ASSEMBLY OF THE CDC 6600 COMPASS LANGUAGE ON THE IBM 360/65 COMPUTER A HIGH-LEVEL LANGUAGE APPROACH

A THESIS

Presented to

The Faculty of Graduate Studies and Research

The University of Manitoba

in partial fulfilment

of the requirements for the degree

Master of Science in the Department of Computer Science

by

Robert Groff

January 1973

ABSTRACT

The development of a load-and-go assembler for the CDC 6600 computer written in a high-level language is presented in this thesis. A description of the benefits of using high-level languages for writing this type of software is given along with a description of the load-and-go assembler. The assembler, when linked with the interpreter of the CDC instructions, can be run on the IBM 360/65 as a teaching tool for the CDC Compass language.

ACKNOWLEDGEMENTS

I wish to express my appreciation to my thesis supervisor, Robert J. Collens, for his advice throughout this project.

I am thankful to M. Doyle and J. Lawless for reading and commenting on the thesis.

I also wish to thank Gerald Neufeld for the use of his assembly language I/O routines, and Bill Kocay whose programs helped in debugging the interpretive system.

Robert K. Groff

January 1973

TABLE OF CONTENTS

CHAPTER I

THE APPROACH TO WRITING THE ASSEMBLER

1.0 INTRODUCTION

The best way to study the various computing machines and computer languages is by programming them. Since a variety of languages are available on large computers it is possible for students to learn many high-level languages and the assembly language of the available computer. But what about the other computers and assembly languages one may be required to know at other computer installations? To enable students to familiarize themselves with these machines and assembly languages, it is necessary for simulators and interpreters to be written for the available computer.

The initial goals in planning the project described in this thesis were:

1) To use a high-level language at the start and make any changes to a low-level language later if they were deemed necessary.

2) To have a load-and-go assembler since it was felt this would minimize turnaround time.

1.1 HISTORY OF ASSEMBLERS

Assemblers were the first translators. An assembly language is essentially a symbolic representation of the machine code with a one-to-one conversion ratio. In two passes through a set of instructions, assemblers first produce a symbol table and then convert the symbolic opcode and addresses to machine code.

Some computers were hard to program in their own machine language and this encouraged the development of programming languages. The first commercially available computer, the Univac I (1951), was relatively easy to program in its own code. It was a decimal, alphanumeric machine with mnemonic instructions that were easy to remember and use. The 12-character word made scaling of many fixed-point calculations fairly easy. It was not always easy to see the advantage of assembly systems and compilers that were slow and clumsy in a machine of only 12,000 characters of high-speed storage (200 microseconds average access time per 12-character word).

The IBM 701 (1952) was the first commercially marketed large-scale binary computer. The best known programming language on the 701 was Speedcode (a language that made the one-address binary fixed-point machine appear to be a three-address decimal floating-point computer with index registers). More than almost any language, Speedcode

demonstrated the extent to which users were willing to sacrifice computer speed for the sake of programming convenience.

The IBM 650 was an easily programmed drum computer. It lacked floating-point hardware, a feature that was later available. A program could be optimized by proper placement of the instructions on the drum. SOAP (Symbolic Optimizer and Assembly Program) did the assembly and the optimizing for the programmer. The optimization feature of this program made it valuable to the programmer.

With the introduction of magnetic core storage the IBM 704 was produced with many hardware features that had previously been software features, such as index registers and the ability to do floating-point arithmetic. These features removed the necessity of interpretive systems such as Speedcode. SAP (Share Assembly Program) was an early assembler used on the IBM 704 computer in 1956. These early assembly languages were much like the MIX (see Knuth vol. 1) assembly language. Both computers and assemblers became more complex with the coming of machines like the CDC 1640 and the IBM 7090. These machines are similar to the MIX machine.

After two generations most assembly systems became full-scale computer-oriented compilers with features such as extensive libraries of subroutines, macro generators and program organization features.

1.2 IMPLEMENTATION LANGUAGE

The aims of this project were to create a teaching tool for students and to experiment in writing computer systems in high-level languages. The teaching tool is the resulting interpretive system for the CDC 6600 computer, written in FORTRAN.

FORTRAN was chosen as the high-level language for several reasons:

1) The WATFIV compiler was available for testing algorithms and modules as they were written, allowing fast turnaround and giving good error diagnostics.

2) The FORTRAN H compiler [14] was available. FORTRAN H is an optimizing compiler and removes inefficiencies in the user's source code. Other extra features, not available in most FORTRAN compilers, make it an excellent compiler for writing software. FORTRAN H allows the use of LOGICAL*1 variables, which permits the comparison of characters one at a time. The function SHFTR(L,N) is a built-in function which allows you to shift a bit string L, N places to the right. The opposite left shift function and other bit manipulation functions are also available.

3) Since a FORTRAN IV compiler is available on most large machines, the interpreter can be run at other installations with few modifications. Although there are differences between the various FORTRAN compilers, the

program can be run without extensive rewriting of the program.

4) Most students of computer science are familiar with FORTRAN and will be able to modify or add to the simulator if this is desired.

5) It is easier to code in high-level languages and easier to debug the code. Program bugs are difficult to find and fix in software written in assembly language.

6) In most cases the number of statements required to write a given algorithm is less in the high-level language.

7) High-level languages make it easier to write large programs. This results in the development of better software using less computer time. The average programmer does not write better code than FORTRAN H produces when they write in assembly language. Thus it is quite likely that systems written in high-level languages would be better than those written in assembly languages.

The Burroughs 5500 operating system was written in Algol 60. The operating system has all the advantages of being written in a high-level language and is also fast, efficient, and small in program size. These features are thought to be reserved for systems written in assembly language. With the availability of good optimizing compilers this need not be true.

1.3 SPECIFICATIONS FOR THE ASSEMBLER

This project consisted of writing a load-and-go assembler which would simulate the CDC Compass assembler. An IBM 360/65 was used as the host machine for the load-and-go assembler. Some of Compass's advanced features such as micros, macros and many of its psuedo-opcodes were left out. This omission provided for a simpler language. Because the assembler's main purpose was to be used as a teaching device, some special properties were added to the assembler.

These desirable properties of such a teaching language are given below:

1) The assembler should not require a large amount of core. We considered a limit to be about 100K.

