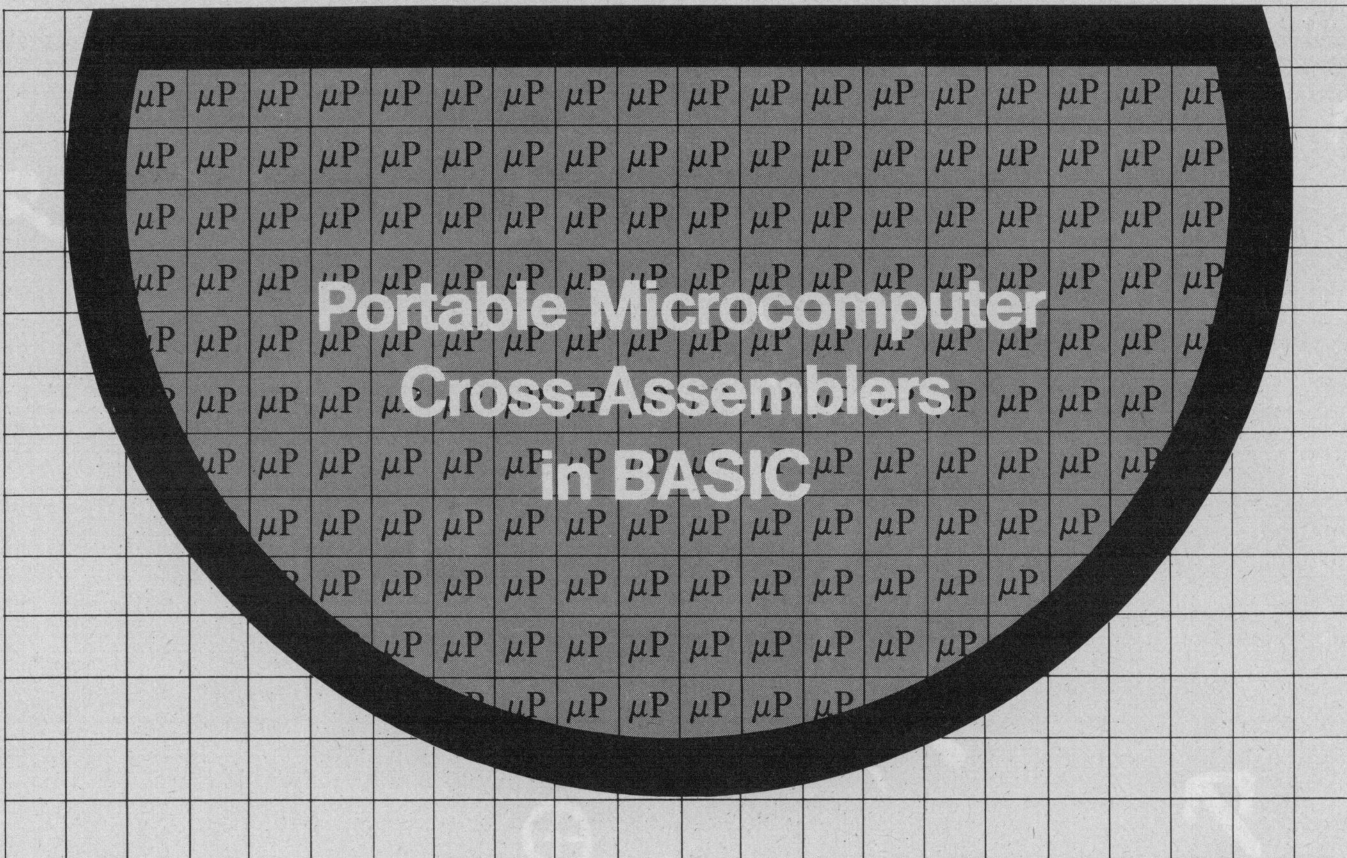


Portable Microcomputer Cross-Assemblers in BASIC

Steven W. Conley
University of Arizona

Introduction

With the tremendous number of microprocessors on the market today it is becoming increasingly hard to find cross assemblers that run on the particular in-house computer system or timesharing service available. Since most minicomputer systems and almost all timesharing systems have BASIC interpreters or compilers available, this seems to be the most reasonable language choice for writing cross assemblers. It also has a very good trace and debug facility, which is especially useful if you are writing medium-long programs, such as cross-assemblers.

Another advantage of a cross-assembler written in BASIC is the ease with which options can be added. For example, one of the new microprocessors by MOS Technology, Inc. is "pin-compatible" with the M6800 by Motorola, but it is not "bit-compatible." In other words the mnemonics like TST must be assembled into two different bit patterns for the machines, but they both execute similarly. Thus, if you were using the MOS Technology device as a second source for Motorola (or vice versa) you could not use the same ROMs. With a cross-assembler such as described here, a switch could be included which would allow the same assembler to generate code for both machines. This article presents a technique for writing cross-assemblers which is both modular (as far as possible in BASIC) and can be used for many different microprocessors with little modification.

The Assembly Language

To begin with we define an assembly language. This does not have to correspond exactly with the manufacturer's suggested assembly language—and in fact may be better in two ways: human readability and machine readability. For example, many assembly languages define a statement label as beginning in column 1 and terminating with spaces:

```
LABEL    AND    R1,ALPHA
```

This works fine until the programmer forgets the spaces in front of an instruction:

```
AND    R1,ALPHA
```

All the assembler can do in this case is to assign AND as the label for that location, and upon seeing an "instruction" R1, rejecting the statement. Alternatively we can define a label as terminating with a colon:

```
LABEL:    AND    R1,ALPHA
```

Now there is no ambiguity and we can correctly "parse" the statement. There are many other examples. The important thing to realize is that the manufacturer's assembly language is not sacred, and many times can be improved upon or at least changed to match the assembly languages you may be using on other machines.

