<tr>
<td></td>
<td>Address in the FLUB program</td>
</tr>
<tr>
<td></td>
<td>Value of an integer expression constructed by the user of STAGE2</td>
</tr>
</tbody>
</table>

The VAL field must also be capable of holding the integer —1. This is the only negative number which is ever stored in the VAL field. We can therefore guarantee that the contents of the VAL field is always positive if we bias it by adding 1. If the topmost character (represented by 255) never occurs, then biasing does not require another bit.

If we were to settle on an eight bit VAL field, using bias by 1 to ensure that the contents is non-negative, then the maximum allowable string length would be 254. Depending upon the intended application of STAGE2 this might be quite adequate. If we envisage the need for longer strings the number of bits assigned to the VAL field could be increased.

The PTR field must be able to hold an address and on System/360 this could be taken to imply a length of twenty-four bits. However, by using base registers and constraints on memory addresses this length can be reduced. Here is a case where the choice of a target computer does not fix the size of an address. A choice is still open to the programmer. He may trade a saving in space for additional complexity in the generated code. Let us examine how this trade could be made.

Together the FLG and VAL fields require ten bits. If four bytes were allocated to a FLUB word this would leave twenty-two bits for the PTR field. By suitable choice of origin every FLUB word could be guaranteed to have an address whose last two bits were 0. Thus the last two bits of the address carry no information and may be dropped, bringing the length of an address down to twenty-two bits. The only operations which actually use the PTR field to access memory are GET and STO (Figure 2), and hence they are the only two which must adjust its value.

Program addresses may also be stored in PTR fields. They are manipulated by the operations TO "BY" and RETURN BY " and represent return addresses for subroutines. By inserting the pseudo operation CNOP into the definition of TO "BY", it is possible to force the return point to a full word boundary. The return address will therefore always end in two zero bits. This wastes a maximum of two bytes per subroutine call, and therefore does not represent a serious space penalty. TO "BY" would shift the address right by two bit positions before placing it in the PTR and RETURN BY " would shift it left after extracting it.

The value of an integer expression constructed by the user of STAGE2 might be either positive or negative. This means that one bit of the PTR field must be reserved for a sign,
which implies that either the size of the field should be increased or the range of addresses restricted. Increasing the size of the field would require either an increase in the word length or a decrease in the size of VAL. Neither alternative is attractive although either is possible. If we restricted the length of an address to twenty-one bits only half of the possible address space of System/360 could be made available. In practice, however, this will not prove a serious limitation. Let us therefore assume the following field lengths:

- **FLG:** two bits
- **VAL:** eight bits (maximum string length = 254)
- **PTR:** twenty-two bits ($-2^{22} < \text{value of user constructed expression} < 2^{22}$)

It is important to realize that there is nothing sacred about these lengths. They represent a reasonable compromise in a System/360 implementation which is expected to handle short strings and relatively large values of user constructed expressions.

Most FLUB instructions operate on one field of a register independently of the other fields of the same register. Effectively, each register is treated as three separate entities. It is therefore convenient to consider a field rather than a register as the basic unit of information for most instructions. If this is done, the number of instructions which must be implemented is decreased. For example, it is no longer necessary to consider the addition of two VAL fields and the addition of two PTR fields as separate instructions. Both are simply instances of an instruction which adds the contents of one field to that of another field and leaves the result in a third field.

In order to reduce the number of instructions in this way, all fields must be represented identically. The amount of space wasted by this representation is small and non-existent on some computers. We have already pointed out that the fields of a register should be held in unpacked form to avoid overheads. This implies that each field must occupy at least one storage location. Only if each location has insufficient storage for a PTR field does the uniform representation waste space.

Space will certainly be wasted by a uniform representation on System/360. Each FLG field could be stored in a single byte, since byte move and compare operations are available. Addition and subtraction must be performed on the VAL field so that it is more conveniently represented as a half word. Only the PTR field actually requires a full word. This mixed representation of the registers fills a storage area of 152 bytes, while a uniform representation uses 432 bytes. Neither figure represents an excessive amount of storage so that the decision would probably be made on other grounds: half word operations take somewhat longer to execute than their full word equivalents but byte move and compare instructions are slightly faster than the sequence of full word operations required to achieve the same effect.

Another possible organization is to use bytes, half words and full words respectively for the fields of a register, but to place all fields of a single register into the same double word. This would waste one byte per register but it would allow the same representation of data in registers and memory. (Each memory word would be twice as long as in the previous example.) Such an implementation would increase the speed of STAGE2 by eliminating the packing and unpacking associated with GET and STO, at the cost of halving the available memory space.

Because the fields of a FLUB word are generally represented differently in registers than in memory, it is possible to compute values which cannot be stored. For example, full word arithmetic operations on the PTR field can create numbers larger than $2^{32}$. Such numbers could not, however, be stored in the FLUB memory. The value resulting from
a STO–GET sequence would depend upon the exact implementation of these operations. Thus it may be possible for the user of STAGE2 to actually evaluate an arithmetic expression correctly, even though its value exceeds the capacity of the PTR field of a memory word. This is because arithmetic expressions are evaluated using a stack whose top two words are held in registers. For certain expressions, only these top two elements of the stack will be required and hence partial results will never be stored. (The final result of the expression is either tested or converted to a character string—it is not stored as an integer in a FLUB word.)

Two I/O packages have been used for FLUB record I/O operations. The first, and most widely distributed, was originally developed for the TEXED machine. It will be discussed at some length later in the paper. This package was unsatisfactory for a number of reasons, and was replaced in mid-1970. A complete description of the replacement (which is used for AIM1 as well as FLUB) is beyond the scope of this paper. Basically, each record I/O operation becomes a subroutine call which could be written in FORTRAN as follows:

```
JFLG = IOOP(JOP, JVAL, LB, 1, LIM)
JFLG = flag field of register
JVAL = value field of register
JOP = -1 for read
     0 for rewind
     1 for write
LB = base address of the line buffer
LIM = is set by a read operation to index the first free space in the line buffer,
     and contains the first free space in the line buffer before a write operation
```

The character I/O operations are simple fetches and stores. There are two variables which index the line buffer, one for reading (LBR) and one for writing (LBW). Each index is set to 1 by the corresponding record I/O operation. VAL' = CHAR moves the character indexed by LBR into the VAL field of the specified register and increments LBR. The line buffer is at least one word longer than the longest line which can be read, and READ NEXT stores -1 in the word indexed by LIM. We rely upon the user to check for this datum and hence we normally provide no check on the value of the index. Such a check could easily be provided and was used during the development of STAGE2. Once the program was checked out, however, the test was removed to improve efficiency.

CHAR = VAL' is somewhat more complex, and is often not implemented as in-line code. Figure 3 shows the logic of this operation. MAX is one larger than the length of the line to be output. Note that no carriage return is actually written into the line buffer; it is only necessary to set the limit for the write call to the standard interface.

The message operation is also implemented by a call on the standard interface. Each message may be stored separately and written directly from its storage area by specifying appropriate arguments to IOOP. Alternatively, the operation could construct the appropriate message in the line buffer and write it from there. A third possibility is to have a single 'skeleton' message and fill it out with the four characters specified in the message operation before writing it.

**Critique**

In the previous section we pointed out that a uniform representation of the fields of a FLUB register was possible. This is the key to the major shortcoming in the design. There
is no good reason for carrying the word structure required to store a tree over to all other data types.