2) Batch processing should be available.

3) Turnaround should be very fast.

4) Error diagnostics should be very good and simple debug features should be available. For examples see below.

The assembler has all of these features. Each error diagnostic includes a brief description of the error. For example, the statement

```
SB2 B1
```

will cause the error message;

$$$ERROR$$$--- STRING B1 IS NOT AN OPCODE. PROBABLE CAUSE, MISSING BLANK.

The above error occurred because the opcode started before

column three of the card.

Since the assembler was written for student use it has many error messages to make debugging easier and has special debug features not available in the Compass assembler. These are the TRACEON and TRACEOFF features that allow you to trace the execution of the program instruction by instruction.

The combined assembler-interpreter [4] takes 102 K of core to run. Batch processing is available and turnaround is very fast. The program assembles statements at a rate of more than 6000 cards per minute.

1.4 SIMULATORS and INTERPRETERS

Simulators and interpreters make the host computer mimic another computer or complex system through software. Interpretive assemblers differ from real assemblers in that the machine code produced is not generally for the host machine but for an interpreter that translates and executes the assembled code on the host machine.

Machines are simulated to give students other examples of computers to study or easier machines to study. The Spectre, Gamma 70 and Mix computers are examples of simulated hypothetical computers that are easier to understand and program than most real machines. In the hypothetical machine, many features are left out so that

only a very basic machine is produced. It is used as a teaching tool to enable students to better grasp the workings of a basic stored-program machine. A short description of the SPECTRE machine is given here as an example.

The Spectre machine is a simple configuration with an input unit (card reader), an output unit (line printer), a central memory of 1000 words, a central processor and an arithmetic unit. The machine has 2 registers used by programmers for arithmetic operations, the AC (Accumulator), and the MQ (Multiplier Quotient) and 2 internal registers, the IAR (Instruction Address Register) and the IR (Instruction Register). Each register holds a ten-digit signed integer except the IAR which holds only three digits (the address of the next instruction to be executed). The IR holds the instruction currently being executed.

The memory of the Spectre computer holds 1000 ten-digit signed integers with values between -9999999999 and +9999999999. These are addressed 000 to 999.

The machine can be programmed directly in machine code, or in the assembly language, Spectre MAP. The assembler is a simple two pass program that does a one-to-one conversion of the symbolic opcodes and addresses to the machine equivalent. Since there are no index registers, address modification is used to perform indexing. Spectre is written

in IBM 360 assembly language as other simulators have been. [15]

With our simulator, students are able to study a new assembly language that runs on a real computer and familiarize themselves with a computer quite different in design from the IBM 360/65, the host computer.

1.5 A DESCRIPTION OF THE CDC 6600

The CDC 6600 is similar to the older model CDC machines in many ways:

1) The number of possible opcodes is 64.

2) The machine uses binary representation for numbers.

3) One's complement notation is used for negative numbers. This means there are two zeros, a plus zero and a minus zero.

The word length in CDC machines increased from 48 bits in the CDC 1604 and CDC 3600 to 60 bits in the CDC 6600. The CDC 6600 computer is a large multipurpose machine with up to 10 Peripheral Processor Units (PPU's) which handle all I/O. The PPU's can do a small amount of arithmetic. Each has a memory of 4096 words of 12 bits each. The PPU's are assigned communication areas of 8 60-bit words each in central memory. The first word is the input register and the second the output register. The remainder is a message buffer. Each PPU has a 18-bit accumulator register and a repertoire of 61

instructions. The computer has 12 data channels. PPU's can read or write, using peripheral equipment, into their own memory using any of the channels and can read or write directly into central memory from their own memory.

The Central Processor Unit (CPU) does the computing. It has access to a central memory of up to 131,072 words of 60 bits each.

Input to the central memory is received as five 12-bit words combined in a buffer and transmitted to one 60-bit word. For output the CPU can set a word in memory for a PPU to read.

A floating-point number is represented in memory as k.2**n, where 2000 (the bias base 8 or excess 1024 base 10 notation) + n is stored in the 12 high-order bits (the leftmost bit being the sign of the number), and k (the integer portion) in the other 48 bits of the 60-bit word. The representation of the 60 bits labelled from right to left by 0,1,2,.....59 is given below.

59	48	47	0
S	EXPONENT	INTEGER k	

If n (base 8) is negative, it is changed to one's complement notation and added to 2000. For example (the following numbers are base 8), if n is -1 it is converted to 7776 and added to 2000 giving 11776. The high-order octal digit is

ignored, giving a biased exponent of 1776. Therefore there are two zero exponents, 2000 and 1777. 1777 is minus zero and is used for indefinite results.

CPU instructions can be 15-bit instructions or 30-bit instructions. Respectively, they take the following forms fm i j k or fm i j K. The fm part is a 6-bit function code (opcode). The parts i, j, and k are 3-bit register designators. The part K in the 30-bit instruction is an 18-bit address. The 60-bit words of memory can be filled with two 30-bit, four 15-bit or one 30-bit and two 15-bit instructions. Any one of these can be a null operation.

The CPU has eight X registers, eight A registers and eight B registers. The 60-bit X registers act as accumulators, the A registers hold 18-bit addresses and the 18-bit B registers are for indexing. A and X registers are linked so that the following functions occur if An is referenced:

1) n=0 no effect on the X registers.

2) n=1,2,3,4 or 5 the contents of the address referenced by An are loaded into Xn.

3) n=6 or 7 contents of the Xn register are unloaded into the address referenced by An.

There are ten distinct functional units in the CPU to speed execution. These units are the floating add unit, the long add unit, the divide unit, the shift unit, the boolean

unit, the branch unit, two multiply units and two short add units. Essentially this means that ten operations can be running simultaneously if the programmer can make the best use of the machine. The CPU has an instruction stack of thirty-two, 15-bit registers to hold instructions prior to execution, saving fetch time.

