# Interactive machine-language programming*

BUTLER W. LAMPSON

*University of California, Berkeley*

## INTRODUCTION

The problems of machine language programming, in the broad sense of coding in which it is possible to write each instruction out explicitly, have been curiously neglected in the literature. There are still many problems which must be coded in the hardware language of the computer on which they are to run, either because of stringent time and space requirements or because no suitable higher level language is available.

It is a sad fact, however, that a large number of these problems never run at all because of the inordinate amount of effort required to write and debug machine language programs. On those that are undertaken in spite of this obstacle, a great deal of time is wasted in struggles between programmer and computer which might be avoided if the proper systems were available. Some of the necessary components of these systems, both hardware and software, have been developed and intensively used at a few installations. To most programmers, however, they remain as unfamiliar as other tools which are presented for the first time below.

In the former category fall the most important features of a good assembler:[1,2] macro-instructions implemented by character substitution, conditional assembly instructions, and reasonably free linking of independently assembled programs. The basic components of a debugging system are also known but relatively unfamiliar.[3,4] For these the essential prerequisite is an *interactive* environment, in which the power of the computer is available at a console for long periods of time. The batch processing mode in which large systems are operated today of course precludes interaction, but programs for small machines are normally debugged in this way, and as time-sharing becomes more widespread the interactive environment will become common.

It is clear that interactive debugging systems must have abilities very different from those of off-line systems. Large volumes of output are intolerable, so that dumps and traces are to be avoided at all costs. To take the place of dumps, selective examination and alteration of memory locations are provided. Traces give way to breakpoints, which cause control to return to the system at selected instructions. It is also essential to escape from the switches-and-lights console debugging common on small machines without adequate software. To this end, type-in and type-out of information must be symbolic rather than octal where this is convenient. The goal, which can be very nearly achieved, is to make the symbolic representation of an instruction produced by the system identical to the original symbolic written by the user. The emphasis is on convenience to the user and rapidity of communication.

The combination of an assembler and a debugger of this kind is a powerful one which can reduce by a factor of perhaps five the time required to write and debug a machine language program. A full system for interactive machine language programming (IMP), however, can do much more and, if properly designed, need not be more difficult to implement. The basic ideas behind this system are these:

1. Complete integration of the assembler and the debugging system, so that all input goes through the same processor. Much redundant coding is thus eliminated, together with one of two different languages serving the same purpose: to specify instructions in symbolic form. This concept requires that code be assembled directly into core—or into a core image on secondary

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract SD-185.

storage. Relocatable output and relocatable loaders are thereby done away with. (A remark on terminology: It will be convenient in the sequel to speak of the "assembler" and the "debugger" in the IMP system. These terms should be understood in the light of the foregoing: different parts of the same language are being referred to, rather than distinct languages.)

2. Commands for editing the symbolic source program. The edit commands simultaneously modify the binary program in core and the symbolic on secondary storage. Corrections made during debugging are thus automatically incorporated into the symbolic, and the labor of keeping the latter current is almost eliminated.
3. A powerful string-handling capability in the assembler, which makes it quite easy to write macros for compiling algebraic expressions, to take a popular example, which can be handled in a few other systems but rather clumsily. The point is not that one wants to write such macros, but that in particular applications one may want macros of a similar degree of complexity.

These matters are discussed in more detail below. We consider the assembler first and then the debugger since the command language of the latter makes heavy use of the assembler's features.

Before beginning the discussion it may be well to describe briefly the machine on which this system is implemented. It is a Scientific Data Systems 930, a 2-microsecond, single-address computer with indirect addressing and one index register. Our system includes a drum which is large enough to hold for each user all the symbolic for a program being debugged, together with the system, a core image of the program and some tables. Backup storage of at least this size is essential for the editing features of the IMP system. The rest of the system could be implemented after a fashion with tapes.

## THE BASIC ASSEMBLER

The input format of the assembler was originated on the TX-O at M.I.T. It has been adopted by DEC for most of its machines, but is unknown or unpopular elsewhere in the industry. Although it looks strange at first, it has substantial advantages in terms of simplicity, both for the user and for the system. The latter is a nonnegligible consideration, equally often ignored and overemphasized.

