DESIGN AND IMPLEMENTATION OF AN IBM ASSEMBLY LANGUAGE ASSEMBLER

by

HONG KIEN ONG

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

BACHELOR OF SCIENCE
IN COMPUTER SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1980

Signature of Author Signature redacted
Department of Electrical Engineering
May 9, 1980

Certified by Signature redacted
Stuart E. Madnick
Thesis Supervisor

Accepted by Signature redacted
David Adler
Chairman, Departmental Committee on Theses

Design and Implementation of an IBM Assembly Language Assembler

by
Hong Kien Ong

Submitted to the Department of Electrical Engineering on May 1980 in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering

ABSTRACT

ASSMBLR is an assembler for a subset of the IBM System/370 assembly language. This assembler is coded in PL/1. Rather than handling the complete instruction set of the assembly language, ASSMBLR accepts a sufficiently large subset to permit its use for teaching purposes. Since ASSMBLR was designed in part to be used as a pedagogical tool to demonstrate the workings of an assembler, emphasis was placed on simplicity and clarity.

Thesis Supervisor: Prof. Stuart E. Madnick
Associate Professor of Management Science
M.I.T. Sloan School of Management

TABLE OF CONTENTS

1 INTRODUCTION

ASSMBLR is an assembler designed to translate a subset of the IBM System/370 Basic Assembly Language into machine code. The program itself is written in PL/1. There are several factors that provided the motivation for this project. First, this mini-assembler could be used as an example to illustrate the structure of assemblers in general. Second, and more specifically, a working IBM machine language simulator exists at M.I.T. Together with this simulator, ASSMBLR could be used to actually run student programs in assembly language on machines other than the IBM 360/370. In fact, this assembler was written to be used on the Prime 400 minicomputer of the East Campus Computer Facility at M.I.T. and was compiled using version 17.3 of the PL/1-G compiler on the Prime machine.

In its current form, the program reads source assembly language programs from a file and prints out the object deck, a machine code listing in hexadecimal, on the user's terminal. With slight modification, the output could be diverted into a file and hence used as the source program for the simulator.

2 NATURE AND FUNCTION OF AN ASSEMBLER

A machine language is a very low level language that allows a programmer to work with the instructions that are actually executed by the particular machine for which the language was designed. Because of this, machine language allows the programmer to control much of what he really wants the machine to execute. At the same time, having to write in a language that directly translates into a sequence of ones and zeroes makes programming in machine language a very tedious job. In machine language, the programmer could use mnemonic names for the machine instructions, but he has to explicitly compute all the addresses of the operands. This is where the assembler becomes very useful.

An assembler is a program that accepts as input an assembly language program, and converts it into the corresponding program in machine language. The main benefit of writing programs in assembly language is that the programmer can now use symbolic labels and leave the assembler to compute the actual addresses that correspond to these labels. Yet, he does not have to sacrifice control over the actual machine instructions to be executed, something he would have to if he were to program in a higher level language. Assembly language also permits the use of symbolic constants which in machine language must be written in the precise formats that directly correspond to their representations in bits.

Besides converting programs from one language into another, the assembler also recognizes certain instructions that do not generate actual machine code. Rather, these instructions provide control information to the assembler. These instructions are called assembler instructions, as distinct from machine instructions. There is a one-to-one correspondence between machine instructions and the operations that are executed by the machine. There is no such correspondence for assembler instructions. Assembly language also provides additional power to the programmer by allowing him to refer to other program sections, the addresses of which are not known to the programmer until these programs have been loaded into the computer's memory.

2.1 Design Criteria

Throughout the design process, importance was placed on producing a modular but compact assembler that could handle a reasonably large set of instructions. The structure of this assembler reflects that emphasis on modularity and clarity.

The costs and benefits of using table driven algorithms were also considered. Because of the nature of an assembler, a substantial amount of information was necessarily required in one form or another, and it was decided that there was not much benefit to be derived from trying to eliminate whatever little additional table-stored information that is, strictly speaking, redundant. For instance, eliminating the machine length field in the machine Instruction Table could have reduced the memory requirements for the table by about 300 bytes. However, it is uncertain that the additional computation that would have had to be done, had this information been deleted, would not have taken up that much memory anyway.

As for the design of the overall structure and the division of certain sections into separate routines, emphasis was given to designing routines that were flexible and could be applied to several similar but non-identical operations. For instance, many of the the same routines were applied for both literals implicitly defined as operands, and constants defined through the DC and DS instructions. By paying particular attention to modularity and functionality we have attempted to make the program modular and readable but at the same time avoid the proliferation of individual and specialized routines.

2.2 Approach of Thesis

This thesis was arranged bearing in mind a reader who has some familiarity with assembly languages and machine languages in general, not necessarily those of IBM's System

360/370.

The section immediately following this briefly describes the meanings of the assembler instructions that are accepted by ASSMBLR. The various formats of the actual machine instructions are described in Chapter 3. These descriptions are not meant to be sufficiently detailed as to enable a person to learn assembly language programming. Nonetheless, they should provide enough detail to enable the reader to read through the thesis without having to consult IBM manuals. Next, in Chapter 4, we describe the databases that are employed by ASSMBLR. These data bases contain both static information such as those pertaining to the different lengths and formats of the various instructions, and dynamic information that is required to convert mnemonic addresses into numerical values.

In Chapter 5 we come to the main body of the thesis. This chapter describes the actual division of ASSMBLR into the main procedure body and its internal and external subroutines. The main actions taken by the ASSMBLR in both passes are described. In Pass 1 the assembler primarily collects all the symbolic definitions and stores them in the appropriate tables, assigning numeric values whereever necessary. In pass 2, ASSMBLR uses the tables to generate the actual addresses in place of the symbolic references. It also generates the actual object code in this pass. The various subroutines perform various functions that can be grouped as logical entities. The actual function of each of the routines are described in this chapter. By Chapter 6 the reader should have read enough to understand the operation of this assembler. Hence this chapter describes the contraints on the assembly language program that is to be given to ASSMBLR as a source program. These contraints are by no means exhaustive. They describe primarily the additional contraints that are not imposed by the convention for the full IBM Basic Assembly Language.

2.3 Scope of Thesis

Specifically, ASSMBLR accepts eight out of the full set of assembler instructions permitted in IBM Basic Assembly Language. These instructions are CSECT, USING, DROP, DC, DS, EQU, LTORG, and END. CSECT and END are used to mark the beginning and end of a program respectively. USING specifies the use of a certain register as a base register, while drop indicates that a particular register is no more available to be used as a base register. EQU defines a correspondence between a symbol and a value, that is, it defines the meaning (or value) of a mnemonic symbol. DC instructions are used to define various types of data that are to be included in the object program, while DS instructions reserve storage space in the object program. Finally, an LTORG instruction instructs the assembler to generate data for all the literals that have been used so far, or since the previous LTORG instruction. Literals are similar to data defined by DC instructions except that in this case the actual assignment of memory for the data is handled by the assembler.

3 INSTRUCTION FORMATS AND TRANSFORMATIONS

There are six different machine instruction formats in the IBM Basic Assembly Language. Of these six, five are implemented for (or recognized by) ASSMBLR. ASSMBLR also accepts extended mnemonic instructions, but not floating point instructions.

3.1 RR Format Instructions

RR type instructions are instructions which take both operands from registers. The machine instruction operand format is R1,R2 and their bit representation is shown in Figure 1. These instructions are two bytes long. In its database, ASSMBLR encodes RR Type instructions as being of type 1. There are two exceptions to this; they are the SPM and SVC instructions. Although they both have RR Format, their operand fields are slightly different from the other RR instructions. The SVC instruction takes only one operand and its bit representation is shown in Figure 2. This instruction is encoded as type 2. The SPM instruction does not have a second operand either. However, its sole operand only takes up four bits, unlike the SVC instruction whose operand takes up a whole byte. The SPM instruction is recognized by ASSMBLR as a type 3 instruction and its bit format is shown in Figure 3.

3.2 RX Format Instructions

RX machine instructions have the format R1,D2(X2,B2) or M1,D2(X2,B2) in machine language. ASSMBLR recognizes these instructions as being of type 4. The machine instruction for RX format instructions is shown by Figure 4.

3.3 RS Format Instructions

RS format instructions have one of the following two operand formats: R1,R3,D2(B2) or R1,M3,D2(B2). Both are considered as type 5 instructions by our assembler. The machine representation is shown in Figure 5. The RS format instructions that are the exceptions to this rule are the shift instructions, both the arithmetic shifts and the logical shifts. These instructions are encoded as being of type 6, and have the operand format R1,D2(B2) which is represented in the machine as in Figure 6.

3.4 SI Format Instructions

SI format instructions are 4 bytes long and have the operand fields D1(B1),I2. Their actual machine representation can been seen in Figure 7. They are known as type 7 instructions.

3.5 S Format Instructions

S format instructions are not implemented for this particular assembler. All instances of type S instructions will be assembled as having 'ØØ' as their op-code.

3.6 SS Format Instructions

There are two different groups of SS instructions. One group has the operand fields D1(L1,B1),D2(L2,B2) and the machine representation of Figure 8. The other has the operand fields of type D1(L,B1),D2(B2) and their machine representation is shown in Figure 9. SS instructions are 6 bytes long. In ASSMBLR the first group is recognized as being of type 8 and the second as being of type 9.

3.7 Illegal Instructions

Instructions whose mnemonics are not recognized by ASSMBLR are treated as being illegal and they are assembled as a series of zeroes.

3.8 Extended Mnemonics

These Instructions are mnemonics for conditional branches and are actually equivalent to the BC or BCR instructions. The various mnemonics correspond to different branching conditions otherwise encoded as the first operand, or mask value, of the BC or BCR instructions. ASSMBLR converts these instructions into the full code for the BC or BCR instructions, whichever happens to be appropriate.

```
|Op-Code|  R1   |  R2   |
Ø      7,8   11,12    15
```

Figure 1: RR Instructions

```
| SVC   |        I      |
Ø      7,8             15
```

Figure 2: SVC Instruction

```
|Op-Code|  R1   |XXXXXXX|
Ø      7,8   11,12     15
```

Figure 3: SPM Instruction

```
|Op-Code|  R1   |  X2   |  B2   |   D2     |
Ø      7,8   11,12   15,16  19,20        31
```

Figure 4: RX Instructions

```
|Op-Code|  R1   | R3/M3 |  B2   |   D2     |
Ø      7,8   11,12   15,16  19,20        31
```

Figure 5: RS Instructions

```
 ___________________________________________
|Op-Code|  R1   |XXXXXXX|  B2   |    D2     |
Ø       7,8   11,12   15,16  19,2Ø         31
```

Figure 6: Shift Instructions

```
 ___________________________________________
|Op-Code|          I            |  B2   |    D2     |
Ø       7,8                   15,16  19,2Ø         31
```

Figure 7: SI Instructions

```
 _______________________________________________________________
|Op-Code|  L1   | L2/I3 |  B1   |    D1     |  B2   |    D2     |
Ø       7,8   11,12   15,16  19,2Ø       31,32   35,36        47
```

Figure 8: SS1 Instructions

```
 _______________________________________________________________
|Op-Code|          L1           |  B1   |    D1     |  B2   |    D2     |
Ø       7,8   11,12   15,16  19,2Ø       31,32   35,36        47
```

Figure 9: SS2 Instructions

4 DATA BASES

The data bases used by the assembler can be roughly grouped into two separate types:

i) databases that contain information known before execution of the assembler. These store information such as the mnemonic names and op-codes of instructions. The values stored in these data bases are done through the 'initial' option of the declaration.

ii) databases that store information collected dynamically during execution of ASSMBLR. Most of this information is collected during Pass 1 of the assembler and used during Pass 2. An example of this is the information regarding the use of base registers and the contents of these registers.

4.1 Static Data Bases

4.1.1 Pseudo-Op Table (POT)

The Pseudo-Op Table contains a list of pseudo-op instructions recognised by ASSMBLR: DC, DS, LTORG, EQU, CSECT, USING, DROP, END. This table is used to index into the relevant blocks that contain the actions required for each of these pseudo-ops.

4.1.2 Extended Mneumonic Table (EOT)

```
1  EOT(32)
```

```
    2  NAME
    2  TYPE
    2  OP1
```

The Extended Op-code Table contains information on the extended mnemonics for branching conditions: NAME has the mnemonic names of the instructions. TYPE distinguishes between the RR and RX format instructions. OP1 contains the values of the condition code masks corresponding to these extended mnemonic instructions.

4.1.3 Machine Instruction Table (MOT)

```
1  MOT(131)
    2  NAME
    2  TYPE
    2  BIN_CODE
    MLEN
```

This table contains the name, type, binary code and length of the relevant machine instructions. NAME uniquely identifies the mnemonic name of the machine instruction. For to each instruction, TYPE encodes the corresponding instruction that enables us to form the actual machine instruction. BIN_CODE provides the corresponding binary op-code for the machine instruction, while MLEN indicates its length. This data base has some redundancy because the instruction length could, in fact, be deduced from the instruction type. However, we decided that it was worth using a little more memory space in order to simplify the process of computing the instruction length.

4.2 Dynamic Data Bases

4.2.4 Symbol Table(ST)

```
1  ST(1ØØ)
   2  SYMBOL
   2  VALUE
   2  RELOC
   2  DEFN
SYMBOL_NO
```

The symbol table maintains a list of the labels (SYMBOL) used in the program, the location (VALUE) in which each label is defined, its relocatability (RELOC), and the line where the symbol was defined (DEFN). In addition, a variable called SYMBOL_NO is used to keep track of the total number of labels in the symbol table.

4.2.5 Literal Table (LT)

```
1 LT(2Ø)
   2  LABEL
   2  VALUE
LT_TOP
```

LT_BOT

The Literal Table is similar to the Symbol Table except it keeps track of the literals used (LABEL) and their addresses (VALUE). Two variables, LT_TOP and LT_BOT, are also used to point to respectively the top and the bottom of the Literal Table. The Literal Table is more complicated than the Symbol Table in that literal definitions are not explicit and hence the table has to be checked to ensure that multiple references to the same literal do not result in multiple entries unless a 'LTORG' pseudo-op separates the two references. LT_TOP keeps track of the top of the Literal Table since the last 'LTORG' and LT_BOT points to the bottom of the table thus far.

4.2.6 LTORG Table (LTORG)

LTORG is an array that is used in conjunction with the Literal Table. Each time an LTORG pseudo-op is executed, the current index in the Literal Table is entered into LTORG. This provides us with information on the relevant sections of the Literal Table in between LTORG instructions. LTORG_C is used to indicate which entry of LTORG should be used at the next LTORG pseudo-op.

