Kolsky

A FAST ASSEMBLY TECHNIQUE USING APL

H. J. MYERS

Declassified
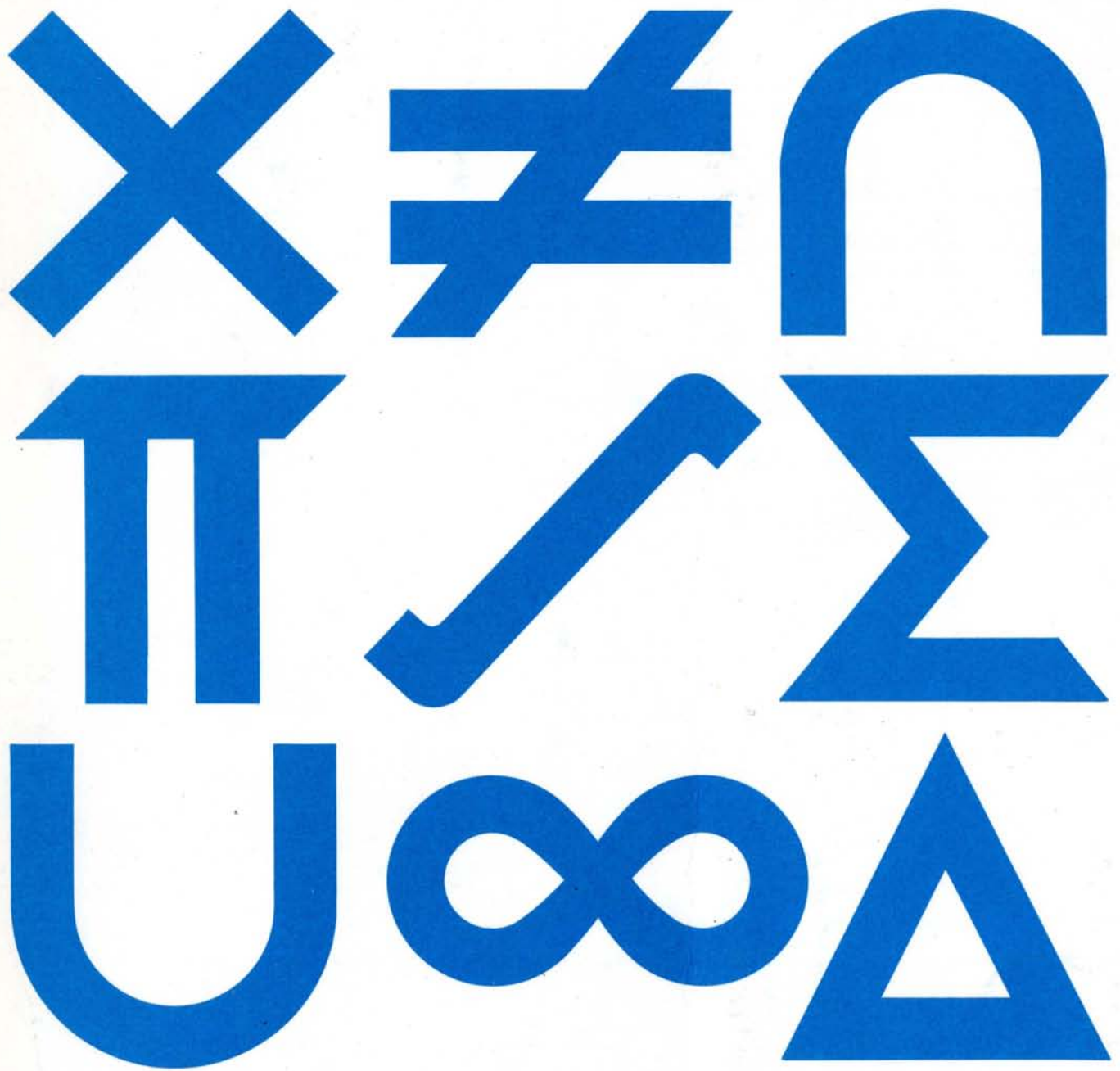
1974 IBM PALO ALTO SCIENTIFIC CENTER REPORTS
IBM CONFIDENTIAL AND IBM INTERNAL USE ONLY

ZZ20-6426   March 1974
P. SMITH and K. PRICE — An Architectural and
Design Overview of SCAMP (42 p.) IBM Confidential
until March 1984, Limited Distribution

ZZ20-6427   October 1974
HARRY F. SMITH, JR. — VM/7 Virtual Memory for
the S/7 (68 p.) IBM Confidential until November 1979.

1975 IBM PALO ALTO SCIENTIFIC CENTER REPORTS
IBM CONFIDENTIAL AND IBM INTERNAL USE ONLY

ZZ20-6428   February 1975
A. HASSITT and L. E. LYON — The APL Assists
(RPQ S00256) (115 p.) IBM Internal Use Only

ZZ20-6429   June 1975
M. J. BENISTON, R. J. CREASY, A. HASSITT,
J. W. LAGESCHULTE, L. E. LYON — Writing an
APL/CMS Auxiliary Processor (with complete example
code) (76 p.) IBM Internal Use Only

1976 IBM PALO ALTO SCIENTIFIC CENTER REPORTS
IBM CONFIDENTIAL AND IBM INTERNAL USE ONLY

ZZ20-6431   June 1976
H. J. MYERS - A Fast Assembly Technique
using APL (19 p.) IBM Internal Use Only

ZZ20-6495 August 1975
Abstracts of IBM Confidential and IBM Internal Use
Only Palo Alto Scientific Center Reports (43 p.)
IBM Confidential

The availability of reports is correct as of the printing date of this report.

* These reports are available only on the need to know basis, please contact the Scientific Center for information on copies.

● Copies are no longer available from the Scientific Center.

A Fast Assembly Technique using APL

H. Joseph Myers

IBM Scientific Center

P. O. Box 10500

Palo Alto, California 94304

JUNE 1976

ABSTRACT

A technique is described which reduces the cost of producing assemblers for a wide variety of machine architectures. Assembly is accomplished by executing each instruction of the source program as an APL function. An assembler has been generated capable of speeds of about 2000 lines per minute in an APL environment on an IBM System 370/145.

Index Terms for the IBM Subject Index

APL
Assemblers
Performance

INTRODUCTION

The recent years have seen the introduction of many microcomputers in a wide variety of machine architectures. The prices of these machines are extremely low (on the order of a few hundred dollars). Neither the vendors nor the users of these machines can invest much capital in programming support for them without losing the advantages of their low cost.

APL presents an excellent environment for low cost programming. The nature of the APL language also makes it attractive for implementing computer simulators. (Such simulators could be used by vendors to validate their machine designs, and by buyers to check out their application programs before actual acquisition of the harware.) It is naturally desirable to provide an assembler in the same environment as the simulator. However, assemblers written in APL usually execute very slowly (about 100 times slower than comparable assemblers written in machine language on the same computer).