A program is initiated in the CPU when a PPU loads the program into central memory. The PPU must build an 'exchange jump package' and then do an exchange jump (an exchange jump is a special PPU opcode, see below). This package is a sequence of sixteen 60-bit words in central memory which are used to set the initial values of the registers A0 to A7, X0 to X7, B1 to B7, P, RA, FL and EM. The P (Program Address) register is the address of the next CPU instruction. RA is the reference address, which is used for program relocation. Any reference to a memory location K in the program automatically becomes K+RA, making programs relocatable. FL (Field Length) defines the number of consecutive memory locations used by the program. The CPU will stop if the program references an address outside the range specified by RA and FL. EM (Exit Mode) defines the conditions that are allowed to stop the operation of the CPU.

The exchange jump is a PPU instruction specifying the address of an exchange package that is currently in memory. The initial conditions for the next program are set storing

the previous contents of A, X, B, P, RA, FL and EM registers in the exchange package. That is, the exchange packages of the old and new programs are interchanged and the operation of the new program starts at the instruction in location P+RA.

The SCOPE operating system uses one PPU for the system display routine, and one as the Monitor PPU, which keeps continuous watch on the program operating in the CPU and passes messages from the program to other PPU's. The Monitor is in permament overall control of the system. Other PPU's cannot perform any function not approved in advance. The eight other PPU's remain in a pool, looping on a read of their input register until a request is received. The Monitor selects a PPU and instructs it to carry out the operation by putting a significant word into its input register. The PPU obeys any request, then indicates to the Monitor via its output buffer that it is finished. The Monitor periodically reads the output register of each PPU looking for requests and zeroes the registers whenever the request has been satisfied. The Monitor also handles requests from peripheral devices and allocates a PPU to process these jobs. PPU's are primarily slaves of the Monitor but occasionally a PPU must request cooperation of other PPU's. These requests are routed through the Monitor. To prevent two PPU's from trying to use the same I/O

channel, the Monitor maintains a list of channels and their status. A PPU must request the Monitor to assign a channel to it. Commonly used PPU programs are stored in central memory.

The CDC 6600 is a multiprogrammed computer. The executing program requests I/O by storing the request at a specific location in memory and stopping. The Monitor then acts according to the type of I/O requested. If it is a buffered request, the Monitor accepts the request and restarts the program. Later the CPU can examine a flag to see if the request has been fulfilled. A circular buffer is used and the PPU can fill it at the same time the CPU empties it. If it is an unbuffered request, it implies that the program cannot continue until the request is fulfilled. The Monitor passes the request to a PPU, then does an exchange jump to start another program. Later when the I/O request has been completed the first program can be restarted.

The Compass assembler runs under the SCOPE operating system. Compass language is similar to most advanced assembly languages in that it has macro capabilities and many pseudo-opcodes. The main difference with respect to other languages is that the Compass opcodes depend on the operands. That is, the machine language code cannot be produced until the syntax of the operand is known.

A detailed description of the language and register usage may be found in the Compass Reference Manual. [13]

CHAPTER 2

USING HIGH-LEVEL LANGUAGES

2.0 INTRODUCTION

The load-and-go assembler was written totally in FORTRAN H except for the subroutines to do I/O. This seems to be a much better approach to writing large programs than writing them in assembly languages. There are reasons for using assembly languages but with the development of compilers for writing systems software that produce good optimized code most of the benefits of using assembly language disappear. However, it is felt that a knowledge of assembly language and the computer being used aids in the writing of good code in high-level languages.

2.1 HIGH-LEVEL VS LOW-LEVEL LANGUAGES

The quality of code produced depends on the programmer and the compiler or assembler being used. The code in low-level languages is much more dependent on the programmer than the code from a high-level language. The FORTRAN H compiler will fix-up inefficient FORTRAN code for the programmer in many cases. Code is not improved by assemblers.

It is possible for a good assembler programmer to outcode the FORTRAN H compiler but only with a lot of work

to ensure that the code being written is optimal. Large systems take much time to develop in any language. This time could be reduced by using high-level languages.

The following two quotations discuss the positive and negative aspects of assembly coding.

" There are a number of reasons why assembly language is used.

1) Assembly language has no peers for producing efficient programs, that is, programs efficient in terms of compiled program size and speed of running. Efficiency does matter. People often pay lip service to it and then write extremely inefficient programs. High efficiency reduces operating costs and makes things possible that are impossible if the programs are inefficient. Two examples are fast console response to filing operations in a multi-access system and wringing the most out of a small (say 8K) machine. The skill and ingenuity of the programmer also have a great effect on the efficiency, but this is just as true for high-level languages as for assembly code.

2) Assembly language enables you to get as close to the machine as is possible. It does not prevent any operation that the machine is capable of executing from being programmed. This is certainly not true of high-level languages. Further, the discipline of learning assembly language ensures that the programmer learns about the machine. High-level languages screen the programmer from the machine. While this attribute is highly desirable in many applications, system programs can benefit from the programmer thoroughly understanding the actual machine and its capabilities.

3) Assembly language can be quickly compiled; this is a great convenience when developing programs, especially on-line with a single user or multi-access machine.

4) On-line debugging systems are an enormous aid to debugging at the assembly language level. Typical facilities are interrogation and setting of registers, searching for bit patterns and break points, free running and interpretive execution and so on. Correction of errors with high-level languages

normally involves editing the source document, recompiling and reloading.

5) Assembly language systems do not require the use of a run-time system to perform functions such as space allocation. This leaves the system programmer with the maximum flexibility to organize his system program to suit the problem at hand." [6]

For these reasons almost all system programs are written in assembly language. Are these reasons sufficient to make programming in assembly language desirable?

" The answer is no, but unfortunately it is still the most suitable language on many machines. Briefly, because they are well known, the very grave disadvantages of assembly language are:

1) Apart from the few who delight in such intricacies, most people find assembly language programs harder to write, read, understand, debug and maintain than high-level language programs.

2) It provides the poorest conceptual framework for the programmer to express the computing operations he wants performed.

3) It is completely machine dependent, thus requiring any machine language program to be completely rewritten when it is transferred to a different machine." [6]

Since programming in assembly language is undesirable for these reasons, methods must be found to take advantage of the good properties of assembly languages. Compilers can produce efficient code if they are well written and have a good optimization feature, such as the one in the FORTRAN H compiler. Machines can be designed to be compatible with a high-level language. The B-6500 was designed with this feature in mind. Debug features of debug compilers can be

just as fast and efficient as the assembly language debug features, and also can be much easier to use. For example, WATFIV produces excellent diagnostics.