4.2.7 Base Register Table (BT)

```
1  BT(16)
```

2 REGISTER

2 VALUE

BASE_NO

The entries in the Base Register Table are kept in ascending order according to the contents of the registers. Each structure entry has a register number and the contents associated with the register. By having the registers sorted in a linear order, it is easy for us to look for the appropriate base register, i.e. the register which would result in the smallest displacement for the instruction operand address. BASE_NO keeps track of the number of base registers currently in use.

5 DESCRIPTION OF PROGRAM

This section describes the actual working of the main procedure, ASSMBLR, and its various external and internal routines. Wherever possible, routines are made external. This allows us to make changes in the main program without having to recompile all the routines, thereby reducing compilation time for the main program. The internal routines are mostly those that make references to global variables.

5.1 Main Procedure

5.1.1 Pass 1

The main function of Pass 1 is to build the Symbol Table and the Literal Table, by associating numeric address values to the mnemonic labels and literal definitions. The value of a label defined in a machine instruction or a DS/DC instruction is the value of the location counter at the point the label appears in the name field. The location counter is pre-adjusted for boundary alignment where necessary. For EQU statements, the label value is the value of the expression in the operand field. Therefore, the expression must only contain constants, or symbols whose values have been previously defined.

Literal definitions, on the other hand, are stored without their address values until space is actually allocated for them, either explicitly through a LTORG instruction or implicitly upon execution of an END statement. Multiple references to the same literal do not generate multiple copies of that literal unless the references are separated by LTORG instructions. In other words, literals can only have address values that are greater than the addresses of the instructions where these literals are defined.

5.1.2 Pass 2

Pass 2 utilizes the tables formed in Pass 1 to convert mnemonic labels and literal references into mnemonic addresses relative to the start of the program. The other primary function of Pass 2 is to generate the machine code.

Modularity is maintained in this section by having separate routines perform the code generating functions. MOTGEN is called when the instruction is a machine instruction, EOTGEN is called for extended op-code instructions, and LITGEN is called for literals and DC/DS instructions.

When a USING instruction is executed, the particular register referred to in the instruction is designated as a base register, and its number is inserted into the Base Register Table. The contents of the register is assumed to be the value of the first operand in the USING instruction. A DROP instruction does the exact opposite. It results in the particular register being dropped from the Base Register Table.

Execution of a LTORG statement or the END statement causes ASSMBLR to call LITGEN to generate machine code for all the literals, in the Literal Table, which have not been generated into machine code by previous LTORG statements.

5.2 Internal Routines

5.2.1 EOTGEN

EOTGEN generates the output when the operator is an extended instruction.

5.2.2 MOTGEN

MOTGEN generates the output when the operator is a machine instruction.

5.2.3 LITGEN

LITGEN generates the output when the instruction is a DC or DS pseudo-op. In the case of a DS pseudo-op, nothing is really generated since only storage space is allocated. LITGEN is also called when a LTORG or END pseudo-op is executed. In this case LITGEN creates the output for all the literals that have to be generated because of the LTORG or END instruction.

5.2.4 EVAL

EVAL evaluates a given expression string and returns its binary value. It also returns as a parameter the relocatability of the expression and sets a flag if the expression is illegal.

5.2.5 MULTDIV

MULTDIV is primarily a routine that simplifies the task of EVAL. It is called by EVAL to evaluate only expressions that contain no '+' or '-' operators, and it returns the value of the expression. It also checks for relocatability errors.

5.2.6 VAL

VAL is a routine that takes a character string and passes back the value of the argument. If the argument parameter is a character representation for a number, the result parameter is set to the value of that number. If the argument parameter is a symbol, the result is set to the value of the symbol in the Symbol Table. If the argument parameter is a '*', its value is the current value of the location counter.

5.2.7 SEPARATE_FIELDS

SEPARATE_FIELDS, as the name suggests, separates the current instruction line into its label (LABEL_FIELD), instruction (INST_FIELD), and operands (OP_FIELD). The changes are done through global variables and no parameters are passed.

5.2.8 LTGET

LTGET finds the entry of the given character string in the Literal Table. It takes as a parameter the character string representing the literal and returns the index to the entry in the Literal Table. If the literal does not exist, it returns a Ø.

5.2.9 LTSTO

LTSTO does the opposite of LTGET. Given a literal string as parameter, it stores this literal in the Literal Table if no such entry exists since the last LTORG instruction. If an identical literal is already stored in the table, nothing is done.

5.2.1Ø STGET

STGET is very similar to LTGET, except it searches the Symbol Table instead of the Literal Table. It takes as parameter the character string representing the symbol and returns the index of the entry in the Symbol Table. If the symbol is not declared in the table, it returns a Ø.

5.2.11 BRSET

BRSET is a routine that computes the base-displacement format for a given address. The address could be an expression or in base-displacement form. If the address is already in base-displacement form, all BRSET does is it takes the base and displacement parameters which are passed to BRSET as character strings, and converts them into their bit string representations. If an expression is passed as the argument, BRSET evaluates the expression, calls BRGET to find the appropriate base register and converts both the base register and displacement into the appropriate bit strings. In addition, BRSET checks for relocatability errors.

5.3 External Routines

5.3.1 CVB

CVB is a general routine which converts a character string that represents two hexadecimal digits into a bit string that is eight bits long (a byte).

5.3.2 CHARGET

CHARGET is a utility routine that returns the EBCDIC

8-bit representation for a given character argument. CHARGET uses a table to find the representation that corresponds to the given character.

5.3.3 POTGET

POTGET is similar to LT_GET and LTSTO, and returns the index of its argument in the Pseudo-Op Table.

5.3.4 EOTGET

Like POTGET, EOTGET get searches a table, except in this case the table searched is the Extended Instruction Table (EOT).

5.3.5 MOTGET

MOTGET searches the Machine Instruction Table for the appropriate entry and returns the index corresponding to the entry in the table. This routine uses a binary search algorithm.

5.3.6 BOUND

BOUND is a procedure that is used to align the location counter to the appropriate address boundary. BOUND takes two arguments: the location counter and the alignment count. The location counter is set to the smallest value, larger than or equal to the counter value, that is divisible by the alignment count.

5.3.7 DLENGTH

The purpose of this routine is to calculate the length of a literal or a constant. Given a character string that represents a literal or a constant, DLENGTH will return the length of the literal or constant, in bytes.

5.3.8 GET_OPER

GET_OPER is used to separate the leftmost operand from a list of operands. Given a parameter input of the operand field, GET_OPER separates the first operand in the string and truncates the input string to exclude the extracted operand (and the following comma, if there is one). The resulting operand is returned as a parameter.

5.3.9 GET_LTRL

GET_LTRL is very similar to GET_OPER except that it can only be called when the leftmost operand in the input string is a literal. GET_LTRL then separates the leftmost literal from the input string. The input string is truncated as is the case with GET_OPER.

5.3.10 PARSOP

PARSOP parses a string representing an address into its base register, index register, and displacement fields if these are explicitly stated in the operand. If the operand is a simple expression without any parentheses, PARSOP does not do anything to the string. If there are three components in the input argument, they are broken up in A, B, and C where A contains the displacement, B the index register, and C the base register. If there are two compnents, C is returned as a null string. If there is a comma within the parentheses, indicating that a third argument is implicit, that argument results in a null string being assigned to the corresponding return parameter.

5.3.11 PARSLIT

PARSLIT parses a literal into its duplication factor, type, modifier, and nominal value. Any missing fields result in null strings being returned for the corresponding fields.

5.3.12 NEXTOK

NEXTOK extricates that part of a given argument up to but excluding the next operand. That is, it pulls out the leftmost term in the expression. That term could be a symbol, a self-defining term or a location counter reference (*).

5.3.13 NEXTPLUS

This routine is only used by EVAL. Its function is to separate the leftmost part of an expression, up to the first '+' or '-' sign. The resulting expression is then given to MULTDIV by EVAL.

5.3.14 REGDROP

REGDROP performs most of what is required by the DROP instruction. It removes the relevant base register from the Base Register Table and makes the register no longer available to be used as a base register. It is also used by USING to clear the previous entry for a register that is reassigned as a base register through the USING instruction.

5.3.15 BTSTO

In conjunction with REGDROP, BTSTO performs what is required of a USING statement. It creates an entry in the Base Register Table for the given register, and also stores its value in the table. Note that this routine has nothing to do with the actual contents of the register. All it takes as the value of the contents is whatever value it is given in the USING statement, regardless of whether the value is correct or not.

5.3.16 BRGET

BRGET is the routine called by MOTGEN or EOTGEN when either routine need to convert an address into base-displacement format. BRGET does not actually compute its base-displacement, it merely returns the number of the register that should be used as a base register. If none are available, it returns a Ø.

5.4 An Example

One of the major functions of an assembler is to compute the value of expressions and convert these into numeric addresses. In order to elucidate the process that ASSMBLR goes through in this computation, let us now consider as an example the instruction:

```
ST   4,HERE+A*6
```

where HERE is the label of an instruction, and A is a constant defined through an EQU statement. Assume that in this case HERE has a relative address of 3Ø and A has the absolute value 4. Since there is neither a label nor a literal definition, nothing much is done in Pass 1. In Pass 2, after this card is read, SEPARATE_FIELDS is called, and it sets LABEL_FIELD to null, INST_FIELD to 'ST', and OP_FIELD to 'HERE+A*6'.

After calling MOTGET to search the MOT, Pass 2 calls MOTGEN to generate the machine code. When MOTGEN is called, it in turn callss GET_OPER to separate the two operands. Next MOTGEN calls EVAL to evaluate '4' and uses its value, 4, as the first operand. Then MOTGEN goes on to look at 'HERE+A*6'.

MOTGEN's first step in evaluating 'HERE+A*6' is to call PARSOP to see if this is already in base-displacement format. It is not, so PARSOP returns the whole expression to MOTGEN. Next BRSET is called to convert the expression into base-displacement format.

BRSET evaluates the expression by calling EVAL. EVAL first calls VAL to get the value of 'HERE' and then calls MULTDIV to get the value of 'A*6'. VAL is only called when the term is an expression without any operators. MULTDIV is called when the expression contains only '*' or '/' operators. VAL will return the address value 30 and MULTDIV will return the value 24. EVAL adds the two and returns 54 to the calling procedure, BRSET.

BRSET uses this 54 and calls BRGET to select an appropriate base register if there are any available. BRGET returns the number of the base register to be used. Then BRSET uses this to compute the displacement value and returns both the base register and displacement values.

Finally, MOTGEN takes these values, arranges them in the proper format, and prints out the machine code representation that corresponds to this instruction. Part of the calling sequence for this example is shown in Figure 1Ø.

```
                  BRSET
                   / \
                  /   \
                 /     \
             EVAL     BRGET
              /\
             /  \
            /    \
     MULTDIV  MULTDIV
        /        /\
       /        /  \
      /        /    \
    VAL      VAL    VAL
```

Figure 1Ø: Calling Sequence

6 CONSTRAINTS

ASSMBLR is implemented for a subset of the assembly language for the IBM System 360/370. Therefore, there are additional constraints imposed on programs meant to be assembled by ASSMBLR.

6.1 General Program Structure

1. Only 1 control section is allowed and the first instruction must be a CSECT.
2. Labels must start on column 1.
3. ASSMBLR implements the floating fields format. Hence the various fields must be separated by 1 or more spaces between fields.
4. There can be no spaces between operands or within operands.
5. No macros are allowed.

6.2 Instructions

1. No floating point instructions are allowed.
2. No S format instructions are allowed.
3. Instruction op-codes must be mnemonic.
4. If an instruction operand field contains a literal, the

literal must be the last operand for the instruction.

5. Length attributes in SS instructions must be explicitly inserted.

6.3 Expressions

1. Valid expressions are those in which constants (or self-defining terms in IBM jargon) are written as decimal integers. Signs are optional.

2. The allowed operators are +, -, /, and *. Parentheses are not allowed.

6.4 Constants and Literals

1. Only constants and literals of type C, X, B, F, H, A are recognized.

2. Literals must start with the '=' symbol.

3. Symbol length attribute references are not allowed.

4. The duplication factor must be an unsigned decimal integer, if one is used at all.

5. For type 'C' constants or literals, only a subset of the full EBCDIC character set is implemented. Specifically, quotes (') are not permitted.

6. Only one address can be specified within the parentheses of an A type constant.

7. For X type constants the number or characters enclosed in quotes must be even. For B type constants, the number of binary digits enclosed within the quotes must be an integral multiple of 8.

8. Each constant must include a type attribute. For example, DC F'7,8,9' should be written as
DC F'7',F'8',F'9'.

7 CONCLUSION

ASSMBLR demonstrates that by using a high level language it is possible to create a reasonably flexible machine independent assembler. Planning in the form of careful division of the program into highly utilized and highly flexible modules is very important if complexity is to be contained. Our approach was to design many of the modules such that they were sufficiently flexible and they can handle a reasonably large number of cases, yet care was taken to ensure that these modules were not so generalized as to be very complicated. This approach has enabled us to design and implement an assembler that is reasonably flexible and yet maintain program readability.

As a general rule, ASSMBLR can be used as an example in the design of cross assemblers as well. The target machine language and source assembly langauge need not belong to the same machine. The necessary criterion that will enable us to write a cross assembler is that we have a table in which each valid assembler instruction in the source language is mapped into a certain set of instructions in the target machine language. No doubt such an assignment is likely to be complicated by machine differences, such as different numbers of registers, or different addressing schemes. Yet, the same design principles applied to ASSMBLR could be extended to such a case in order to contain the explosion of complexity.