A technique has been developed that overcomes this speed drawback, and allows production of (non-macro) assemblers with performance in the neighborhood of their machine-language counterparts. It also reduces the time to produce an assembler in an APL environment from 4-5 man-weeks to one or two man-days. The rapid availability and low cost of this type of assembler will be of considerable benefit to both vendors and buyers of this new breed of inexpensive computer.

OVERVIEW OF THE METHOD

In order to understand the method of fast assembly, one should first think about the functions an assembler performs. Each line of the source program must be scanned to isolate the tokens of the language. ("Tokens" are labels, op-codes, parentheses, commas and other atomic components that make up the assembler statements.) Labels must be entered into a symbol table; op-codes must be looked up in an op-code table to select the appropriate actions for each line. These and many other language processing functions must be carried out by the assembler. The nature of these language processing functions is not unique to assemblers. Indeed, the APL system also performs many of them.

The typical approach to building an assembler in an APL environment would consist of writing (in APL) subroutines that would read a line of source code, break it up into tokens, store labels in a symbol table, look up op-codes, and so forth. The fast assembly technique involves harnessing these functions already inherent in the APL system itself. By doing this we not only avoid the coding of these functions, but achieve dramatic performance benefits because these functions (within the APL system) are coded in machine language.

The organization of a typical two-pass assembler is diagrammed in

Figure 1. Pass 1 is principally concerned with allocating storage for each instruction and constant, and with assigning values to symbolic labels in the program. Pass 2 uses information developed by pass 1 to assemble the bit patterns of instructions and constants that constitute the program into a loadable format, and to list the results. A symbol table and inter-pass file constitute the principal data linking the two passes. The symbol table contains labels and their values, and the inter-pass file contains a copy of the source program, usually encoded for efficient interpretation by pass 2.
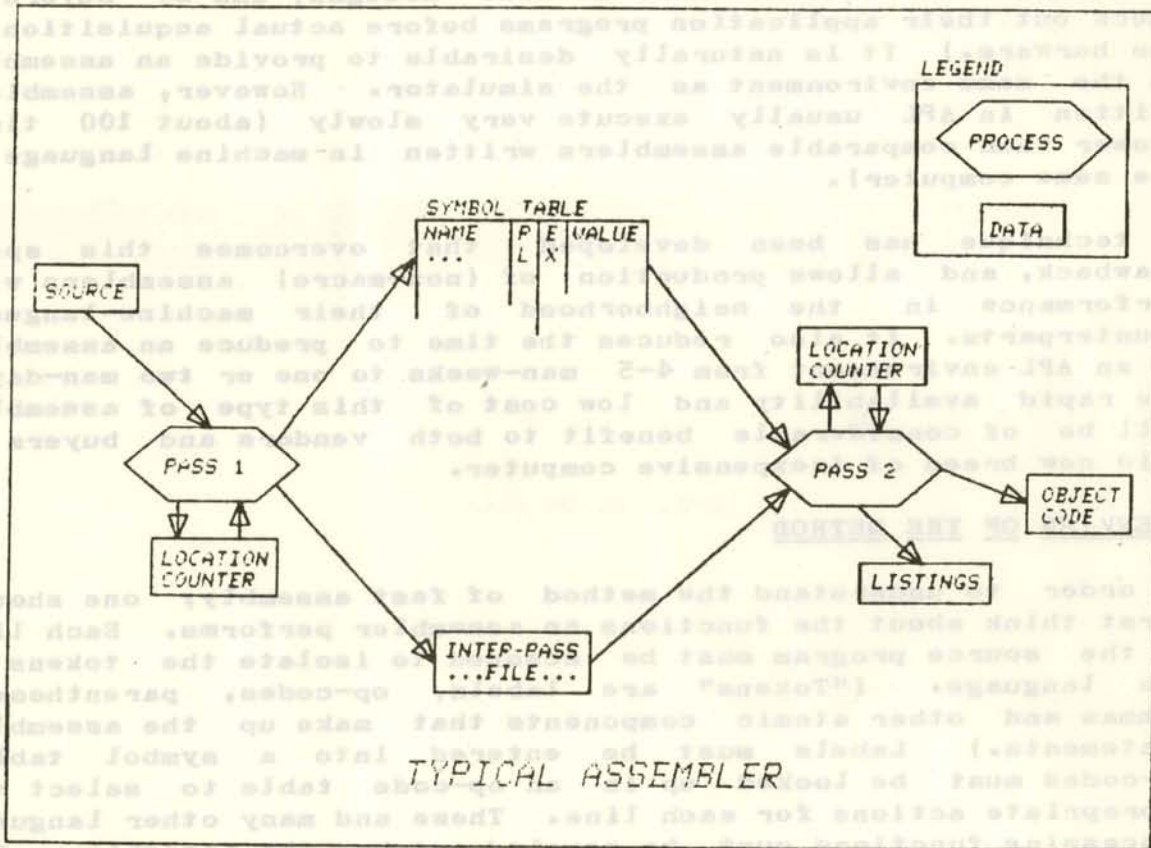

Figure 1.

The fast assembly technique takes advantage of a similarity between the syntaxes of APL and assembly language. One can view a line of assembly code as a function call. The instruction mnemonic is the name of the function. The operand fields of the instruction, separated by commas, are catenated to form the (vector) argument of the function. Thus the APL function *ADD* would generate the bit pattern for an ADD instruction in the target machine. (Note that throughout this report the names of APL functions and variables will be italicized.) With this view, an assembly source program would be an APL function consisting of a series of calls upon *ADD* and other such "assembly functions".