Table 1. Assembly language for IMP16L

1. Since the IMP16L is a 16-bit machine, we will use a base 16 radix (hexadecimal. This radix when used as an expression will be preceded by a "#." Decimal radix numbers will have no # character.
2. Symbols will be six alphanumeric (A-Z, 0-9) characters maximum and will be followed by a colon, and will be used as labels. They can be assigned values by an "=" followed by an expression.
3. Comments will be any graphic (printable) characters and space preceded by a semicolon.
4. All operator mnemonics will be three or four characters long, terminated by a space.
5. All operands will be separated with commas.
6. The current location counter will be indicated by "."
7. Macros or assembler directives will be preceded by "."

For example, here is a section of assembler code:

```
.       =       100     ;start code at location 100
;                       this is a comment only line
AC0     =       0       ;define the four registers
AC1     =       1       ;we could call them R1 thru R4 instead
AC2     =       2       ;or even AC,SCRATC,IND1,IND2
AC3     =       3
BLANK   =       #20     ;ascii blank is hex 20
IEN     =       9       ;interrupt enable bit in processor status word
;
START:  LD      AC2,XLOC        ;load AC2 from memory location XLOC
        LI      AC0,BLANK       ;load AC0 immediate with a blank
        ST      AC0,-1(AC2)     ;store AC0 into location pntd to by AC2
                                ;minus 1 (this is indexed addressing)
        AISZ    AC2,-1          ;add -1 to AC2 and skip if zero
        BOC     IEN,T2          ;branch if interrupt enable on to T2
        PUSH    AC2             ;push AC2 onto the stack
        RTS     3               ;return (from subroutine) to the address
                                ;on the stack +3
TEMP:   .WORD   0               ;reserve a location and store a zero in it
XLOC:   .WORD   BLANK           ;reserve another with the value of BLANK in it.
        .WORD   START           ;reserve a word with the address START in it.
        .END
```

Using the National Semiconductor IMP16L as an example microprocessor (because of its extensive and varied instruction set), this discussion will point out a few problems with writing assemblers which are not encountered in simpler machines such as the Intel 8080, or the Motorola 6800. An assembly language for the IMP16L is defined in Table 1.

The Host Machine

For the purposes of this paper the minicomputer for which the cross assembler is written is a PDP11/40 running the RT11 operating system. RT11 BASIC is very similar to DEC10, HP, and PDP11 RTSTS BASIC, to name a few. The only real requirement of the BASIC is that it must have file manipulation and string manipulation. The reason for the latter is obvious, but the file manipulation may not be so obvious. It is required for two reasons: the cross-assembler family described here is multi-pass; and on small main memory machines the cross-assembler may have to be "chained" in order to have enough room for large symbol tables. Most BASIC dialects with file manipulation also have the CHAIN feature which is known to FORTRAN users as overlays.

In the following tabulation several features of the RT11/BASIC are explained for those not familiar with it:

\	used to separate statements on a single line
$	last character of a "string" variable name
&	concatenates string variables
#	used to indicate file number
POS(string,char,n)	returns position in string of 1st occurrence of char, starting with nth character in string.
SEG$(string,m,n)	returns the segment of string from the mth character to the nth character.
TRM$(string)	removes trailing blanks from the string.
VAL(string)	considers the string to be a numeral string and returns the value (can include "–" and ".")

An interesting feature of this and other BASICs is the ability to form the IF-THEN-ELSE construct, even though ELSE is not a feature of the language:

```
10  IF X >= 30 THEN 20 \ X=30 \ GO TO 25
20  X=X+10
25  continue
```

This is equivalent to: IF X>=30 then X=X+10 ELSE X=30.

One limitation of RT11 BASIC is its lack of integer variables. This would serve to reduce the core requirements considerably in a string-processing type program.

```
100     PRINT "IMP16 CROSS ASSEMBLER - STEVEN CONLEY"
110     PRINT "SOURCE FILE:";        \ INPUT F1$
130     PRINT "OPTIONS:";            \ INPUT A$
140     F2$=F1$&".IMP"
150     F3$=F1$&".TMP"
160     F4$=F1$&".LST"
170     OPEN F2$ AS FILE #2
180     OPEN F3$ FOR OUTPUT AS FILE #3
190     OPEN F4$ FOR OUTPUT AS FILE #4
```

Figure 1.
Disk file initialization section of cross-assemblers

The Initialization Section

The first section of the program asks for the source program name, opens the file as "name.IMP," and opens files 2 and 3 as "name.TMP" and "name.LST." Any options are also requested at this time. If CHAINing is used, this might be the extent of the first overlay (see Figure 1).

Next we initialize the hexadecimal conversion table, the permanent symbol table, and a few variables. This might be the extent of the second overlay (see Figure 2).

Lines 20000-20010 set up a hex-ascii and ascii-hex table. Lines 20030-20050 set up and initialize the opcode permanent symbol table. Notice that the opcodes are grouped according to type—e.g., one-word versus two-word instructions. Lines 20300-20320 initialize the user symbol table, the first symbol being the "." location counter. T1$ is the string array where the symbol names will go and T1 is where the symbol values will go. M1 is the maximum number of symbols, and P3 is the next available slot pointer.

Useful Subroutines

In Figure 3 we have a listing of the general subroutines for Pass 1 of the assembler. These were written with speed of implementation and ease of debugging rather than speed of execution in mind. For example, the symbol table search is linear rather than the normal "hash" table search. Because of the modularity of the subroutines, faster routines can be substituted for the major bottleneck parts of the code after the assembler is running.

Subroutine 10000-10090 is an internal-binary to hexadecimal-string conversion routine. A check is made for 16-bit overflow (the IMP-16 is a 16-bit machine). Step 10035 is a "trick" to form the 16-bit two's complement of a number represented internally as a negative floating point number (standard BASIC representation). The hexadecimal string is formed by concatenation on the right of an entry from the hexadecimal table indexed by the integer part of the result of division by a power of 16. This shows how a routine can be written independent of the host-computer word length, as long as the mantissa of the floating point word exceeds 16 bits.