APPENDIX A

Implemented Machine Instructions

Instruction Name	Mnemonic	Type
Add	A	4
Add	AR	1
Add Decimal	AP	8
Add Halfword	AH	4
Add Logical	ALR	1
Add Logical	AL	4
AND	NR	1
AND	N	4
AND	NI	7
AND	NC	9
Branch and Link	BALR	1
Branch and Link	BAL	4
Branch on Condition	BCR	1
Branch on Condition	BC	4
Branch on Count	BCTR	1
Branch on Count	BCT	4
Branch on Index High	BXH	5
Branch on Index Low or Equal	BXLE	5
Compare	CR	1
Compare	C	4
Compare and Swap	CS	5
Compare Decimal	CP	8
Compare Double and Swap	CDS	5
Compare Halfword	CH	4
Compare Logical	CLR	1
Compare Logical	CL	4
Compare Logical	CLC	9
Compare Logical	CLI	7
Compare Logical Characters Under Mask	CLM	5
Compare Logical Long	CLCL	1
Convert to Binary	CVB	4
Convert to Decimal	CVD	4
Divide	DR	1
Divide	D	4
Divide Decimal	DP	8
Edit	ED	9
Edit and Mark	EDMK	9
Exclusive OR	XR	1
Exclusive OR	X	4
Exclusive OR	XI	7
Exclusive OR	XC	9
Execute	EX	4

Insert Character	IC	4
Insert Characters under Mask	ICM	5
Insert Storage Key	ISK	1
Load	LR	1
Load	L	4
Load Address	LA	4
Load and Test	LTR	1
Load Complement	LCR	1
Load Control	LCTL	5
Load Halfword	LH	4
Load Multiple	LM	5
Load Negative	LNR	1
Load Positive	LPR	1
Load Real Address	LRA	4
Monitor Call	MC	7
Move	MVI	7
Move	MVC	9
Move Long	MVCL	1
Move Numerics	MVN	9
Move with Offset	MVO	8
Move Zones	MVZ	9
Multiply	MR	1
Multiply	M	4
Multiply Decimal	MP	8
Multiply Halfword	MH	4
OR	OR	1
OR	O	4
OR	OI	7
OR	OC	9
Pack	PACK	8
Read Direct	RDD	7
Set Program Mask	SPM	2
Set Storage Key	SSK	1
Shift and Round Decimal	SRP	8
Shift Left Double	SLDA	6
Shift Left Double Logical	SLDL	6
Shift Left Single	SLA	6
Shift Left Single Logical	SLL	6
Shift Right Double	SRDA	6
Shift Right Double Logical	SRDL	6
Shift Right Single	SRA	6
Shift Right Single Logical	SRL	6
Signal Processor	SIGP	5
Store	ST	4
Store Character	STC	4
Store Characters under Mask	STCM	5
Store Control	STCTL	5
Store Halfword	STH	4
Store Multiple	STM	5

Store then AND System Mask	STNSM	7
Store Then OR System Mask	STOSM	7
Subtract	SR	1
Subtract	S	4
Subtract Decimal	SP	8
Subtract Halfword	SH	4
Subtract Logical	SLR	1
Subtract Logical	SL	4
Supervisor Call	SVC	3
Test Under Mask	TM	7
Translate	TR	9
Translate and Test	TRT	9
Unpack	UNPK	8
Write Direct	WRD	7
Zero and Add Decimal	ZAP	8