The basic idea is that the assembler processes each line of input as an *expression* (unless it is a directive or macro call).[5] The expression is evaluated and the value is put into core at the word addressed by the location counter, after which the location counter is advanced by 1. Expressions are made up of *operands,* which may be symbols, constants, numeric or alphanumeric and parenthesized subexpressions; and *operators.* Available operators are +, —, *, /, .AND, .OR, .NOT with their usual meaning and precedence; .E (equals), .G (greater), .GE, .L, .LE, .NE, which are binary operators with precedence less than +, and yield 1 or 0 depending on whether the indicated relation holds between the operands or not; and #, a unary operator with lowest precedence which causes its operand to be taken as a literal. This means that it is assigned a storage location, which is the same as the location assigned to other literals with the same value, and the address of this location is the value of the literal. Blanks have the following significance: Any string of blanks not at the beginning or end of an expression is taken as a single plus sign. An expression is terminated by carriage return or semicolon. Several instructions may therefore be written on one physical line. This trivial feature proves in practice to have significant advantages.

It is not immediately clear how instructions are conveniently written as expressions, and in fact the scheme used depends on the fact that the object machine is a single-address, word-oriented computer with a reasonable number of modifiers in a single instruction. It would work on the PDP-6, but not on the IBM 7030.

The idea is simple: all operation code mnemonics are predefined symbols with values equal to the octal encodings of the instructions. On the SDS 930, for instance, LDA (load A) is defined as 7600000 (all numbers are in octal). The expression LDA+200 then evaluates to 7600200. When the convention about spaces is invoked, the expression

```
LDA 200
```

evaluates to the same thing, which is just the instruction we expect from this symbolic line in a conventional assembler.

Modifiers are handled in the same spirit. In the 24 bit word of the 930 there is an index bit, which is the second from the left, and an indirect bit, which is the tenth. With the predefined symbols

```
I=40000
X=20000000
```

the expression

```
LDA I 200 X
```

evaluates to 27640200. In more conventional form it would look like this:

```
LDA* 200,2
```

There is little to choose between them for brevity or

clarity. Note that the order of the terms in the expression is arbitrary.

The greatest advantages of the uniform use of expressions accrue to the assembler, but the programmer gains a good deal of flexibility. Examples will readily occur to the reader.

Using this convention the implementation of the basic assembler is very simple. Essentially all that is required is an expression analyzer and evaluator, which will not run to more than three or four hundred instructions on any machine. Because all assembly is into core, there is no such thing as relocatability.

Two rather conventional methods are provided for defining symbols. A symbol appearing at the beginning of a line and followed by a comma is defined to be the current value of the location counter. Such a symbol may not be redefined. In addition, a line such as

```
SYM=4600
```

defines SYM. Any earlier definition is simply overridden. The right side may of course be any expression which can be evaluated.

The special symbol . refers to the location counter. It may appear on the left of a = sign. Thus, the line

```
A,      .=.  40
```

is equivalent to

```
A       BSS 40
```

in a conventional assembler.

Note that the first punctuation character in a line of input to the assembler must be comma or space. The character . is not a punctuation character, but behaves exactly like a letter. Symbols reserved by the system begin with dot ordinarily. For convenience in forming negative addresses, the symbol .. is provided with a permanent value such that ..—1 is —1 truncated to the address field. On the 930, a two's complement machine with a 14 bit address field, .. is 40000.

Strings of characters encoded in ASCII may be written surrounded by single or double quotes, ' ' or " ". If the string is less than 4 characters in length, it is equivalent to the number obtained by left-justifying it in a 24-bit word. Otherwise, it must appear alone on a line and generates enough words to accommodate all its characters. Strings in single quotes are scanned for : and & (see below); those in double quotes are taken literally.

The characters space * signal a comment, which is ignored up to the next carriage return. An initial * also has this effect.