Care should be taken in the documentation of BASIC programs because all variables are global. This may not appear to be a serious problem at first, but when the program length exceeds approximately two pages, the chance of choosing the same variable again is very high, especially when limited to a single letter followed by a one digit number. It becomes necessary to document all internal "scratch" variables:

Subroutine 10000-10090

FUNCTION:	internal binary to hexadecimal string conversion.
INPUTS:	B1 is input binary string.
OUTPUTS:	H1$ is output hex string.
SCRATCH VARIABLE:	I,X1,X2,X3
GLOBAL EFFECTS:	if input > 65535, L4$ = "C", and H1$ = 5 blanks.
SUBROUTINES CALLED:	none
GLOBALS USED:	previously initialized BIN-HEX table: H$

Documentation of this sort will be very valuable to both your successors and yourself after 3 months or so.

Subroutine 10100-10190 is a hexadecimal-string to internal-binary conversion routine. It has error checks for illegal characters and more than 4 hex digits. Subroutine 10200-10260 returns the value of an operand—either symbol, decimal, or hexadecimal. If the operand does not start with #, –, 0-9, then a linear search of the symbol table T1$ is made. If the entry is not found, the index T3 will be equal to M1. Subroutines 10300, 10400, 10500 and 10600 are self explanatory once you see that C2 is the current character pointer in the line. Subroutine 10700-10850 sets the next operand, expecting it to be defined. It also checks for a comma after the operand, and sets a flag E=1 if it is not there.

Pass 1 The next section (Figure 4) is the "first pass" of the assembler. The flow chart in Figure 5 helps to explain Pass 1 and its relation to the other passes over the source code. The function of the first pass is to produce a "symbol" table and an intermediate source file. The symbol table contains all user defined symbols and statement labels. The intermediate file has the following structure:

LOC   CODE   FLAG   ORIGINAL SOURCE

The LOC is the memory address (sometimes called location counter; hence LOC), CODE is the index into the opcode table, FLAG is a special statement indicator, and ORIGINAL SOURCE is the untouched source line. To generate the LOC, it is necessary to know the word length of each instruction. This is one of the reasons for the opcode table arrangement. The CODE makes Pass 2 faster by allowing a branch on that value to an instruction group specific

```
20000 DIM H$(15) \ FOR I=0 TO 15 \ READ H$(I) \ NEXT I
20010 DATA "0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"

20030 M2=70
20040 DIM O1$(71),O2(71)
20050 FOR I=0 TO 70 \ READ O1$(I),O2(I) \ NEXT I

20060 DATA "RADD",12288,"RXCH",12416,"RCPY",12417,"RXOR",12418
20065 DATA "RAND",12419
20067 REM 0-4 *****

20070 DATA "ROL",22528,"ROR",22528,"SHL",23552,"SHR",23552
20075 REM 5-8 *****

20080 DATA "AISZ",18432,"LI",19456,"CAI",20480
20085 REM 9-11 ****

20090 DATA "PUSH",16384,"PULL",17408,"XCHRS",21504
20095 REM 12-14 ***

20100 DATA "LD",32768,"LDI",36864,"ST",40960,"STI",45056,"ADD",49152
20105 DATA "SUB",53248,"SKG",57344,"SKNE",61440
20107 REM 15-22 ***

20110 DATA "AND",24576,"OR",26624,"SKAZ",28672
20115 REM 23-25 ***

20120 DATA "JMP",8192,"JMPI",9216,"JSR",10240,"JSRI",11264,"ISZ",30720
20125 DATA "DSZ",31744
20127 REM 26-31 ***

20130 DATA "BOC",4096
20135 REM 32 ******

20140 DATA "SFLG",2048,"PFLG",2176
20145 REM 33-34 ***

20150 DATA "RIN",1024,"ROUT",1536,"RTS",512,"RTI",256
20155 REM 35-38 ***

20160 DATA "PUSHF",128,"PULLF",640,"HALT",0,"NOP",12417,"ISCAN",1296
20165 REM 39-43****

20170 DATA "SETST",1792,"CLRST",1808,"SKSTF",1856
20175 DATA "SETBIT",1824,"CLRBIT",1840,"CMPBIT",1888,"SKBIT",1872
20177 REM 44-50 ***

20180 DATA "JINT",1312,"JMPP",1280,"JSRP",768,"JSRT",896
20185 REM 51-54 ***

20197 REM TWO WORD FROM HERE ON
20200 DATA "MPY",1152,"DIV",1168,"DADD",1184,"DSUB",1200
20210 DATA "LDB",1216,"LLB",1216,"LRB",1216,"STB",1232,"SLB",1232
20215 DATA "SRB",1232
20217 REM 55-64 ***

20220 DATA ".WORD",65,".BYTE",66,".ASCII",67
20225 DATA ".LOCAL",68,".NAME",69,".END",70
20227 REM *********

20300 M1=300 \ DIM T1$(301),T1(301)
20320 T1$(0)="." \ T1(0)=0 \ P3=1 \ T1(M1)=0
```

Figure 2.
Initialization section of IMP-16 cross-assembler

subroutine. The FLAG is also an addition to speed up Pass 2. It indicates comment only lines by having the flag equal to "C." Symbol assignment statements are also converted to comments after they have served their purpose..

In Figure 4, statements 210 and 220 are the source input statements. When the end of file on the source is reached, a branch to Pass 2 is made. Statement 225 is simply a pacifier for impatient programmers on slow timesharing systems. It prints a period on his terminal each time a line is processed. Figure 6 is a flowchart of Pass 1.