Furthermore, execution of this source program (in the proper context) would actually perform the entire assembly. In essence, this view is the core of the method we will call the "fast assembly" technique.

```
        ∇ SOURCE
[1]     ⍝ SAMPLE SOURCE PROGRAM
[2]     ENTRY A4,ERR
[3]     EXTRN'X1,X2'
[4]     ADD NB,NX
[5]     A1:ADDI NB,0
[6]     IF NX,GT,NB,A1
[7]     CGOTO NB,ERR,A1,A2,A3,A4,A5,A6
[8]     ⍝ START OF BRANCH GROUP
[9]     A2:ADD NA,NC
[10]    A3:EQU A2+4
[11]    A4:ADD NB,NX-1
[12]    A5:ADD(A1+1),X2
[13]    A6:ADDI X1,⁻5
[14]    ⍝ CONSTANTS
[15]    NB:DC 3
[16]    NA:DS 20
[17]    NX:DC 5
[18]    NC:ORG 100
[19]    ERR:DC A4
[20]    END
        ∇
```
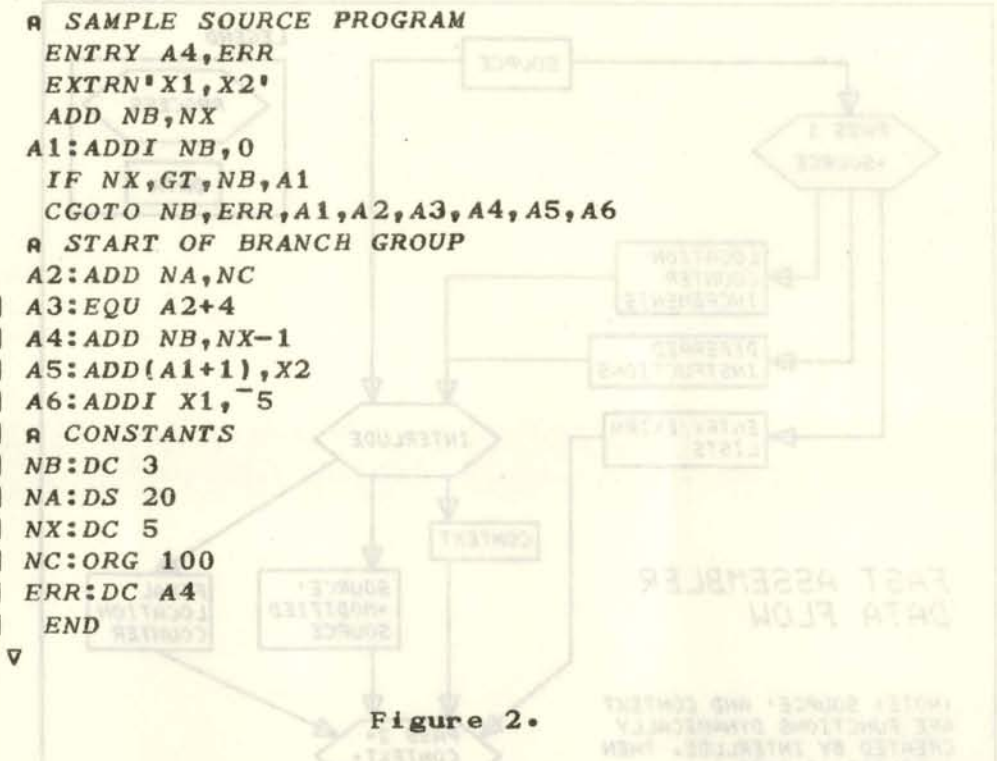
Figure 2.

A sample source program is shown in Figure 2. This program is both an APL program and an assembly source program for a hypothetical computer. When SOURCE (in Figure 1) is executed, the first line is skipped because it is a comment. The second line invokes the function ENTRY. ENTRY performs the ENTRY assembly function (described in detail later). The third line invokes the EXTRN function, the fourth line, the ADD function and so on. By constraining the syntax of the assembly language to conform to that of APL we can cause this program to take on a dual function of allowing the language processing functions of the APL system to be applied to the task of assembly. As a result, approximately two orders of magnitude in speed improvement can be achieved over coding these language functions in APL.

The data flow for the fast assembly technique is shown in Figure 3. Pass 1 executes the source program, SOURCE for example, to collect storage allocation information. Each function called by SOURCE is capable of operating in each of two modes -- pass 1 mode, and pass 2 mode. Because APL line labels have values that are APL line numbers, (not related to assembly values), operand fields are ignored during pass 1. (Operand fields of machine instructions typically are not evaluated during pass 1 anyway.) Instead, those instructions ordinarily requiring operand

evaluation during pass 1 are deferred. (Their pass 1 action is to place themselves on a deferral list.) After pass 1 but before pass 2, an "interlude" process is carried out. The function of the interlude is to create the pass 2 context for the second execution of *SOURCE*.

```
                    ┌──────────┐              LEGEND
                    │  SOURCE  │           ┌──────────────┐
                    └──────────┘           │  ⬡ PROCESS   │
      ⬡ PASS 1                             │              │
        =SOURCE                            │  ▭ DATA      │
                                           └──────────────┘
        ┌──────────────┐
        │  LOCATION    │
        │  COUNTER     │
        │  INCREMENTS  │
        └──────────────┘
        ┌──────────────┐
        │  DEFERRED    │
        │  INSTRUCTIONS│
        └──────────────┘
        ┌──────────────┐         ⬡ INTERLUDE
        │  ENTRY/EXTRN │
        │  LISTS       │
        └──────────────┘
   FAST ASSEMBLER      ┌─────────┐
   DATA FLOW           │ CONTEXT │
                       └─────────┘  ┌──────────┐   ┌──────────┐
                                    │ SOURCE'  │   │ FINAL    │
   (NOTE: SOURCE' AND CONTEXT       │ =MODIFIED│   │ LOCATION │
   ARE FUNCTIONS DYNAMICALLY        │ SOURCE   │   │ COUNTER  │
   CREATED BY INTERLUDE, THEN       └──────────┘   └──────────┘
   EXECUTED AS PASS 2.)
                          ⬡ PASS 2=
                            CONTEXT+
                            SOURCE'

                   ┌──────────┐    ┌─────────────┐
                   │ LISTINGS │    │ OBJECT CODE │
                   └──────────┘    └─────────────┘
```
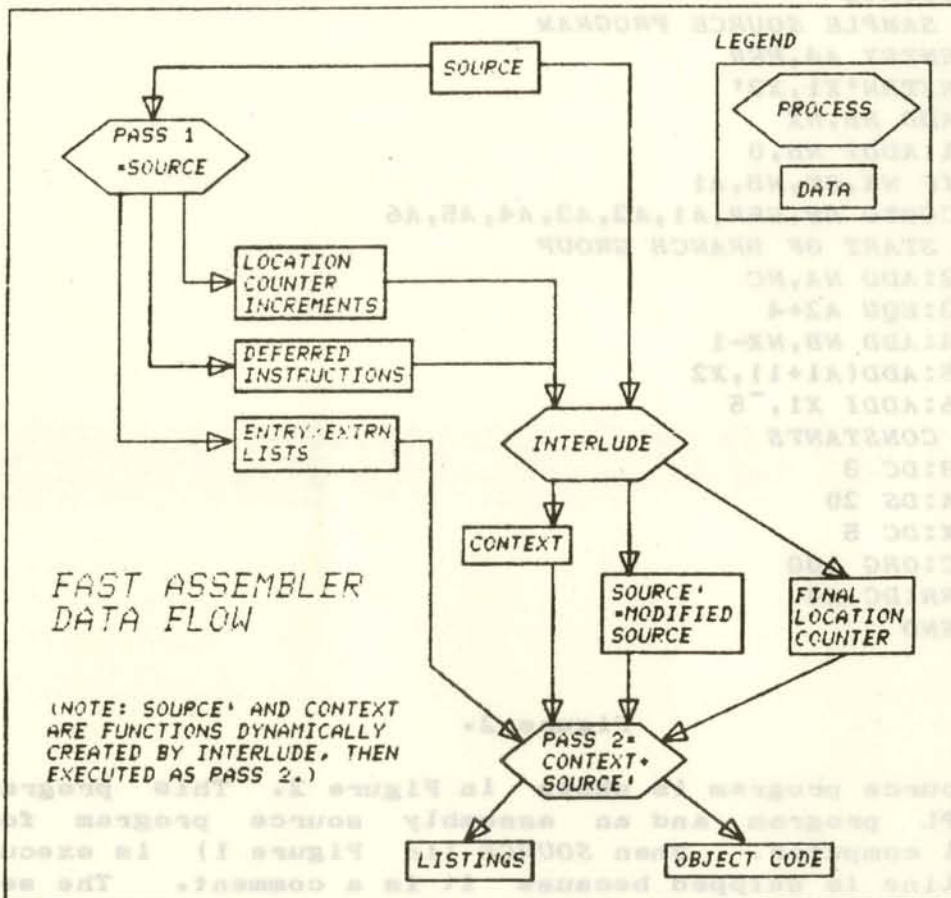
Figure 3.