There remains one point about the basic assembler which is crucially important to the implementation: the treatment of undefined symbols. When an expression is encountered during assembly, there is no guarantee that it can be evaluated, since all the symbols in it may not be defined. This is the reason why most assemblers are two pass: the first pass serves to define the symbols. The increase in speed obtained by looking at the symbolic only once is so great, however, that it is worth a good deal of trouble. Even if every expression contains an undefined symbol on the first pass, it still takes only one-fifth as long to evaluate the already analyzed expressions as to read the input again, and this for a program with no macros. The assembler therefore keeps track of undefined expressions explicitly.

There is a general way of doing this, in which the undefined expression, translated for convenience into reverse Polish, is added to a list of such expressions, together with the address of the word it is to occupy. At suitable intervals this list is scanned and all the newly defined expressions are evaluated and inserted in the proper locations. For complex expressions there is no avoiding some such mechanism, and it has the advantage of simplicity. It is, however, wasteful of storage and also of time, since an expression may be examined many times while it is on the list before it can be evaluated. One important case can be treated much more efficiently, and this is the case of an instruction with an undefined address, which includes perhaps 90% of the occurrences of undefined expressions.

For example, when the assembler sees this code:

```
X,  BRU A   *BRANCH UNCONDITIONAL
    LDA B
A,  STA C
```

the instruction at X has an undefined address which becomes defined when the label A is encountered. This situation can be kept track of by putting in the symbol table entry for A the location of the first word containing A as an address. In the address of this word we put the location of the second such word, and so build a list through all the words containing the undefined symbol A as an address. The list is terminated by making the address field point to itself. When the symbol is defined we simply run down the chain and fill in the proper value. This scheme will work as long as the address field contains only A, since there is then no other information which must be preserved. Note that no storage is wasted and that when A is defined the correct address can be filled in very quickly.

## STRINGS AND MACROS

The description of the basic assembler is now complete, except for a few nonessential details, and we turn to the macro and string handling facility. There is a uniform method for delimiting strings of characters, which may be illustrated by the assignment of such a string as the value of a symbol:

```
A = <B,(C,D),E,F>
```

In order to describe the result of using A after this assignment, we introduce a distinction between the appearance of a symbol in a *literal* and in a *normal* context.

A symbol inside string brackets < > or single quotes or in a macro argument is in a literal context; all other contexts but one are normal. In a normal context, the value of the symbol, whether a string or a number, is substituted for the symbol. In a literal context, on the other hand, the characters of the symbol are passed on unaltered. The case of a symbol on the left side of an assignment is an exceptional one; such a symbol is of course not normally evaluated.

To permit the value of a symbol to be obtained in a literal context, the convention is introduced that a colon preceding the symbol causes it to be evaluated if the colon is at the top level of parentheses, brackets, and quotes. If its value is a string, the characters of the string replace the symbol; if it is a number the shortest string of digits which can represent the number in the prevailing radix replaces the symbol. Colon in a normal context is illegal.

For convenience in delimiting string names a second colon may follow a name preceded by a colon. This second colon serves only to delimit the name and is otherwise ignored. Thus if

AB = <XYZ>

then

<:AB> = <XYZ> and <:AB:CD> = <XYZCD>

There are times when it is desirable to force evaluation of a symbol in a normal context when it would normally pass unevaluated. The character & preceding the symbol has this effect; it is exactly like : except that it acts only in a normal context. Continuing the previous example:

VW&AB = VWXYZ and

&AB = 12 is equivalent to XYZ = 12.

A string may be thought of as having two kinds of structure:

1. It is composed of a sequence of characters.
2. It is composed of a sequence of substrings delimited by commas not enclosed in parentheses, brackets, or quotes.

With reference to the first structure, a single character may be selected by a subscript enclosed in brackets. Referring to the string assigned to A, we note that

A[2] is <,>, A[6] is <D>, and A[7] is <)>.

By an obvious extension of this notation,

A[3,7] is <(C,D)> and A[9,11] is <E,F>.

Subscripts which reference the substring structure are enclosed in parentheses. Thus

A(1) = <B> and A(2) = <C,D>.

Note that a single pair of parentheses surrounding a substring is removed. Subscripting may be iterated:

A(2)(2) = <D>.

Subscripting is applied only to a string-valued symbol which is in a normal context or is evaluated by a colon. Subscripting of a name on the left side of an assignment forces it to be evaluated even if it is not preceded by a colon.