Why have almost all systems in the past been written in assembly languages? This has been true because of the advantages of assembly languages as mentioned above and because the code most high-level languages have produced in the past has been very inefficient. This need not be the case. Other advantages of high-level languages make them very acceptable for writing large programs.

> " The advantages of coding in a high-level language lie in increased programmer productivity, fewer bugs in the code, fewer programmers required, increased flexibility in the product, 'readability' of the source code ('self-documentation') and some degree (unspecified) of machine independence (better called 'portability')." [2]

> " Portability is the property of a system which permits it to be mapped from one environment to a different environment." [8]

Thus using high-level languages can have many advantages. With improved compilers having built-in optimizing features, it seems likely that high-level languages will be used much more in the future for writing systems software.

It seems that having both a debug compiler and a production compiler is a good approach to produce efficient programs in high-level languages in a relatively short period of time. It is unlikely that a good debug compiler

can also produce good, efficient code and be efficient in both areas. Therefore, it is reasonable to have compilers such as WATFIV for debugging and one, such as FORTRAN H, for producing good code. This is the approach that IBM seems to be taking with their new PL1 compilers. With this approach most error checking could be taken out of the production compiler and the error checking would then be done before-hand using the debug compiler.

Assembly language still has a place in producing software until a good system writing compiler is produced. Where sections of code are found to be very inefficient and it is felt that writing the code in assembly language would produce a significant improvement, it would be reasonable to do this. This was done with the CDC load-and-go assembler. The I/O routines were replaced with assembly language subroutines, which avoided the use of FIOCS and resulted in much shorter run-time for the program. This mixed language development approach is being taken in many systems to-day; for example, in programs such as SPSS.

Writing software in high-level languages has already produced some very good systems and programs. For example, FORTRAN H was written in FORTRAN H.

> " I want to defend the FORTRAN H compiler. It is perhaps the most complex FORTRAN compiler that has ever been written. It is unfair to compare it with compilers written in machine language, that do not do the same functions. It is indeed doubtful whether it could have been written in anything but a high-level

language. I know of other successful uses of high-level languages for writing assemblers and compilers. The point of using high-level languages is to suppress unnecessary detail, which is vital in complex problems." [7]

If high-level languages are used extensively for writing systems software, it will be reasonable to design computers that are well integrated with the language being used. The hardware will be designed so that the computer can perform the software functions as quickly as possible. Thus the computer should be very fast when using the language for which it was designed.

The B-6500, a Burroughs computer, was designed so that high-level languages (especially ALGOL) could be used exclusively in programming the machine. Making the hardware of a computer compatible with a high-level language seems to be a very good approach for producing good software. The following quotations indicate a growing trend to promote the use of high-level languages in all areas of software writing.

" For almost a decade now, the people in Burroughs have been writing their compilers and operating systems in higher-level languages, primarily ALGOL 60 or ALGOL 60 with extensions. Many say that the software systems that have been delivered are quite respectable." [1]

" The implementation of the operating systems and compilers by use of higher-level languages (and all that it implies) is one of the most important (if not the most important) reason for the success that the software provided with the B-5500 and B-6500 has enjoyed." [1]

CHAPTER 3

THE ASSEMBLER

3.0 IMPLEMENTATION FEATURES

The CDC load-and-go assembler was designed to make the host computer look as much like the CDC 6600 computer as possible. The host computer was an IBM 360/65 machine which has 32-bit words. The CDC machine to be simulated has 60-bit words.

This meant using double precision (ie. two words) for every word of code produced. This left four unused bits for every CDC word produced. These extra bits were used to contain flags, such as the trace flag, for the trace feature that was implemented.

The documentation for the CDC 6600 computer uses an octal notation to represent a word of memory. This means that every 3 bits in the CDC word form a digit between 0 and 7. This gives 20 digits to the 60-bit word. Since the simulator was written for the IBM 360/65, 30 bits of an IBM word were used to make one-half of a CDC word. An IBM 32-bit word can hold an integer as large as 2**31-1. The words were filled digit by digit as the statements were assembled. To store the ten digits in one word they had to be taken one at a time, multiplied by the appropriate power of 8 and added to the word being created. When this was done one-half of a

CDC word in the 30 low-order bits of an IBM word was formed. This word had the same bit pattern as the equivalent CDC half-word would have if assembly were performed on a CDC 6600 computer.

It was decided to have a memory of 512 words in the interpreted CDC machine. In the Block Data routine all the tables are initialized to certain values and can easily be changed. Hence memory size can be altered easily by changing one small routine which initializes constants. The output from the program resembles the output from a CDC Compass assembler. This is achieved by storing each statement until the END card is encountered and then printing the output.

The load-and-go assembler was written in FORTRAN using the WATFIV and FORTRAN H compilers. It was written in modular form. These modules are fairly small and uncomplicated so that the resulting program can be understood by others without much difficulty. Most of the large routines are fairly well segmented so that they are really just a combination of smaller modules.

The input to the assembler is a subset of the Compass assembly language. This subset includes all the opcodes for the CDC 6600 CPU, and a few of the psuedo-opcodes that are necessary for assembly. Opcodes to perform I/O are added to the instruction set. Debug features are included as psuedo-opcodes. Detailed error messages are provided.