Most strings used in STAGE2 are created and destroyed in a predictable manner and occupy blocks of contiguous words. Thus, there is no need for explicit linkage. The length of each string is given and no flags are required to mark the end. When integers are stored in memory the FLG and VAL fields only rarely contain useful information. We might therefore save considerable space in the memory by having three data types (tree nodes, characters and integers) rather than one.

```
CHAR = VAL'

Char: ≠ 1

< LBW: MAX

≥

Set LIM = LBW
Set FLG' to one

Move VAL' to the line buffer
Set FLG' to zero

Set LBW = LBW + 1.

Done

Figure 3. The character output operation
```

The saving in space would only be realized, of course, if the target machine structure took the proper form. On System/360, bytes could be used to store characters and full words to store tree nodes and integers. The length of a string could be represented by an integer rather than a VAL field, thus easing the 254 character restriction. Similarly, the limit on the value of an arithmetic expression could be raised without a storage penalty.

We would require distinct operations for the abstract machine to transmit the various data types between registers and memory. Once in the registers, however, all data would be treated uniformly. Thus only one set of arithmetic and conditional operations would be necessary. This eliminates eight operations while the memory transfers add four.

If the character data type were stored in a form compatible with the standard I/O interface, then the two character I/O operations and the line buffer could also be eliminated. The number of operands for READ NEXT and WRITE NEXT would be increased, allowing the programmer to specify data transmissions involving any area of memory.

Our measurements on the existing version of STAGE2 show that these changes would also improve its running speed. Approximately 20 per cent of the total running time is spent moving characters into and out of the line buffer, a task which could be eliminated. Another 13 per cent goes to moving character strings from one area of memory to another. Although this time could not be eliminated, it would be reduced if explicitly linked lists did not have to be set up.

On the whole, the performance of the current version of STAGE2 is adequate on medium and large scale computers. Memory size is a serious limitation on small machines.
where higher packing densities could be obtained by using three data types instead of one. Expansion times depend upon the complexity of the macros but are generally reasonable (150 source lines per minute on an ICL 4/70 when translating FORTRAN-like text to assembly code).

**TEXED**

**Design rationale**

The final form of the FLUB machine was strongly influenced by the limitations of SIMCMP. Once FLUB has been realized on the target computer, STAGE2 is available and SIMCMP may be discarded. TEXED was the first abstract machine whose design was based on the use of STAGE2, rather than SIMCMP, as the tool for realization. It was used to implement a comprehensive text manipulation system called MITEM.

At the time TEXED was designed we had not had sufficient experience with FLUB to perceive the shortcomings noted in the previous section. We felt that the basic FLUB architecture would provide a convenient machine for the implementation of a program to manipulate text. MITEM is a context editor which, in its simplest mode of operation, accepts lines of text from a READ stream, edits them according to instructions taken from the CONTROL stream and outputs the modified lines to a WRITE stream. Success or failure of the modification is reported on a PRINT stream (the READ and WRITE streams would usually be disk files in a multiaccess system; the CONTROL and PRINT streams would be allocated to a user's console). For simple copying operations, the transfers from one file to another are performed outside the memory of the TEXED machine. However, if the line has to be searched or modified, then it is placed in the TEXED memory in the form of a list. The VAL field holds a character, the PTR field a link to the next character and the FLG field a marker to indicate end-of-list. The basic editing operations such as locate, delete, replace and insert a character string are then programmed in terms of operations on lists.

The register structure of FLUB was retained but the number of operations was increased to fill in some of the gaps in the FLUB set (e.g. VAL ' = ', TO ' IF VAL ' GE ', see Figure 2). Further, since STAGE2 was available to translate TEXED, we were able to provide ourselves with a more convenient language for programming MITEM. Thus we removed the restrictions on FLUB operands to single characters imposed by SIMCMP and permitted the use of strings as identifiers for manifest and character constants. The former allowed an installation to vary some of the characteristics of the program in a simple manner; the latter provided a convenient method of describing operations on characters in a machine independent way, e.g. the instruction

\[
\text{VAL A} = \text{Z}
\]

stores the integer value of the character Z in the VAL field of register A. The register/memory transfer operations also differ from those in FLUB since they reflect the fact that the TEXED memory may be divided into an arbitrary number of arrays by declarations of the form

**DECLARE ARRAY '*'**

The first parameter is the array name and the second is the number of TEXED words to be allocated to it. Thus the operation

\[
\text{SET Y} = \text{CHANNEL(X)}
\]
stores in register Y, the word in the array CHANNEL addressed by the PTR field of X and is equivalent to the GET operation of FLUB. The reverse transfer is effected by the operation

\[ \text{SET CHANNEL}(X) = Y \]

Each operation assumes that the PTR field contains the target computer address relative to the base address of the array. On the other hand, in the operation

\[ \text{SET} \quad ' = ' \text{(VAL ')} \]

the VAL field is assumed to contain an index which may therefore have to be adjusted when the instruction is translated. The more conventional indexing operation on an array is thus permitted.

The restriction on labels was also removed in TEXED and identifiers may be used instead of two-digit integers. Further the pseudo operations

\[ \text{PROC '() and ENDPROC} \]

were introduced to delimit procedures. The first parameter in PROC is the procedure identifier and the second is a list of formal parameters specifying register fields. Entry and exit to procedures are effected by

\[ \text{CALL '()} \quad \text{and RETURN} \]

statements. The second parameter in the CALL statement is a list of actual parameters which may be register fields, manifest constants or character constants.

Two additional 'hardware' features were added to FLUB in creating TEXED. The first was a push down stack for temporary storage of registers and for the transmission of parameters. The second was a number of 'flip-flops' which allowed the program to interrogate its environment and alter its behaviour accordingly. For example, if the program is running interactively and an error is detected, then it must inform the user and wait for him to respond; on the other hand, in batch mode, its only course of action is to terminate since the execution of further commands may corrupt the file. The instruction

\[ \text{TO ' IF BATCH} \]

tests the BATCH mode 'flip-flop' and transfers control if it is set. Similarly the operation

\[ \text{TO ' IF INTERRUPT} \]

allows the user to regain control at the console. This feature is needed since it is possible to initiate complex searches and repetitive operations which may be time consuming. If the user realizes that he has made a mistake he can cancel the operation and recover the initial position.

The most significant difference between FLUB and TEXED is the complexity of the I/O operations required to support MITEM; in the simplest case, MITEM needs one more device than STAGE2 with input coming from READ and CONTROL streams and output going to WRITE and PRINT streams. More complex situations require more complex behaviour. For example, consider the problem of moving a block of text from one position in a file to another. This involves deleting the text with respect to the WRITE stream and outputting it to a DELETE stream. Subsequent lines are then input from the READ stream and output to the WRITE stream until the new position is reached. The DELETE stream is then rewound and copied onto the WRITE stream. Thus, in addition to being able to access many I/O devices, MITEM must be able to perform control operations such as rewind, endfile and backspace.
A still more complex example is that of merging two files on the basis of their contents, possibly making modifications as well. This requires the ability to connect the READ stream to either of two devices and to alter the connection without altering the current position of either device. The user first connects the READ stream to the first file and issues editing commands in the normal way. When he arrives at line \( x \) he switches the READ stream to the second file and continues. At line \( y \) of the second file he switches back to the first file to process line \( x \) and subsequent lines. Notice that line \( x \) must be examined \textit{before} switching from the first file to the second and must \textit{also} be available for processing after the user has switched back.