In the pass 2 context all APL labels are redefined to have the assembly-related values determined by pass 1. To do this, the interlude process creates a "context function" and modifies the original source program. Pass 2 consists of invocation of the context function which establishes the new context, executes the deferred instructions, and finally calls the modified source program. In the pass 2 mode all of the functions invoked by *SOURCE* (modified) generate object code and associated listings. Figure 4 shows the call topography of the fast assembler. (In Figure 4 levels of call are shown by indentation. E.g. *ASM* calls *SOURCE*, Δ*INTERLUDE* and Δ*CONTEXT*.)

```
ASM
    | SOURCE = Pass 1
    | Assembler Instructions (Pass 1 mode)
    | Machine Instructions
    |   | Code Generator (Pass 1 mode)
    | ΔINTERLUDE (build ΔCONTEXT and ΔPASSTWO)
    | ΔCONTEXT = Pass 2
    |   | Deferred Instructions (ORG, EQU, DS)
    |   | ΔPASSTWO (SOURCE modified) = Pass 2 text
    |   |   | Assembler Instructions (Pass 2 mode)
    |   |   | Machine Instructions
    |   |   |   | Code Generator (Pass 2 mode)
```

Figure 4.

Because the language functions native to APL need not be explicitly present in the fast assembler, its size is also considerably reduced. A fast assembler will consist of about 160 lines of basic function written in APL, plus two additional APL lines for each instruction in the target machine. For a machine of 80 instructions the assembler will consist of 160+2x80 or 320 lines of APL code. (A traditionally coded assembler [1] required about 500 lines of APL code.) This does not tell the whole story because the 160 machine instruction lines are quite simple and rapidly coded with little probability of error. The base 160 lines need little modification from one assembler to another.

Another advantage of the technique is that the source program can be edited with the standard APL function editor. No separate source program editor need be provided.

DETAILS OF THE METHOD

The APL listings for a sample fast assembler are displayed in the appendix. We will examine below how it works in contrast to a typical two-pass assembler. The sample assembler supports object code relocation and generates code for a hypothetical machine. The target machine has 16-bit words, but is addressable in 8-bit bytes. Its instructions are variable in length and consist of of one or more words. The first word of each instruction holds its op-code.

The sample fast assembler supports the following typical assembler instructions: EQU, ORG, ENTRY, EXTRN, DS (define storage), DC (define constant) and END. It is assumed that the reader is familiar with at least one assembly language, and that the functions of these assembler instructions are known to him. (See [2] for an example of a typical assembler language.) We will now describe the method in detail, using the sample program called SOURCE shown in Figure 2.

The user enters

     ASM 'SOURCE'

to invoke the assembler. The results of the assembly are left in
some APL global variables (described later). This information is
sufficient for some post processor (not described in this report)
to form a relocatable object module of any desired format. (The
function *DUMP* displays object code and relocation information to
demonstrate this.)

Typical Pass 1.

A typical assembler will perform certain initializations (e.g.,
set a location counter to zero) and start the first of two passes
over the source program. A typical pass 1 would perform the
following functions:

1) Tokenize each line, copy the encoded line to an
   external file (for later use by pass 2), and extract
   the label and instruction mnemonic.

2) For those lines containing a label, place the label
   in a symbol table. Except for EQU and ORG
   instructions, place the current location counter
   value in the symbol table entry for the label and
   set the "relocation" bit on. In any case, advance
   the location counter by an amount depending on the
   instruction.

3) For EQU, ORG and DS instructions, the operand field
   must be evaluated. Evaluation must take into
   account the relocation attribute of symbolic values.
   It also requires parsing and evaluation of the
   operand for an infix algebraic expression. EQU
   assigns its operand value (including relocation bit)
   to its label (in the symbol table). ORG and DS
   increment the location counter by the amount
   computed from their operands. ORG assigns the new
   value to its label if any is present.

4) ENTRY looks up or enters each label from its operand
   into the symbol table. The "entry" bit for each of
   these labels is set on.

5) EXTRN enters each of the labels in its operand into
   the symbol table and sets the "external" bit on.

After pass 1, all source program lines are on an external
(inter-pass) file in an encoded form. All of the labels in the
symbol table have been assigned a value and had their relocation,
entry and external attribute bits set. Before we go on to
describe pass 2, let's see how the fast assembler handles the
first pass.

Fast Assembler Pass 1.

The fast assembler, *ASM*, first establishes a special environment (consisting of constants and empty lists) and then executes *SOURCE*. (Recall that each line of *SOURCE* is an APL function with a name that is an assembler mnemonic.) During the first pass the following actions are carried out.

   1) Machine functions — such as ADD (using the function △GENWDS) insert into the vector △LCX a count of the number of addressable units of storage they use. (△LCX has one element for each line in *SOURCE*.)

   2) Assembly functions *EQU*, *ORG* and *DS* record their line numbers on a list, thereby deferring their executions until the end of the first pass. These are among the few instructions that cause manipulations of the source program as text.

   3) Comments are ignored.

   4) *ENTRY* records its arguments on a list.

   5) *EXTRN* converts its operands into APL variables and assigns them external symbol values. (Note that the *EXTRN* operands must be quoted so as to avoid evaluation by the APL interpreter — line 3 in *SOURCE*. This is the most noticeable intrusion of APL syntax into the syntax of the assembly language. More will be said about syntax in the section on drawbacks.) *EXTRN* is the other instruction that causes manipulation of the source program as text.

Fast Assembler Interlude.

At the end of pass 1 *ASM* is not in the same state as the typical assembler. The location counter increments are held in a vector △LCX. *EQU*, *ORG* and *DS* have been deferred because the values of labels during pass 1 are those of APL line numbers, not location counter values. The deferred instructions are the only ones which must have their operands evaluated before pass 2 starts. When that evaluation takes place, the labels must have the proper values. To this end, a function called △INTERLUDE is invoked at the end of pass 1. The purpose of the interlude function is to cause the APL label variables (A1, A2, NX, ERR etc.) of the source program to take on their assembly values. Once this is done, the deferred functions (*EQU*, *ORG* and *DS*) can be executed and final location counter assignments can be made.