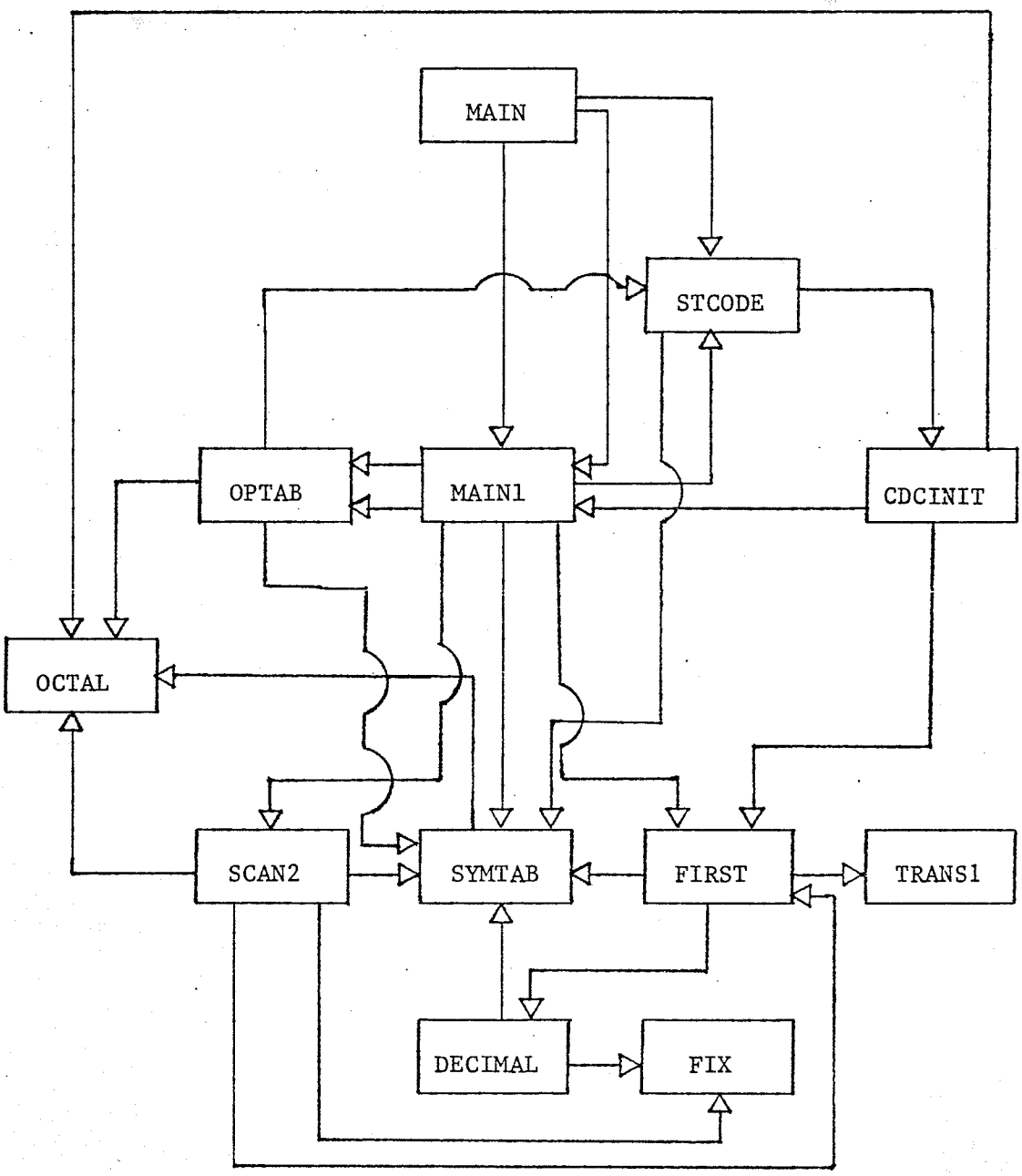
The assembler was first written in WATFIV which allows for easy debugging of the algorithms. Separate scan routines were written for the labels and opcodes, and others for the operand field. One routine was written to handle constants and literals (see Compass Reference Manual). A symbol table routine was written to handle simple linear tables. This included a table for literals. Characters and integers had to be translated into CDC's form. A routine to do this translation was written. Floating-point numbers were translated using a different routine.

Base 10 and base 8 numbers were allowed, making it necessary to check for the base along with the many other modifiers allowed for each constant and literal (see Compass Reference Manual).

The organization of the program is such that there is really no mainline program but a system of subroutines which carry out specific tasks. There is an entry point at TEMP, the first routine.

FIGURE 1

Macro Flowchart of the Load-and-Go Assembler



3.1 THE SUBROUTINES

The BLOCK DATA routine has already been briefly mentioned. It contains all COMMON statements containing all the tables and constants that may require changing. They are initialized in this routine to make changes in the size of tables fairly uncomplicated. This includes changes in memory size, symbol and literal table length, and the number of executable statements per program.

The subroutine called MAIN is used to read the card images one at a time. The input may be CDC Compass language statements or data for the executing program. Each new program initializes constants in this routine and this routine calls others so that their constants can be initialized. Scanning starts in subroutine MAIN with a search for labels and opcodes. When these are found the appropriate routines are called. Literals are recognized by the '=' sign in the MAIN routine and routine FIRST is called to translate the literal into a form that can be easily handled. The SCAN routine is called to scan the variable field if no literal has been detected. On return from routine SCAN, MAIN reads a new card and processes it, unless an end-of-file has been reached.

The symbol table routine, SYMTAB, first initializes the tables when called by MAIN. It will be noticed that, in batch processing, only entries that have been used are

re-initialized with each new program in a batch. The whole table is initialized with the first program of the batch. The symbol table has five kinds of entries; constants and labels that already have values, undefined variables from the operand field, labels from SET statements which can have their values changed by other SET statements, and the location pointer value if it is used in the program. These are entered and distinguished only by their name. Each of them is given a value if defined. The value is left at -1 if the entry remains undefined and a pointer is set so that the value can be added into memory when the END statement is encountered. This is a feature of a one pass assembler. If any variables in the symbol table remain undefined an error is produced at the end of the program.

Literals can be from 1 to 4 CDC words long. They have a separate table. There is a pointer to the next available location in the table. For each literal a pointer is kept which points to the statement that produced it. The literal value is stored only once.

The translation routine, TRANS, takes both integer and character literals and produces the internal representation for the CDC machine using a table of values for each possible character. Since each character is represented by two base 8 digits, there can be a character set of at most 64 characters. Character strings can be 40 characters long,

with 10 characters to a CDC word. Integers and floating-point numbers are up to 20 digits in length if they are literals, but constants can only be numbers less than 777777 base 8. Larger numbers produce an error message. This limit is set since CDC has an addressing capability of only 18 bits. The CDC interpreter has a memory of 1000 (base 8) words numbered in octal from 000 to 777. If an address is too large for the interpreter an error is generated at execution time.

Character strings are converted character by character into the IBM word. Integers are more difficult since they can be base 8 or base 10. A special case is made for base 8 numbers with no modifiers so that any desired bit string can be easily produced. Otherwise, integers are handled in a more general manner using two IBM words, building the integer digit by digit to resemble the CDC internal form.