These requirements led to the design of an I/O package in which the I/O devices are conceptualized as files connected to the TEXED machine by channels. Each channel is associated with a channel buffer which can hold one line (a record) from a file. The normal record I/O operations (READ NEXT ' and WRITE NEXT ') transfer information between channels and the line buffer using the channel buffers as intermediate storage. The operations WRITE CURRENT ' and READ CURRENT ' transfer a line between the line buffer and the appropriate channel buffer. Basic control operations (REWIND ', BACKSPACE ' and ENDFILE ') are also available.

In designing these I/O operations, we anticipated that we might be able to satisfy the requirements of a wide range of abstract machines. Since the environment must be recoded for each computer, the use of a common package would allow the implementation effort to be spread over a number of abstract machines.

The I/O package was implemented as a collection of subroutines with integer arguments. A version exists in standard ANSI FORTRAN and this serves both as a document for describing the logic of the I/O package and as a mechanism to enable the system to be bootstrapped with a minimum of effort on any machine equipped with a FORTRAN compiler. Full details of the implementation have been presented elsewhere. Experience has shown that the FORTRAN version is very slow and hence assembly code versions of the environment have been provided for a number of machines. Implementation times for these have varied from man-weeks to man-months, a point that will be discussed later.

**Implementation**

An implementation of TEXED is essentially the same as an implementation of FLUB for the same target machine. Since the implementation is carried out by STAGE2 rather than by SIMCMP, however, it is possible to tailor the generated program to meet particular constraints. We shall refer to this as the 'adaptability' of the program and illustrate it from two points of view:

1. Adaptation to user and system requirements.
2. Adaptation to specific hardware.

The first depends mainly on the characteristics of the program and the way in which it is constructed; to a lesser extent, it implies modification of the abstract machine since unused 'hardware' facilities may be removed during adaptation. The second is concerned with the way in which the abstract machine is mapped onto the real one. This depends largely upon the structures of the two machines but may also be influenced by the characteristics of the program.

In any on-line system, one would expect to find a text editor. These vary from simple editors which permit the insertion, replacement and deletion of lines on the basis of line numbers to ones which provide the user with powerful facilities for manipulating the text
on the basis of context. In writing a portable text editor to operate in many systems, we were faced with the problem of attempting to satisfy a wide range of user requirements. If too many facilities were included in the program, then it might become too large to include in some systems; on the other hand, the lack of some facilities might result in the program not being adopted. Further, some facilities might prove difficult or even impossible to implement in a given system, e.g. the interrupt flip-flop. The problem was solved by including many facilities and allowing an installation to select those which it wished to make available to its users. In MITEM, six versions and a number of options are incorporated within the one body of text. Before generation, the version and options required are declared and STAGE2 then adapts the input text accordingly. This involves the selection and omission of declarations, routines and code sequences within a routine. Any optional routine is preceded by a statement of the form

\[ \text{DELETE IF VERSION LT } n \]

where \( n \) is the version number. On encountering this statement, STAGE2 stops processing the text and skips to the end of the routine if the current version requested is less than \( n \). Individual statements and declarations may be marked for selective inclusion by preceding them with headers of the form \(+ m\) or \(- n\). The former denotes lines to be included for versions greater than or equal to \( n \), the latter, lines for versions less than \( n \). Similarly, statements which implement optional facilities are preceded by a subsidiary header of the form \(/m\) where \( m \) is the option number. Thus a line with headers

\[ +4 /2 \]

will only cause code to be generated if option two has been requested and the version required is four or greater. The resultant programs vary from MITEM1 (a simple context editor operating on four streams and channels) to MITEM6 (which manipulates eight streams and up to thirty-two channels). Excluding the I/O, the size of MITEM6 is three times that of MITEM1. Typically, in a medium size system, one might make MITEM3 available for everyday use and reserve MITEM6 for the more esoteric editing operations. The choice can be made on the basis of how much core the installation has available for such programs.

In discussing adaptation to specific hardware, we shall present two examples which illustrate:

(a) Optimization for speed (minimization of CPU time).

(b) Optimization for space (minimization of core occupancy).

The first of these was carried out on an ICL KDF9 for both FLUB and TEXED. The actual machine has fifteen registers, each of three fields; the abstract machines have thirty-six such registers. The problem is therefore one of register allocation. We started with a naïve realization in which the register fields were mapped into core. We altered the macros to generate code for gathering statistics and ran the resultant program on a comprehensive set of test data to reveal which registers were most frequently used. These were then assigned to the hardware registers and the macros rewritten to generate code conditionally depending on which registers of the abstract machine were involved in the operation. At the same time, other forms of optimization were applied (use of immediate operands for operations involving constants, special treatment for the operands 0 and 1, etc.). Measurements showed that the optimized programs were about twice as fast as the unoptimized ones. It is important to realize that only a few man-days of effort were required to produce such optimized versions.
In the second example, the goal was to make MITEM as small as possible on the ICL 4/70, a machine with an architecture similar to that of System/360. Our core was limited at the time and we had to fit MITEM6 into a small interactive partition to continue testing. We already had an optimized version in assembly code but this would only run in the batch partition. The approach taken was to realize TEXED as an interpreter.

The form of the interpreter code reflects both the structure of the abstract machine and the characteristics of the program itself. TEXED has 108 register fields and permits an unspecified number of constants. MITEM uses about 110 constants so that the total number of operands is less than 256. Although there are more than 256 labels in the entire program, there are less than this number in any one procedure. Moreover, the number of procedures is less than 256. Since MITEM contains no direct jumps into or out of any procedure, labels are strictly local. This means that any operand or label can be specified by a single byte. Each procedure has an array of local addresses and there are two global arrays; one for register fields and constants, the other for procedure addresses. The one-byte operands and labels are interpreted as indices to these arrays.

There are twenty-eight operations provided in the TEXED interpreter and it occupies 1000 bytes on the 4/70. Each instruction is four bytes long and consists of an operation code and up to three operands. This uniform representation wastes some space but simplifies the interpreter and speeds up its inner loop. The interpreter instructions are summarized in Table III.

The size of the interpretive version of MITEM is about 40 per cent of that of the optimized assembly code version (excluding the I/O). Lack of suitable facilities in the 4/70 operating system prevented us from measuring the increase in CPU time exactly. However, twenty instructions must be obeyed in the interpreter to execute the TEXED operation

\[ \text{PTR}' = \text{'} + \text{'} \]

as opposed to three in the directly executable version; for an unconditional branch, the ratio is 20:1. This suggests that the interpreter is at least an order of magnitude more expensive with respect to CPU time.

In presenting the above results, we hope that the reader will not assume that we are proposing that a program should exist in a variety of forms for a given system. The transformations carried out on TEXED were in response to specific situations. What we have tried to illustrate is the degree of control one can exercise over the form of the running program by using the macro processing approach. In no case was any change made to the source statements of MITEM.

The applications of this type of adaptability are numerous. For example, during the construction of an operating system, one can proceed directly with the writing of a module and defer the decision about its final form until after it has been integrated with the remainder of the system and the appropriate measurements have been made. Infrequently used modules can operate interpretively to minimize core and backing store occupancy; critical sections can be optimized for speed to maintain throughput and response; other modules may employ a combination of modes.

Critique

TEXED has provided a reasonable vehicle for the development of a text manipulation system and we have not found maintenance or enhancement difficult to carry out. The degree of optimization obtainable has been quite satisfactory. However, based on our
experience with moving the program, we conclude that the machine suffers from a number of design faults.