The important variables in Pass 1 are:

L2$	LOC field in the output temp file
L3$	CODE field in the output temp file
L4$	FLAG field in the output temp file
T1(0)	memory address (location) counter
C1	character pointer

The errors checked for are: multiply defined symbol, no colon after label, symbol table overflow, undefined opcode, and various character errors detected by the subroutines.

```
10000 REM BINARY TO HEX CONVERSION , B1 IS BINARY, H1$ IS HEX
10010 IF B1>65535 THEN 10080
10020 H1$=" " \ X3=B1
10025 IF X3>=0 THEN 10030 \ X3=65535+X3+1
10030 FOR I=3 TO 0 STEP -1
10040 X1=16^I \ X2=INT(X3/X1) \ X3=X3-X2*X1
10050 H1$=H1$&H$(X2)
10060 NEXT I
10070 RETURN
10080 PRINT "OVF IN BINHEX" \ L4$="C" \ H1$="     "
10090 RETURN

10100 REM HEXADECIMAL TO BINARY CONVERSION
10110 X1=LEN(H1$) \ B1=0
10120 IF X1>4 THEN 10180
10130 FOR I=0 TO (X1-1) \ S1$=SEG$(H1$,X1-I,X1-I)
10140 FOR I1=0 TO 15
10150 IF H$(I1)=S1$ THEN 10170
10160 NEXT I1 \ PRINT "ILLEGAL CHAR IN HEX NUMBER" \ GO TO 10190
10170 B1=B1+I1*(16^I) \ NEXT I \ RETURN
10180 PRINT ">4 HEX DIGITS"
10190 L4$="C" \ B1=0 \ RETURN

10200 REM SEARCH SYMBOL TABLE FOR ELEMENT S2$, RETURN VALUE V1
10201 T3=0 \ S4$=SEG$(S2$,1,1)
10202 IF S4$>"9" THEN 10210 \ IF S4$>="0" THEN 10204
10203 IF S4$<>"-" THEN 10206
10204 V1=VAL(S2$) \ RETURN
10205 RETURN
10206 IF SEG$(S2$,1,1)<>"#" THEN 10210
10207 H1$=SEG$(S2$,2,256) \ GOSUB 10100 \ V1=B1\RETURN
10210 FOR T3=0 TO M1
10220 IF S2$=T1$(T3) THEN 10250
10230 NEXT T3
10250 V1=T1(T3)
10260 RETURN

10300 REM SEARCH OPCODE TABLE FOR OPCODE S2$, RETURN NUMBER I1
10310 FOR I1=0 TO M2
10320 IF S2$=O1$(I1) THEN 10360
10330 NEXT I1
10360 RETURN

10400 REM FIND CHAR BEFORE NEXT BLANK OR TAB
10410 GOSUB 10600 \ C2=C2-1
10460 RETURN

10500 REM LOOK FOR NEXT CHAR EXCEPT TAB OR SPACE
10510 X$=SEG$(L$,C2,C2)
10515 IF C2>80 THEN 10590
10520 IF X$<>" " THEN 10530 \ C2=C2+1 \ GO TO 10510
10530 IF X$<>"   " THEN 10590 \ C2=C2+1 \ GO TO 10510
10590 RETURN

10600 REM LOOK FOR 1ST TAB OR SPACE
10610 X$=SEG$(L$,C2,C2)
10620 IF X$=" " THEN 10690
10630 IF X$="    " THEN 10690 \ C2=C2+1 \ GO TO 10610
10690 RETURN

10700 REM GET NEXT SYMBOL AND VALUE C2 IS POINTER, IF LAST E1=1
10710 E1=0 \ C1=C2
10720 GOSUB 10600 \ C3=C2 \ REM FIND LAST PLACE
10730 C2=C1
10740 IF SEG$(L$,C2,C2)="," THEN 10760 \ IF C2>=C3 THEN 10750
10745 C2=C2+1 \ GO TO 10740
10750 E=1 \ C2=C3
10760 C2=C2+1 \ S2$=SEG$(L$,C1,C2-2) \ GOSUB 10200
10770 IF T3<>M1 THEN 10850
10780 PRINT "UNDEFINED SYMBOL:",S2$,L$
10790 V1=0
10850 RETURN
```

Figure 3. General cross-assembler subroutines

```
210 IF END #2 THEN 30000
220 INPUT #2:L$
225 PRINT ".";
230 C1=0 \ L2=T1(0) \ L3=0 \ L4$=" "
240 L$=TRM$(L$) \ L$=L$&" " \ X$=SEG$(L$,1,1)
250 IF X$=";" THEN 370
260 IF X$<>" " THEN 265 \ C2=1 \ GO TO 550
265 IF X$<>"     " THEN 270 \ C2=1 \ GO TO 550

270 C2=POS(L$,":",1)
280 IF C2>0 THEN 500
285 C2=1
290 GOSUB 10400 \ S2$=SEG$(L$,1,C2) \ GOSUB 10200
350 IF T3=M1 THEN 390
355 IF T3=0 THEN 390 \ REM I.E. A "."
360 PRINT "MULTIPLY DEFINED SYMBOL:",S2$
370 L4$="C" \ GO TO 1000

390 IF SEG$(L$,C2+2,C2+2)="=" THEN 420
400 PRINT "NO : ?",L$ \ GO TO 370 \ REM TREAT AS COMMENT

420 GOSUB 10400 \ S3$=SEG$(L$,C1,C2) \ C2=C2+4
450 GOSUB 10700 \ REM EVALUATE SYMBOL
455 IF S3$<>"." THEN 460 \ T1(0)=V1 \ GO TO 370
460 IF P3>M1 THEN 463 \ T1$(P3)=S3$ \ T1(P3)=V1 \ P3=P3+1 \ GO TO 370

463 PRINT "SYMBOL TABLE OVERFLOW:",S3$ \ STOP

500 T1$(P3)=SEG$(L$,1,C2-1) \ T1(P3)=T1(0) \ P3=P3+1 \ C2=C2+1
540 REM START HERE LOOKING FOR OPCODE
550 GOSUB 10500 \ IF S2$=";" THEN 370
620 C1=C2 \ GOSUB 10400 \ S2$=SEG$(L$,C1,C2) \ GOSUB 10300
655 T1(0)=T1(0)+1
657 IF T1(0)<65536 THEN 660 \ T1(0)=0
660 IF I1<>M2 THEN 670 \ PRINT "UNDEFINED OPCODE:",S2$
665 L3=42 \ L4$=" " \ GO TO 1000

670 IF I1<55 THEN 690 \ IF I1>64 THEN 690 \ T1(0)=T1(0)+1
690 L3=I1

1000 B1=L2 \ GOSUB 10000 \ L2$=H1$
1030 B1=L3 \ GOSUB 10000 \ L3$=H1$
1060 P$=SEG$(L2$&" "&L3$&" "&L4$&" "&L$,1,72)
1080 PRINT #3:P$ \ GO TO 210
```