APPENDIX B

B.1 The Main Program

```
ASSMBLR: PROC OPTIONS(MAIN);


/***********************************************************
* DATABASE DECLARATIONS                                    *
***********************************************************/

DECLARE

    POT(8) CHAR(5) VARYING STATIC EXTERNAL INIT(
    'DC','DS','LTORG','EQU','CSECT','USING','DROP','END'),

    1 EOT(32) STATIC EXTERNAL,
      2 NAME CHAR(5) VARYING INIT(
        'B','BE','BER','BH','BHR',
        'BL','BLR','BM','BMR','BNE',
        'BNER','BNH','BNHR','BNL','BNLR',
        'BNM','BNMR','BNO','BNOR','BNP',
        'BNPR','BNZ','BNZR','BO','BOR',
        'BP','BPR','BR','BZ','BZR',
        'NOP','NOPR'),
      2 TYPE BIT(1) ALIGNED INIT(
        '1'B,'1'B,'Ø'B,'1'B,'Ø'B,
        '1'B,'Ø'B,'1'B,'Ø'B,'1'B,
        'Ø'B,'1'B,'Ø'B,'1'B,'Ø'B,
        '1'B,'Ø'B,'1'B,'Ø'B,'1'B,
        'Ø'B,'1'B,'Ø'B,'1'B,'Ø'B,
        '1'B,'Ø'B,'Ø'B,'1'B,'Ø'B,
        '1'B,'Ø'B),
      2 OP1 BIT(4) INIT(
        '1111'B,'1ØØØ'B,'1ØØØ'B,'ØØ1Ø'B,'ØØ1Ø'B,
        'Ø1ØØ'B,'Ø1ØØ'B,'Ø1ØØ'B,'Ø1ØØ'B,'Ø111'B,
        'Ø111'B,'11Ø1'B,'11Ø1'B,'1Ø11'B,'1Ø11'B,
        '1Ø11'B,'1Ø11'B,'111Ø'B,'111Ø'B,'11Ø1'B,
        '11Ø1'B,'Ø111'B,'Ø111'B,'ØØØ1'B,'ØØØ1'B,
        'ØØ1Ø'B,'ØØ1Ø'B,'1111'B,'1ØØØ'B,'1ØØØ'B,
        'ØØØØ'B,'ØØØØ'B);

DCL
    1 MOT(131) STATIC EXTERNAL,
      2 NAME CHAR(5) INIT(
        'A','AH','AL','ALR','AP',
        'AR','BAL','BALR','BC','BCR',
        'BCT','BCTR','BXH','BXLE','C',
        'CDS','CH','CL','CLC','CLCL',
        'CLI','CLM','CLR','CLRIO','CP',
    'CR','CS','CVB','CVD','D',
    'DP','DR','ED','EDMK','EX',
    'HDV','HIO','IC','ICM','IPK',
    'ISK','L','LA','LCR','LCTL',
    'LH','LM','LNR','LPR','LPSW',
    'LR','LRA','LTR','M','MC',
    'MH','MP','MR','MVC','MVCL',
    'MVI','MVN','MVO','MVZ','N',
    'NC','NI','NR','O','OC',
    'OI','OR','PACK','PTLB','RDD',
    'RRB','S','SCK','SCKC','SH',
    'SIGP','SIO','SIOF','SL','SLA',
    'SLDA','SLDL','SLL','SLR','SP',
    'SPKA','SPM','SPT','SPX','SR',
    'SRA','SRDA','SRDL','SRL','SRP',
    'SSK','SSM','ST','STAP','STC',
    'STCK','STCKC','STCM','STCTL','STH',
    'STIDC','STIDP','STM','STNSM','STOSM',
    'STPT','STPX','SVC','TCH','TIO',
    'TM','TR','TRT','TS','UNPK',
    'WRD','X','XC','XI','XR',
    'ZAP'),
  2 TYPE FIXED BIN(15) INIT(
    4,4,4,1,8,
    1,4,1,4,1,
    4,1,5,5,4,
    5,4,4,9,1,
    7,5,1,1Ø,8,
    1,5,4,4,4,
    8,1,9,9,4,
    1Ø,1Ø,4,5,1Ø,
    1,4,4,1,5,
    4,5,1,1,1Ø,
    1,4,1,4,7,
    4,8,1,9,1,
    7,9,8,9,4,
    9,7,1,4,9,
    7,1,8,1Ø,7,
    1Ø,4,1Ø,1Ø,4,
    5,1Ø,1Ø,4,6,
    6,6,6,1,8,
    1Ø,2,1Ø,1Ø,1,
    6,6,6,6,8,
    1,1Ø,4,1Ø,4,
    1Ø,1Ø,5,5,4,
    1Ø,1Ø,5,7,7,
    1Ø,1Ø,3,1Ø,1Ø,
    7,9,9,1Ø,8,
    7,4,9,7,1,
    8),
2 BIN_CODE BIT(8) INIT(
  '5A'B4,'4A'B4,'5E'B4,'1E'B4,'FA'B4,
  '1A'B4,'45'B4,'Ø5'B4,'47'B4,'Ø7'B4,
  '46'B4,'Ø6'B4,'86'B4,'87'B4,'59'B4,
  'BB'B4,'49'B4,'55'B4,'D5'B4,'ØF'B4,
  '95'B4,'BD'B4,'15'B4,'ØØ'B4,'F9'B4,
  '19'B4,'BA'B4,'4F'B4,'4E'B4,'5D'B4,
  'FD'B4,'1D'B4,'DE'B4,'DF'B4,'44'B4,
  'ØØ'B4,'ØØ'B4,'43'B4,'BF'B4,'ØØ'B4,
  'Ø9'B4,'58'B4,'41'B4,'13'B4,'B7'B4,
  '48'B4,'98'B4,'11'B4,'1Ø'B4,'ØØ'B4,
  '18'B4,'B1'B4,'12'B4,'5C'B4,'AF'B4,
  '4C'B4,'FC'B4,'1C'B4,'D2'B4,'ØE'B4,
  '92'B4,'D1'B4,'F1'B4,'D3'B4,'54'B4,
  'D4'B4,'94'B4,'14'B4,'56'B4,'D6'B4,
  '96'B4,'16'B4,'F2'B4,'ØØ'B4,'85'B4,
  'ØØ'B4,'5B'B4,'ØØ'B4,'ØØ'B4,'4B'B4,
  'AE'B4,'ØØ'B4,'ØØ'B4,'5F'B4,'8B'B4,
  '8F'B4,'8D'B4,'89'B4,'1F'B4,'FB'B4,
  'ØØ'B4,'Ø4'B4,'ØØ'B4,'ØØ'B4,'1B'B4,
  '8A'B4,'8E'B4,'8C'B4,'88'B4,'FØ'B4,
  'Ø8'B4,'ØØ'B4,'5Ø'B4,'ØØ'B4,'42'B4,
  'ØØ'B4,'ØØ'B4,'BE'B4,'B6'B4,'4Ø'B4,
  'ØØ'B4,'ØØ'B4,'9Ø'B4,'AC'B4,'AD'B4,
  'ØØ'B4,'ØØ'B4,'ØA'B4,'ØØ'B4,'ØØ'B4,
  '91'B4,'DC'B4,'DD'B4,'ØØ'B4,'F3'B4,
  '84'B4,'57'B4,'D7'B4,'97'B4,'17'B4,
  'F8'B4),
2 MLEN FIXED BIN(15) INIT(
  4,4,4,2,6,
  2,4,2,4,2,
  4,2,4,4,4,
  4,4,4,6,2,
  4,4,2,4,6,
  2,4,4,4,4,
  6,2,6,6,4,
  4,4,4,4,4,
  2,4,4,2,4,
  4,4,2,2,4,
  2,4,2,4,4,
  4,6,2,6,2,
  4,6,6,6,4,
  6,4,2,4,6,
  4,2,6,4,4,
  4,4,4,4,4,
  4,4,4,4,4,
  4,4,4,2,6,
  4,2,4,4,2,
  4,4,4,4,6,
  2,4,4,4,4,
        4,4,4,4,4,
        4,4,4,4,4,
        4,4,2,4,4,
        4,6,6,4,6,
        4,4,6,4,2,
        6);

DCL
    1 ST(1ØØ),
      2 SYMBOL CHAR(8),
      2 VALUE FIXED BIN(31),
      2 RELOC BIT(1),
      2 LEN FIXED BIN(15),
      2 DEFN FIXED BIN(31),
    SYMBOL_NO FIXED BIN(15),

    1 LT(2Ø),
      2 LABEL CHAR(1Ø) VARYING,
      2 VALUE FIXED BIN(31),
    LT_TOP FIXED BIN(15),
    LT_BOT FIXED BIN(15);

DCL 1 BT(16) STATIC EXTERNAL,
      2 REGISTER FIXED BIN(15),
      2 VALUE FIXED BIN(31),
    BASE_NO FIXED BIN(15) STATIC EXTERNAL;

DCL LTORG(5) FIXED BIN(15),
    LTORG_C FIXED BIN(15);
/************************************************************
* OTHER DECLARATIONS                                        *
************************************************************/

DCL CVX ENTRY(BIT(*),CHAR(*) VAR,FIXED BIN),
    GET_OPER ENTRY(CHAR(*) VAR,CHAR(*) VAR),
    GET_LTRL ENTRY(CHAR(*) VAR,CHAR(*) VAR),
    BOUND ENTRY(FIXED BIN,FIXED BIN),
    POTGET ENTRY(CHAR(*) VAR) RETURNS(FIXED BIN),
    EOTGET ENTRY(CHAR(*) VAR) RETURNS(FIXED BIN),
    MOTGET ENTRY(CHAR(*) VAR) RETURNS(FIXED BIN),
    DLENGTH ENTRY(CHAR(*) VAR) RETURNS(FIXED BIN),
    NEXTOK ENTRY(CHAR(*) VAR,CHAR(*) VAR),
    NEXTPLUS ENTRY(CHAR(*) VAR,CHAR(*) VAR),
    PARSLIT ENTRY(CHAR(*) VAR,CHAR(*) VAR,CHAR(*) VAR,
        CHAR(*) VAR,CHAR(*) VAR),
    PARSOP ENTRY(CHAR(*) VAR,CHAR(*) VAR,CHAR(*) VAR,
        CHAR(*) VAR),
    REGDROP ENTRY(FIXED BIN(15)),
    BTSTO ENTRY(FIXED BIN(15),FIXED BIN(15)),
    BRGET ENTRY(FIXED BIN(15),FIXED BIN(15))
```

```
        RETURNS(FIXED BIN(4)),
    CVB ENTRY(CHAR(*)) RETURNS(BIT(8)),
    CHARGET ENTRY(CHAR(*)) RETURNS(BIT(8));

DCL INDATA EXTERNAL FILE;

DCL PS BIT(1),

    INPUT(2ØØ) CHAR(8Ø) VARYING,
    CARD_NO FIXED BIN(15),
    CARD_CHAR(8Ø) VARYING,
    LINE_NO FIXED BIN(15),
    LABEL_FIELD CHAR(8) VARYING,
    INST_FIELD CHAR(5) VARYING,
    OP_FIELD CHAR(8Ø) VARYING,
    (OP,OP1,OP2,OP3) CHAR(8Ø) VARYING,
    LIT CHAR(8Ø) VARYING,
    T CHAR(1) VARYING,
    FLAG BIT(1),
    TVAL FIXED BIN(15),
    IND FIXED BIN(15),
    (I,J) FIXED BIN(15),
    LC FIXED BIN(15),
    L FIXED BIN(15),
    RELOC BIT(1),
    (FOUND,FIRST) BIT(1),
    (REGVAL,REGNO) FIXED BIN(15);

DCL LF CHAR(2) VARYING;

/*************************************************************
* OTHER INITIALIZATIONS                                      *
*************************************************************/

ON ENDFILE(INDATA)
    BEGIN;
        CARD_NO = I - 1;
        I = 2ØØ;
    END;

    SYMBOL_NO = Ø;
    LT_TOP = Ø;
    LT_BOT = Ø;
    BASE_NO = Ø;

    PS = 'Ø'B;

/*************************************************************
* PASS 1                                                     *
*************************************************************/
```

```
DO I = 1 TO 200;
    GET FILE(INDATA) EDIT(INPUT(I))(A);
END;

LC = 0;
DO LINE_NO = 1 TO CARD_NO;

    CARD = INPUT(LINE_NO);
    IF SUBSTR(CARD,1,1)='*' THEN GOTO NEXT1;

    CALL SEPARATE_FIELDS;
    I = POTGET(INST_FIELD);
    GOTO PLAB1(I);


PLAB1(0):   /*MACHINE OR EXTENDED INSTRUCTION*/

    I = EOTGET(INST_FIELD);
    IF I^=0 THEN
        IF (EOT(I).TYPE) THEN L=4;
        ELSE L = 2;
    ELSE DO;
        I = MOTGET(INST_FIELD);
        IF I=0 THEN PUT SKIP LIST ('ERROR');
            ELSE L = MOT(I).MLEN;
    END;
    CALL BOUND(LC,2);

    IF LABEL_FIELD ^= '' THEN DO;
        SYMBOL_NO = SYMBOL_NO + 1;
        ST(SYMBOL_NO).SYMBOL = LABEL_FIELD;
        ST(SYMBOL_NO).VALUE = LC;
        ST(SYMBOL_NO).RELOC = '1'B;
        ST(SYMBOL_NO).DEFN = LINE_NO;
    END;

    OP1 = '';
    OP2 = '';
    OP3 = '';

    CALL GET_OPER(OP_FIELD,OP1);
    IF OP_FIELD ^= '' THEN CALL GET_OPER(OP_FIELD,OP2);
    IF OP_FIELD ^= '' THEN CALL GET_OPER(OP_FIELD,OP3);
    IF OP3 ^= '' THEN
        IF SUBSTR(OP3,1,1)='=' THEN DO;
            CALL LTSTO(OP3);
            IF SUBSTR(OP2,1,1)='=' THEN PUT LIST('ERROR');
        END;
        ELSE;
```

```
     ELSE IF OP2^='' THEN
         IF SUBSTR(OP2,1,1)='=' THEN
              CALL LTSTO(OP2);

    LC = LC + L;

    GOTO NEXT1;

PLAB1(1):;    /*DC INSTRUCTION*/
PLAB1(2):     /*DS INSTRUCTION*/

    L = Ø;
    FIRST = '1'B;
    DO WHILE (OP_FIELD ^='');
        CALL GET_LTRL(OP_FIELD,OP);
        FOUND = 'Ø'B;
        DO I = 1 TO LENGTH(OP) WHILE (^FOUND);
            T = SUBSTR(OP,I,1);
            IF INDEX('CFHXBA',T)^=Ø THEN FOUND = '1'B;
        END;
        IF (T='A'|T='F') THEN CALL BOUND(LC,4);
        IF T='H' THEN CALL BOUND(LC,2);
        IF (FIRST & (LABEL_FIELD ^='')) THEN DO;
            SYMBOL_NO = SYMBOL_NO + 1;
            ST(SYMBOL_NO).SYMBOL = LABEL_FIELD;
            ST(SYMBOL_NO).VALUE = LC;
            ST(SYMBOL_NO).RELOC = '1'B;
            ST(SYMBOL_NO).DEFN = LINE_NO;
            FIRST = 'Ø'B;
        END;
        LC = LC + DLENGTH(OP);
    END;

    GOTO NEXT1;

PLAB1(3):     /*LTORG PSEUDO-OP*/

    IF LT_TOP<LT_BOT THEN
    DO I = (LT_TOP + 1) TO LT_BOT;
        LIT = SUBSTR(LT(I).LABEL,2);
        FOUND = 'Ø'B;
        DO J = 1 TO LENGTH(LIT) WHILE (^FOUND);
            T = SUBSTR(LIT,J,1);
            IF INDEX('CFHXBA',T) ^=Ø THEN FOUND = '1'B;
        END;
        IF (T='A'|T='F') THEN CALL BOUND(LC,4);
        IF T='H' THEN CALL BOUND(LC,2);
        LT(I).VALUE = LC;
        LC = LC + DLENGTH(LIT);
    END;
    LT_TOP = LT_BOT;
    LTORG_C = LTORG_C + 1;
    LTORG(LTORG_C) = LT_TOP;
    GOTO NEXT1;

PLAB1(4):     /*EQU PSEUDO-OP*/

    TVAL = EVAL(OP_FIELD,RELOC,FLAG);
    I = STGET(LABEL_FIELD);
    IF I=Ø THEN DO;
        SYMBOL_NO = SYMBOL_NO + 1;
        I = SYMBOL_NO;
        ST(I).SYMBOL = LABEL_FIELD;
        ST(I).VALUE = TVAL;
        ST(I).RELOC = RELOC;
        ST(I).DEFN = LINE_NO;
    END;
        ELSE PUT SKIP LIST('LABEL DUPLICATED:',LABEL_FIELD);

    GOTO NEXT1;

PLAB1(5):;    /*CSECT PSEUDO-OP*/
PLAB1(6):;    /*USING PSEUDO-OP*/
PLAB1(7):     /*DROP PSEUDO-OP*/
    GOTO NEXT1;

PLAB1(8):     /*END PSEUDO-OP*/

    IF LT_TOP<LT_BOT THEN
    DO I = (LT_TOP + 1) TO LT_BOT;
        LIT = SUBSTR(LT(I).LABEL,2);
        FOUND = 'Ø'B;
        DO J = 1 TO LENGTH(LIT) WHILE (^FOUND);
            T = SUBSTR(LIT,J,1);
            IF INDEX('CFHXBA',T) ^=Ø THEN FOUND = '1'B;
        END;
        IF (T='A'|T='F') THEN CALL BOUND(LC,4);
        IF T='H' THEN CALL BOUND(LC,2);
        LT(I).VALUE = LC;