Two operations, .L and .LC, determine respectively the number of substrings and the number of characters in their arguments. Thus

.L(A) = 4, .L(A(2)) = 2 and .LC(A) = 11.

Having dealt with the general machinery for handling strings, we now turn to the slight refinement which adds macros with arguments to the system. This takes the form of a modification to the ordinary line assigning a string to a symbol, which permits an argument string to be specified. Thus

```
STORE <ARG> =
<.RPT.FOR T = 1, .L(ARG(2)),1
<ST&ARG(1)  ARG(2)(T)>>
```

defines a macro with two arguments, the first a string which, when appended to <ST>, creates a store instruction, and the second a list of locations to be stored into. Whenever STORE is used, the string of characters beginning with the first following nonblank character and ending with a line delimiter or unmatched right parenthesis is made the value of ARG. The string which is the value of STORE is then substituted for it as usual.

STORE might be called with

```
STORE A,(S1,S2,S3)
```

which is, because of the definition, equivalent to

```
.RPT.FOR T = 1,3,1
<STA <S1,S2,S3> (T)>
```

To complete the expansion we must consider the .RPT directive which has been used above. This directive causes the string which follows to be scanned repeatedly. It takes one of two forms:

1. .RPT N <...>

which causes N repetitions, or

2. .RPT.FOR J = nl,n2,n3 <...>

which causes (n2—nl)/n3 + 1 repetitions with J initially set to nl, and then incremented by n3 until it exceeds n2. Zero repetitions are possible. The n3 may be elided if it is 1.

The STORE macro call above may now be seen to expand into

```
STA S1
STA S2
STA S3
```

We illustrate with two further examples. The first is a generalized MOVE macro which takes as its arguments a sequence of pairs of lists. The first list of each

pair specifies the locations to load from, while the second gives the corresponding locations to store into. A list may of course have only one element.

```
        MOVE <ARG> =
<.RPT.FOR S1 = 1, .L(ARG),2
*THIS LINE STEPS THROUGH THE PAIRS OF
  LISTS
<.RPT.FOR S2 = 1, .L(ARG(S1))
*THIS LINE STEPS THROUGH THE ELEMENTS
  OF ONE PAIR OF LISTS

< LDA ARG(S1)(S2)
< STA ARG(S1 + 1)(S2) >>>
```

thus

```
        MOVE A,B,C,D
```

becomes

```
        LDA   A
        STA   B
        LDA   C
        STA   D
```

So does

```
        MOVE  (A,C),(B,D)
```

Suppose that we have some two-word data structures to manipulate. We can attach to the name of each structure a string of the form <A,B>. A is the address of the first word of the structure, B of the second. A macro can do this and assign the storage.

```
            TW <ARG> =
<           TWS1 = TWS + 1
            ARG(1) = <TW:TWS,TW:TWS1>
        TW&TWS, 0
        TW&TWS1, 0
            TWS = TWS + 2 >
```

Now, if we call TW twice after setting TWS to 1:

```
        TW      A
        TW      B
```

we will have given A the value <TW1,TW2> and B the value <TW3,TW4> and defined the four TW symbols.

We can now use A and B in the MOVE macro. In fact

```
        MOVE    A,B
```

expands to

```
        LDA     TW1
        STA     TW3
        LDA     TW2
        STA     TW4
```

With the addition of one more device we can proceed to the definition of a very grandiose macro. The directives .IF and .ELSF, used thus:

```
        .IF     E1  <...>
        .ELSF   E2  <...>
            .
            .
            .
        .ELSF   En  <...>
```

