INTERACTIVE MACHINE LANGUAGE PROGRAMMING

Butler W. Lampson
University of California
Berkeley, California

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.\(^1\,2\) Granted that less than half of the binary instructions generated in the past year had their origin in assembly language, it remains likely that much more than half the instructions executed originated in this way. 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; 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 are relatively unfamiliar.\(^3\) 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 is 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

\(^*\)The work described in this paper was supported by the Advanced Research Projects Agency of the Department of Defense under contract SD-185.
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 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 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 in the following. 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 ASSEMBLER

The input format of the IMP assembler is a rather unusual one. Originated on the TX-0 at MIT, 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 non-negligible consideration, often equally 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). The expression is evaluated and put into core at the word addressed by the location counter, and 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: +, -, ×, /, A, V, = with their usual meaning and precedence; =, <, and >, which are binary operators with precedence less than +, and which yield 0 or 1 depending on whether the indicated relation holds between the operands; and ≠, a unary operator with lowest precedence which causes its operand to be taken as a literal — i.e., 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.

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 sin­
gle instruction. It would work on the PDP-6, but
not on Stretch.

The idea is simple: all operation code mnemon­
ics 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 in­
struction 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 pro­
grammer 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 relo­
catability.

Two rather conventional methods are provided
for defining symbols. A symbol appearing at the
left edge of a line is defined as the current value of
the location counter. Such a symbol may not be re­
defined. In addition, a line such as
SYM = 4600
defines SYM. Any earlier definition is simply over­
ridden. The right side may of course be any expres­
sion which can be evaluated.

The special symbol \( \langle \) refers to the location
counter. It may appear on the left of a \( \leftrightarrow \) sign. Thus, the line

\[
\text{A } \leftrightarrow \text{ 40}
\]

is equivalent to

\[
\text{A BSS 40}
\]

in a conventional assembler.

There remains one point about the basic assem­
blers which is crucially important to the implemen­
tation: 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. Because the IMP
assembler must accept typewriter input, it cannot be
two pass and must therefore keep track of unde­
defined 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 special case can be treated much
more efficiently, and this is the case of an instruc­
tion with an undefined address, which includes at
least 90 percent of the occurrences of undefined
expressions.

For example, when the assembler sees this code:

\[
\text{X BRU A \hfill branch unconditional} \\
\text{LDA B} \\
\text{A STA C}
\]

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 con­
taining the undefined symbol A as an address. The
list is terminated by filling the address field with
ones. 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.

The description of the basic assembler is now complete, except for a few nonessential details, and we turn to the macro facility. Macros are handled in a standard manner, which the following example should sufficiently illustrate:

STORE MACRO ARG1, ARG2
  IRP TEMP=ARG2 indefinite repeat
  STA ARG1 TEMP
  ENDR
ENDM STORE

called with
STORE A,(S1,S2,S3)

becomes after argument substitution
IRP TEMP=S1,S2,S3
STA TEMP
ENDR

That is, this string of characters is seen by the assembler as though it were in the symbolic input.

A macro may be defined with more arguments that it is called with, in which case the extra arguments are made either null strings or generated symbols. No more arguments are collected than are called for by the definition. An argument is normally collected literally, character by character, but a colon appearing before a macro name will cause it to be expanded. To provide additional flexibility, two directives called STACK and UNSTACK are provided which respectively suspend the analysis of the current expression and resume it.

Some unusual things may be done with this much machinery. Consider the macro

LIT MACRO ARG, GEN
  STACK
  TEMP::<LITERALS
  GEN
  LITERALS=::LITERALS 1
  TEMP
  UNSTACK
  GEN
ENDM LIT

Called with
LDA LIT 20

it will assign a storage location, say LITERALS+10, put 20 in it, and assemble

LDA LITERALS+10

There are many other ways of writing this macro using the list features discussed below.