The SCAN routine scans all operands except literals. Expressions are allowed to have up to four entries. That is, whenever K is allowed in an operand any combination of up to a total of four variables, constants or a single location pointer value is a legal entry in the operand field. One of the variables can be undefined per expresion, since there is only one pointer table value for each 30-bit instruction. The pointer table contains pointers to the symbol table entry of the undefined variable, or to a literal table entry

in the instruction to which it refers. Only addition and subtraction are allowed in expressions. Undefined variables cannot be preceeded by a minus sign.

The operand field is scanned for any delimiters or constants. Any value beginning with a digit is assumed to be a constant and subroutine FIRST is called.

When the variables or constants have been separated, the symbol table routine is called via various entry points to place the new values in the table. Registers are also decoded in this routine. The type and number of the register must be found to produce the opcode.

The instruction is in a vector of 10 entries, one for each digit of the instruction. If it is a 15-bit instruction only five digits are required and half the vector remains zero.

The OPTAB routine is used for decoding opcodes. OPTAB contains tables of possible opcodes and pseudo-opcodes. The opcode cannot be completely decoded until the syntax of the operand is discovered in the SCAN or FIRST routines. The opcode depends on the registers, variables or constants in the operand field. Many opcodes have registers combined into their format. The number and type of these registers must be decoded in this routine.

Pseudo-opcodes set flags and cause the assembler to do certain functions rather than produce any code. For example,

TRACEON sets a flag indicating that the following instructions are to be traced when they are executed by the interpreter.

The E$$$$E routine is called whenever an error or warning is found. The message is printed out and a flag is set to indicate that an error has occurred.

STCODE is the routine that translates the vector of digits into a 15 or 30-bit CDC instruction. The length of the instruction is determined by a flag and the appropriate course is taken. There are certain conditions that cause the instruction to be 'forced upper'. This means that, instead of filling a CDC word that already has one or more instructions in it, this instrucion must be placed at the beginning of a CDC word and therefore moved up to the next CDC word in memory. The remaining sections of the CDC word are filled with null opcodes by the assembler.

When the END statement is encountered, literals are taken from their table and put into memory after the program to create the literal pool. Their addresses are recorded so that the instructions concerned can have their addresses resolved. The pointers to other undefined variables in the symbol table are checked. If they are non-zero the address of the instruction must be resolved. If an address remains undefined an error is recorded.

The output is now ready to be printed. Its format is as

close to CDC Compass assembler format as possible.

FIRST is the routine that puts literals and constants into a form that can easily be used by other routines to produce the internal CDC format. Scanning for alphabetic codes must be done to find any modifiers that are present. When this has been completed we must decide whether we have an integer, a floating-point number or a character string so that the proper routine can be called.

Routine FIX1 has two entry points. Entry FIX2 translates the exponent of the floating-point number into a form we can use in the DECMAL routine. Entry FIX forms an integer value and checks the base of the number. If it is base 8 it is converted to base 10.

DECMAL accepts floating-point IBM numbers and translates them into floating-point CDC numbers. The floating-point number is finally put into the literal table.

There is a function called OCTAL that is used to translate base 10 numbers to base 8.

3.2 ALGORITHMS

The algorithms used in this program vary from the fairly simple label scan procedures to the more complicated algorithms used to translate numbers into CDC internal form.

The label and opcode scan algorithms are encountered in MAIN. Scanning consists of a search for special characters

used as delimiters or blanks which mark the end of a field. Error messages are given for illegal characters in the label field. A label must start with an alphabetic character. It is truncated if it is longer than 8 characters. The label is then passed to the symbol table routine and entered in the table with its value, the address of the instruction presently being decoded.

The opcode can be up to 11 characters in length. It is handled in a similar fashion. It is then passed to the opcode table routine to check for valid opcodes. Since some opcodes can be of the form SX.SYMBOL we use a method of equivalencing variables so that the symbol can be referenced by a single variable. This symbol has the same restrictions on it as the labels and variables have that are used in the operand field. The symbol must be assigned a value before it is used or an error message will be printed. The opcodes are broken down into three forms and three tables are used to hold the variety of possible codes. These are single character codes, two character codes and pseudo-opcodes which can be up to eight characters long. Since the machine translation of an instruction depends partly on the symbolic opcode and partly on the operand, the algorithms had to allow the opcode to change when the syntax of the operand was known.

3.2.1 SYMBOLIC ADDRESSES AND LITERALS

All tables are initialized at the start of each new batch. Only used entries in the tables are re-initialized for each program in a particular batch. This involves keeping track of the last element in any table and re-initializing the table up to and including that element. This saves time when batch processing.

Figure 2. Organization of the Symbol Table in the Load-and-Go Assembler

SYMBOL	ENTRY NUMBER	VALUE
LABEL1	1	13
LABEL2	2	20
OPERAND	3	-1
*	4	25
	5	-1

The names and uses of the pointers that appear in the SYMTAB routine follow:

FINAL - points to the next available location, 5 in this case.

INIT - points to the symbol being referenced.

The VALUE table is initialized at -1. If it remains -1 the symbol is undefined and JPOINT (the pointer from memory to the symbol table) is set to the value of

INIT. The values in this table are all base 10.

The symbol table shown in Figure 1 is a simple list. The literal table is similar to the symbol table except that any entry may take up more than one place in the table. Linear search is used in both tables to locate an entry, since it was felt that short student programs would be the main source of programs run using this interpreter.

Figure 3. Organization of the Literal Table in the Load-and-Go Assembler

LITERAL	ENTRY NUMBER	POINTER
0102030405	1	5
0607080910	2	5
1112131415	3	5
1617181920	4	5
	5	7

The names and uses of the pointers that are used in the SYMTAB routine follow:

LITEND - points to the next available space (5 above) and is always odd.

I27 - points to the literal being entered into the table.

JPOINT - is set to (I27+1)/2+L500 which is the CDC address for the top of the literal table plus the length of CDC memory.

L500 - is the length of CDC memory.

The literal table shown in Figure 2 is a simple list. The literal table is used to create the literal pool at the end of the translated program. The literal in the table in Figure 2 is the numeric representation of the first 20 members of the alphabet.