cause each $E_i$ in turn to be evaluated until one is greater than zero. The string following this one is then scanned and the rest of the structure ignored.

```
*THIS MACRO COMPILES AN ARITHMETIC EXPRESSION CONSISTING OF SINGLE-
*LETTER VARIABLES, BINARY + AND - AND PARENTHESES.  IT CALLS THE
*MACRO ERROR IF THE EXPRESSION IS NOT WELL FORMED.

    ARITH <ARG> =
  <  EXPR=<:ARG(1).>      *APPEND . TO THE EXPRESSION
    STK=<*>               *INITIALIZE THE STACK WHICH HANDLES
                          *PARENTHESES
    J=1                   *INITIALIZE THE CHARACTER POINTER
    TI=0                  *INITIALIZE THE TEMPORARY STORAGE COUNTER

*IF TEMPORARY STORAGE IS REQUIRED IT IS ASSIGNED AS TEMP1,
*TEMP2, ETC., AND TI KEEPS TRACK OF THE NEXT AVAILABLE LOCATION.

    X1                    *THIS IS THE MACRO WHICH DOES THE WORK
    .IF T .NE '.'         <ERROR> >

*CHECK THAT EXPRESSION WAS NOT TERMINATED BY A RIGHT PARENTHESIS.
*THIS MACRO COLLECTS A SUB-EXPRESSION CONSISTING OF OPERANDS
*STRUNG TOGETHER WITH + AND -.  IF THE SUBEXPRESSION IS A SINGLE
*VARIABLE, COP (CURRENT OPERAND) WILL BE THAT VARIABLE ON EXIT.
*OTHERWISE IT WILL BE EMPTY.
    X1  =
  <  COP = <***>          *ENSURE THAT COP IS NOT EMPTY INITIALLY

*AN EMPTY COP MEANS THAT CODE HAS BEEN ASSEMBLED LEAVING A VALUE
*IN THE A REGISTER.  IF COP IS A LETTER, IT IS THE VARIABLE
*WHICH IS THE CURRENT OPERAND.

    OPERAND               *GET THE FIRST OPERAND
    .RPT .FOR E=1,1,0     *E IS SET TO 2 WHEN THERE ARE NO MORE + OR -
                          *SIGNS
  <  T=':EXPR[J]'         *EXPECTING AN OPERATOR OR TERMINATION
    J=J+1
    .IF T .E '.' .OR T .E ')'  <E=2>

*SET E TO TERMINATE THE LOOP IN THIS CASE.

    .ELSF  T .E '+'  <COMPILE ADD,ADD>
    .ELSF  T .E '-'  <COMPILE SUB,(CMA;ADD)>

*IF A + OR - IS PRESENT, GET THE SECOND OPERAND AND COMPILE CODE.

    .ELSF 1  <ERROR>      *OTHERWISE, ERROR
    > >                   *CLOSE LOOP AND MACRO

*THIS MACRO COLLECTS THE SECOND OPERAND OF A BINARY OPERATOR AND
*CONSTRUCTS CODE TO PERFORM THE SPECIFIED OPERATION.  IT USES ITS
*FIRST ARGUMENT IF THE FIRST OPERAND IS IN THE A REGISTER, ITS
*SECOND ARGUMENT IF THE SECOND OPERAND MUST BE IN A AND THE FIRST
*TAKEN FROM MEMORY.

    COMPILE <CARG> =
  <  OPERAND              *GET THE SECOND OPERAND
    .IF  .LC(COP)  .G 0

*IN THIS CASE THE SECOND OPERAND IS A SINGLE VARIABLE.

  <  .IF  .LC (PREVOP)  .G 0   <LDA PREVOP>

*IF THE FIRST OPERAND IS ALSO A VARIABLE (OR A TEMP LOCATION)
*BRING IT INTO A

    CARG(1)  COP >        *AND COMPILE CODE
    .ELSF 1       <CARG(2) PREVOP>

*OTHERWISE THE SECOND OPERAND MUST BE IN A, AND THE FIRST IN MEMORY

    COP=< >       >

*SET COP TO INDICATE A VALUE IN A AND CLOSE THE MACRO.

*THIS MACRO COLLECTS AN OPERAND, WHICH MAY BE A PARENTHESIZED
*SUBEXPRESSION

    OPERAND=
  <  T = ':EXPR[J]'       *GET THE NEXT CHARACTER
    J=J+1                 *IT SHOULD BE A LETTER OR (
    .IF T .E '('
  <  .IF .LC(COP) .E 0

*IF WE ALREADY HAVE A VALUE IN A IT MUST BE SAVED IN TEMPORARY
*STORAGE WHILE THE SUBEXPRESSION IS EVALUATED.

  <  TI = TI +1;
    STA TEMP&TI           *CONSTRUCT A TEMP LOCATION TO SAVE IT IN
    COP=<TEMP:TI>  >      *AND REMEMBER IT IN COP
    STK=<:COP,:STK>       *STICK COP ON THE FRONT OF STK
    X1
    .IF T .NE '('  <ERROR>
    E=1                   *RESET THE TERMINATION SWITCH FOR X1
    PREVOP=<:STK(1)>      *SET PREVOP TO THE OLD COP WHICH WAS SAVED
    STK=<:STK(2,.L(STK))> >

*REMOVE OLD COP FROM STK AND TERMINATE THIS CASE.  X1 HAS SET COP

    .ELSF  T .GE 'A' .AND T .LE 'Z'

*IF T IS A LETTER (RECALL THAT THE CHARACTER CODE IS ASCII)

  <  PREVOP=<:COP>
    COP=<:EXPR[J-1]>  >
    .ELSF  1  <ERROR>  >
```