The first major deficiency is that the structure of TEXED does not adequately reflect operations on strings. Since it is an extension of FLUB, the basic operations are those concerned more with lists and trees than with strings. MITEM is programmed accordingly

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register transfer</td>
<td>Destination</td>
<td>Source</td>
<td></td>
<td>REG A = B</td>
</tr>
<tr>
<td>Field transfer</td>
<td>Destination</td>
<td>Source</td>
<td></td>
<td>FLG X = 2</td>
</tr>
<tr>
<td>Addition</td>
<td>Result</td>
<td>First operand</td>
<td>Second operand</td>
<td>VAL A = B + C</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Result</td>
<td>First operand</td>
<td>Second operand</td>
<td>PTR X = Y - Z = 0</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Result</td>
<td>First operand</td>
<td>Second operand</td>
<td>PTR A = B * COUNT</td>
</tr>
<tr>
<td>Division</td>
<td>Result</td>
<td>First operand</td>
<td>Second operand</td>
<td>PTR X = Y / Z</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>Label</td>
<td>First operand</td>
<td>Second operand</td>
<td>TO XYZ</td>
</tr>
<tr>
<td>Conditional jump EQ</td>
<td>Label</td>
<td>First operand</td>
<td>Second operand</td>
<td>TO NEXT IF FLG A = 1</td>
</tr>
<tr>
<td>Conditional jump NE</td>
<td>Label</td>
<td>First operand</td>
<td>Second operand</td>
<td>TO PQR IF VAL B NE = C</td>
</tr>
<tr>
<td>Conditional jump GT</td>
<td>Label</td>
<td>First operand</td>
<td>Second operand</td>
<td>TO XYZ IF PTR A GT TEN</td>
</tr>
<tr>
<td>Conditional jump GE</td>
<td>Label</td>
<td>First operand</td>
<td>Second operand</td>
<td>TO ALPHA IF PTR X GE Y</td>
</tr>
<tr>
<td>Call, 2 parameters</td>
<td>Routine</td>
<td>Parameter 1</td>
<td>Parameter 2</td>
<td>CALL OUT(VAL A, TEN)</td>
</tr>
<tr>
<td>Call, 1 parameter</td>
<td>Routine</td>
<td>Parameter 1</td>
<td></td>
<td>CALL IN(VAL X)</td>
</tr>
<tr>
<td>Call, no parameters</td>
<td>Routine</td>
<td>Parameter 1</td>
<td></td>
<td>CALL PROCESS</td>
</tr>
<tr>
<td>Procedure entry,</td>
<td>Routine</td>
<td>Parameter 1</td>
<td>Parameter 2</td>
<td>PROC OUT(PTR A, PTR B)</td>
</tr>
<tr>
<td>2 parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure entry,</td>
<td></td>
<td>Parameter 1</td>
<td></td>
<td>PROC IN(VAL A)</td>
</tr>
<tr>
<td>1 parameter</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure exit</td>
<td></td>
<td></td>
<td></td>
<td>RETURN</td>
</tr>
<tr>
<td>Memory to register</td>
<td>Destination</td>
<td>Source PTR</td>
<td>Array</td>
<td>SET Y = CHANNEL(X)</td>
</tr>
<tr>
<td>Register to memory</td>
<td>Source</td>
<td>Destination</td>
<td>Array</td>
<td>SET CHANNEL(A) = B</td>
</tr>
<tr>
<td>Clear array</td>
<td>Array</td>
<td>Size</td>
<td></td>
<td>CLEAR ARRAY CHANNEL</td>
</tr>
<tr>
<td>Push register</td>
<td>Source</td>
<td>Stack pointer</td>
<td>Stack array</td>
<td>PUSH A ON STORE</td>
</tr>
<tr>
<td>Push field</td>
<td>Source</td>
<td>Stack pointer</td>
<td>Stack array</td>
<td>PUSH VAL B ON STORE</td>
</tr>
<tr>
<td>Pop register</td>
<td>Destination</td>
<td>Stack pointer</td>
<td>Stack array</td>
<td>POP X FROM STORE</td>
</tr>
<tr>
<td>Pop field</td>
<td>Destination</td>
<td>Stack pointer</td>
<td>Stack array</td>
<td>POP PTR Y FROM STORE</td>
</tr>
<tr>
<td>Empty stack</td>
<td>Stack pointer</td>
<td></td>
<td></td>
<td>EMPTY STORE STACK</td>
</tr>
<tr>
<td>I/O transfers</td>
<td>Output</td>
<td>Input</td>
<td>Operation code</td>
<td>READ NEXT X</td>
</tr>
<tr>
<td>Load message</td>
<td>Message</td>
<td>number</td>
<td></td>
<td>LOAD MESSAGE QUIT</td>
</tr>
<tr>
<td>Terminate</td>
<td></td>
<td></td>
<td></td>
<td>STOP</td>
</tr>
</tbody>
</table>

with strings set up as lists and string manipulations encoded as operations on lists. If TEXED is implemented on a machine which has hardware instructions for manipulating strings, then it cannot take advantage of these. It is clear now that it would have been possible to design a more specialized abstract machine whose basic operations and data types were concerned just with the manipulation of strings. This could have been placed in a hierarchy above a machine in which string operations were expressed as operations on arrays of integers and mapped onto it in a machine independent way. For computers
without special hardware, one could have mapped the low level machine to minimize the amount of effort required. However, if a computer had the appropriate hardware then one could map the high level machine directly to obtain a more efficient version of the program. This deficiency in TExED highlights one of the major problems: the design of an abstract machine is not a simple task and one does not have a completely clear idea of what the design should be until after the algorithm has been encoded.

The second major drawback concerns the environment. In some cases considerable effort has been required to implement it in assembly code to obtain an efficient version. Further, although we attempted to design a generalized system we have now encountered situations where it does not provide the necessary facilities. The problem is that the design of the I/O system is too heavily biased towards the requirements of MITEM. Although it also satisfies the needs of STAGE2 (and SIMCAMP), it has too rigid a structure (particularly with regard to the organization of buffers) to qualify as a 'general' system. In the new version of the I/O, the boundary between abstract machine and environment has been moved so that the buffers are controlled by the abstract machine. The environment has thereby been simplified and its only function now is to control the flow of information to and from the channels. Its realization on actual machines is simpler and efficient versions are more readily obtainable.

AIM1

Design rationale

AIM1 is intended to be suitable for interpreters for a wide variety of algebraic languages. We have produced a BASIC interpreter using this abstract machine, but we recognize that BASIC includes few of the complexities seen in ALGOL or PL/1. For this reason we consider AIM1 to be only the first of several iterations.

The principal data types provided by AIM1 are integers, floating point numbers (reals) and strings. Integers are the most heavily used, appearing as counters, character codes, flags and memory addresses. An indefinite number of integer registers may be declared by name. Five operations (addition, subtraction, multiplication, integer division and remaindering) are available to combine integer values (register contents and constants). The assignment statement [Figure 4(a)] is the means by which values of integer registers are changed. The reader should ponder the definition of 'expression' [Figure 4(a)], since many other instructions of the AIM1 (assembly) language involve this same object. It should be noted that operators have no precedence; expressions are evaluated strictly from left to right.

As an example of the AIM1 language, consider the sequence of assignments shown in Figure 4(b). ONE, TWO, THREE and FOUR are assumed to be the constants 1–4 respectively. The MOD operator yields the remainder on integer division. Thus registers AO, BO and CO will contain 1, 10 and 11 after these three statements have been executed.