The IRP operation used above is not new, but it is not well known. It causes the lines in its range, which is delimited by a matching ENDR, to be processed repeatedly by the assembler. Each time around the argument, TEMP in this case, is replaced by one of the subarguments, which are the character strings following the = sign and separated by commas. The entire process is rather similar to a macro call, and subarguments are processed according to the rules for macro arguments, except that parentheses are not removed. Thus the IRP generated by the expansion of the macro discussed above in turn expands into

STA S1                TEMP replaced by S1
STA S2                TEMP replaced by S2
STA S3                TEMP replaced by S3

Two extensions of this device:
IRP A,B=A1,B1,A2,B2,A3,B3 $ C=C1,C2
A,B,C
ENDR

expands into
A1,B1,C1,
A2,B2,C2
A3,B3,C2

We illustrate with another macro definition:
MOVEMACRO ARG
IRP TEMPA, TEMPB=ARG
IRP TEMPC=:TEMPCA $ TEMPD
=:TEMPB
LDA TEMPC
STORE TEMPD
ENDR
ENDR
ENDM

Called by
MOVE (A1,(B1,C1),A2,B2)

this expands into
LDA A1
STA B1
STA B2
LDA A2
STA B2

Suppose that we have some two-word data structures which we wish to manipulate. We can define each of them as a macro, using another macro to do the definition and reserve storage:

TW MACRO ARG, GENA, GENB
GENA 0
Now, if we call TW:

```
TW A
TW B
```

we can then use the newly defined macros A and B in the move macro. In fact:

```
MOVE (A,B)
```

after character substitution both in the macro body and in the first IRP body is:

```
IRP TEMPA,TEMPB=A,B
IRP TEMPC=.G0001, .G0002 $ TEMPD
    =.G0003, .G0004
```

which expands to:

```
LDA .G0001
STA .G0003
LDA .G0002
STA .G0004
```

There are two other repeat directives,

```
RPT expression
ENDR
```

which repeats its scope the number of times specified by the expression, and

```
CRPT expression
ENDR
```

which repeats its scope, reevaluating the expression each time until it is \( < 0 \).

Finally, there is a conditional directive:

```
IF expression
ELSE expression
ENDF
```

which causes the lines between the first IF or ELSF whose argument is \( > 0 \) to be assembled and everything else to be ignored.

The implementation of all this is quite straightforward, and very similar for macros and repeats. The body of the macro definition or repeat is collected as a character string, with markers replacing the arguments, and saved away. Each time it is called, the routine which delivers characters to the assembler, which we will call CHAR, is switched from the input medium to the saved string. Arguments for a macro call or IRP are likewise saved as strings. The characters coming from a definition are monitored for the argument marker, and if it is found CHAR is switched again, to the argument string. Whenever any of these strings ends, CHAR is switched back to the string it was working on before.

All this machinery is of course recursive. The only restriction is that macro definitions and repeats must be properly nested. Note that because of the implementation technique just described a macro definition may contain anything, including other definitions. The other definitions of course are not made until the macro is called.

The most novel feature of this assembler is the string or list-manipulating features available to the programmer, which allow him to define macros to perform functions normally regarded as the prerogative of a compiler. A list may be assigned to a symbol as its value by:

```
SYM←[any string not containing an unbalanced right bracket]
```

The string is saved literally as the value of SYM, with one exception: the character : causes the following symbol to be expanded if it is a macro or list name, just as it does in macro arguments. The structure : [string] is equivalent to the string alone.

Once SYM has been equated to a list, any use of it is exactly equivalent to writing the contents of the string, including the brackets. Exception: If SYM appears within brackets or as a macro argument and is not preceded by : it is transmitted literally. In most contexts a string enclosed in brackets has the same effect as one not so enclosed.

Thus the sequence:

```
SYMA←[A]
SYMB←[B]
SYMC←[:SYMA:SYMB,CD]
```

will leave SYMC with the value [AB,CD]. The ' has the same function as it does in a macro definition.

A symbol equated to a list (or, as always, the explicit list itself) may be subscripted in two ways:

In the above example, SYMC[2] is equivalent to B (i.e., a subscript in bracket selects a single character). More generally, SYMC[2,5] is equivalent to B,CD.

The other form of subscripting selects a segment of a list delimited by commas: SYMC(1) is the same as AB.