This macro, called by

```
ARITH ((A + B) — (C— D))
```

would generate

```
LDA A
ADD B
STA TEMP1
LDA C
SUB D
CNA
ADD TEMP1
```

Note that there are only three lines in the definition which actually generate code. The temporary storage location TEMP1 must be defined elsewhere.

The implementation of all this is quite straightforward. When a string is encountered, it is collected character by character, due attention being paid to colons, ampersands, brackets, and quotes, and stored away. When it is referenced, the routine which delivers characters to the assembler, which we will call CHAR, is switched from the input medium to the saved string. This process is of course recursive. When the string which is the current source of characters ends, CHAR is switched back to the string it was working on before. All the various occurrences of strings are treated perfectly uniformly, except that in the case of macro definitions the substrings of the argument string are delimited when the latter is collected to improve the efficiency. Perfectly arbitrary nesting of the various constructs is possible because of the recursiveness of the string collection and reference routines.

In the interests of efficiency the .IF directive is not handled in this way, since its subject string is scanned either once or not at all. All that is necessary is a flag which indicates whether an .ELSF directive is to be considered or ignored.

## THE DEBUGGING SYSTEM

An interactive debugging system should not be designed for the occasional user. Its emphasis must be on completeness, convenience, and conciseness, not on highly mnemonic commands and self-explanatory output. The basic capabilities required are quite simple in the main, but the form is all important because each command will be given so many times.

One essential, completely symbolic input and output is half taken care of by the assembler. The other half is easier than it might seem: given a word to be printed in symbolic form, the symbol table is scanned for an exact match on the opcode bits. If no match is found, the word is printed as a number. Otherwise the opcode mnemonic is printed, indirect and index bits are checked, the proper symbols printed, and the table is scanned for the largest symbol not greater than the remainder of the word. This symbol is printed out, followed if necessary by a + and a constant.

The most fundamental commands are single characters, possibly preceded by modifiers. Thus to examine a register the user types

```
/x1-3; LDA I NUTS + 2
```

where the system's response is printed in capitals. This command may be preceded by any combination of modifiers:

- C for printout in constant form
- S for printout in symbolic form
- O for octal radix
- D for decimal radix
- R for relative (symbolic) address
- A for absolute address
- H for printout as ASCII characters
- I for printout as signed integer
- N for no printing of addresses
- L (load) for no printing of register contents

The modifiers hold until the user types a carriage return or gives another / command.

For examining a sequence of registers, the commands + and — are available. The former examines the preceding register, the latter the following register. In the absence of a carriage return the modifiers of the last examination hold. The → command examines the register addressed by the one last examined.

The contents of a register may be modified after examination simply by typing the desired new contents. Note that the assembler is always part of the command processor, and that debugging commands are differentiated by their format from words to be assembled (as noted above, an assembler line has comma or space at its first punctuation character, and all debugger lines have some other initial punctuation character). Furthermore, debugging commands may occur in macros, so that very elaborate operations can be constructed and then called on with the two or three characters of a macro name.