△INTERLUDE forms an APL function called △CONTEXT shown in Figure 5. In this function all labels from *SOURCE* are made into local APL variables. Each is assigned a value determined by its line number and the value in the corresponding position of a variable

named ΔLC. $(\Delta LC\leftarrow+\backslash\Delta K65,\Delta LCX)$ Recalling that ΔLCX contains only location counter increments, the reader will realize that ΔLC contains the location counter setting without taking into account the effect of ORG and DS functions. Note that the value for each label is augmented by the contents of $\Delta\Delta$. $\Delta\Delta$ is an adjustment (initially $\Delta K65$) due to location counter manipulation by the ORG and DS functions. $\Delta K65=2*16$ and is a relocation bit appended to all location counter values. (More will be said about relocation bit strategy later.) Note that the deferred functions are interleaved with the assignments of the labels. They are all in the same order as they appeared in $SOURCE$.

```
          ∇  ΔCONTEXT;ΔΔ;ΔPASSTWO;A1;A2;A3;A4;A5;A6;NB;NA;NX;NC;ERR
    [1]      ΔΔ←ΔK65+0×ΔLV←¯1+ρ□LC
    [2]      A1    ←ΔΔ+  6
    [3]      A2    ←ΔΔ+ 40
    [4]      A3← 10ΔEQU A2+4
    [5]      A4    ←ΔΔ+ 46
    [6]      A5    ←ΔΔ+ 52
    [7]      A6    ←ΔΔ+ 58
    [8]      NB    ←ΔΔ+ 64
    [9]      NA← 16ΔDS 20
    [10]     NX    ←ΔΔ+ 66
    [11]     18 ΔORG 100
    [12]     NC    ←ΔΔ+ 68
    [13]     ERR   ←ΔΔ+ 68
    [14]     ΔMEM←([ .5×ΔK65|[/ΔLC+ΔLCX,0)ρ0
    [15]     ΔLC[ΔEQL[;0]]←ΔEQL[;1]
    [16]     ⍙□FX ΔF
          ∇
```

Figure 5.

By the time $\Delta CONTEXT$ reaches line 16 (see Figure 5) all labels are defined and ΔLC has the location counter values for each of the lines of the source program. We are then in the same position as the typical assembler was at the end of pass 1, and are ready to begin pass 2.

Typical Pass 2.

At the beginning of pass 2 the typical assembler opens an object code output file. It then emits into the buffer of this file entry and external symbol information from the symbol table. Then in pass 2 each line of encoded text is read from the inter-pass file and the following functions are performed.

1) For machine instructions and other bit generators (such as DC), operand fields are evaluated. Evaluation of operands requires parsing of infix algebraic expressions. The results of evaluation are packed according to the format requirements of each instruction. The packed data and its location counter value are emitted to the output buffer. The

location counter is advanced as it was in pass 1.

2) When listing is required, the generated data,
location counter value and source line image is
formatted and placed into an output listing file.

At the end of pass 2, the symbol table is printed with values and
cross reference information for each label. Error messages, if
any, are printed just before or after the symbol table. Finally,
relocation information from the symbol table is sent to the
output buffer, and assembly is completed.

Fast Assembler Pass 2.

Pass 2 execution is similarly straight-forward in the fast
assembler. ΔINTERLUDE, in addition to preparing ΔCONTEXT, also
prepared SOURCE for pass 2 execution. The preparation consisted
of removing all the labels, and changing the header line to
ΔPASSTWO. Figure 6 shows this new version of SOURCE.

```
      ∇ ΔPASSTWO
 [1]  ⍝ SAMPLE SOURCE PROGRAM
 [2]    ENTRY A4,ERR
 [3]    EXTRN'X1,X2'
 [4]    ADD NB,NX
 [5]    ADDI NB,0
 [6]    IF NX,GT,NB,A1
 [7]    CGOTO NB,ERR,A1,A2,A3,A4,A5,A6
 [8]  ⍝ START OF BRANCH GROUP
 [9]    ADD NA,NC
[10]    EQU A2+4
[11]    ADD NB,NX-1
[12]    ADD(A1+1),X2
[13]    ADDI X1,¯5
[14]  ⍝ CONSTANTS
[15]    DC 3
[16]    DS 20
[17]    DC 5
[18]    ORG 100
[19]    DC A4
[20]    END
      ∇
```

Figure 6.

ΔCONTEXT (on line 16) calls ΔPASSTWO (the text image of which was
left in ΔF by ΔINTERLUDE) and the following actions are carried
out by the assembler functions called from ΔPASSTWO.

1) Machine instructions (through the function ΔGENWDS)
place the proper data and relocation bits into the
vector ΔMEM. If listing is required, the function
ΔPRT is called upon. Machine instructions
(including DC) are the only instructions whose

operands are evaluated during pass 2. When they are evaluated, the values of labels are those established by ΔCONTEXT.

2) Functions EQU, ORG, DS and EXTRN only list (their functions having been completed before pass 2).

3) Comments are not executed. Therefore in order to list them, the print routine looks at the line following each one it prints to see if the successor is a comment. If it is, the successor is printed (and its successor checked). This procedure will guarantee listing of all comments except one appearing on line 1. For this case ASM must perform the check and call ΔPRT if required.

4) ENTRY forms all of its listed items (entry labels) into the matrix ΔENL. The values of the items are taken from ΔLC. Listing is performed as required.

At the end of pass 2 (if a listing is requested) the symbol table is printed. Error messages, if present, are listed and assembly is complete. The equivalent of the object code file is held in the global variables ΔMEM, ΔENL, and ΔEXL.

ERROR CHECKING

Many of the errors in the source program will be detected by APL itself. If there are any syntax errors they will occur in pass 1. Assembly will stop and the user can usually correct them by editing the source program, and then resuming the assembly as he would the execution of any APL program. This should not be confusing because the APL error messages come out in the context of the source program. The code displayed is familiar to the user. This is contrary to the usual case where an APL error message is in the context of the assembler -- a program the user did not write. APL checking also eliminates considerable code that would have to be included in the typical assembler.

Value errors will occur either during pass 1 (when a label is misspelled or missing), or during the interlude (when the operand of a deferred instruction is not defined earlier in the source program). If the error in either of these cases is not in the line at which the assembler stopped, the assembly must be aborted before the correction is made. Otherwise, the line causing the error may be modified and the assembly resumed.

The assembler makes a number of checks itself ΔGENWDS checks data and relocation bits it is passed for compatability. If they don't match an error message is issued, but the assembly continues. EQU, ORG and DS check their operands for proper shape and value and issue any needed error messages. All error messages are set up by a common routine, ΔERR. ΔERR places the message and line number on an error list. If no listing is

requested, the source line image is included on the list. At the
end of assembly, any accumulated error messages are printed
following the symbol table.

Some errors will escape detection. For example, duplicate labels
will not be noticed. Some relocatable expressions (like A+B, A+X
and X+1, where A and B are relocatable labels, and X is an
external label) will be wrong without being noted. These could
be detected at additional cost of assembly speed. There are no
attempts to catch errors introduced through malicious use (such
as real numbers or quoted strings in the operand fields). These
errors will cause the assembler to stop with some APL error
message (probably INDEX or DOMAIN error).