Floating point numbers are used by the BASIC program which is to be run. We employ a conventional stack, ARITHSTACK, as the floating point arithmetic register. This avoids the problem of temporary storage allocation during expression evaluation and permits a compact 'zero address' representation for instructions. Similar considerations regarding addressing and temporary storage apply to strings. We have therefore defined a STRINGSTACK to hold the strings of current interest.

A third stack, LINKSTACK, is provided for return addresses. Thus all subroutines of AIM1 may be recursive provided that the programmer is careful about storing local
variables. In addition to easing the task of constructing recursive routines, LINKSTACK makes a 'relative return' straightforward. The return from a subroutine is accomplished by an EXIT statement. EXIT 0 is a command to transfer control back to the instruction immediately following the CALL which activated the subroutine. More generally, EXIT $n$ (where $n$ is an integer) causes $n$ instructions following the CALL to be skipped; these instructions are all assumed to be unconditional jumps. Hence, if PROC XYZ has an

$$<\text{assignment}> ::= <\text{reg}> = <\text{expr}>$$

$$<\text{expr}> ::= <\text{item}> \mid <\text{expr}> \text{ operator } <\text{item}>$$

$$<\text{item}> ::= <\text{reg}> \mid <\text{constant}>$$

$$<\text{operator}> ::= + \mid - \mid * \mid / \mid \text{MOD}$$

[Note: 'reg' is an abbreviation of (integer) register. 'expr' is an abbreviation of (expression)]

(a)

$AO = \text{ONE}$

$BO = AO + \text{FOUR} + \text{TWO}$

$CO = BO \text{ MOD THREE} + BO$

(b)

Figure 4. AIM1 assignment statements

EXIT 2 instruction within it, then every CALL XYZ instruction in the program will (normally) be followed by two unconditional jumps (the first is to cater for EXIT 0 orders and the second for EXIT 1 orders). Thus a subroutine can return to different positions in the calling program, depending upon some test.

In addition to the registers and stacks, AIM1 has a memory which is capable of holding integers, reals and strings. Each data type may occupy a different amount of storage and these amounts may vary from one target computer to another. No fixed relationship (such as 'a real requires twice as much storage as an integer') exists among them. We have therefore postulated that each data type may be stored only in certain memory locations. AIM1 provides operations which will 'round up' the contents of an integer register if it does not address a memory location capable of storing a particular data type. There is one such operation for each data type.

The basic data manipulation operations provided by AIM1 are summarized in Table IV. Groups 0 and 1 contain all register/register and arithmetic operations, while group 2 shows the memory transfers. None of these operations require changes in the data types of their operands. Operations in group 3, however, transmit information between units which hold different data types. In fact, the whole purpose of these operations is to perform type conversion.

FIX TO <reg> rounds the top entry of ARITHSTACK to obtain an integer. If truncation is desired, then TRUNCATE (group 1a) may be used before fixing. Both of these
operations require non-negative floating point operands. This restriction is imposed so that the implementor may safely use a simple shift for truncation. Experience has indicated that it is normally a positive number which must be made an integer, so that the restriction is not burdensome. If a negative number must be converted, it is a simple matter to program this conversion in the AIM1 language.

Table IV. Basic manipulation in AIM1

<table>
<thead>
<tr>
<th>Group</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\text{ADD} \quad \text{SUBTRACT} \quad \text{MULTIPLY} \quad \text{DIVIDE} \quad \text{NEGATE} \quad \text{REVERSE} \quad \text{ERASE} \quad \text{DUPLICATE} \quad \text{TRUNCATE})</td>
<td>Used for all REGISTER/REGISTER operations</td>
</tr>
<tr>
<td>1a</td>
<td>(\text{ADD} \quad \text{SUBTRACT} \quad \text{MULTIPLY} \quad \text{DIVIDE} \quad \text{NEGATE} \quad \text{REVERSE} \quad \text{ERASE} \quad \text{DUPLICATE} \quad \text{TRUNCATE})</td>
<td>This subgroup is the various arithmetic and manipulative operations available which affect only the ARITHSTACK. Only the top one or two elements of the stack are affected</td>
</tr>
<tr>
<td>1b</td>
<td>(\text{ERASE STRING} \quad \text{CONCATENATE} \quad \text{USE} \quad \text{TO SPLIT})</td>
<td>This subgroup is analogous to (1a) except the stack involved is the STRING STACK</td>
</tr>
<tr>
<td>2a</td>
<td>(\text{MEM}(\langle \text{exprn}\rangle) = \langle \text{exprn}\rangle \quad \langle \text{reg}\rangle = \text{MEM}(\langle \text{exprn}\rangle) \quad \text{ROUND UP} \quad \langle \text{reg}\rangle \quad \text{FOR INTEGRERLTS})</td>
<td>This subgroup is for REGISTER/MEMORY communication</td>
</tr>
<tr>
<td>2b</td>
<td>(\text{STORE}(\langle \text{exprn}\rangle) \quad \text{TO STACK} \quad \text{STACK TO STORE}(\langle \text{exprn}\rangle) \quad \text{ADD} \quad \text{STORE}(\langle \text{exprn}\rangle) \quad \text{SUBTRACT} \quad \text{STORE}(\langle \text{exprn}\rangle) \quad \text{MULTIPLY} \quad \text{BY STORE}(\langle \text{exprn}\rangle) \quad \text{DIVIDE} \quad \text{BY STORE}(\langle \text{exprn}\rangle) \quad \text{ROUND UP} \quad \langle \text{reg}\rangle \quad \text{FOR FPRLTS})</td>
<td>ARITHSTACK/MEMORY communication</td>
</tr>
<tr>
<td>2c</td>
<td>(\text{STORE}(\langle \text{exprn}\rangle) \quad \text{TO STRINGSTACK} \quad \text{STRINGSTACK TO STORE}(\langle \text{exprn}\rangle) \quad \text{ROUND UP} \quad \langle \text{reg}\rangle \quad \text{FOR STRELTS})</td>
<td>STRINGSTACK/MEMORY communication</td>
</tr>
<tr>
<td>3a</td>
<td>(\text{FIX TO} \quad \langle \text{exprn}\rangle \quad \text{FLOAT} \quad \langle \text{exprn}\rangle)</td>
<td>ARITHSTACK/REGISTER communication</td>
</tr>
<tr>
<td>3b</td>
<td>(\text{START READING STRING IN STACK} \quad \text{STRINGSTACK/REGISTER communication} \quad \langle \text{reg}\rangle \quad \text{FROM STRING IN STACK} \quad \text{START WRITING STRING IN STACK} \quad \langle \text{exprn}\rangle \quad \text{TO STRING IN STACK})</td>
<td>STRINGSTACK/REGISTER communication</td>
</tr>
</tbody>
</table>

A string may be packed or unpacked using the instructions in group 3b. The ‘cursor’ which moves along the string must be initialized, and then subsequent transmission operations advance it.

Note the presence of a number of redundant operations in group 2b. For example, ADD STORE (\(\langle \text{exprn}\rangle\)) could be replaced by a pair of operations:

\[
\text{STORE} (\langle \text{exprn}\rangle) \quad \text{TO STACK} \\
\text{ADD}
\]
This combination would occur quite frequently and hence coding effort is reduced by providing a single instruction. Also, a single instruction is easier to recognize for optimization purposes than two adjacent ones. These redundant operations provide a simple example of a hierarchy; we shall show a more complex one below.