To increase the flexibility of debugging macros, the unary operator @ is defined. The value of @ SYM3 is the *contents* of location SYM3. With this operator, macros may be defined to type out words depending on very complicated conditions. A simple example is

```
TG<A> =
< .RPT.FOR TEMP = A(1),37777,1
  *SCAN THROUGH ALL OF STORAGE STARTING
      AT THE LOCATION GIVEN BY
  *THE FIRST ARGUMENT
< .IF @ TEMP E. (A)2
  *IF THE CURRENT LOCATION MATCHES THE
```

```
        SECOND ARGUMENT, THE SCAN IS OVER
</TEMP;                 *PRINT OUT THE
        TEMP1 = TEMP      CONTENTS
        TEMP = 37777    *SAVE THE ADDRESS
        >>>>            *AND TERMINATE
                          THE SCAN
```

Called with

TG 100,20

it will type out the first location after 100 with contents greater than 20.

Another important command causes an expression to be typed in a specified format. Thus if SYM has the value 1253 then

= sym; 1253

would be the result of giving the = command. All the modifiers are available but the normal mode of typeout is constant rather than symbolic. If no expression is given, the one most recently typed is taken. Thus, after the above command, the user might try

s= ; SYM (the system's response the symbolic equivalent of 1253, follows the ;)

It is often necessary to search storage for occurrences of a particular word. This may be done with a macro, as indicated above, but long searches would be quite slow. A faster search can be made with

↑expression;

which causes all the locations matching the specified expression to be typed out. The match may be masked, and the bounds of the search are adjustable. This command takes all the typeout modifiers as well as

E

which searches for a specified effective address (including indexing and indirect addressing) and

X

which searches for all exceptional words (which do *not* match). For additional flexibility the user may specify a macro which will be executed each time a matching word is found.

In addition to being able to examine and modify his program, the user also needs to be able to run it. To this end he may start it at a specified location with

,G location

If he wishes to monitor its progress he may insert breakpoints at certain locations with the command

,B location

This causes execution of the program to be interrupted at the specified location. Control returns to the system, which types some useful information and awaits further commands. An alternate form of this command is

,B location,marco name

which causes the specified macro to be executed at each break, instead of returning control directly to the typewriter. Very powerful conditional tracing may be done in this way.

After a break has occurred, execution of the program may be resumed with the ,P command. The breakpoint is not affected. To prevent another break until the breakpoint has been passed n time the form

\n;

may be used. Modifiers may precede the command.

To step through the program, instruction by instruction, the command ,S may be used instead of ,P. It allows one instruction to be executed and then breaks again. $n; allows n instructions to be executed before breaking. A fully automatic trace has been deliberately omitted, but presents no difficulties in principle.

## THE EDITOR

There remains one feature of great importance in the IMP system, the symbolic editor. The debugger provides facilities, which have already been described, for modifying the contents of core. These modifications, however, are not recorded in the symbolic version of the program. To permit this to be done, so that reloading will result in a correctly updated binary program, several commands are available which act both on the assembler binary and on the symbolic.

This operation is not as straightforward as it might appear, since there is no one to one correspondence between lines of symbolic and words of binary. Addresses given to the debugger of course refer to core locations, but for editing it is more convenient to address lines of symbolic. To permit proper correlation of these line references with the binary program, a copy of the symbolic file is made during loading with the address of the first and last assembled words explicitly appended to each line. Since the program is not moved around during editing, these numbers do not change except locally. When a debugging session is complete, the edited symbolic is rewritten without this information.

We illustrate this with an example. Consider the symbolic and resulting binary

```
S1  MOVE A,B   (200,201)   S1  LDA   A   200
                               STA   B   201
    ADD C      (202,202)       ADD   C   202
    STORE D,E  (203,204)       STA   D   203
                               STA   E   204
S2  BRU S1     (205,205)   S2  BRU   S1  205
```

and the editing command

```
,I   S2-1        insert before line S2-1
     SUB F
```

which gives rise to the following:

```
S1  MOVE A,B   (200,201)   S1    LDA   A     200
                                 STA   B     201
    ADD C      (202,1512)        BRU   .END  202
```

```
    SUB F       (1513,1513)        BRU  .END 1   203
    STORE D,E   (1514,204)         STA  E        204
S2  BRU S1      (205,205)      S2  BRU  S1       205
                                   ...
                               .END ADD  C        1512
                                    SUB  F        1513
                                    STA  D        1514
                                    BRU  S1 4     1515
                                    BRU  S1 5     1516
```

All the BRU (branch unconditional) instructions are inserted to guarantee that the right thing happens if any of the instructions causes a skip. The alternative to this rather simple-minded scheme appears to be complete reassembly, which has been rejected as too slow. The arrangement outlined will deal correctly with patches made over other patches; although the binary may come to look rather peculiar, the symbolic will always be readable.