Figure 4.
Pass 1 of IMP-16 cross assembler

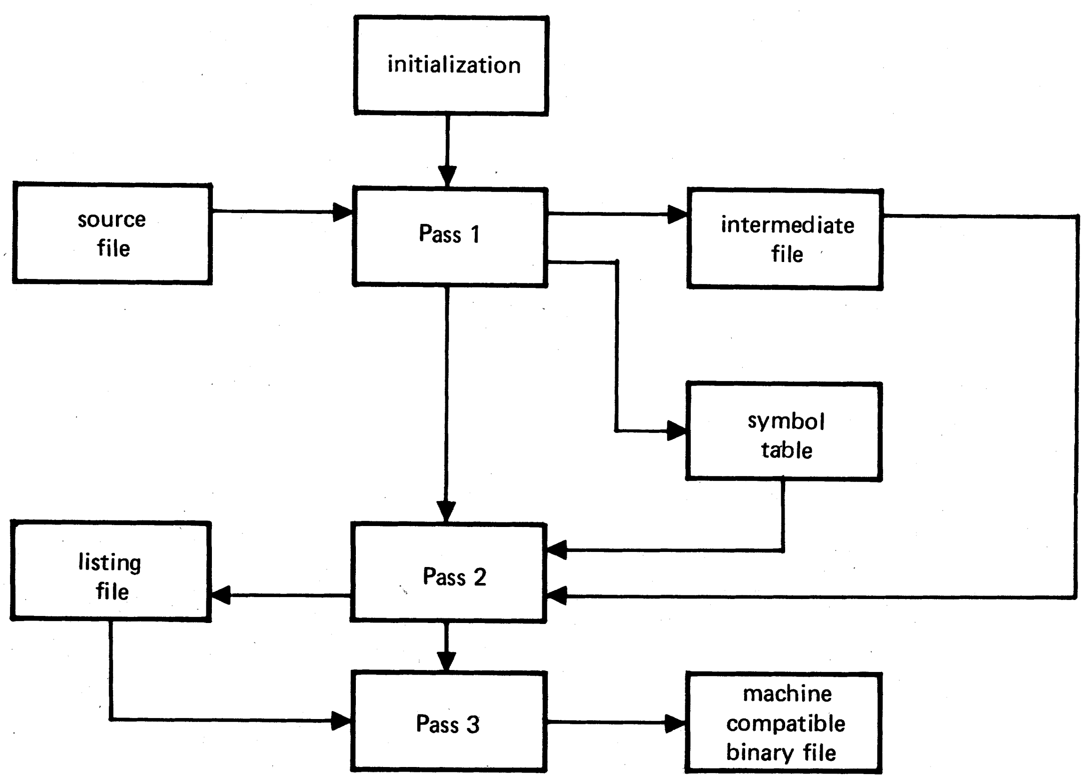


Figure 5.
Flow chart of cross-assembler

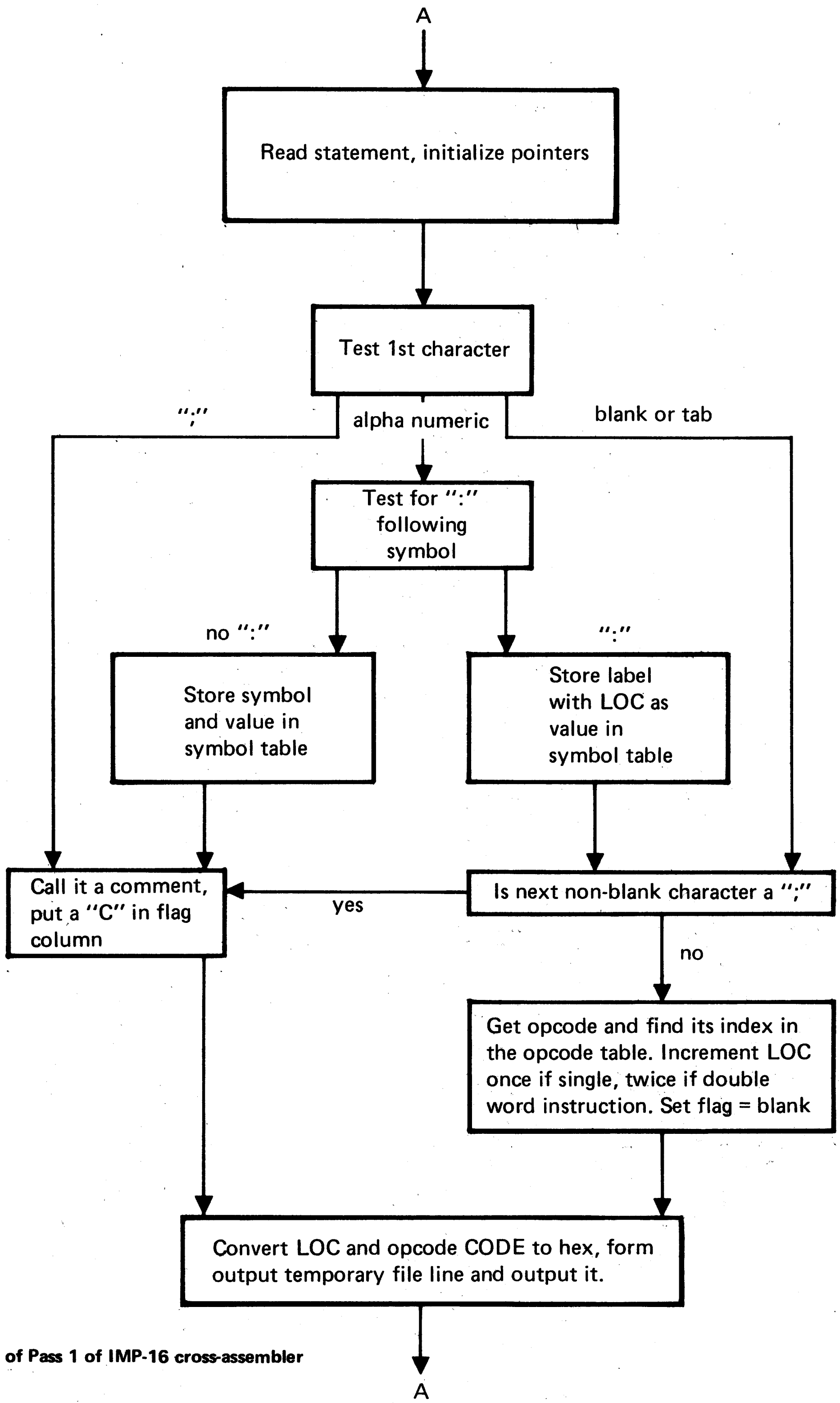


Figure 6. Flow chart of Pass 1 of IMP-16 cross-assembler

Pass 2 Pass two is the most difficult of the three passes. This is where the code becomes instruction-specific. For lack of space, only one type instruction will be followed through here. The entire BASIC code for Pass 2 and its associated subroutines is in Figure 7.

The instruction we will follow through is a "SKAZ"—skip-if-accumulator-zero. In Pass 1, we assigned it an index of decimal 25. The temporary file entry looks like this:

```
0100 0019 LABEL: SKAZ AC0,LOOP3 ;goto loop 3 if AC0=0
```

We enter the BASIC code at line 30100. We set X$ equal to the flag which is blank. Next we set H1$ equal to the CODE field, convert it to hex, and in line 30211 we position the character pointer C1 after the opcode and a tab or space (30213). Statements 30215-30218 position the character pointer to the first character of the operand field. We then branch out to the different opcode groups. From 30230 we go to 30640. We then call a general subroutine to get the register number. This subroutine 10900 verifies that the value of the symbol indicating the register is within the

```
10900 GOSUB 10700
10910 IF V1<0 THEN 10920 \ IF V1>3 THEN 10920
10915 RETURN
10920 PRINT "VALUE TOO LARGE FOR FIELD:",V1,L$
10930 L4$="N"
10940 V1=0 \ RETURN

11000 GOSUB 10700
11010 IF V1<-128 THEN 11050 \ IF V1>127 THEN 11050
11020 IF V1<0 THEN 11030 \ RETURN
11030 V1=256+V1 \ RETURN
11050 GO TO 10920

11100 GOSUB 11000
11110 IF V1<0 THEN 11120 \ RETURN
11120 PRINT "NEG VALUE IN FIELD:",L$
11130 GO TO 10930

11200 GOSUB 11100
11210 IF V1=0 THEN 11220 \ RETURN
11220 PRINT "ZERO FIELD VALUE:",L$
11230 GO TO 10930

11300 PRINT "NOT ENOUGH ARGUMENTS:",L$
11310 L4$="N" \ V1=0 \ RETURN