Table V lists the control operations of AIM1. The last four conditionals in group 2a are used to test integers to ascertain whether or not they are character codes which represent

<table>
<thead>
<tr>
<th>Group</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TO &lt;label name&gt;</td>
<td>Unconditional jump</td>
</tr>
<tr>
<td></td>
<td>LOC &lt;label name&gt;</td>
<td>Label declaration</td>
</tr>
<tr>
<td>1</td>
<td>PROC &lt;name&gt;</td>
<td>Simple type of procedure heading</td>
</tr>
<tr>
<td></td>
<td>CALL &lt;name&gt;</td>
<td>Call of a procedure with no parameter</td>
</tr>
<tr>
<td></td>
<td>PROC &lt;name&gt; (?!)</td>
<td>Procedure with parameter and/or some local variables</td>
</tr>
<tr>
<td></td>
<td>CALL &lt;name&gt; (&lt;exprn&gt;)</td>
<td>Parameter 1 is the formal parameter name, if present</td>
</tr>
<tr>
<td></td>
<td>EXIT &lt;integer&gt;</td>
<td>Parameter 2 is a list of local variables</td>
</tr>
<tr>
<td>2a</td>
<td>TO ' IF &lt;exprn&gt; &lt;relop&gt; &lt;item&gt;</td>
<td>The S/R return mechanism</td>
</tr>
<tr>
<td></td>
<td>TO ' IF &lt;exprn&gt; ALPHABETIC</td>
<td>'Conditionals involving registers</td>
</tr>
<tr>
<td></td>
<td>TO ' IF &lt;exprn&gt; NOT ALPHABETIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TO ' IF &lt;exprn&gt; NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TO ' IF &lt;exprn&gt; NONNUMERIC</td>
<td></td>
</tr>
<tr>
<td>2b</td>
<td>TO ' IF STORE(&lt;exprn&gt;) &lt;relop&gt; STACK(1)</td>
<td>Conditions involving floating point numbers</td>
</tr>
<tr>
<td></td>
<td>TO ' IF STORE(&lt;exprn&gt;) &lt;relop&gt; STACK(2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TO ' IF STACK(1) &lt;relop&gt; STACK(2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TO ' IF ARITHSTACK FULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TO ' IF ARITHSTACK EMPTY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TO ' IF OVERFLOW</td>
<td></td>
</tr>
<tr>
<td>2c</td>
<td>TO ' IF STRINGS EQUAL</td>
<td>Conditions involving stringstack</td>
</tr>
<tr>
<td></td>
<td>TO ' IF STRINGS NOT EQUAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TO ' IF STRINGSTACK FULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TO ' IF STRINGSTACK EMPTY</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BRANCH ON &lt;exprn&gt;</td>
<td>Multi-way branch instruction</td>
</tr>
</tbody>
</table>

Where

<item> ::= <name of a constant> | <register name>.
<name> ::= <letter> | <name> <letter> | <name> <digit>.
<relop> ::= = | NS | LT | GT | LE | GE.

letters or digits. The motivation is clear—AIM1 does not have a character code of its own (it is usually preferable to use the codes of the particular real machine) and because the tests in question are a very common operation in syntax analysis, we would like them to be as efficient as possible.

The BRANCH instruction of group 3 is simply a computed goto statement. An implementation should assume that this instruction will be followed by a number of unconditional jumps equal to the maximum value expected for the expression. If the expression evaluates to zero then the order following the BRANCH is executed; if one then a single jump will be skipped, etc.
The other instructions of Table V are self explanatory and the data manipulation picture is complete with a word about constants. Real numeric constants are declared as in the following example:

\[
\text{FPCONST ALPHA} = -63P41E + 12
\]

The syntax of the number part is as found in ALGOL 60 except that the decimal point is written as 'P' and the sub-ten is written as 'E' (this use of 'P' is forced by the fact that the character sets of many machines are quite small and the decimal point character is already used to signal end of instruction). Reference to a floating point constant is made by the construct

\[\text{FPCONST <name>}\]

Such a reference may appear anywhere that \text{STORE(<exprn>)} or \text{STACK(2)} may appear and so we have orders such as

\[
\text{ADD FPCONST HALF}
\]

and

\[
\text{TO CRASH5 IF STACK(1) LT FPCONST ABC}
\]

\[\text{STRING <text>}\] declares a string constant. The value of the constant, \text{<text>}, is also its name. A string constant is referenced by using \text{STRING <text>} in place of \text{STORE(<exprn>)}. For example, a string \text{BLOGGS} could be declared and used by

\[
\text{STRING BLOGGS}
\]

\[\text{STRING BLOGGS TO STRINGSTACK}\]

The input/output structure of AIM1 is designed to use the latest version of the I/O package. All transfers are line-at-a-time and an AIM1-coded subroutine package is available which manages buffers, provides logical information streams and delivers single characters on demand (removing spaces if appropriate). As seen in Table VI (group 1), there are also open and close orders for files. The noteworthy point about \text{OPEN} is that the name of the file which is to be opened can be the result of some computation since it is taken from \text{STRINGSTACK}.

A number of special instructions (Table VI, group 2) enhance the capabilities of the processor and/or increase its efficiency. The segmentation instruction seems an almost essential option. Space is at a premium on small machines and even on large multi-programming systems interactive programs need to be small. The two instructions

\[
\text{TO ' IF INTERRUPT and TO ' IF BPT <exprn> ON}
\]

are important if the user is on-line and needs to communicate with his running program. Indeed, one of the two is essential if the programmer is to interact in any fashion other than question/response type dialogue.

Two more memory data elements are introduced with the \text{HASH(')} and \text{ORD(')} statements of group 2; we will call these elements \text{HASEL} and \text{ORDELT} respectively (the ordinary integer data element is referred to as an \text{INTEGRERLT}). Each of these new data types is a pair of integers and is useful when the quantum of memory assigned to an \text{INTEGRERLT} on a particular computer can hold more information than will ever be found in an \text{INTEGRERLT}.

\text{ORDELTS} were invented to represent internally the program that an interpreter interprets. The two integers involved are an opcode (for which we allow 256 different values), and an operand which may be an address. Hence the second field needs the same
Table VI. Input/output and special instructions