RELOCATION CONVENTIONS

For this particular machine architecture (16-bit words) it is
convenient to include the relocation bits as part of the label
value. These bits are the 17th and 18th bits (counting from the
right) of a binary representation of the label value. Bit 17 is
one if the value is relocatable. Bit 18 is one if the label is
an external label. These values are easily tested for relocation
type determination by the loader. The object code vector, ΔMEM,
readily holds one 16-bit word plus two relocation bits per
element. (On a S/370 implementation of APL up to 56 bits can be
held per element.) The final format of the relocatable object
code is beyond the scope of this report. Such a format depends
heavily upon the relocating loader requirements. However,
sufficient information is produced by the assembler to allow the
construction of any desired format. Inclusion of an object code
formatter would not appreciably increase assembly time.

MACROS

This report describes only a basic assembler that has no macro
capability. Implementation of macros so that macro definitions
could appear as part of the source program would lead to
relatively slow text processing. However, one can, without
significant loss of execution speed, implement what are
classically called "built in" macros. That is, one can implement
APL functions which generate multiple machine instructions per
invokation. Such APL functions can take on all of the properties
generally associated with conditional macros. The only
difference between these macros and definable macros is that they
operate in terms of "inside the assembler" rather as part of the
source language.

DRAWBACKS

The fast assembly technique described above has a number of
drawbacks, none judged to be serious. The source program format
is dictated by APL syntax requirements. Labels must appear
followed by a colon. (Some people will view this as an
advantage.) Comments can appear only on comment lines (a

distinct disadvantage). Operands must be evaluated right to left without operator precedence. This means that all operands but the rightmost must be enclosed in parentheses if they contain an operator. (See line 12 in *SOURCE*.) Program labels cannot be the same as op-codes because all names are in the same APL symbol table. The labels *DUMP* and *ASM* can't be used, though this restriction could be removed. (Note that all internal assembler functions and variables have names beginning with 'Δ'.) Neither more complete error checking, macro processing nor label-use recording can be achieved without considerable loss in assembly speed. Some features such as literals, hexadecimal and EBCDIC data specification are not included in the sample assembler, but could be added with little cost in speed or implementation time.

TIMINGS AND CONCLUSIONS

The sample assembler has been tested and timed to a limited extent on an IBM S/370/145 (under VM/370) and on an IBM 5100. The timing formulas for assemblies with and without listings are shown below. The output from the assembly of our sample program is show in Figure 7 at the end of this report.

```
on S/370/145 (with microcode assist)                    Maximum
     with listing     seconds = .037xLINES + .141    1608 lpm
     without listing  seconds = .029xLINES + .106    2077 lpm

on 5100
     with listing     seconds = 5.71xLINES + 22.5      11 lpm
     without listing  seconds = 2.70xLINES + 16.7      22 lpm
```

The numbers following the formulas (under the heading "Maximum") give the maximum number of lines per minute achievable according to the formulas.

The fast assembler was implemented in two man-days, once the concept was perceived by the author. A similar assembler [1] using "typical" techniques was constructed by the author in about four man-weeks. It is estimated that only one or two man-days would be required to write and check out an assembler for any of a variety of typical machine architectures. This low implementation cost, coupled with the high execution speed brings the cost of the fast APL assembler to the point of viability in the realm of micro-computer economics.

REFERENCES

1) Myers, H. Joseph, and Friedl, Paul J., "A Terminal-Oriented Assembler/Simulator for System/7", IBM Scientific Center Report ZZ20-6412, December 1971. (IBM Internal Use Only.)

2) IBM Coprporation, "IBM System/360 Disk and Tape Operating System Assembler Language", Form C24-3414, 1969.

```
      ASM 'SOURCE '
LOC    OPR   OPND OPND OPND   | SOURCE                              PAGE 1
C--                            1| A SAMPLE SOURCE PROGRAM
0000:                          2|      ENTRY A4,ERR
0000:                          3|      EXTRN'X1,X2'
0000:003B 0040 0056            4|      ADD NB,NX
0006:003D 0040 0000            5|A1    ADDI NB,0
000C:1057 0056 0040 0006       6|      IF NX,GT,NB,A1
0014:0056 0040 0006 0064       7|      CGOTO NB,ERR,A1,A2,A3,A4,A5,A6
     0006 0028 002C 002E
     0034 003A
C--                            8|A START OF BRANCH GROUP
0028:003B 0042 0064            9|A2    ADD NA,NC
002C:                         10|A3    EQU A2+4
002E:003B 0040 0055           11|A4    ADD NB,NX-1
0034:003B 0007 0001           12|A5    ADD(A1+1),X2
003A:003D 0000 FFFB           13|A6    ADDI X1,⁻5
C--                           14|A CONSTANTS
0040:0003                     15|NB    DC 3
0042:                         16|NA    DS 20
0056:0005                     17|NX    DC 5
0064:                         18|NC    ORG 100
0064:002E                     19|ERR   DC A4
0066:                         20|      END
```

```
SYMBOL TABLE
  A1        5    6=R 0006
  A2        9   40=R 0028
  A3       10   44=R 002C
  A4       11   46=R 002E
  A5       12   52=R 0034
  A6       13   58=R 003A
  ERR      19  100=R 0064
  NB       15   64=R 0040
  NA       16   66=R 0042
  NX       17   86=R 0056
  NC       18  100=R 0064
```

```
ENTRIES
  A4   =   1    46
  ERR  =   1   100
```

```
EXTERNAL SYMBOLS
  X1
  X2
```

Figure 7.