11400 REM GET ADDR MODE AND AMOUNT
11410 C1=C2 \ V2=0
11420 GOSUB 10600 \ C3=C2 \ REM FIND SPACE
11430 C2=C1
11440 IF SEG$(L$,C2,C2)="(" THEN 11500 \ IF C2>=C3 THEN 11460
11450 C2=C2+1 \ GO TO 11440
11460 C2=C2+1 \ S2$=SEG$(L$,C1,C2-2) \ GOSUB 10200
11470 IF T3<>M1 THEN 11700
11480 PRINT "UNDEFINED SYMBOL:",S2$,L$
11490 V1=0 \ L4$="N" \ RETURN

11500 S2$=SEG$(L$,C1,C2-1) \ GOSUB 10200
11510 IF T3=M1 THEN 11480
11520 V2=V1 \ C2=C2+1 \ C1=C2
11530 IF SEG$(L$,C2,C2)=")" THEN 11560 \ IF C2>=C3 THEN 11550
11540 C2=C2+1 \ GO TO 11530
11550 PRINT "NO CLOSE PAREN:",L$ \ GO TO 11490
11560 S2$=SEG$(L$,C1,C2-1) \ GOSUB 10200
11570 IF T3=M1 THEN 11480
11580 IF V1>3 THEN 11590 \ IF V1>1 THEN 11610
11590 PRINT "ILLEGAL INDEX REGISTER:",S2$,L$
11600 L4$="N" \ V1=2
11610 V1=V1*256
11620 RETURN

11700 V2=V1
11710 IF V2>255 THEN 11720 \ V1=0 \ RETURN
11720 V1=256
11740 RETURN

11800 S2$="#"&S2$ \ GO TO 10200
```

Figure 7. Pass 2 and associated subroutines **(continued on next page)**

range 0-3. Because opcodes 23, 24, and 25 only allow registers AC0 and AC1, we have another error detection statement in 30690. If it is in error we set the flag field to "N" to indicate error. This field could be used to give an error code with or without the accompanying error message that is given here. Statement 30710 then checks that we have another argument: the address. Next, the branch address must be computed by 11400. The addressing form must be decided upon because the IMP16 has PC relative, base and index forms for this instruction. Statement 30795 finishes the instruction-specific part by computing the machine code for the instruction: opcode-base-from-table + register*offset-in-word + addressing-mode + address-offset. Control is transferred to the general routine 45000, which composes the listing line, outputs it, prints a period for the impatient user, and returns for another line. The listing file is as follows:

ADDRESS MACHINE-CODE FLAG SOURCE-STATEMENT

(Figure 7 continued)

```
11900 H1$=SEG$(L$,2,5) \ GOSUB 10100 \ X=V2-(B1+1)
11910 IF X<-128 THEN 11920 \ IF X<128 THEN 11930
11920 PRINT "OUTSIDE OF PC REL RANGE:",X,L$ \ X=0
11930 IF X>=0 THEN 11940 \ X=256+X
11940 RETURN

12000 GOSUB 10700
12010 IF V1<0 THEN 12020 \ IF V1<16 THEN 12030
12020 GOSUB 10920
12030 RETURN

30000 REM
30045 PRINT \ PRINT \ PRINT \ PRINT
30060 CLOSE #3 \ CLOSE #2
30080 OPEN F3$ FOR INPUT AS FILE #2

30100 IF END #2 THEN 50000
30110 INPUT #2:L$
30115 L4$=" "\X$=SEG$(L$,13,13)
30130 IF X$<>"C" THEN 30200
30140 X$=SEG$(L$,15,255)
30150 P$="                    "&X$
30160 PRINT #4:P$
30170 GO TO 30100

30200 H1$=SEG$(L$,8,11)
30210 GOSUB 10100
30211 C1=POS(L$,O1$(B1)&"       ",15)
30212 IF C1<=0 THEN 30213 \ C1=C1+LEN(O1$(B1)) \ GO TO 30215
30213 C1=POS(L$,O1$(B1)&" ",15)+LEN(O1$(B1))
30215 C2=C1
30217 GOSUB 10500
30218 B2=B1
30220 IF B2>31 THEN 40000
30230 IF B2>14 THEN 30640
30240 IF B2>8 THEN 30500
30250 IF B2>4 THEN 30400
30260 GOSUB 10900 \ REM GET REGISTER
30270 R1=V1 \ IF E1<>1 THEN 30290 \ GOSUB 11300 \ GO TO 30330
30290 GOSUB 10900
30380 L3=R1*1024+V1*256+O2(B2) \ GO TO 45000

30400 GOSUB 10700
30420 R1=V1 \ IF E1=1 THEN 30280 \ GOSUB 11100
30440 IF B2<>6 THEN 30450 \ V1=256-V1
30450 IF B2<>8 THEN 30460 \ V1=256-V1
30460 L3=R1*256+V1+O2(B2) \ GO TO 45000

30500 GOSUB 10900
30510 L3=V1*256
30515 V1=0
30520 IF B2>11 THEN 30540
30525 IF E1<>1 THEN 30530 \ GOSUB 11300 \ GO TO 30540
30530 GOSUB 11000 \ IF V1>=0 THEN 30540 \ V1=V1+256
30540 L3=L3+O2(B2)+V1 \ GO TO 45000

30640 R1=0 \ IF B2>25 THEN 30740
30650 GOSUB 10900 \ REM GET REGISTER
30660 R1=V1
30670 IF B2<23 THEN 30710 \ REM SEPERATE OUT AC0,AC1 ONLY INSTR
30680 IF V1<2 THEN 30710
30690 PRINT "AC2,AC3 ILLEGAL IN:AND,OR,SKAZ:",L$
30700 V1=0 \ L4$="N"
30710 IF E1<>1 THEN 30750
30720 GOSUB 11300 \ GO TO 30750

30740 L3=0
30750 REM MAY BE OF INDEX FORM:-X(ACN)
30760 GOSUB 11400
30770 X=V2 \ IF V1<>256 THEN 30820 \ REM  SEPERATE OUT PC RELATIVE
30780 GOSUB 11900
30795 L3=V1+X+O2(B2)+R1*1024 \ GO TO 45000

30820 IF V1=0 THEN 30840 \ GOSUB 11930 \ GO TO 30790
30840 IF X>=0 THEN 30790 \ PRINT "NEG ADDR?:",X,L$ \ GO TO 30810
```

```
40010 IF B2>43 THEN 40300
40020 IF B2>32 THEN 40100
40030 GOSUB 12000
40060 R1=V1 \ IF E1<>1 THEN 40070 \ GOSUB 11300 \ GO TO 40080
40070 GOSUB 10700 \ V2=V1 \ GOSUB 11900
40080 L3=4096+R1*256+X \ GO TO 45000

40100 R1=0\V1=0
40105 IF B2>38 THEN 40200
40110 IF B2>34 THEN 40170
40120 GOSUB 10700
40130 IF V1<0 THEN 40140 \ IF V1<7 THEN 40150
40140 GOSUB 10920
40150 R1=V1\IF E1<>1 THEN 40170 \ GOSUB 11300
40160 GO TO 40200

40170 GOSUB 10700
40180 IF V1<0 THEN 40190 \ IF V1<127 THEN 40200
40190 GOSUB 10920
40200 L3=256*R1+V1+O2(B2) \ GO TO 45000

40300 IF B2>54 THEN 40500
40310 IF B2>50 THEN 40400
40320 GOSUB 12000
40340 L3=V1+O2(B2) \ GO TO 45000

40400 GOSUB 10700
40405 IF V1<0 THEN 40460
40410 IF B2<>51 THEN 40420 \ V1=V1-288
40415 IF V1>15 THEN 40460
40420 IF B2<>52 THEN 40430 \ V1=V1-256
40425 IF V1>15 THEN 40460
40430 IF B2<>53 THEN 40440 \ V1=V1-256
40435 IF V1>127 THEN 40460
40440 IF B2<>54 THEN 40470 \ V1=V1-65408
40445 IF V1<128 THEN 40470
40460 X=V1 \ GOSUB 11920
40470 IF V1<0 THEN 40460 \ L3=V1+O2(B2) \ GO TO 45000

40500 IF B2>64 THEN 40600
40510 GOSUB 11400
40515 IF V1<>256 THEN 40520 \ V1=0
40520 L3=V1+O2(B2)
40530 GOSUB 45005
40540 IF B2<59 THEN 40550 \ V2=V2*2
40550 IF B2<>61 THEN 40560 \ V2=V2+1
40560 IF B2<>63 THEN 40570 \ V2=V2+1
40570 H1$=SEG$(L2$,2,5) \ GOSUB 10100
40575 B1=B1+1 \ GOSUB 10000 \ L2$=H1$&" "
40580 L3=V2\L$=" " \ L4$=" " \ GOSUB 45010 \ GO TO 30100

40600 IF B2<>65 THEN 40630
40610 GOSUB 10700
40620 L3=V1 \ GO TO 45000

40630 IF B2<>66 THEN 40660
40640 GOSUB 11000
40650 L3=V1 \ GOSUB 11000 \ L3=L3*256+V1 \ GO TO 45000

40660 IF B2<>69 THEN 40700
40670 N1$=SEG$(L$,C2,255)
40680 N1$=TRM$(N1$) \ L4$="C"
40690 GO TO 30140

40700 GOSUB 10700
40710 L3=V1 \ L4$="S" \ L2$="       " \ GOSUB 45010 \ GO TO 30100

45000 GOSUB 45005 \ GO TO 30100
45005 L2$=SEG$(L$,1,6)
45010 B1=L3 \ GOSUB 10000 \ L3$=H1$
45020 L$=SEG$(L$,14,255)
45030 P$=L2$&L3$&" "&L4$&"  "&L$
45033 PRINT #4:P$
45035 IF P9=0 THEN 45040 \ PRINT P$ \ RETURN
45040 PRINT ".";  \ RETURN
```