        LC = LC + DLENGTH(LIT);
    END;
    LT_TOP = LT_BOT;
    LTORG_C = LTORG_C + 1;
    LTORG(LTORG_C) = LT_TOP;

NEXT1:
END;

/**************************************************************
* PASS 2                                                      *
```

```
*************************************************************/

PUT SKIP EDIT('LOC','OBJECT CODE','STMT','SOURCE STATEMENT')
             (X(2),A,X(2),A,X(8),A,X(3),A);
PUT SKIP;

LTORG_C = 1;
LT_TOP = Ø;
LT_BOT = LTORG(1);
LT_TOP = Ø;
LC = Ø;
DO LINE_NO = 1 TO CARD_NO;

    CARD = INPUT(LINE_NO);
    IF SUBSTR(CARD,1,1)='*' THEN DO;
        PUT SKIP EDIT(LINE_NO,CARD)(X(26),F(4),X(1),A);
        GOTO NEXT2;
    END;

    CALL SEPARATE_FIELDS;
    I = POTGET(INST_FIELD);
    GOTO PLAB2(I);

PLAB2(Ø):    /*MACHINE OR EXTENDED INSTRUCTION*/

    I = EOTGET(INST_FIELD);
    IF I^=Ø THEN DO;
        IF (EOT(I).TYPE) THEN L=4;
            ELSE L = 2;
        CALL BOUND(LC,2);
        CALL EOTGEN(I,OP_FIELD);
    END;
    ELSE DO;
        I = MOTGET(INST_FIELD);
        L = MOT(I).MLEN;
        CALL BOUND(LC,2);
        CALL MOTGEN(I,OP_FIELD);
    END;


    LC = LC + L;

    GOTO NEXT2;

PLAB2(1):;    /*DC INSTRUCTION*/
PLAB2(2):     /*DS INSTRUCTION*/

    IND = I;
    L = Ø;
    FIRST = '1'B;
    DO WHILE (OP_FIELD ^='');
        CALL GET_LTRL(OP_FIELD,OP);
        FOUND = 'Ø'B;
        DO I = 1 TO LENGTH(OP) WHILE (^FOUND);
            T = SUBSTR(OP,I,1);
            IF INDEX('CFHXBA',T)^=Ø THEN FOUND = '1'B;
        END;
        IF (T='A'|T='F') THEN CALL BOUND(LC,4);
        IF T='H' THEN CALL BOUND(LC,2);
        CALL LITGEN(OP,'Ø'B);
        LC = LC + DLENGTH(OP);
    END;

    GOTO NEXT2;

PLAB2(3):     /*LTORG PSEUDO-OP*/

    PUT SKIP EDIT(LINE_NO,CARD)(X(26),F(4),X(1),A);
    IF LT_TOP<LT_BOT THEN
    DO I = (LT_TOP + 1) TO LT_BOT;
        LIT = SUBSTR(LT(I).LABEL,2);
        FOUND = 'Ø'B;
        DO J = 1 TO LENGTH(LIT) WHILE (^FOUND);
            T = SUBSTR(LIT,J,1);
            IF INDEX('CFHXBA',T) ^=Ø THEN FOUND = '1'B;
        END;
        IF (T='A'|T='F') THEN CALL BOUND(LC,4);
        IF T='H' THEN CALL BOUND(LC,2);
        CALL LITGEN(LIT,'1'B);
        LC = LC + DLENGTH(LIT);
    END;
    LT_TOP = LT_BOT;
    LTORG_C = LTORG_C + 1;
    LT_BOT = LTORG(LTORG_C);
    GOTO NEXT2;

PLAB2(4):;    /*EQU PSEUDO-OP*/
PLAB2(5):     /*CSECT PSEUDO-OP*/
    PUT SKIP EDIT(LINE_NO,CARD)(X(26),F(4),X(1),A);
    GOTO NEXT2;
PLAB2(6):    /*USING PSEUDO-OP*/
    CALL GET_OPER(OP_FIELD,OP1);
    CALL GET_OPER(OP_FIELD,OP2);
    REGVAL = EVAL(OP1,RELOC,FLAG);
    REGNO = EVAL(OP2,RELOC,FLAG);
    CALL BTSTO(REGNO,REGVAL);
    PUT SKIP EDIT(LINE_NO,CARD)(X(26),F(4),X(1),A);
    GOTO NEXT2;
PLAB2(7):     /*DROP PSEUDO-OP*/
    CALL GET_OPER(OP_FIELD,OP1);
    I = EVAL(OP1,RELOC,FLAG);
    CALL REGDROP(I);
    PUT SKIP EDIT(LINE_NO,CARD)(X(26),F(4),X(1),A);
    GOTO NEXT2;

PLAB2(8):     /*END PSEUDO-OP*/

    PUT SKIP EDIT(LINE_NO,CARD)(X(26),F(4),X(1),A);
    IF LT_TOP<LT_BOT THEN
    DO I = (LT_TOP + 1) TO LT_BOT;
        LIT = SUBSTR(LT(I).LABEL,2);
        FOUND = 'Ø'B;
        DO J = 1 TO LENGTH(LIT) WHILE (^FOUND);
            T = SUBSTR(LIT,J,1);
            IF INDEX('CFHXBA',T) ^=Ø THEN FOUND = '1'B;
        END;
        IF (T='A'|T='F') THEN CALL BOUND(LC,4);
        IF T='H' THEN CALL BOUND(LC,2);
        CALL LITGEN(LIT,'1'B);
        LC = LC + DLENGTH(LIT);
    END;
    LT_TOP = LT_BOT;

NEXT2:
END;


/************************************************************
* SEPARATE_FIELDS SEPARATE THE CURRENT INPUT IN 'CARD' INTO *
* THREE FIELDS: 'LABEL_FIELD', 'INST_FIELD', 'OP_FIELD'     *
************************************************************/

SEPARATE_FIELDS: PROCEDURE;

DCL I FIXED BIN(15),
    TCARD CHAR(8Ø) VARYING;

    LABEL_FIELD = '';
    INST_FIELD = '';
    OP_FIELD = '';
    TCARD = CARD;
    I = INDEX(TCARD,' ');
    IF I>9 THEN PUT SKIP LIST ('LABEL TOO LONG');
    IF I>1 THEN DO;
        LABEL_FIELD = SUBSTR(TCARD,1,I-1);
        TCARD = SUBSTR(TCARD,I);
    END;
    DO WHILE (SUBSTR(TCARD,1,1) = ' ');
        TCARD = SUBSTR(TCARD,2);
    END;
    I = INDEX(TCARD,' ');
    IF I>6 THEN PUT SKIP LIST ('INST TOO LONG');
    IF I=Ø THEN DO;
        INST_FIELD = TCARD;
        TCARD = '';
    END;
    ELSE DO;
        INST_FIELD= SUBSTR(TCARD,1,I-1);
        TCARD = SUBSTR(TCARD,I);
    END;

    DO WHILE (INDEX(TCARD,' ')=1);
        TCARD = SUBSTR(TCARD,2);
    END;
    I = INDEX(TCARD,' ');
    IF I=Ø THEN OP_FIELD = TCARD;
        ELSE OP_FIELD = SUBSTR(TCARD,1,I-1);

END SEPARATE_FIELDS;

/*************************************************************
* LTGET GETS THE INDEX OF CHAR STRING 'X' IN THE LITERAL     *
* TABLE.                                                     *
*************************************************************/


LTGET: PROCEDURE(X) RETURNS(FIXED BIN(15));

DCL X CHAR(*) VARYING,
    I FIXED BIN(15),
    FOUND BIT(1);

    FOUND = 'Ø'B;

    IF LT_BOT=Ø THEN RETURN(Ø);
    DO I = (LT_TOP+1) TO LT_BOT;
        IF LT(I).LABEL = X THEN RETURN(I);
    END;
    RETURN(Ø);

END LTGET;

/*************************************************************
* LTSTO STORES A NEW LITERAL IN THE LITERAL TABLE IF IT      *
* DOES NOT ALREADY EXIST WITHIN THE CURRENT SCOPE            *
* (I.E. SINCE THE PREVIOUS LTORG INSTRUCTION).               *
*************************************************************/
```

```
LTSTO: PROCEDURE(X);

DCL X CHAR(*) VARYING,
    I FIXED BIN(15);

    I = LTGET(X);
    IF I = Ø THEN DO;
        LT_BOT = LT_BOT + 1;
        LT(LT_BOT).LABEL = X;

    END;
    RETURN;

END LTSTO;

/**************************************************************
* STGET GETS THE INDEX OF ITS ARGUMENT IN THE SYMBOL TABLE. *
* IF THE ARGUMENT IS NOT IN THE TABLE, IT RETURNS A Ø.      *
**************************************************************/

STGET: PROCEDURE(ARG) RETURNS(FIXED BIN(15));

DCL ARG CHAR(*) VARYING,
    PLACE FIXED BIN(15),
    I FIXED BIN(15),
    FOUND BIT(1);

    FOUND = 'Ø'B;
    DO I = 1 TO SYMBOL_NO WHILE(^FOUND);
        IF ARG=ST(I).SYMBOL THEN DO;
            FOUND = '1'B;
            PLACE = I;
        END;
    END;
    IF FOUND THEN RETURN(PLACE);
        ELSE RETURN(Ø);
END STGET;

/**************************************************************
* VAL CALCULATES THE VALUE OF A NUMBER OR VARIABLE. IF THE  *
* VARIABLE IS ILLEGAL IT SETS FLAG TO '1'B.                 *
**************************************************************/

VAL: PROCEDURE(ARG,RES,RELOC,FLAG);

DCL ARG CHAR(*) VARYING,
    RES FIXED BIN(15),
    RES8 BIT(8),
    RELOC BIT(1),
    FLAG BIT(1),
    TARG CHAR(8) VARYING,
    T CHAR(1),
    TEMP CHAR(2),
    I FIXED BIN(15);

    FLAG = 'Ø'B;
    RELOC = 'Ø'B;
    TARG = ARG;
    T = SUBSTR(TARG,1,1);
    IF INDEX('Ø123456789',T)>Ø THEN DO;
        RES = TARG;
        RETURN;
    END;
    IF TARG='*' THEN DO;
        RES = LC;
        RELOC = '1'B;
        RETURN;
    END;
    IF T='X' THEN
        IF INDEX(TARG,'''')>Ø THEN DO;
            TEMP = SUBSTR(TARG,3,2);
            RES8 = CVB(TEMP);
            RES = RES8;
            RETURN;
        END;
    IF T='B' THEN
        IF INDEX(TARG,'''')>Ø THEN DO;
            RES8 = SUBSTR(TARG,3,8);
            RES = RES8;
            RETURN;
        END;
    I = STGET(TARG);
    IF I=Ø THEN DO;
        RES = Ø;
        FLAG = '1'B;
        RETURN;
    END;
    RES = ST(I).VALUE;
    RELOC = ST(I).RELOC;
    RETURN;
END VAL;

/*************************************************************
* MULTDIV IS CALLED TO EVALUATE ONLY EXPRESSIONS WITH       *
* NO '+' OR '-' OPERATORS. IT RETURNS THE COMPUTED VALUE    *
* AND ITS RELOCATABILITY.  IF THE EXPRESSION IS             *
* ILLEGAL, THE FLAG IS SET TO '1'B AND Ø IS RETURNED.       *
*************************************************************/
```

```
MULTDIV: PROCEDURE(ARG,RELOC,FLAG) RETURNS(FIXED BIN(15));

DCL ARG CHAR(*) VARYING,
    (RELOC,RELOC2) BIT(1),
    FLAG BIT(1),
    (RES,RES2) FIXED BIN(15),
    (OP1,OP2) CHAR(8Ø) VARYING,
    TARG CHAR(8Ø) VARYING,
    T CHAR(1);

    TARG = ARG;
    RELOC = 'Ø'B;
    FLAG = 'Ø'B;
    CALL NEXTOK(TARG,OP1);
    CALL VAL(OP1,RES,RELOC,FLAG);
    IF FLAG THEN RETURN(Ø);
    DO WHILE(TARG^='');
        T = SUBSTR(TARG,1,1);
        TARG = SUBSTR(TARG,2);
        CALL NEXTOK(TARG,OP2);
        CALL VAL(OP2,RES2,RELOC2,FLAG);
        FLAG = FLAG | RELOC | RELOC2;
        IF FLAG THEN RETURN(Ø);
        IF T='*' THEN RES = RES* RES2;
            ELSE RES = DIVIDE(RES,RES2,15);
    END;
    RETURN(RES);
END MULTDIV;

/************************************************************
* EVAL TAKES AN EXPRESSION AND RETURNS ITS VALUE AND        *
* RELOCATABILITY. IF THE EXPRESSION IS NOT VALID, A Ø IS    *
* RETURNED AND FLAG IS SET TO '1'B.                         *
************************************************************/

EVAL: PROCEDURE(ARG,RELOC,FLAG) RETURNS(FIXED BIN(15));

DCL ARG CHAR(*) VARYING,
    (RELOC,RELOC2) BIT(1),
    FLAG BIT(1),
    (RES,RES2) FIXED BIN(15),
    RELCOUNT FIXED BIN(15),
    (OP1,OP2) CHAR(8Ø) VARYING,
    T CHAR(1),
    I FIXED BIN(15),
    TARG CHAR(8Ø) VARYING,
    FOUND BIT(1);

    RELOC = 'Ø'B;
    FLAG = 'Ø'B;
    TARG = ARG;
    CALL NEXTPLUS(TARG,OP1);
    RES = MULTDIV(OP1,RELOC,FLAG);
    RELCOUNT = BINARY(RELOC);
    IF FLAG THEN RETURN(Ø);
    DO WHILE(LENGTH(TARG)>Ø);
        T = SUBSTR(TARG,1,1);
        TARG = SUBSTR(TARG,2);
        CALL NEXTPLUS(TARG,OP2);
        RES2 = MULTDIV(OP2,RELOC2,FLAG);
        IF FLAG THEN RETURN(Ø);
        IF T='+' THEN DO;
            RES = RES + RES2;
            RELCOUNT = RELCOUNT + BINARY(RELOC2);
        END;
        ELSE DO;
            RES = RES - RES2;
            RELCOUNT = RELCOUNT - BINARY(RELOC2);
        END;

    END;
    IF (RELCOUNT=1) THEN RELOC = '1'B;
        ELSE IF (RELCOUNT=Ø) THEN RELOC = 'Ø'B;
            ELSE FLAG = '1'B;
    IF FLAG THEN RETURN(Ø);
        ELSE RETURN(RES);
END EVAL;

/************************************************************
* EOTGEN GENERATES THE OUTPUT WHEN THE OPERAND IS AN        *
* EXTENDED INSTRUCTION.                                     *
************************************************************/

EOTGEN: PROCEDURE(I,OP_FIELD);

DCL I FIXED BIN(15),
    OP_FIELD CHAR(*) VARYING,
    OP_CODE BIT(8),
    R1_BIT(4),
    (DR2,DX2,DB2) FIXED BIN(4),
    DD2 FIXED BIN(12),