⍝ *APPENDIX: A SAMPLE FAST ASSEMBLER*

```
      ∇ ASM ∆N;∆LSTSW;∆LCX;∆PASS2;∆H4;∆F;∆CONTEXT;∆K65;⎕IO;
        ∆PGN;∆PGH;∆LCT;∆LV;∆EQL;∆ERL;∆SY;∆MT;∆LBL;∆HEX;∆LC
[1]     ∆MT←∆ENL←∆EQL←∆ERL←∆SY←ρ∆PGN←∆LCT←∆PASS2←⎕IO←0
[2]     →⍳2>ρ∆LCX←(1↑ρ∆F←⎕CR ∆N)↑0
[3]     ∆LSTSW←' '∈∆N
[4]     ∆LV←¯1+ρ⎕LC
[5]     ∆LBL←⍳6
[6]     ∆K65←⌊2*16
[7]     ∆H4←4ρ16
[8]     ∆EXL←0 0ρ∆HEX←'0123456789ABCDEF'
[9]     ⍒∆N
[10]    ∆PASS2←1
[11]    ∆PGH←⎕AV[5ρ169],'LOC    OPR   OPND OPND OPND           | ',∆F[0
        ;],' PAGE '
[12]    ⍒⎕FX ∆INTERLUDE
[13]    ∆PRSYM
      ∇

      ∇ Z←∆INTERLUDE;I;J;K;L;M;N
[1]   ⍝ EXTRACT LINE LABELS (FOR SYMBOL TABLE)
[2]     ∆SY←∆F[;∆LBL]
[3]     I←M/⍳ρM←∆SYv.=':'
[4]     ∆SY←∆SY[I;]
[5]     ∆SY←(N←ρ∆SY)ρ(J←v\∆SY=':')⊖∆SY,[¯0.5]' '
[6]   ⍝ EXTRACT EQU/DS LINES (FOR CONTEXT FUNCTION)
[7]     K←∆F[L←(0<∆EQL)/∆EQL;]
[8]   ⍝ CREATE PASS 2 FUNCTION (PASS 1 LESS LABELS)
[9]     ∆F[0;]←(1↓ρ∆F)↑'∆PASSTWO'
[10]    ∆F[I;∆LBL]←Nρ(J←1,0 ¯1↓~J)⊖∆F[I;∆LBL],[¯0.5]' '
[11]    ∆F[∆LBL←I;]←(N←+/J)⌽∆F[I;]
[12]  ⍝ CONVERT EQU/DS TO ∆EQU/∆DS (FOR CONTEXT FUNCTION)
[13]    ∆LC←+\∆K65,∆LCX
[14]    I←(J≠∆SY),'←','∆','∆','+',0 0⍚∆LC[(J←~∆LBL∈∆EQL)/∆LBL]
        ∘.-,∆K65
[15]    K←(-N)⌽'←',(3 0⍚L∘.+,0),'∆',0 1↓(N←¯1+(~J)/N)⌽K
[16]  ⍝ EXTRACT ORG LINES (FOR CONTEXT FUNCTION)
[17]    N←∆F[L←|(∆EQL<0)/∆EQL;]
[18]    N←(⍚L∘.+,0),'∆',N
[19]  ⍝ COMBINE SEGMENTS INTO CONTEXT FUNCTION
[20]    L←21[ρZ←(Z≠' ')/Z←'∆CONTEXT;∆∆;∆PASSTWO',,';',∆SY
[21]    Z←(L↑Z),[¯0.5]L↑'∆∆←∆K65+0×∆LV←¯1+ρ⎕LC'
[22]    I←K ∆VCAT N ∆VCAT I
[23]    ∆EQL←0 2ρK←(N/∆EQL),(~N←∆EQL>0)/∆EQL
[24]    Z←Z ∆VCAT I[⍋(|K),(~∆LBL∈K)/∆LBL;]
[25]    Z←Z ∆VCAT ∆INT1
[26]  ⍝ PREPARE SYMBOL TABLE
[27]    ∆SY←M⌿∆SY
[28]    ∆SY[I/⍳ρI←∆F[;0]='⍝';]←⎕AV[255]
[29]    →⍳'⍝'≠∆F[1;0]
[30]    'C--'∆PRT 1
      ∇
```

```
     ∇ N ∆GENWDS A;I;J;L;M;T
[1]      →∆PASS2/A1
[2]      ∆LCX[ 1↑∆LV↑□LC ]←2×ρN
[3]      →0
[4]   A1:L←⌊0.5×∆K65|T←∆LC[ J←''ρ∆LV↑□LC ]
[5]      →( 0=I←ρN )/A4
[6]      →( N∧.=∆K65≤A-I↑A )/A3
[7]      J ∆ERR'RELOCATION ERROR'
[8]   A3:∆MEM[ L+ιI ]←A
[9]   A4:→ι~∆LSTSW
[10]     I←,(⍉∆HEX[ ∆H4↑T,A ]),' '
[11]     I[4]←':'
[12]     I ∆PRT J
     ∇


     A ASSEMBLER INSTRUCTIONS

     ∇ EQU L;I
[1]      →∆PASS2/A3×∆LSTSW
[2]      I←''ρ1↑∆LV↑□LC
[3]      →( 1=L←ρ,L )/A1
[4]      I ∆ERR(⍕L),' OPERANDS'
[5]   A1:→( ':'∊∆F[ I;∆LBL ] )/A2
[6]      I ∆ERR'LABEL MISSING'
[7]      →0
[8]   A2:∆EQL←∆EQL,I
[9]      →0
[10] A3:∆MT ∆GENWDS ∆MT
     ∇

     ∇ Z←N ∆EQU L
[1]      →( ( 2×∆K65 )>L )/A1
[2]      N ∆ERR'RELOCATION ERROR'
[3]      L←0
[4]   A1:∆EQL←∆EQL,[ 0 ]N,Z←1↑L
     ∇

     ∇ ORG L
[1]      →∆PASS2/A1×∆LSTSW
[2]      ∆EQL←∆EQL,-''ρ∆LV↑□LC
[3]      →0
[4]   A1:∆MT ∆GENWDS ∆MT
     ∇

     ∇ N ∆ORG L
[1]      →( ~2|L )/A1
[2]      N ∆ERR'ODD ORIGIN'
[3]      L←L+1
[4]   A1:∆∆←∆∆+∆LCX[ N ]←( L-∆K65|L )-∆K65|∆LC[ N ]
[5]      ∆EQL←∆EQL,[ 0 ]N,L+∆K65
[6]      ∆LC←+\∆K65,∆LCX
     ∇
```

```
      ∇ DS L
[1]     EQU L
      ∇


      ∇ Z←N ∆DS L
[1]     →(0≤L←1↑L)/A1
[2]     N ∆ERR'ILLEGAL NEGATIVE'
[3]     L←0
[4]   A1:→(∆K65>L)/A2
[5]     N ∆ERR'RELOCATION ERROR'
[6]     L←0
[7]   A2:Z←∆LC[N]
[8]     ∆∆←∆∆+∆LCX[N]←L
[9]     ∆LC←+\∆K65,∆LCX
      ∇


      ∇ ENTRY L
[1]     →∆PASS2/A1←~∆LSTSW
[2]     ∆ENL←∆ENL,L
[3]     →0
[4]   A1:∆MT ∆GENWDS ∆MT
[5]     →ι2=ρρ∆ENL
[6]     ∆ENL←∆SY[∆ENL;],'=',⍕⍉(3,∆K65)⊤∆LC[∆ENL]
      ∇


      ∇ EXTRN ∆L;∆I;∆J;∆
[1]     →∆PASS2/∆1×∆LSTSW
[2]     →ι(0=1↑0ρ∆L)∨0=ρ,∆L
[3]     ∆J←(∆J,ρ∆L)←0,1+∆J←∆I/ιρ∆I←∆L=','
[4]     →ι0ερ∆J←(0=⎕NC ∆J)≠∆J←(ρ∆J)ρ(,∆J←∆Jο.>ι⌈/∆J)\(~∆I)/∆L
[5]     ∆EXL←∆EXL ∆VCAT ∆J
[6]     ∆J←((1↓ρ∆J)↑'∆'),[0]∆J←∆J,'←',⍕((-1↑ρ∆J)ιι1↑ρ∆EXL)ο.+,