The pointers used to resolve addresses at the end of the program may point to either the symbol table or the literal table. These pointers are kept separate by adding a constant to the literal table pointers.

Figure 4. Organization of the Memory and Pointer Vectors

MEMORY		ENTRY NUMBERS		POINTERS	
6120000012	5230000002	0	L500+0	0	0
2112256610	2100615710	1	L500+1	0	0
1161203060	0014046000	2	L500+2	0	0
6140000000	5150000000	3	L500+3	513	4
0	0	4	L500+4	0	0

The memory contains the instructions as in Figure 3. A pointer is non-zero only if the address of the variable or literal is unknown. For example, entry 3 in Figure 3 contains an undefined address. Here the pointer points to a literal table entry. The other pointer at entry L500+3 points to the symbol table. The remaining entries have yet

to be used and are zero.

The pointers to the symbol and literal table are used when the END statement is encountered. A new vector is produced to contain all the pointers to the tables. Thus every entry in the new vector occurs at least once in the pointer vector. The pointer vector has its values changed to point to the corresponding entry in the new vector. The pointer values in the new vector are then changed to the values from the symbol table or the literal addresses to which they point. If the value in the symbol table is undefined an error is printed. The values in the new vector are then converted for entry into the memory vector in the same manner as the instruction, as mentioned below. The pointer vector now links the value in the new vector to the memory vector. The value in the new vector is then added to the proper word in the memory vector. If the instruction in the memory vector started on one-quarter of a CDC word, this must be taken into account and the value from the new vector must be multiplied by the appropriate power of 8 before being added to the memory vector.

3.2.2 INSTRUCTION TRANSLATION

A simple algorithm is used to produce the 30-bit CDC half-word in memory from the 10 element vector of digits that make up the instruction. Each digit is added to the IBM

word of the memory vector after the digit has been multiplied by the appropriate power of 8. In other words MEMORY(I) = DIGIT(10)*8**0 + DIGIT(9)*8**1 + DIGIT(8)*8**2 + DIGIT(7)*8**3 + DIGIT(6)*8**4 + DIGIT(5)*8**5 + DIGIT(4)*8**6 + DIGIT(3)*8**7 + DIGIT(2)*8**8 + DIGIT(1)*8**9. Actually the shift function was used to replace multiplying by powers of 2. This approach was used since 10 digit numbers base 10 greater than 2**31-1 cannot be stored as a single word in the IBM 360/65 computer. Also it is easier to manipulate the digits in the vector rather than in a single word when the operand is decoded.

In scanning the operands only special delimiters, constants, and literals need be distinguished. Variables do not cause any problem. The registers are most difficult to decode since they affect the opcode. The A.SYMBOL type register designator (see Compass Reference Manual) is decoded in a different fashion than with the opcodes. It is broken up at the '.' and the value of the symbol obtained from the symbol table.

To handle expressions the variable and constants are stored in a vector. Their symbol table values are added or subtracted depending on the sign between the given variables. Then their values are broken down into separate digits to be stored in the instruction vector with the opcode, which must be updated depending on the syntax of the

operand.

3.2.3 CONSTANTS AND LITERALS

To decode constants and literals (character strings, integers or floating-point numbers) they must first be identified by the leading digit or '=' sign. Then all modifiers must be found (a list of these modifiers is given in the Compass Reference Manual). The operand is categorized and the numeric modifiers are stored in vectors. Flags for any other modifiers are also kept in vectors. Character strings are designated by special alphabetic character codes. The length of the string must also be known. The procedure to convert a character string begins with a vector of eight IBM zeroed words and the characters are added to a member of the vector one at a time, five to each member, until the end of the string is reached. A character is added by multiplying the member of the vector being used by 64 and adding the value of the character from the table to the member. If the string is designated as a literal it is entered into the literal table; if not, it is a constant and is truncated to three characters.

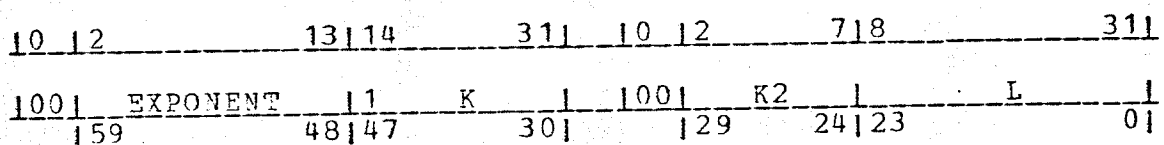
Integers are handled similarly but have many more modifiers. This complicates the problem of conversion. The integer can be from 1 to 20 digits in length. The same vector is used here as is used with the character strings.

The digits are taken one at a time and added to the member of the vector to create a CDC word. This word is made from two IBM words (two members of the vector). The digit is added to the member and then multiplied by the base. The first member must be checked for overflow. The top four bits of the first word are added to the low-order bits of the second member and then masked off the first. This prevents the loss of bits through overflow. If all the digits have been used, the modifiers are then used to finish the translation. If the integer is designated as base 8 with no modifiers, the conversion is simplified. The ten low-order digits are put into the first word by multiplying each digit by the appropriate power of 8. The ten high-order digits are put into the second member in the same fashion.

Floating-point numbers are handled differently. A double precision IBM number is obtained by the routine FIRST. To convert its exponent to base 16, the formula D=(log10(A)+EXP)/log10(16) is used. Here A is the double precision number received from FIRST, EXP is an exponent base 10 from one of the modifiers of the number, and D (the result) is an exponent of 16 such that 16**D=A*10**EXP. The integer part of D is used to form part of the exponent. A is redefined as 16**DFR, where DFR is the fraction part of D. The exponent base 2 is therefore the integer part of D*4 plus the base 2 type modifier. The next step is to break the

double precision value A into component parts. To do this integer K is equivalenced to the first half of A and integer L to the second half of A. Then, since the first eight bits contain the exponent or characteristic, they are masked off and kept in a separate integer minus the bias of 40 (base 16) or 64 (base 10). The masked off exponent is multiplied by 4 and added to D*4 to become part of the base 2 exponent used in the CDC word. K must be normalized since all CDC numbers are stored in a normalized form except when a P type modifier is used (see Compass Reference Manual). K must be shifted to the right until bit 14 of the IBM word is 1 and all bits to the left of bit 14 are zero, leaving room for the exponent to be placed into the high-order bits.