    (R2,X2,B2) BIT(4),
    D2 BIT(12),
    DLC FIXED BIN(24),
    XLC BIT(24),
    (A,B,C) CHAR(2Ø) VARYING,
    (RELOC,FLAG) BIT(1),
    BRVAL FIXED BIN(15),
    TD2 FIXED BIN(15);
    DLC = LC;
    XLC = DLC;
    TD2 = EOT(I).TYPE;
    GOTO TYP(TD2);

TYP(Ø):
    OP_CODE = 'Ø7'B4;
    R1 = EOT(I).OP1;
    DR2 = EVAL(OP_FIELD,RELOC,FLAG);
    R2 = DR2;
    PUT SKIP EDIT(XLC,OP_CODE,R1,R2)
                 (B4(6),X(1),B4(2),B4(1),B4(1));
    GOTO NEXT;

TYP(1):
    OP_CODE = '47'B4;
    R1 = EOT(I).OP1;
    CALL PARSOP(OP_FIELD,A,B,C);
    IF B^='' THEN DX2 = EVAL(B,RELOC,FLAG);
        ELSE DX2 = Ø;
    X2 = DX2;
    CALL BRSET(A,C,B2,D2,FLAG);
    PUT SKIP EDIT(XLC,OP_CODE,R1,X2,B2,D2)
        (B4(6),X(1),B4(2),B4(1),B4(1),X(1),B4(1),B4(3));

NEXT:
    PUT EDIT(LINE_NO,CARD)(COL(27),F(4),X(1),A);
END EOTGEN;


/************************************************************
* MOTGEN GENERATES THE OUTPUT WHEN THE INSTRUCTION IS       *
* A MACHINE INSTRUCTION.                                    *
************************************************************/

MOTGEN: PROCEDURE(I,OP_FIELD);

DCL I FIXED BIN(15),
    OP_FIELD CHAR(*) VARYING,
    OP_CODE BIT(8),
    (R1,R2,R3,X2,B1,B2) BIT(4),
    (DR1,DR2,DR3,DX2,DB1,DB2) FIXED BIN(4),
    (D1,D2) BIT(12),
    (DD1,DD2) FIXED BIN(12),
    BYTE BIT(8),
    DBYTE FIXED BIN(8),
    (L1,L2) BIT(4),
    (DL1,DL2) FIXED BIN(4),
    DLC FIXED BIN(24),
    XLC BIT(24),
    (OP1,OP2,OP3) CHAR(32) VARYING,
    (A,B,C) CHAR(2Ø) VARYING,
    (RELOC,FLAG) BIT(1),
    TD2 FIXED BIN(15),
    J FIXED BIN(15),
    BRVAL FIXED BIN(15);

    DLC = LC;
    XLC = DLC;
    OP_CODE = MOT(I).BIN_CODE;
    J = MOT(I).TYPE;
    GOTO TYP(J);

TYP(1):
    CALL GET_OPER(OP_FIELD,OP1);
    CALL GET_OPER(OP_FIELD,OP2);
    DR1 = EVAL(OP1,RELOC,FLAG);
    R1 = DR1;
    DR2 = EVAL(OP2,RELOC,FLAG);
    R2 = DR2;
    PUT SKIP EDIT(XLC,OP_CODE,R1,R2)
        (B4(6),X(1),B4(2),B4(1),B4(1));
    GOTO NEXT;

TYP(2):
    OP1 = OP_FIELD;
    DR1 = EVAL(OP1,RELOC,FLAG);
    R1 = DR1;
    DR2 = Ø;
    R2 = DR2;
    PUT SKIP EDIT(XLC,OP_CODE,R1,R2)
                 (B4(6),X(1),B4(2),B4(1),B4(1));
    GOTO NEXT;

TYP(3):
    OP1 = OP_FIELD;
    DBYTE = EVAL(OP1,RELOC,FLAG);
    BYTE = DBYTE;
    PUT SKIP EDIT(XLC,OP_CODE,BYTE)
                 (B4(6),X(1),B4(2),B4(2));
    GOTO NEXT;

TYP(4):
    CALL GET_OPER(OP_FIELD,OP1);
    CALL GET_OPER(OP_FIELD,OP2);

    DR1 = EVAL(OP1,RELOC,FLAG);
    R1 = DR1;

    IF SUBSTR(OP2,1,1)='=' THEN DO;
```

```
        J = LTGET(OP2);
        TD2 = LT(J).VALUE;
        DB2 = BRGET(TD2,BRVAL);
        IF DB2=Ø THEN DD2 = TD2;
            ELSE DD2 = TD2 - BRVAL;
        DX2 = Ø;
        B2 = DB2;
        D2 = DD2;
        X2 = DX2;
    END;
    ELSE DO;
        CALL PARSOP(OP2,A,B,C);
        IF B^='' THEN DX2 = EVAL(B,RELOC,FLAG);
            ELSE DX2 = Ø;
        X2 = DX2;
        CALL BRSET(A,C,B2,D2,FLAG);
    END;
    PUT SKIP EDIT(XLC,OP_CODE,R1,X2,B2,D2)
        (B4(6),X(1),B4(2),B4(1),B4(1),X(1),B4(1),B4(3));
    GOTO NEXT;
TYP(5):
    CALL GET_OPER(OP_FIELD,OP1);
    CALL GET_OPER(OP_FIELD,OP2);
    CALL GET_OPER(OP_FIELD,OP3);

    DR1 = EVAL(OP1,RELOC,FLAG);
    R1 = DR1;

    DR3 = EVAL(OP2,RELOC,FLAG);
    R3 = DR3;


    IF SUBSTR(OP3,1,1)='=' THEN DO;
        J = LTGET(OP3);
        TD2 = LT(J).VALUE;
        DB2 = BRGET(TD2,BRVAL);
        IF DB2=Ø THEN DD2 = TD2;
            ELSE DD2 = TD2 - BRVAL;
        B2 = DB2;
        D2 = DD2;
    END;
    ELSE DO;
        CALL PARSOP(OP3,A,B,C);
        CALL BRSET(A,B,B2,D2,FLAG);
    END;
    PUT SKIP EDIT(XLC,OP_CODE,R1,R3,B2,D2)
        (B4(6),X(1),B4(2),B4(1),B4(1),X(1),B4(1),B4(3));
    GOTO NEXT;

TYP(6):
    CALL GET_OPER(OP_FIELD,OP1);
    CALL GET_OPER(OP_FIELD,OP2);

    DR1 = EVAL(OP1,RELOC,FLAG);
    R1 = DR1;
    DR3 = Ø;
    R3 = DR3;
    DB2 = Ø;
    B2 = DB2;

    TD2 = EVAL(OP2,RELOC,FLAG);
    DD2 = TD2;
    D2 = DD2;
    PUT SKIP EDIT(XLC,OP_CODE,R1,R3,B2,D2)
        (B4(6),X(1),B4(2),B4(1),B4(1),X(1),B4(1),B4(3));
    GOTO NEXT;

TYP(7):
    CALL GET_OPER(OP_FIELD,OP1);
    CALL GET_OPER(OP_FIELD,OP2);
    CALL PARSOP(OP1,A,B,C);
    CALL BRSET(A,B,B1,D1,FLAG);
    DBYTE = EVAL(OP2,RELOC,FLAG);
    BYTE = DBYTE;
    PUT SKIP EDIT(XLC,OP_CODE,BYTE,B1,D1)
        (B4(6),X(1),B4(2),B4(2),X(1),B4(1),B4(3));
    GOTO NEXT;

TYP(8):
    CALL GET_OPER(OP_FIELD,OP1);
    CALL GET_OPER(OP_FIELD,OP2);
    CALL PARSOP(OP1,A,B,C);
    CALL BRSET(A,C,B1,D1,FLAG);
    DL1 = EVAL(B,RELOC,FLAG);
    L1 = DL1;

    CALL PARSOP(OP2,A,B,C);
    CALL BRSET(A,C,B2,D2,FLAG);
    DL2 = EVAL(B,RELOC,FLAG);
    L2 = DL2;

    PUT SKIP EDIT(XLC,OP_CODE,L1,L2,B1,D1,B2,D2)
        (B4(6),X(1),B4(2),B4(1),B4(1),X(1),B4(1),B4(3),
         X(1),B4(1),B4(3));
    GOTO NEXT;

TYP(9):
    CALL GET_OPER(OP_FIELD,OP1);
    CALL GET_OPER(OP_FIELD,OP2);
    CALL PARSOP(OP1,A,B,C);
    CALL BRSET(A,C,B1,D1,FLAG);
    DBYTE = EVAL(B,RELOC,FLAG);
    BYTE = DBYTE;

    IF SUBSTR(OP2,1,1)='=' THEN DO;
        J = LTGET(OP2);
        TD2 = LT(J).VALUE;
        DB2 = BRGET(TD2,BRVAL);
        IF DB2=Ø THEN DD2 = TD2;
            ELSE DD2 = TD2 - BRVAL;
        B2 = DB2;
        D2 = DD2;
    END;
    ELSE DO;
        CALL PARSOP(OP2,A,B,C);
        CALL BRSET(A,B,B2,D2,FLAG);
    END;
    PUT SKIP EDIT(XLC,OP_CODE,BYTE,B1,D1,B2,D2)
        (B4(6),X(1),B4(2),B4(2),X(1),B4(1),B4(3),
         X(1),B4(1),B4(3));
    GOTO NEXT;

TYP(1Ø):
    PUT SKIP EDIT(XLC,'ØØØØ ØØØØ')
                 (B4(6),X(1),A);

NEXT:
    PUT EDIT(LINE_NO,CARD)(COL(27),F(4),X(1),A);
END MOTGEN;

/************************************************************
* LITGEN GENERATES THE OUTPUT WHEN THE INSTRUCTION IS A DC  *
* OR DS INSTRUCTION OR WHEN AN LTORG OR END STATEMENT       *
* RESULTS IN LITERALS BEING GENERATED.                      *
************************************************************/

LITGEN: PROCEDURE(ARG,CODE);

DCL ARG CHAR(*) VARYING,
    TARG CHAR(8Ø) VARYING,
    CODE BIT(1),
    (DUPFAC,MOD,NOMVAL) CHAR(2Ø) VARYING,
    TYP CHAR(1) VARYING,
    (DLEN,VDUPFAC) FIXED BIN(15),
    IND FIXED BIN(15),
    CHARSTR CHAR(2Ø) VARYING,
    DONE BIT(1),
    (I,J) FIXED BIN(15),
    T CHAR(1),
    T2 CHAR(2),
    BYTE BIT(8),
    (LBYTE,LWORD) FIXED BIN(15),
    FVALUE FIXED BIN(31),
    BFVALUE BIT(32),
    HVALUE FIXED BIN(16),
    BHVALUE BIT(16),
    DLC FIXED BIN(24),
    XLC BIT(24),
    (RELOC,FLAG) BIT(1);

    TARG = ARG;
    LBYTE = Ø;
    DONE = 'Ø'B;
    DLC = LC;
    XLC = DLC;
    CALL PARSLIT(TARG,DUPFAC,TYP,MOD,NOMVAL);
    PUT SKIP EDIT(XLC,' ')(B4(6),A);
    IF INST_FIELD='DS' THEN GOTO NEXT;
    IND = INDEX('CFHXBA',TYP);
    IF DUPFAC='' THEN VDUPFAC = 1;
        ELSE VDUPFAC = DUPFAC;

    GOTO TLAB(IND);

TLAB(1):      /*CCCCC*/
    DLEN = LENGTH(NOMVAL)-2;
    CHARSTR = SUBSTR(NOMVAL,2,DLEN);
    IF (MOD^='') THEN DO;
        DLEN = SUBSTR(MOD,2);
        IF DLEN<LENGTH(CHARSTR) THEN
            CHARSTR = SUBSTR(CHARSTR,1,DLEN);
        ELSE DO;
            DO WHILE(LENGTH(CHARSTR)<DLEN);
                CHARSTR = CHARSTR || ' ';
            END;
        END;
    END;
    DONE = 'Ø'B;
    DO I = 1 TO VDUPFAC WHILE(^DONE);
        DO J = 1 TO DLEN WHILE(^DONE);
            T = SUBSTR(CHARSTR,J,1);
            BYTE = CHARGET(T);
            PUT EDIT(BYTE)(B4(2));
            LBYTE = LBYTE + 1;
            IF LBYTE>=8 THEN DONE = '1'B;
        END;
    END;
    GOTO NEXT;

TLAB(2):      /*FFFFF*/
```

```
    LWORD = LENGTH(NOMVAL) - 2;
    FVALUE = SUBSTR(NOMVAL,2,LWORD);
    BFVALUE = UNSPEC(FVALUE);
    DO I = 1 TO VDUPFAC WHILE(^DONE);
        PUT EDIT(BFVALUE)(B4(8));
        LBYTE = LBYTE + 4;
        IF LBYTE>4 THEN DONE = '1'B;
    END;
    GOTO NEXT;

TLAB(3):     /*HHHHH*/
    LWORD = LENGTH(NOMVAL) - 2;
    HVALUE = SUBSTR(NOMVAL,2,LWORD);
    BHVALUE = HVALUE;
    DO I = 1 TO VDUPFAC WHILE(^DONE);
        PUT EDIT(BHVALUE)(B4(4));
        LBYTE = LBYTE + 2;
        IF LBYTE>6 THEN DONE = '1'B;
    END;
    GOTO NEXT;

TLAB(4):     /*XXXXX*/
    DLEN = LENGTH(NOMVAL)-2;
    CHARSTR = SUBSTR(NOMVAL,2,DLEN);
    DLEN = DIVIDE(DLEN,2,15);
    IF (MOD^='') THEN DO;
        DLEN = SUBSTR(MOD,2);
        IF (2*DLEN)<LENGTH(CHARSTR) THEN
            CHARSTR = SUBSTR(CHARSTR,1,2*DLEN);
        ELSE DO;
            DO WHILE(LENGTH(CHARSTR)<DLEN);
                CHARSTR = CHARSTR || 'ØØ';
            END;
        END;
    END;
    DO I = 1 TO VDUPFAC WHILE(^DONE);
        DO J = 1 TO DLEN WHILE (^DONE);
            T2 = SUBSTR(CHARSTR,2*J-1,2);
            BYTE = CVB(T2);
            PUT EDIT (BYTE)(B4(2));
            LBYTE = LBYTE + 1;
            IF LBYTE>=8 THEN DONE = '1'B;
        END;
    END;
    GOTO NEXT;
TLAB(5):     /*BBBBB*/
    DLEN = LENGTH(NOMVAL)-2;
    BYTE = SUBSTR(NOMVAL,2,DLEN);
    DO I = 1 TO VDUPFAC WHILE(^DONE);
        PUT EDIT(BYTE)(B4(2));
        LBYTE = LBYTE + 1;
        IF LBYTE >=8 THEN DONE = '1'B;
    END;
    GOTO NEXT;
TLAB(6):      /*AAAAA*/
    DLEN = LENGTH(NOMVAL)-2;
    CHARSTR = SUBSTR(NOMVAL,2,DLEN);
    FVALUE = EVAL(CHARSTR,RELOC,FLAG);
    BFVALUE = UNSPEC(FVALUE);
    DO I = 1 TO VDUPFAC WHILE(^DONE);
        PUT EDIT(BFVALUE)(B4(8));
        LBYTE = LBYTE + 4;
        IF LBYTE >4 THEN DONE = '1'B;
    END;

NEXT:
    PUT EDIT(' ')(COL(26),A);
    IF CODE='Ø'B THEN
        PUT EDIT (LINE_NO,CARD)(F(4),X(1),A);
    ELSE DO;
        TARG = '=' || TARG;
        PUT EDIT (TARG)(X(2Ø),A);
    END;
END LITGEN;

/***********************************************************
* BRSET GETS A BASE REGISTER IF ANY EXIST.                 *
***********************************************************/

BRSET: PROCEDURE(ARG1,ARG2,BR,DISP,ERROR);

DCL (ARG1,ARG2) CHAR(*) VARYING,
    BR BIT(4),
    DISP BIT(12),
    ERROR BIT(1),
    (RELOC,FLAG) BIT(1),
    DBR FIXED BIN(4),
    DDISP FIXED BIN(12),
    TDISP FIXED BIN(15),
    BRVAL FIXED BIN(15);

    IF ARG2='' THEN DO;
        TDISP = EVAL(ARG1,RELOC,FLAG);
        ERROR = (^RELOC)|FLAG;
        DBR = BRGET(TDISP,BRVAL);
        IF DBR =Ø THEN DDISP = TDISP;
            ELSE DDISP = TDISP - BRVAL;
    END;
    ELSE DO;
        DBR = EVAL(ARG2,RELOC,FLAG);
```

```
            ERROR = RELOC|FLAG;
            DDISP = EVAL(ARG1,RELOC,FLAG);
            ERROR = ERROR|RELOC|FLAG;
        END;
        IF ERROR THEN DO;
            DBR = Ø;
            DDISP = Ø;
        END;
        BR = DBR;
        DISP = DDISP;
    END BRSET;
    END ASSMBLR;
```