          2×∆K65
[7]     ⍝⎕FX ∆J
[8]     →0
[9]   ∆1:∆MT ∆GENWDS ∆MT
      ∇


      ∇ END
[1]     →A1×∆PASS2∧∆LSTSW
[2]   A1:∆MT ∆GENWDS ∆MT
      ∇


      ∇ Z←I ∆VCAT J
[1]     Z←0,⁻1↑(ρI)⌈ρJ
[2]     Z←((Z[ρI]↑I),[0](Z[ρJ]↑J)↑J
      ∇


      ∇ J ∆ERR M
[1]     M←⎕AV[73],(4 0⍕J),': ',M
[2]     →∆LSTSW/A1
[3]     M←(29↑M),'|',∆F[J;]
[4]   A1:∆ERL←∆ERL,M
      ∇
```

```
      ∇ I ∆PRT J;L
[1]     →ι~∆LSTSW
[2]    A1:→(0<∆LCT←∆LCT-1)/A2
[3]     ⎕←(5×1=∆PGN)↓∆PGH,▼∆PGN←∆PGN+1
[4]     ∆LCT←60
[5]    A2:→(' '=1↑I)/A3
[6]     L←((L≠⎕AV[255])/L←∆SY[J;]),∆F[J;]
[7]     ⎕←(25↑I),(4 0▼J),'|',L
[8]    A3:→(25≥ρI)/A4
[9]     ⎕←25↑I←'          ',25↓I
[10]    →A1
[11]   A4:→ι(1↑ρ∆F)≤J←J+1
[12]    I←'C--'
[13]    →A1×'ʀ'=∆F[J;0]
      ∇

      ∇ ∆PRSYM;I;J;CR
[1]     CR←⎕AV[73]
[2]     →(~∆LSTSW)/A3
[3]     →(0∊ρ∆SY←(~∆SY[;0]∊' ',⎕AV[255])≠∆SY)/A2
[4]     →((∆LCT-5)>2+1↑ρ∆SY)/A1
[5]     ⎕←(∆LCT+6)ρ⎕AV[169]
[6]    A1:⎕←CR,'SYMBOL TABLE'
[7]     ∆SY←∆SY,0 0▼∆LBL,[0.5](×J)×∆K65|I←|J←∆LC[∆LBL]
[8]     J←∆SY,'=',' R'[2|⌊I≠∆K65],' ',⍒∆HEX[∆H4⊤,J]
[9]     ⎕←' ',J[⍋'ABCDEFGHIJKLMNOPQRSTUVWXYZ'ιJ[;0];]
[10]    →(0∊ρ∆ENL)/A2
[11]    ⎕←CR,'ENTRIES'
[12]    ⎕←' ',∆ENL
[13]   A2:→(0∊ρ∆EXL)/A3
[14]    ⎕←CR,'EXTERNAL SYMBOLS'
[15]    ⎕←' ',∆EXL
[16]   A3:I←⎕EX ∆EXL
[17]    →ι0=ρ∆ERL
[18]    ⎕←CR,'ERRORS:',∆ERL
      ∇

      ꞧ MACHINE INSTRUCTIONS

      ∇ ADD L
[1]    ꞧADD T,F
[2]    0 1 1 ∆GENWDS 59,L
      ∇

      ∇ ADDI L
[1]    ꞧADDI T,FI
[2]    0 1 0 ∆GENWDS 61,L
      ∇

      ∇ GOTO L
[1]    0 1 ∆GENWDS 85,L
      ∇
```

```
      ∇ CGOTO L
[1]     ⍝ CGOTO IX,ERR,L1,L2,...,LN
[2]     (0 1 0,1↓L=L)∆GENWDS 86,L[0],(¯2+⍴L),1↓L
      ∇

      ∇ IF L
[1]     ⍝IF A,CP,B,LOC (WHERE CP= GT, EQ, GE, LT, NE OR LE)
[2]     0 1 1 1 ∆GENWDS(87+L[1]),1 0 1 1/L
      ∇

      ∇ IFI L
[1]     ⍝IFI A,CP,BI,LOC (WHERE CP= GT, EQ, GE, LT, NE OR LE)
[2]     0 1 0 1 ∆GENWDS(32855+L[1]),1 0 1 1/L
      ∇


      ∇ DC L
[1]     ⍝DC V1,V2,...,VN
[2]     (L≥∆K65)∆GENWDS L←,L
      ∇



      ⍝ AUXILIARY FUNCTIONS

      ∇ Z←∆
[1]     →ι~Z←∆PASS2
[2]     Z←∆LC['' ⍴∆LV↑⎕LC]
      ∇


      ∇ Z←DUMP N;I;J;K;L;CR;⎕IO
[1]     Z←0⍴CR←⎕AV[73+⎕IO←0]
[2]     N←⌊|2↑N≠2
[3]     I←N[0]
[4]     N←(¯1+⍴∆MEM)⌊⌈/2↑N
[5]     A1:→ιN<I
[6]     →((K≠1)∧J[0]∧.=J←∆MEM[I+ιK←8⌊1+N-I])/A2
[7]     Z←Z,CR,(¯1↓∆CVH I×2),': ',∆CVB J
[8]     →A1,I←I+K
[9]     A2:L←I
[10]    A3:→(N<I←I+K)/A4
[11]    →(J[0]∧.=∆MEM[I+ιK←8⌊1+N-1])/A3
[12]    A4:Z←Z,CR,'    ',(∆CVH 2×L),'THRU ',(∆CVH 2×I-1),'
        CONTAIN ',∆CVB J[0]
[13]    →A1
      ∇


      ∇ Z←∆CVH N
[1]     Z←,(⍉'0123456789ABCDEF'[16 16 16 16⊤,N]),' '
      ∇


      ∇ Z←∆CVB N
[1]     Z←3 16 16 16 16⊤,N
[2]     Z←' RX'[Z[0;]×N≥0],[0]'0123456789ABCDEF'[1 0↓Z]
[3]     Z←,⍉1 0 1 1 1 1 0↘Z
      ∇
```

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHOR(S) : H.J. Myers	9. SUBJECT INDEX TERMS APL Assemblers Performance
2. TITLE : A Fast Assembly Technique Using APL	
3. ORIGINATING DEPARTMENT Palo Alto Scientific Center	
4. REPORT NUMBER ZZ20-6431	

5a. NUMBER OF PAGES 19	5b. NUMBER OF REFERENCES 0	

6a. DATE COMPLETED May 5, 1976	6b. DATE OF INITIAL PRINTING June 1976	6c. DATE OF LAST PRINTING

7. ABSTRACT :

A technique is described which reduces the cost of producing assemblers for a wide variety of machine architectures. Assembly is accomplished by executing each instruction of the source program as an APL function. An assembler has been generated capable of speeds of about 2000 lines per minute in an APL environment on an IBM System 370/145.

8. REMARKS :

PADC - 18 - IBM

IBM Palo Alto Scientific Center, P. O. Box 10500, Palo Alto, California 94304