Pass 3 The final pass consists of printing the symbol table in alphanumeric order, and then outputting the machine code in a form suitable for the microprocessor loader or for a ROM burner, etc. The symbol table code is in Figure 8. It is a simple "bubble" sort, again not chosen for speed, but for its small core requirement and ease of debugging.

```
50000 CLOSE #4
50010 IF A$="SYM" THEN 50020 \ STOP
50020 PRINT \ PRINT
50030 FOR J=0 TO P3-1
50035 X$="ZZZZZZZZ"

50040 FOR K=0 TO P3-1
50050 IF T1$(K)<X$ THEN 50060 \ GO TO 50080
50060 IF T1$(K)>"." THEN 50070 \ GO TO 50080
50070 X4=K \ X$=T1$(K)
50080 NEXT K

50090 B1=T1(X4) \ GOSUB 10000 \ PRINT T1$(X4),H1$
50095 T1$(X4)=" "
50100 NEXT J
50110 STOP
```

Figure 8. Symbol table sort and listing section

The generation of the machine code in a suitable format is not described here because it is not only target machine dependent, but application dependent too (ROM burner, paper tape, floppy disk, etc.). It is quite simple to write since the code is already available in the listing file. The listing file must be opened and read sequentially to pick off addresses, if needed, and the absolute code, converted to binary from the hex string format, and output in the proper format.

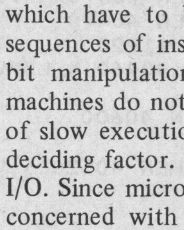
Conclusion

Besides the basic BASIC assembler described here, many types of preprocessing programs could be envisioned. One example is a macroprocessor. Other preprocessors, such as the many forms of PL/1 languages for microprocessors, could then feed the macroprocessor. All of these could be written in BASIC as separate programs with intermediate files—e.g., source PL/1, macro, assembly, temporary for assembler, and finally machine code. Each of these could be programmed totally independently once the "languages" were defined.

When you are programming on a BASIC-only system, which many timesharing systems are, the source program of the target computer (e.g., IMP-16) could be written and edited entirely under the BASIC editor. The cross assembler could simply throw away the line numbers on input, or better yet, label the output listing with the line numbers on the extreme left. This is especially useful with error messages, because the user can then use the editor to list just that line for correction without searching the entire program to try to locate the line in error.

If paper tape is available, the cross assembler could produce the absolute machine code directly for loading into the microprocessor. If a relocating loader is available on the microprocessor, a relocatable binary tape could be produced. In the absence of a relocating loader, this could also be programmed in BASIC to operate on disk binary files previously created by the cross assembler to produce an absolute binary paper tape. Of course, this final tape could also be in hexadecimal or BPNF format for a ROM burner. Another alternative involves using one of the new floppy disk subsystems such as the TEC Corporation's DISCO-TEC. This is a buffered floppy disk system with two RS232 serial ports. This unit could be inserted between the user's terminal and the timesharing computer to produce binary files on the disk, then connected between the terminal and the microprocessor's RS232 I/O port to load the microprocessor. The beauty of this system is the hardware and software transparency—i.e., the two CPU's function exactly as if they were communicating with a terminal with paper tape.

A microprocessor simulator written in any high-level language presents two basic problems: speed and accuracy of simulation. With regard to speed of simulated execution, many microprocessors contain a number of internal flags which have to be exactly simulated to cover all possible sequences of instructions. There is also a great amount of bit manipulation, especially if the word lengths of the machines do not match. Even if you can live with the cost of slow execution, the problem of accuracy becomes the deciding factor. This accuracy problem is most apparent in I/O. Since microprocessor programs are most often directly concerned with I/O, it becomes quite difficult to verify routines which will depend on these external factors. One of the solutions to this problem is having the simulator keep track of the simulated execution time. Then subroutines can be written which vary these simulated external parameters in a random way within the range of the actual device parameters. Of course, this increases the simulation execution time again. In general, if the microprocessor is available with a front panel, or a simulated terminal front panel (e.g., MOS Technology's TIM and KIM), simulation on the target machine itself will be superior in time and money to a simulation on a timesharing CPU. ■



Steve Conley is currently finishing his PhD requirements at the University of Arizona in electrical engineering, with a dissertation on a DARE microprocessor system running under a BASIC operating system. The DARE program is a continuing effort at the U of A to replace the analog computer with the digital in both simulation and control applications. He received the BS in computer science at Vanderbilt in 1971 and the MSEE at U of A in 1973. During the past two years he has been a consultant in hardware and software for several mini/microcomputer firms. His interests lie in systems programming, real-time/multiprogramming systems, and (for fun) computer music and games.