B.2 The External Subroutines

```
/*************************************************************
* CVB CONVERTS TWO HEXADECIMAL CHARACTERS INTO A BIT STRING *
* (BYTE).                                                   *
*************************************************************/

CVB: PROCEDURE(ARG) RETURNS(BIT(8));

DCL ARG CHAR(*),
    T(2) CHAR(1),
    NIB(2) BIT(4),
    RES BIT(8),
    (I,J) FIXED BIN(15),
    FOUND BIN(1);

DCL 1 HEX_TABLE(16) STATIC,
      2 TITS BIT(4) ALIGNED INIT(
        'ØØØØ'B,'ØØØ1'B,'ØØ1Ø'B,'ØØ11'B,
        'Ø1ØØ'B,'Ø1Ø1'B,'Ø11Ø'B,'Ø111'B,
        '1ØØØ'B,'1ØØ1'B,'1Ø1Ø'B,'1Ø11'B,
        '11ØØ'B,'11Ø1'B,'111Ø'B,'1111'B),
      2 HEX CHAR(1) INIT(
        'Ø','1','2','3','4','5','6','7',
        '8','9','A','B','C','D','E','F');


    T(1) = SUBSTR(ARG,1,1);
    T(2) = SUBSTR(ARG,2,1);
    NIB(1) = 'Ø'B4;
    NIB(2) = 'Ø'B4;
    DO I = 1 TO 2;
        FOUND = 'Ø'B;
        DO J = 1 TO 16 WHILE(^FOUND);
            IF (T(I)=HEX_TABLE(J).HEX) THEN DO;
                NIB(I) = HEX_TABLE(J).TITS;
                FOUND = '1'B;
            END;
        END;
    END;
    RES = NIB(1) || NIB(2);
    RETURN(RES);
END CVB;

/*************************************************************
* GET_OPER TAKES THE GIVEN INPUT IN 'OP_FIELD' AND RETURNS  *
* THE FIRST OPERAND IN 'RESULT'. 'OP_FIELD' IS TRUNCATED TO *
* EXCLUDE THE FIRST OPERAND (AND THE FOLLOWING COMMA IF     *
* IT EXISTS).                                               *
*************************************************************/
```

```
GET_OPER: PROCEDURE(OP_FIELD,RESULT);

DCL OP_FIELD CHAR(*) VARYING,
    RESULT CHAR(*) VARYING,
    (ICOM,ILP,IRP,IEQ) FIXED BIN(15),
    TEMP CHAR(32) VARYING,
    (TOPF,TRES) CHAR(80) VARYING;

    TOPF = OP_FIELD;
    TRES = '';
    ICOM = INDEX(TOPF,',');
    ILP = INDEX(TOPF,'(');
    IRP  = INDEX(TOPF,')');
    IEQ  = INDEX(TOPF,'=');

    IF ICOM = 0 THEN DO;
        RESULT = TOPF;
        OP_FIELD = '';
        RETURN;
    END;

    IF ((ICOM>IEQ)&(IEQ^=0)) THEN DO;
PUT SKIP LIST('OH NO!');
        CALL GET_LTRL(TOPF,TRES);
        OP_FIELD = TOPF;
        RESULT = TRES;
    END;

    IF (ICOM>ILP)&(ICOM<IRP) THEN DO;
        RESULT = SUBSTR(TOPF,1,IRP);
        IF IRP<LENGTH(TOPF) THEN
             OP_FIELD = SUBSTR(TOPF,IRP+2);
        ELSE OP_FIELD = '';
        RETURN;
    END;

    RESULT = SUBSTR(TOPF,1,ICOM-1);
    OP_FIELD = SUBSTR(TOPF,ICOM+1);

END GET_OPER;

/*************************************************************
* GET_LTRL TAKES GIVEN INPUT IN 'OP_FIELD' AND RETURNS THE   *
* FIRST LITERAL IN 'RESULT'. 'OP_FIELD' IS TRUNCATED TO      *
* EXCLUDE THE LITERAL (AND THE FOLLOWING COMMA IF IT         *
* EXISTS).                                                   *
*************************************************************/

GET_LTRL: PROCEDURE(OP_FIELD,RESULT);
```

```
DCL OP_FIELD CHAR(*) VARYING,
    RESULT CHAR(*) VARYING,
    IND FIXED BIN(15),
    I FIXED BIN(15),
    T CHAR(1),
    FOUND BIT(1),
    (TOPF,TRES) CHAR(80) VARYING;

    I = 1;
    FOUND = '0'B;
    TOPF = OP_FIELD;
    TRES = '';

    DO WHILE (I<=LENGTH(TOPF)&^FOUND);
        T = SUBSTR(TOPF,I,1);
        IF INDEX('CFHXBA',T)>0 THEN FOUND = '1'B;
        I = I + 1;
    END;

    IF (^FOUND) THEN PUT SKIP LIST('ERROR');
    ELSE DO;
        IND = INDEX(TOPF,'''');
        IF ((IND>0)&(T='C')) THEN DO;
            TRES = SUBSTR(TOPF,1,IND);
            TOPF = SUBSTR(TOPF,IND+1);
            IND = INDEX(TOPF,'''');
            TRES = TRES || SUBSTR(TOPF,1,IND);

            IF LENGTH(TOPF)>IND
                THEN TOPF = SUBSTR(TOPF,IND+2);
                ELSE TOPF = '';
        END;
        ELSE DO;
            IND = INDEX(TOPF,',');
            IF IND=0 THEN DO;
                TRES = TOPF;
                TOPF = '';
            END;
            ELSE DO;
                TRES = SUBSTR(TOPF,1,IND-1);
                TOPF = SUBSTR(TOPF,IND+1);
            END;
        END;
    END;
    OP_FIELD = TOPF;
    RESULT = TRES;

END GET_LTRL;

/**************************************************************
```

```
* BOUND TAKES LC AND ALIGNS IT TO THE BOUNDARY SPECIFIED BY *
* COUNT.                                                    *
*************************************************************/

BOUND: PROCEDURE(LC,COUNT);

DCL (LC,COUNT) FIXED BIN(15),
    TEMP FIXED BIN(15),
    (TLC,TCOUNT) FIXED BIN(15),
    DONE BIT(1);

    DONE = 'Ø'B;
    TLC = LC;
    TCOUNT = COUNT;

    DO WHILE (^DONE);
        TEMP = DIVIDE(TLC,TCOUNT,15);
        IF TLC=TEMP*TCOUNT THEN DONE = '1'B;
            ELSE TLC = TLC + 1;
    END;
    LC = TLC;
END BOUND;

/************************************************************
* POTGET GETS THE INDEX OF ITS ARGUMENT IN POT. IF ITS      *
* ARGUMENT IS NOT IN POT IT RETURNS A Ø.                    *
*************************************************************/

POTGET: PROCEDURE(ARG) RETURNS(FIXED BIN(15));

DCL POT(8) CHAR(5) VARYING STATIC EXTERNAL INIT(
      'DC','DS','LTORG','EQU','CSECT','USING','DROP','END');

DCL ARG CHAR(*) VARYING,
    PLACE FIXED BIN(15),
    I FIXED BIN(15),
    FOUND BIT(1);

    FOUND = 'Ø'B;
    PLACE = Ø;

    DO I = 1 TO 8 WHILE (^FOUND);
        IF ARG = POT(I) THEN DO;
            FOUND = '1'B;
            PLACE = I;
        END;
    END;
    RETURN(PLACE);
END POTGET;
```

```
/***********************************************************
* EOTGET GETS THE INDEX OF ITS ARGUMENT IN EOT. IF THE      *
* ARGUMENT IS NOT IN EOT IT RETURNS A Ø.                    *
***********************************************************/

EOTGET: PROCEDURE(ARG) RETURNS(FIXED BIN(15));

DCL 1 EOT(32) STATIC EXTERNAL,
      2 NAME CHAR(5) VARYING INIT(
        'B','BE','BER','BH','BHR',
        'BL','BLR','BM','BMR','BNE',
        'BNER','BNH','BNHR','BNL','BNLR',
        'BNM','BNMR','BNO','BNOR','BNP',
        'BNPR','BNZ','BNZR','BO','BOR',
        'BP','BPR','BR','BZ','BZR',
        'NOP','NOPR'),
      2 TYPE BIT(1) ALIGNED INIT(
        '1'B,'1'B,'Ø'B,'1'B,'Ø'B,
        '1'B,'Ø'B,'1'B,'Ø'B,'1'B,
        'Ø'B,'1'B,'Ø'B,'1'B,'Ø'B,
        '1'B,'Ø'B,'1'B,'Ø'B,'1'B,
        'Ø'B,'1'B,'Ø'B,'1'B,'Ø'B,
        '1'B,'Ø'B,'Ø'B,'1'B,'Ø'B,
        '1'B,'Ø'B),
      2 OP1 BIT(4) INIT(
        '1111'B,'1ØØØ'B,'1ØØØ'B,'ØØ1Ø'B,'ØØ1Ø'B,
        'Ø1ØØ'B,'Ø1ØØ'B,'Ø1ØØ'B,'Ø1ØØ'B,'Ø111'B,
        'Ø111'B,'11Ø1'B,'11Ø1'B,'1Ø11'B,'1Ø11'B,
        '1Ø11'B,'1Ø11'B,'111Ø'B,'111Ø'B,'11Ø1'B,
        '11Ø1'B,'Ø111'B,'Ø111'B,'ØØØ1'B,'ØØØ1'B,
        'ØØ1Ø'B,'ØØ1Ø'B,'1111'B,'1ØØØ'B,'1ØØØ'B,
        'ØØØØ'B,'ØØØØ'B);

DCL ARG CHAR(*) VARYING,
    PLACE FIXED BIN(15),
    (TOP,BOT) FIXED BIN(15),
    FOUND BIT(1);

    TOP = 1;
    BOT = 32;
    FOUND = 'Ø'B;

    DO WHILE(^FOUND&(TOP<=BOT));
        PLACE = TOP + BOT;
        PLACE = DIVIDE(PLACE,2,15);
        IF ARG=EOT(PLACE).NAME THEN FOUND = '1'B;
        ELSE IF ARG>EOT(PLACE).NAME THEN TOP = PLACE + 1;
                ELSE BOT = PLACE - 1;
    END;
    IF FOUND THEN RETURN(PLACE);
```

```
        ELSE RETURN(Ø);
END EOTGET;

/*************************************************************
* MOTGET GETS THE INDEX OF ITS ARGUMENT IN EOT. IF ITS        *
* ARGUMENT IS NOT IN EOT IT RETURNS A Ø.                      *
*************************************************************/

MOTGET: PROCEDURE(ARG) RETURNS(FIXED BIN(15));

DCL 1 MOT(131) STATIC EXTERNAL,
      2 NAME CHAR(5) INIT(
        'A','AH','AL','ALR','AP',
        'AR','BAL','BALR','BC','BCR',
        'BCT','BCTR','BXH','BXLE','C',
        'CDS','CH','CL','CLC','CLCL',
        'CLI','CLM','CLR','CLRIO','CP',
        'CR','CS','CVB','CVD','D',
        'DP','DR','ED','EDMK','EX',
        'HDV','HIO','IC','ICM','IPK',
        'ISK','L','LA','LCR','LCTL',
        'LH','LM','LNR','LPR','LPSW',
        'LR','LRA','LTR','M','MC',
        'MH','MP','MR','MVC','MVCL',
        'MVI','MVN','MVO','MVZ','N',
        'NC','NI','NR','O','OC',
        'OI','OR','PACK','PTLB','RDD',
        'RRB','S','SCK','SCKC','SH',
        'SIGP','SIO','SIOF','SL','SLA',
        'SLDA','SLDL','SLL','SLR','SP',
        'SPKA','SPM','SPT','SPX','SR',
        'SRA','SRDA','SRDL','SRL','SRP',
        'SSK','SSM','ST','STAP','STC',
        'STCK','STCKC','STCM','STCTL','STH',
        'STIDC','STIDP','STM','STNSM','STOSM',
        'STPT','STPX','SVC','TCH','TIO',
        'TM','TR','TRT','TS','UNPK',
        'WRD','X','XC','XI','XR',
        'ZAP'),
      2 TYPE FIXED BIN(15) INIT(
        4,4,4,1,8,
        1,4,1,4,1,
        4,1,5,5,4,
        5,4,4,9,1,
        7,5,1,10,8,
        1,5,4,4,4,
        8,1,9,9,4,
        10,10,4,5,10,
        1,4,4,1,5,
        4,5,1,1,10,
  1,4,1,4,7,
  4,8,1,9,1,
  7,9,8,9,4,
  9,7,1,4,9,
  7,1,8,1Ø,7,
  1Ø,4,1Ø,1Ø,4,
  5,1Ø,1Ø,4,6,
  6,6,6,1,8,
  1Ø,2,1Ø,1Ø,1,
  6,6,6,6,8,
  1,1Ø,4,1Ø,4,
  1Ø,1Ø,5,5,4,
  1Ø,1Ø,5,7,7,
  1Ø,1Ø,3,1Ø,1Ø,
  7,9,9,1Ø,8,
  7,4,9,7,1,
  8),
2 BIN_CODE BIT(8) INIT(
  '5A'B4,'4A'B4,'5E'B4,'1E'B4,'FA'B4,
  '1A'B4,'45'B4,'Ø5'B4,'47'B4,'Ø7'B4,
  '46'B4,'Ø6'B4,'86'B4,'87'B4,'59'B4,
  'BB'B4,'49'B4,'55'B4,'D5'B4,'ØF'B4,
  '95'B4,'BD'B4,'15'B4,'ØØ'B4,'F9'B4,
  '19'B4,'BA'B4,'4F'B4,'4E'B4,'5D'B4,
  'FD'B4,'1D'B4,'DE'B4,'DF'B4,'44'B4,
  'ØØ'B4,'ØØ'B4,'43'B4,'BF'B4,'ØØ'B4,
  'Ø9'B4,'58'B4,'41'B4,'13'B4,'B7'B4,
  '48'B4,'98'B4,'11'B4,'1Ø'B4,'ØØ'B4,
  '18'B4,'B1'B4,'12'B4,'5C'B4,'AF'B4,
  '4C'B4,'FC'B4,'1C'B4,'D2'B4,'ØE'B4,
  '92'B4,'D1'B4,'F1'B4,'D3'B4,'54'B4,
  'D4'B4,'94'B4,'14'B4,'56'B4,'D6'B4,
  '96'B4,'16'B4,'F2'B4,'ØØ'B4,'85'B4,
  'ØØ'B4,'5B'B4,'ØØ'B4,'ØØ'B4,'4B'B4,
  'AE'B4,'ØØ'B4,'ØØ'B4,'5F'B4,'8B'B4,
  '8F'B4,'8D'B4,'89'B4,'1F'B4,'FB'B4,
  'ØØ'B4,'Ø4'B4,'ØØ'B4,'ØØ'B4,'1B'B4,
  '8A'B4,'8E'B4,'8C'B4,'88'B4,'FØ'B4,
  'Ø8'B4,'ØØ'B4,'5Ø'B4,'ØØ'B4,'42'B4,
  'ØØ'B4,'ØØ'B4,'BE'B4,'B6'B4,'4Ø'B4,
  'ØØ'B4,'ØØ'B4,'9Ø'B4,'AC'B4,'AD'B4,
  'ØØ'B4,'ØØ'B4,'ØA'B4,'ØØ'B4,'ØØ'B4,
  '91'B4,'DC'B4,'DD'B4,'ØØ'B4,'F3'B4,
  '84'B4,'57'B4,'D7'B4,'97'B4,'17'B4,
  'F8'B4),
2 MLEN FIXED BIN(15) INIT(
  4,4,4,2,6,
  2,4,2,4,2,
  4,2,4,4,4,
  4,4,4,6,2,
          4,4,2,4,6,
          2,4,4,4,4,
          6,2,6,6,4,
          4,4,4,4,4,
          2,4,4,2,4,
          4,4,2,2,4,
          2,4,2,4,4,
          4,6,2,6,2,
          4,6,6,6,4,
          6,4,2,4,6,
          4,2,6,4,4,
          4,4,4,4,4,
          4,4,4,4,4,
          4,4,4,2,6,
          4,2,4,4,2,
          4,4,4,4,6,
          2,4,4,4,4,
          4,4,4,4,4,
          4,4,4,4,4,
          4,4,2,4,4,
          4,6,6,4,6,
          4,4,6,4,2,
          6);

DCL ARG CHAR(*) VARYING,
    PLACE FIXED BIN(15),
    (TOP,BOT) FIXED BIN(15),
    FOUND BIT(1);

    TOP = 1;
    BOT = 131;
    FOUND = 'Ø'B;

    DO WHILE(^FOUND&(TOP<=BOT));
        PLACE = TOP + BOT;
        PLACE = DIVIDE(PLACE,2,15);
        IF ARG=MOT(PLACE).NAME THEN FOUND = '1'B;
        ELSE IF ARG>MOT(PLACE).NAME THEN TOP = PLACE + 1;
                ELSE BOT = PLACE - 1;
    END;
    IF FOUND THEN RETURN(PLACE);
        ELSE RETURN(Ø);
END MOTGET;

/************************************************************
* PARSLIT SEPARATES THE LITERAL IN ARG INTO ITS DUPLICATION *
* FACTOR, TYPE, MODIFIER, NOMINAL VALUE.                    *
************************************************************/

PARSLIT: PROCEDURE(ARG,DUPFAC,TYP,MOD,NOMVAL);
```

```
DCL (ARG,DUPFAC,TYP,MOD,NOMVAL) CHAR(*) VARYING,
    TARG CHAR(8Ø) VARYING,
    T CHAR(1),
    I FIXED BIN(15),
    FOUND BIT(1);

    TARG = ARG;
    DUPFAC = '';
    TYP = '';
    MOD = '';
    NOMVAL = '';
    FOUND = 'Ø'B;

    DO WHILE(^FOUND);
        T = SUBSTR(TARG,1,1);
        IF INDEX('CFHXBA',T)=Ø THEN DO;
            DUPFAC = DUPFAC || T;
            TARG = SUBSTR(TARG,2);
        END;
        ELSE FOUND = '1'B;
    END;

    TYP = T;
    TARG = SUBSTR(TARG,2);

    IF LENGTH(TARG)>Ø THEN DO;
        IF INDEX('XCB',T)^=Ø THEN DO;
            I = INDEX(TARG,'''');
            IF I>Ø THEN DO;
                MOD = SUBSTR(TARG,1,I-1);
                NOMVAL = SUBSTR(TARG,I);
            END;
            ELSE MOD = TARG;
        END;
        ELSE NOMVAL = TARG;
    END;

END PARSLIT;

/************************************************************
* DLENGTH TAKES THE LITERAL IN 'ARG' AND RETURNS ITS        *
* LENGTH, IN BYTES.                                         *
************************************************************/

DLENGTH:PROCEDURE(ARG) RETURNS(FIXED BIN(15));

DCL ARG CHAR(*) VARYING,
    TARG CHAR(8Ø) VARYING,
        (DUPFAC,MOD,NOMVAL) CHAR(2Ø) VARYING,
        TYP CHAR(1) VARYING,
        (DLEN,VDUPFAC) FIXED BIN(15),
        IND FIXED BIN(15),
        TEMP FLOAT BIN(23),

        PARSLIT ENTRY(CHAR(*) VAR,CHAR(*) VAR,CHAR(*) VAR,
                      CHAR(*) VAR,CHAR(*) VAR);

        TARG = ARG;
        DLEN = Ø;

        CALL PARSLIT(TARG,DUPFAC,TYP,MOD,NOMVAL);
        IND = INDEX('CFHXBA',TYP);
        IF DUPFAC='' THEN VDUPFAC = 1;
            ELSE VDUPFAC = DUPFAC;
        GOTO TLAB(IND);

    TLAB(1):    /*CCCCC*/

        IF MOD = '' THEN
            IF NOMVAL='' THEN DLEN = 1;
            ELSE DLEN = LENGTH(NOMVAL)-2;
        ELSE DLEN = SUBSTR(MOD,2);

        DLEN = VDUPFAC * DLEN;

        GOTO NEXT1;

    TLAB(2):    /*FFFFF*/
        DLEN = VDUPFAC * 4;
        GOTO NEXT1;

    TLAB(3):    /*HHHHH*/
        DLEN = VDUPFAC * 2;
        GOTO NEXT1;

    TLAB(4):    /*XXXXX*/
        IF MOD^='' THEN DO;
            DLEN = SUBSTR(MOD,2);
            DLEN = DLEN * VDUPFAC;
        END;
        ELSE IF NOMVAL='' THEN DLEN = 1;
            ELSE DO;
                TEMP = LENGTH(NOMVAL)-2;
                TEMP = TEMP/2;
                DLEN = CEIL(TEMP);
                DLEN = DLEN * VDUPFAC;
            END;
        GOTO NEXT1;
```

```
TLAB(5):      /*BBBBB*/
    IF MOD^='' THEN DO;
        DLEN = SUBSTR(MOD,2);
        DLEN = DLEN * VDUPFAC;
    END;
    ELSE DO;
        TEMP = LENGTH(NOMVAL)-2;
        TEMP = TEMP/8;
        DLEN = CEIL(TEMP);
        DLEN = DLEN * VDUPFAC;
    END;
    GOTO NEXT1;

TLAB(6):      /*AAAAA*/
    DLEN = VDUPFAC * 4;

NEXT1:
RETURN(DLEN);
END DLENGTH;

/*************************************************************
* NEXTOK SEPARATES THE NEXT OPERAND FROM ARG.               *
*************************************************************/

NEXTOK: PROCEDURE(ARG,RES);

DCL (ARG,RES) CHAR(*) VARYING,
    I FIXED BIN(15),
    T CHAR(1),
    TARG CHAR(80) VARYING,
    FOUND BIT(1);

    TARG = ARG;
    FOUND = '0'B;
    DO I = 2 TO LENGTH(ARG) WHILE(^FOUND);
        T = SUBSTR(TARG,I,1);
        IF INDEX('+-*/',T)>0 THEN DO;
            FOUND = '1'B;
            RES = SUBSTR(TARG,1,I-1);
            TARG = SUBSTR(TARG,I);
        END;
    END;
    IF (^FOUND) THEN DO;
        RES = TARG;
        TARG = '';
    END;
    ARG = TARG;

END NEXTOK;
```

```
/***************************************************************
* NEXTPLUS SEPARATES A STRING UP TO THE FIRST '+' OR '-'      *
* SIGN. THE INPUT STRING IS TRUNCATED.                        *
***************************************************************/

NEXTPLUS: PROCEDURE(ARG,RES);

DCL (ARG,RES) CHAR(*) VARYING,
    I FIXED BIN(15),
    T CHAR(1),
    TARG CHAR(8Ø) VARYING,
    FOUND BIT(1);

    TARG = ARG;
    FOUND = 'Ø'B;
    DO I = 2 TO LENGTH(ARG) WHILE(^FOUND);
        T = SUBSTR(TARG,I,1);
        IF INDEX('+-',T)>Ø THEN DO;
            FOUND = '1'B;
            RES = SUBSTR(TARG,1,I-1);
            TARG = SUBSTR(TARG,I);
        END;
    END;
    IF (^FOUND) THEN DO;
        RES = TARG;
        TARG = '';
    END;
    ARG = TARG;

END NEXTPLUS;

/***************************************************************
* PARSOP PARSES AN OPERAND INTO THE R, X, AND D FIELDS.       *
***************************************************************/

PARSOP: PROCEDURE(OP,A,B,C);

DCL (OP,A,B,C) CHAR(*) VARYING,
    (ICOM,ILP,IRP) FIXED BIN(15),
    TOPF CHAR(8Ø) VARYING;

    TOPF = OP;
    A = '';
    B = '';
    C = '';
    ICOM = INDEX(TOPF,',');
    ILP = INDEX(TOPF,'(');
    IRP = INDEX(TOPF,')');

    IF ILP=Ø THEN DO;
```

```
        A = TOPF;
        RETURN;
    END;
    A = SUBSTR(OP,1,ILP-1);
    IF ICOM=Ø THEN DO;
        B = SUBSTR(TOPF,ILP+1,IRP-ILP-1);
        RETURN;
    END;
    IF (ICOM=ILP+1) THEN DO;
        C = SUBSTR(OP,ICOM+1,IRP-ICOM-1);
        RETURN;
    END;
    B = SUBSTR(OP,ILP+1,ICOM-ILP-1);
    C = SUBSTR(OP,ICOM+1,IRP-ICOM-1);
END PARSOP;

/************************************************************
* REGDROP DROPS THE USE OF 'REGNO' AS A BASE REGISTER.     *
************************************************************/

REGDROP: PROCEDURE(REGNO);

DCL 1 BT(16) STATIC EXTERNAL,
      2 REGISTER FIXED BIN(15),
      2 VALUE FIXED BIN(31),
    BASE_NO FIXED BIN(15) STATIC EXTERNAL;

DCL REGNO FIXED BIN(15),
    (I,J) FIXED BIN(15),
    DONE BIT(1);

    DONE = 'Ø'B;
    DO I = 1 TO BASE_NO WHILE(^DONE);
        IF REGNO = BT(I).REGISTER THEN DO;
            DO J = I TO BASE_NO-1;
                BT(J) = BT(J+1);
            END;
            BASE_NO = BASE_NO - 1;
            DONE = '1'B;
        END;
    END;
END REGDROP;

/************************************************************
* BTSTO ADDS 'REGNO' AS A BASE REGISTER WITH CONTENTS      *
* 'REGVAL'. THE BASE REGISTERS ARE SORTED SUCH THAT THE    *
* FIRST ENTRY IN BT CONTAINS THE SMALLEST LC VALUE.        *
************************************************************/

BTSTO: PROCEDURE(REGNO,REGVAL);
```

```
DCL 1 BT(16) STATIC EXTERNAL,
      2 REGISTER FIXED BIN(15),
      2 VALUE FIXED BIN(31),
    BASE_NO FIXED BIN(15) STATIC EXTERNAL;

DCL (REGNO,REGVAL) FIXED BIN(15),
    I FIXED BIN(15),
    DONE BIT(1);

DCL REGDROP ENTRY(FIXED BIN(15));

    BT(BASE_NO+1).REGISTER = REGNO;
    BT(BASE_NO+1).VALUE = REGVAL;
    DONE = '0'B;

    CALL REGDROP(REGNO);

    DO I = BASE_NO TO 1 BY -1 WHILE (^DONE);
        IF BT(I).VALUE>REGVAL THEN DO;
            BT(I+1) = BT(I);
            BT(I).REGISTER = REGNO;
            BT(I).VALUE = REGVAL;
        END;
        ELSE DONE = '1'B;
    END;
    BASE_NO = BASE_NO + 1;
END BTSTO;

/**************************************************************
* BRGET GETS THE BASE REGISTER AND ITS CONTENTS FOR A GIVEN *
* SYMBOLIC ADDRESS.                                          *
**************************************************************/

BRGET: PROCEDURE(REGVAL,BRVAL) RETURNS(FIXED BIN(4));

DCL 1 BT(16) STATIC EXTERNAL,
      2 REGISTER FIXED BIN(15),
      2 VALUE FIXED BIN(31),
    BASE_NO FIXED BIN(15) STATIC EXTERNAL;

DCL (REGVAL,BRVAL) FIXED BIN(15),
    REGNO FIXED BIN(4),
    I FIXED BIN(15),
    DONE BIT(1);

    DONE = '0'B;
    REGNO = 0;
    BRVAL = 0;
    IF BASE_NO=0 THEN RETURN(0);
    DO I = BASE_NO TO 1 BY -1 WHILE (^DONE);
        IF REGVAL>=BT(I).VALUE THEN DO;
            REGNO = BT(I).REGISTER;
            BRVAL = BT(I).VALUE;
            DONE = '1'B;
        END;
    END;
    RETURN(REGNO);
END BRGET;

/************************************************************
* CHARGET GETS THE HEXADECIMAL BIT STRING REPRESENTATION    *
* FOR A GIVEN CHARACTER ARGUMENT.                           *
************************************************************/

CHARGET: PROCEDURE(ARG) RETURNS(BIT(8));

DCL ARG CHAR(*),
    I FIXED BIN(15);

DCL 1 CT(63) STATIC,
      2 SYMBOL CHAR(1) INIT(
        'Ø','1','2','3','4',
        '5','6','7','8','9',
        'A','B','C','D','E',
        'F','G','H','I','J',
        'K','L','M','N','O',
        'P','Q','R','S','T',
        'U','V','W','X','Y',
        'Z','.','<','(','+',
        '|','&','!','$','*',
        ')',';','^','-','/',
        ',','
        ':','#','=','"','~',
        '{','}','\'),
      2 VALUE BIT(8) ALIGNED INIT(
        'FØ'B4,'F1'B4,'F2'B4,'F3'B4,'F4'B4,
        'F5'B4,'F6'B4,'F7'B4,'F8'B4,'F9'B4,
        'C1'B4,'C2'B4,'C3'B4,'C4'B4,'C5'B4,
        'C6'B4,'C7'B4,'C8'B4,'C9'B4,'D1'B4,
        'D2'B4,'D3'B4,'D4'B4,'D5'B4,'D6'B4,
        'D7'B4,'D8'B4,'D9'B4,'E2'B4,'E3'B4,
        'E4'B4,'E5'B4,'E6'B4,'E7'B4,'E8'B4,
        'E9'B4,'4B'B4,'4C'B4,'4D'B4,'4E'B4,
        '6A'B4,'5Ø'B4,'5A'B4,'5B'B4,'5C'B4,
        '5D'B4,'5E'B4,'5F'B4,'6Ø'B4,'61'B4,
        '6B'B4,'6C'B4,'6D'B4,'6E'B4,'6F'B4,
        '7A'B4,'7B'B4,'7E'B4,'7F'B4,'A1'B4,
        'CØ'B4,'DØ'B4,'EØ'B4);
    DO I = 1 TO 63;
        IF CT(I).SYMBOL=ARG THEN RETURN(CT(I).VALUE);
    END;
    RETURN('ØØ'B4);
END CHARGET;
```