<table>
<thead>
<tr>
<th>Group 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LINE</strong> <em>(exprn1), (exprn2) TO (exprn3)</em></td>
</tr>
<tr>
<td>Outputs a line of E2 characters to channel E3. E1 is</td>
</tr>
<tr>
<td>the address of a block of store which contains the line. [Ei is the</td>
</tr>
<tr>
<td>value of <em>(exprn i)</em>]</td>
</tr>
<tr>
<td><strong>LINE</strong> <em>(exprn1), (reg) FROM (exprn2)</em></td>
</tr>
<tr>
<td>Inputs a line from channel E2 to the address given by E1. It records</td>
</tr>
<tr>
<td>the number of characters in the register</td>
</tr>
<tr>
<td><strong>OPEN FILE ON CHANNEL</strong> <em>(exprm)</em></td>
</tr>
<tr>
<td>Opens the file whose name is given as the top element in STRINGSTACK</td>
</tr>
<tr>
<td><strong>CLOSE CHANNEL</strong> <em>(exprm)</em></td>
</tr>
<tr>
<td>Close the file on the channel given</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET</strong> <em>(name)</em> <strong>SEGMENT</strong></td>
</tr>
<tr>
<td>Read in the named segment and transfer control to it. The memory and</td>
</tr>
<tr>
<td>the values of global variables are undisturbed and available to all</td>
</tr>
<tr>
<td>segments</td>
</tr>
<tr>
<td><strong>TO ' IF INTERRUPT'</strong></td>
</tr>
<tr>
<td>A test to see if the user wants attention</td>
</tr>
<tr>
<td><strong>TO ' IF BKPT</strong> <em>(exprm)</em> ON**</td>
</tr>
<tr>
<td>Test breakpoint switch whose number is the value of the exprn. (If it</td>
</tr>
<tr>
<td>exists)</td>
</tr>
<tr>
<td><strong>HASH</strong>(exprm) = *(exprm), <em>(reg)</em></td>
</tr>
<tr>
<td>MEMORY/REGISTER communication</td>
</tr>
<tr>
<td><strong>ORDER</strong>(exprm) = *(exprm), <em>(reg)</em></td>
</tr>
<tr>
<td><strong>FAIL</strong> <em>(integer)</em> <strong>MESSAGE</strong> <em>(text)</em></td>
</tr>
<tr>
<td>Failure message declaration</td>
</tr>
<tr>
<td><strong>FAIL</strong> <em>(integer)</em></td>
</tr>
<tr>
<td>Occurrence of failure whose number is given</td>
</tr>
</tbody>
</table>

space as an INTEGERELT. Eight bits is an appropriate amount of space to reserve for
the opcode part of an order for these reasons:

(1) Our experience with a BASIC interpreter indicated that six bits was only adequate
if some convenience was sacrificed. For more complex algebraic languages, the
number of operations requiring opcodes would certainly be more; it is likely that
even seven bits (giving only .50 per cent more operations than used in BASIC)
would also severely lessen convenience.

(2) On several machines encountered so far, eight bits is all that is available if ORDELTS
are to be useful.

Clearly two INTEGERELTS can represent the same information as one ORDELT, but
in many circumstances may take up twice the space.

HASHELTS are designed to make efficient use of storage when constructing a hash
table. (Each entry in the hash table will be a HASHELT.) The two integer fields involved
here are for a label number and the corresponding address. If the language is one of those
which have numeric labels, then the label number will just be the label itself. Otherwise the
label number will be the address of a string which is the label itself. Again, a HASHELT
could be implemented by two INTEGERELTS but to do so would be inefficient in many
implementations (both in terms of space and speed). Note that HASHELTS and
ORDELTS are not the same thing; on some machines an ORDELT will use the same
space as an INTEGERELT while each HASHELT will use twice as much.

The instruction FAILURE MESSAGE *(integer) *(text)* is simply a declaration of the
 correspondence between a piece of text (the failure message) and an integer (the failure
number). The macro for implementing this order may (if space is exceptionally limited) generate no code at all. Any diagnostics will consist of a single integer and no descriptive text. The effect of the FAIL (integer) order is to record the failure number in a predetermined register, initialize the reading of the failure message (if one was generated), clear LINKSTACK and transfer control to a label in the main routine with a predetermined name. It is true that these instructions are not strictly necessary since the same effect can be realized by cunning use of conditional expansion in STAGE2, ordinary string constants and the multiple exit feature of AIM1. However, experience on several machines has shown there is a substantial saving in space (mainly because of the frequent occurrence of the instruction) and a bonus in ease of coding.

Figure 5 is an example of AIM1 code.

PROC BINSRCH(A,B,C).
.THIS IS A SUBROUTINE TO LOOK UP A TABLE OF REAL NUMBERS
.USING THE BINARY SEARCH METHOD. THE TABLE CONTAINS ONLY
.REAL NUMBERS AND ITS BOUNDS ARE GIVEN BY THE VARIABLES
.BA (ADDRESS OF FIRST ENTRY) AND NUM (NUMBER OF ENTRIES).
.THE SINGLE ARGUMENT FOR THE PROCEDURE IS A REAL NUMBER
.IN THE F.P. STACK. IF THIS NUMBER IS NOT FOUND IN THE
.TABLE THEN THE ROUTINE SHOULD EXIT 0 ELSE IT SHOULD
.EXIT 1 WITH THE ADDRESS OF THE NUMBER IN XX.

A = ZERO.
B = NUM.

LOC BINLOOP.
TO BIXO IF B = A.
C = B - A / TWO + A.
XX = C * FPSIZE + T1.
TO BIX1 IF STORE(XX) = STACK(1).
TO BINLAB1 IF STORE(XX) GT STACK(1).
TO BIX0 IF A = C.
A = C.
TO BINLOOP.
.
LOC BINLAB1.
B = C.
TO BINLOOP.
.
LOC BIXO.
EXIT 0.
.
LOC BIX1.
EXIT 1.

Notes:
A period is used to signal the end of lno, hence the commentary above is ignored.
There is no parameter for the procedure but there are.
three local variables.
ZERO, TWO, FPSIZE are constants defined elsewhere
as 0,2 and (size of each real number in memory).

Figure 5. AIM1 coding example
Implementation of AIM1

There are about one hundred instructions for AIM1, requiring implementation of about the same number of macros. Most of these orders expand to three or four lines of assembly code plus the code generated from AIM1 expressions (the expansion of expressions tends to be one or two lines of code per operand). As a rule instructions go to fairly obvious in-line code and writing macros is not a difficult job for someone familiar with the target machine. Some instructions are best handled by generating subroutine calls; 'CONCATENATE, TO' IF STRINGS NOT EQUAL and the I/O instructions are examples of these.

As with FLUB, some thought must be given to memory element size assignments. The integer registers are comparatively few in number and are referenced frequently, so efficiency of the code generated for register references is a more important consideration than the space occupied by each register. Normally there is a unit of memory called a WORD which is large enough to hold an address, is directly addressable and is the unit of information which most naturally takes part in integer arithmetic. Each integer will probably map into one of these words.

The size of the floating point memory element depends upon the accuracy required and the hardware or software available. If the string features of AIM1 are implemented directly, then a string will usually be a few words of packed characters. There will often be more dilemma involved in the choice of memory size for HASHELT and ORDELT since a tradeoff between data space, program space and speed will be involved. Table VII gives some of the options for machines with different word lengths. Some entries in this table may cause surprise at first. The option presented in the line tagged **, for instance, seems inferior to the prescription of the previous line. But, on a machine such as the IBM System/360, the code to manipulate arbitrary triplets of bytes is more cumbersome than that required to work with full words.

Hierarchies of abstract machines have been used extensively in several implementations of AIM1 (recall the earlier description of this technique). It is possible to implement AIM1 in terms of a machine of about a fifth the complexity. The instruction set of this
simpler machine (call it NUTS) is a subset of the instruction set of AIM1 selected in such a way that the other instructions of AIM1 can be written in terms of the orders of NUTS. The big difference between AIM1 and NUTS is that NUTS has no 'hardware' for floating point operations, string operations, or HASHELT and ORDELT instructions. One can run AIM1 programs on the NUTS (abstract) machine (i.e. translate AIM1 programs into the NUTS language) with the aid of the appropriate macros; so, if these macros are in hand, the implementation of AIM1 via this hierarchical design will proceed very rapidly.