To give the user access to the readable symbolic the command

,L symbolic line address[,symbolic line address];

(where the contents of the brackets is optionally included) causes the specified block of lines to be printed. Two other edit commands are available:

,D symbolic line address[,symbolic line address];

which deletes the specified block of lines, and

,C same arguments;

which deletes and then inserts the text which follows. Deleting S1 1 from the original program would result in binary as follows

```
S1      LDA   A
        BRU   .END
        BRU   .END 1
        STA   D
        STA   E
S2      BRU   S1
        ...
.END    STA   B
        BRU   S1 3
```

The implementation of these commands is quite straightforward. One entire edit command is collected and the new text, if any, is assembled. Then the changed core addresses are computed and the appropriate record of the symbolic file rewritten.

The scheme has two drawbacks: It does not work properly for skips of more than one instruction or for subroutine calls which pick up arguments from following locations, and it leaves core in a rather confusing state, especially after several patches have been made at the same location. The first difficulty can be avoided by changing large enough segments of the symbolic. The second can be alleviated by reassembly whenever things get too unreadable.

The only other published approach to the problem of patching binary programs automatically is that of Evans,[6] who keeps relocation information and relocates the entire program after each change. This procedure is not very fast, and in any event is not practical for a system with no relocation.

## EFFICIENCY

The IMP system depends for its viability on fast assembly. The implementation techniques discussed in this paper have permitted the first version of the assembler to attain the unremarkable but satisfactory speed of 200 lines per second. Simple character handling hardware would probably double assembly speed on simple assemblies and produce even greater improvement on programs with many macros and repeats.

Using the latter figures, we deduce that a program of 10,000 instructions, a large one by most standards, will load in 25 seconds. This number indicates that the cost of the IMP approach is not at all unreasonable—far more computer time, including overhead, is likely to be spent in the debugging operations which follow this load. When only minor changes are made it is, of course, possible to save the binary core image and thus avoid reloading.

In spite of the speed of the assembler, it is possible that a relocatable loader might be a desirable adjunct to the system. There are no basic reasons why it should not be included.

As to the size of the system, the assembler is about 2500 instructions, the debugger and editor about 2000.

## ACKNOWLEDGMENTS

The ideas in this paper owe a great deal to many stimulating conversations between the author and Peter Deutsch. I am especially indebted to him for the idea that all strings in the input can be handled uniformly with string brackets. A system very similar to this one has been implemented by him for the CDC 3100.

## REFERENCES

1. M. Halpern, "XPOP—A Metalanguage without Metaphysics," *AFIPS Conf. Proc.*, vol. 25, 1964.
2. G. Mealy, "Anatomy of an Assembly System," RAND Corporation (Dec. 1962).
3. S. Boilen et al, "A Time-Sharing Debugging System for a Small Computer," *AFIPS Conf. Proc.*, vol. 23, Spartan Books, Washington D.C., 1963, pp. 51-58.
4. L. P. Deutsch and B. W. Lampson, "DDT—Time-Sharing Debugging System Reference Manual," Project GENIE Doc. 30.40.10 (May 1965).
5. "The MIDAS Assembly Program," internal memorandum, M.I.T.
6. Thomas G. Evans and D. L. Darby, "DEBUG—

An Extension to Current Online Debugging Techniques," *Comm. ACM,* vol. 8, no. 5, pp. 321-25 (May 1965).

7. C. N. Mooers, "TRAC, A Procedure-Describing Language for the Reactive Typewriter," *Comm. ACM* vol. 9, no. 3, pp. 215-219 (Mar. 1966).