To illustrate the use of these features we consider the following macro to compile an expression
with the operators + and −, single character variables and parenthetization:

\[
\begin{align*}
\text{ARITH} & \quad \text{EXPR} \\
& \quad \text{ARG} \leftarrow \text{EXPR} \\
& \quad \text{SB} \leftarrow 0 \\
& \quad L \leftarrow \text{LENGTH}(\text{ARG}) \\
& \quad \text{ARITH1} \\
& \quad \text{ENDM ARITH}
\end{align*}
\]

\[
\begin{align*}
\text{ARITH1} & \quad \text{MACRO OPA,OPB,OP} \\
& \quad \text{OP} \leftarrow 0 \\
& \quad \text{OPA} \leftarrow 0 \\
& \quad \text{CRPT SB} < L \quad \text{V} \quad \text{OP} \leftarrow -1 \\
& \quad \text{SB} \leftarrow \text{SB} + 1 \\
& \quad \text{C} \leftarrow "\text{:ARG}\{\text{SB}\}" \\
& \quad \text{IF OP} = 0 \\
& \quad \text{IF C} \leftarrow "("  \\
& \quad \quad \text{\textit{this branch if operator not yet found}} \\
& \quad \text{ARITH1} \\
& \quad \quad \text{OPA} \leftarrow 1 \\
& \quad \quad \text{ELSF C} \leftarrow "-" \\
& \quad \quad \text{OP} \leftarrow 2 \\
& \quad \quad \text{ELSF C} \leftarrow "+" \\
& \quad \quad \text{OP} \leftarrow 1 \\
& \quad \quad \text{ELSF C} \leftarrow ")" \\
& \quad \quad \text{OP} \leftarrow 1 \\
& \quad \quad \text{ELS 1} \\
& \quad \quad \text{OP} \leftarrow -2 \\
& \quad \quad \text{OPB} \leftarrow [:\text{ARG}\{\text{SB}\}] \\
& \quad \quad \text{ENDF} \\
& \quad \quad \text{ELSF C} \leftarrow "("  \\
& \quad \quad \quad \text{\textit{this branch if operator found}} \\
& \quad \quad \quad \text{IF OPA} = 1 \\
& \quad \quad \quad \quad \text{\textit{this branch if second operand is }} () \\
& \quad \quad \quad \text{TIDX} \leftarrow \text{TIDX} - 1 \\
& \quad \quad \quad \text{STA T\text{\textasciitilde}NUM} (\text{TIDX}) \\
& \quad \quad \quad \text{OPB} \leftarrow [:\text{T\textasciitilde}NUM\{\text{TIDX}\}] \\
& \quad \quad \text{ARITH1} \\
& \quad \quad \text{TIDX} \leftarrow \text{TIDX} - 1 \\
& \quad \text{ELS 1} \\
& \quad \text{ARITH1} \\
& \quad \text{ENDF} \\
& \quad \text{IF OP} \leftarrow 2 \\
& \quad \text{CNA}  \quad \text{\textit{complement A register}} \\
& \quad \text{ENDF} \\
& \quad \text{ADD OPB} \\
& \quad \text{ELS 1} \\
& \quad \text{IF OPA} \leftarrow 2 \\
& \quad \text{LDA OPB} \\
& \quad \text{ENDF} \\
& \quad \text{IF OP} \leftarrow 1 \\
& \quad \text{ADD ARG}\{\text{SB}\}
\end{align*}
\]

\[
\begin{align*}
\text{ELS} & \quad \text{OP} \leftarrow 2 \\
& \quad \text{SUB ARG}\{\text{SB}\} \\
& \quad \text{ENDF} \\
& \quad \text{OPA} \leftarrow 1 \\
& \quad \text{OP} \leftarrow 0 \\
& \quad \text{ENDF} \\
& \quad \text{ENDR} \\
& \quad \text{ENDM ARITH}"
\end{align*}
\]

\[
\begin{align*}
\text{NOTES}
\end{align*}
\]

LENGTH is a function which gives the length of the list which is its argument.

NUM evaluates its argument and replaces itself with the decimal encoding of the value. It is useful for constructing a series of symbols over which one wants considerable control.

Double quotes enclosing a string turn it into an alphanumeric constant.

This macro, called by

\[
\text{ARITH } [(A+B)-(C-D)]
\]

would generate

\[
\begin{align*}
\text{LDA A} \\
\text{ADD B} \\
\text{STA T1} \\
\text{LDA C} \\
\text{SUB D} \\
\text{CNA} \\
\text{ADD T1}
\end{align*}
\]

Note that there are only six lines in the definition which actually generate code.