NUTS uses hierarchies to the extreme and would only prove useful if implementation time for AIM1 is so severely restricted that substantial loss of efficiency is forgivable. In all the situations encountered so far this has not been the case, and where any use of hierarchies

<table>
<thead>
<tr>
<th>ORDER(a) = b,c.</th>
<th>MEM(a) = b.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM(a + ISIZE) = c.</td>
</tr>
<tr>
<td>a,b = HASH(c).</td>
<td>a = MEM(c).</td>
</tr>
<tr>
<td></td>
<td>b = MEM(c + ISIZE).</td>
</tr>
<tr>
<td>a FROM STRING IN STACK.</td>
<td>a = MEM(CURS).</td>
</tr>
<tr>
<td></td>
<td>CURS = CURS + ISIZE.</td>
</tr>
<tr>
<td>TO a IF b NOT ALPHABETIC.</td>
<td>TO a IF b LT LOWLETTER.</td>
</tr>
<tr>
<td></td>
<td>TO a IF b GT HIGHLETTER.</td>
</tr>
<tr>
<td>ADD,</td>
<td>PTR = PTR - FPSIZE.</td>
</tr>
<tr>
<td></td>
<td>FACC = STORE(PTR - FPSIZE).</td>
</tr>
<tr>
<td></td>
<td>FACC = FACC + STORE(PTR).</td>
</tr>
<tr>
<td></td>
<td>STORE(PTR - FPSIZE) = FACC.</td>
</tr>
<tr>
<td>ERASE,</td>
<td>PTR = PTR - FPSIZE.</td>
</tr>
<tr>
<td>STORE(a) TO STACK.</td>
<td>FACC = STORE(a).</td>
</tr>
<tr>
<td></td>
<td>STORE(PTR) = FACC.</td>
</tr>
<tr>
<td></td>
<td>PTR = PTR + FPSIZE.</td>
</tr>
</tbody>
</table>

**Notes:**
- ISIZE, FPSIZE, LOWLETTER, HIGHLETTER are the appropriate constants.
- CURS, PTR are private variables.

*Figure 6. Examples of hierarchical macros*

was appropriate, it has not been so extensive. A package of pre-coded macros (see examples in Figure 6) which has been useful is available.14 This set of macros:

1. Implements many of the instructions in terms of some of the others. The HASH operations, the ORDER operations and all the stringstack operations are implemented in terms of INTEGERELTS.
(2) Implementation of the bulk of the ARITHSTACK operations is done in terms of the instruction set

\[
\begin{align*}
\text{FACC} &= \text{STORE}(\langle \text{exprn} \rangle) \\
\text{FACC} &= \text{FACC} + \text{STORE}(\langle \text{exprn} \rangle) \\
\text{FACC} &= \text{FACC} - \text{STORE}(\langle \text{exprn} \rangle) \\
\text{FACC} &= \text{FACC} \times \text{STORE}(\langle \text{exprn} \rangle) \\
\text{FACC} &= \text{FACC} / \text{STORE}(\langle \text{exprn} \rangle) \\
\langle \text{variable} \rangle &= \text{SIGN OF FACC} \\
\langle \text{variable} \rangle &= \text{ADDRESS OF FPCONST} \langle \text{name} \rangle
\end{align*}
\]

A technique was mentioned earlier in the paper in which programs for an abstract machine are codified and interpreted instead of being translated to machine code and executed. This scheme is quite appropriate to AIM1, and calculations made on several machines indicate that the implementor should expect to cut the size of the generated program by half (this figure is based on the mix of instructions in the BASIC interpreter). However, the speed sacrificed by using an interpretive approach (a figure which must be calculated for particular machines) indicates that the technique should be applied sparingly.

Test programs which identify errors in the macros and the associated environment are available. In fact, there is an ordered set of five programs which test and report on the instruction set in the bootstrap sequence.

Critique

Our only major criticism of the design of AIM1 was the initial inclusion of the stacks. The structure is clearly suitable for writing interpreters but we have found that it provides a poor match to most real computers. The power of ARITHSTACK, for instance, is never really needed; it is rare to use more than the top one or two entries. Hence there would be very little extra coding expense involved in using an accumulator for the floating point operations, just as real computers do.

The reason that a stack architecture maps poorly onto existing hardware is twofold. Firstly, a huge proportion of current hardware does floating point operations in some register (if at all). Secondly, the number of instructions needed to support a stack is much larger than the number needed to cater for an accumulator. In all, the implementor must spend more effort coding macros for stack instructions than for accumulator instructions because there are more macros and each macro is more complicated (on the average). Similar remarks apply to the STRINGSTACK. We therefore believe that a better abstract machine design could be developed if one replaced ARITHSTACK by a floating point accumulator and STRINGSTACK by a string accumulator.

It is also clear that optimized code could be generated more easily (less effort in macro coding and more efficient macros) if the concept of an expression were appropriately extended. For example, a straightforward translation of

\[
\begin{align*}
\text{XA} &= \text{MEM(XA)} \\
\text{XA} &= \text{MEM(XA)} \\
\text{TO ALPHAI IF XA LT TWO}
\end{align*}
\]

may generate redundant store and fetch operations for XA. The problem is that some extensions tend to encourage a programming style which will lead to inefficient code and/or complicated macros which take a long time to write for a new machine. Indeed, allowing ALGOL-like expressions is undesirable for this reason.
CONCLUSIONS

In this paper we have described three abstract machine models which we have used to implement portable software. We have attempted to give a critical appraisal of each design, pointing out its strengths and weaknesses. It now remains to summarize the merits (or otherwise) of our approach to portable software, in the light of our experiences.

By using abstract machine models, we have produced programs which can readily be moved from one computer to another. Currently, STAGE2 is operating on twenty-five different computers; MITEM has been implemented on five and BASIC on three. These implementations have resulted in usable software of reasonable efficiency. We have found it easy to adapt to local constraints by altering macro definitions. Typical implementation times have been of the order of 1–6 man-weeks, regardless of the software on the target computer but assuming reasonable access and good turnaround. We therefore assert that the technique is satisfactory for achieving portability.

As we noted in the Introduction, the major difficulty lies in the design of the abstract machine model. This is not an easy task even for experienced programmers. The design is intimately linked with the algorithm for solving the problem and is often not clearly visible until after the algorithm has been encoded. In each of our examples, we have noted design deficiencies and come to the conclusion that there exists a more suitable structure for the underlying model.

Another problem is that a diversity of designs leads to a diversity of languages and hence to an increase in the total cost of realizing many abstract machines. Although each machine is easy to realize, none of the macros can be used to cut the cost of realizing other machines. To a large extent we have avoided this problem with I/O by providing a standard package; we believe that further standardization is the key to our difficulties.

Wilkes\textsuperscript{10} has noted that virtually any algorithm employs a 'core' set of organizational operations. We think that we can capture this set in a 'general purpose' abstract machine and still provide for extensions to suit the user’s problem. This does not contradict in any way our original assertion that one solves a problem by designing an appropriate abstract machine. We do not intend that the programmer should twist his problem to fit the general purpose machine. Rather, the general purpose machine will aid the design of the special purpose one and ensure that the resulting program is highly portable. The most efficient version of the program will still result from a direct mapping of the special purpose machine to the target computer.

GLOSSARY

SIMCMP = Simple macro processor used to translate FLUB
FLUB = Abstract machine used to implement STAGE2
STAGE2 = Macro processor used to translate TEXED and AIMI
TEXED = Abstract machine used to implement MITEM, a text manipulation system
AIMI = Abstract machine used to implement an interpretive BASIC system

REFERENCES