APPENDIX C

EXEC Used to Execute ASSMBLR

The following EXEC macro allows a user to call ASSMBLR by typing the following instructions:

```
    EXEC ASSEMBLE filename
```

where filename is the full name, including pathname, of the file he wants assembled.

ASSEMBLE:

```
    COPY &1 T1ONG>INDATA
    SEG
    VLOAD #ASSMBLR
    LOAD B_ASSMBLR
    LOAD B_ROUTINES
    LI PL1GLB
    LI
    EXECUTE
```

APPENDIX D

A User's Guide to ASSMBLR

This section was written specially for anyone intending to use ASSMBLR.

D.1 Operating Instructions

ASSMBLR is implemented for a subset of the full IBM assembly language. Therefore, in addition to the syntax requirements of the full IBM assembly language, there are additional constraints that you will have to observe. These constraints are stated in section D.2 of this Appendix.

To assemble your program, you must first have your assembly language program in a file. This can be done directly at a terminal. Or, you can punch your program into a deck of cards and use the Prime readcard facility to read your deck into a file. Once you have your program in a file, all you have to do is type the following sets of instructions.

```
A T1ONG SGLP
EXEC ASSEMBLE loginid>filename
```

where loginid is your login id, and filename is the name of the file where you have stored your assembly language program. This will cause ASSMBLR to assemble your program, and you will get the result printed out on your terminal.

D.2 Constraints

1. Only 1 control section is allowed and the first instruction must be a CSECT.

2. No macros are allowed.

3. Maximum length for the program is 2ØØ lines.

4. No floating point instructions are allowed.

5. No S format instructions are allowed.

6. If one of the operands of an instruction is a literal, that literal must be the last operand of the instruction.

7. Length attributes in SS instructions must be explicitly stated.

8. Valid expressions are those in which constants (or self-defining terms) are written as decimal integers. Signs are optional.

9. The only legal operators are '+', '-', '/', and '*'. Parentheses are not permitted in expressions.

1Ø. Only constants and literals of type C, X, B, F, H, A are recognized.

11. Literals must start with the '=' symbol.

12. Symbol length attribute references are not permitted.

13. The duplication factor must be an unsigned decimal integer, if one is used at all.

14. For type C constants or literals, only a subset of the full EBCDIC character set is implemented. Specifically, quotes are not permitted.

15. Only one address can be specified within the parentheses of an A type constant.

16. For X type constants the number or characters enclosed in quotes (') must be even. For B type constants, the number of binary digits must be an integral multiple of 8.

17. Each constant must include a type attribute. For example, DC F'7,8,9' should be written as DC F'7',F'8',F'9'.

18. Only constants and literals of types C or X can have length modifiers.

APPENDIX E

A Sample Session

```
exec assemble indata2
C>COPY INDATA2 INDATA
C>SEG
[SEG rev 17.Ø]
# VLOAD #ASSMBLR
$ LOAD B_ASSMBLR
$ LOAD B_ROUTINES
$ LI PL1GLB
$ LI
$ EXECUTE

  LOC   OBJECT CODE          STMT   SOURCE STATEMENT

                               1 MP1    CSECT
                               2        USING *,15
ØØØØØØ 5C3Ø F03Ø               3        M     INDEX,=F'-16'
ØØØØØ4 593Ø FØ34               4 NEXT   C     INDEX,=F'36'
                               5        USING *,9
ØØØØØ8 411Ø 7ØØØ               6 END    LA    1,Ø(Ø,ROMAN)
ØØØØØC ØAØC                    7        SVC   COUNT
ØØØØØE 951B 9Ø1Ø               8        CLI   SAVER+4,X'1B'
ØØØØ12 Ø7FE                    9        BR    14
ØØØØ14 ØØØØØØØ4               1Ø SAVER  DC    A(END-NEXT)
ØØØØ18 C8C5C8C5C8C5C8C5       11        DC    4CL2'HERE'
ØØØØ2Ø C8C5C8C5D9C5F1F2       12        DC    C'HEHERE12?'
ØØØØ2C ØØØØØØØ8               13        DC    A(END)
                              14 INDEX  EQU   3
                              15 ROMAN  EQU   7
                              16 COUNT  EQU   12
                              17        END
ØØØØ3Ø FFFFFFFØ                                 =F'-16'
ØØØØ34 ØØØØØØ24                                 =F'36'
C>ENDX T$ØØØ1
C>
```