With this example we conclude our discussion of the assembler. The implementation of lists is quite straightforward, though a certain amount of care must be taken about the treatment of colons calling for expansion. A few minor points have been glossed over.

\[
\begin{align*}
\text{THE DEBUGGING SYSTEM}
\end{align*}
\]

A good interactive debugging system must be difficult for the beginner to master. Its emphasis must be on completeness, convenience and conciseness, not on simplicity. 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 and 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

```
/S1-3; 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

The modifiers hold until the user types a carriage return.

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. (This is not difficult, because the only thing which may occur at the beginning of a line of assembler code is a label.) 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 SYM 3 is the contents of location SYM 3. With this operator macros may be defined to type out words depending on very complicated conditions. A simple example is

```
TG MACRO A, B
```

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;
```

would be the result of giving the = command. All the modifiers are available but the normal mode of type-out 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
```

For convenience, ← abbreviates S=.

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 type-out modifiers as well as

- **E** which searches for a specified effective address (including indexing and indirect addressing) and
- **N** which searches for all 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, macro 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 times the form n;P may be used.

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

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 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

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI MOVE A, B (200,201)</td>
<td>S1 LDA A 200</td>
</tr>
<tr>
<td>SI LDAA 200</td>
<td>STA B 201</td>
</tr>
<tr>
<td>ADD C (202,202)</td>
<td>ADD C 202</td>
</tr>
<tr>
<td>STORE D, E (203,204)</td>
<td>STA D 203</td>
</tr>
<tr>
<td></td>
<td>STA E 204</td>
</tr>
<tr>
<td>S2 BRU S1 (205,205)</td>
<td>S2 BRU S1 205</td>
</tr>
<tr>
<td>and the editing command</td>
<td></td>
</tr>
<tr>
<td>;I S2-1 insert before line S2-1</td>
<td></td>
</tr>
<tr>
<td>SUB F</td>
<td></td>
</tr>
</tbody>
</table>

which gives rise to the following:

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI MOVE A, B (200,201)</td>
<td>S1 LDA A 200</td>
</tr>
<tr>
<td>S1 LDA A</td>
<td>STA B 201</td>
</tr>
<tr>
<td>ADD C (202,202)</td>
<td>ADD C 202</td>
</tr>
<tr>
<td>SUB F (1513,1513)</td>
<td>SUBF .END 1 203</td>
</tr>
<tr>
<td>STORE D, E (1514,204)</td>
<td>STA E 204</td>
</tr>
<tr>
<td>S2 BRU S1 (205,205)</td>
<td>S2 BRU S1 205</td>
</tr>
<tr>
<td>ENDADD C 1512</td>
<td></td>
</tr>
<tr>
<td>SUB F 1513</td>
<td></td>
</tr>
<tr>
<td>STA D 1514</td>
<td></td>
</tr>
<tr>
<td>BRU S1 4 1515</td>
<td></td>
</tr>
<tr>
<td>BRU S1 5 1516</td>
<td></td>
</tr>
</tbody>
</table>

All the BRU (branch unconditional) instructions are inserted to guarantee that the right thing happens if any of the instructions causes a skip. Multiple skips, or subroutine calls which pick up arguments from subsequent locations, are not handled correctly. 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

;S symbolic line address [,symbolic 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 would result in binary as follows

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 LDA A</td>
<td></td>
</tr>
<tr>
<td>BRU .END</td>
<td></td>
</tr>
<tr>
<td>BRU .END 1</td>
<td></td>
</tr>
<tr>
<td>STA D</td>
<td></td>
</tr>
<tr>
<td>STA E</td>
<td></td>
</tr>
</tbody>
</table>
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, 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 will be installed shortly on our 930; it is expected to double assembly speed on simple assemblies and to 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 2,500 instructions, the debugger and editor about 2,000.

ACKNOWLEDGMENTS

The ideas in this paper owe a great deal to many stimulating conversations between the author and L. Peter Deutsch of the University of California.

REFERENCES

2. The MIDAS Assembly Program, MIT, Cambridge, Mass.