Figure 5. Construction of a CDC Word from Two IBM Words



The first line in the above diagram shows how the IBM words are numbered, with the first two high-order bits being ignored in the CDC word numbered below. This shows how the CDC word is produced in general. The bit pattern lost by shifting K must be saved in integer K2. If the P flag indicates that a modifier P is present the exponent must be set accordingly. The number is not normalized if the P modifier flag is set. For example =1.0P15 is a

floating-point literal with a P modifier whose value is 15. The CDC number in internal representation generated for this number is 1760000000000100000. The integer L must be shifted in the same fashion as K was. The bit pattern saved in K2 is added to the upper portion of L. Any bits lost when L is shifted are not saved. Next the exponent is added to K. If the number produced is supposed to be negative, one's complement notation is used and all the bits in K and L are switched. The value is then stored as a CDC word in the literal table.

2.3 OBSERVATIONS AND REFINEMENTS

Many changes were made to the assembler as progress was made on the project. Algorithms changed as better ones were found. Since the WATFIV compiler was used, testing of algorithms was a fairly simple task. The faster and more efficient algorithms were used in the final implementation. For example, the algorithm to find the CDC exponent was changed to the one described above from one that was found to be more complicated and slower when tested under WATFIV.

CHAPTER 4

PROGRAM ANALYSIS

4.0 TESTING THE I/O ROUTINES

The IBM FORTRAN I/O routines that the FORTRAN H compiler uses to do I/O were found to be the most time consuming part of the load-and-go assembler. The assembler was run using a monitoring program, PROGLOOK, (Appendix G). This program outputs graphs of the percent of CPU time spent in each section of the program being monitored. The length of these sections can be controlled by the user. In testing the load-and-go assembler the section length was 64 bytes. The graphs showed that by far the largest percent of CPU time was spent in the FORTRAN library routines, IHCECOMH, entry point FDIOCS#, and IHCFCVTH, entry points ADCON# and FCVAOUTP. 70 to 75 percent of the CPU time used in the assembler program was spent in these routines. The PROGLOOK program was used to test the load-and-go assembler with the assembly language I/O routines. The results showed that only 8 to 10 percent of the CPU time was spent in these routines with about 5 percent of the CPU time spent in the FORTRAN library routines mentioned above. The reason the assembly language routines are so much faster is that they perform a very specific function rather than being as general as the

FORTRAN routines. The assembly language I/O subroutines were used in the final implementation. This resulted in the load-and-go assembler running from 3 to 5 times faster. If necessary the assembly language routines can be replaced by FORTRAN routines quite easily with a few lines of FORTRAN code.

4.1 A DISCUSSION OF THE FORTRAN H CODE

The code produced by the FORTRAN H compiler was deemed to be very good. While testing the I/O routines the other sections of the load-and-go assembler were also tested. There were no sections that required an extravagant amount of CPU time. An assembly language program was also tested against a FORTRAN H program. Both programs did the same thing. The FORTRAN H program took only minutes to write. The assembly language program had to be rewritten many times to make it only a little faster than the FORTRAN H program. A listing of the assembly language code produced by the FORTRAN H complier showed that the code was very efficient and that it would be very difficult to improve upon the code by rewriting routines in assembly language. It seemed unnecessary to change any routines written in FORTRAN to assembly language since little savings would be gained for the effort involved.

4.2 ASSEMBLY OF A CDC ASSEMBLY LANGUAGE PROGRAM

The CDC program is read one card at a time from unit five, and is stored in a LOGICAL*1 vector, so that each individual character can be tested easily. Labels are found and stored in the symbol table. Opcodes are the second string of characters on the card unless there is no label. They are decoded and the first part of a vector to hold the instruction is produced in routine OPTAB. The next field on the card is the operand field. Literals are easily detected by the '=' sign.

The routine FIRST decodes the syntax and calls TRANS1 to form CDC integer or character representation. DECIMAL is called for decimal numbers. The SYMTAB routine is called to store the literals in the literal table. Control is returned to MAIN1 and then MAIN before STCODE is called to produce the word in memory.

If the operand was not a literal, SCAN2 is called from MAIN1 to decode the operand field. FIRST will be called to decode constants. The instruction vector is completed in the SCAN2 routine before returning to MAIN1 and then MAIN where STCODE is called. With a psudeo-opcode only flags are set when OPTAB is called unless it is the 'END' statement, which causes a call to a section of STCODE to produce all the output the assembler produces except error messages. The routine E$$$E can be called by almost all routines when an

error is encountered. This routine was left out of the flowchart since the further complication seemed unnecessary.

More detailed flowcharts of some of the routines are found in Appendix F.

4.3 FUTURE DEVELOPMENTS

The load-and-go assembler has been written so that it is easy to change table sizes and make additions to the program. If it is desirable to add macros, micros, or more psuedo-opcodes, this can be done without any difficulty.

With the release of IBM's VS/370, the virtual memory operating system, it will be possible to increase the memory size of the interpreter to that of the actual CDC 6600 computer.

CHAPTER 5

CONCLUSIONS

The load-and-go assembler works well. Combined with the interpreter, the assembly system runs under the student-terminal monitor at the University of Manitoba. It is felt that the project has provided what was expected in the way of useful software.

The load-and-go assembler has been tested for more than six months with many programs. Many bugs have been found and fixed and it is hoped that few remain. The use of the program is really just beginning. The interpretive system is to be used in a course to help students learn about the CDC 6600 machine.

The load-and-go assembler is composed of eleven subroutines and one block data routine. There is a total of 1594 FORTRAN statements. The program took about a year and a half to write and test, including the time taken to learn CDC Compass assembly language.

It is hoped that after experimenting with high-level languages when writing this system, people will appreciate the advantages of high-level languages and will make better use of them. It is felt that good compilers are available for both FORTRAN IV and PL1 from IBM for their 360-370 series computers and Burroughs has produced many good

compilers, noteably ALGOL for their B-6500 computer and more good compilers will be available in the future. A good optimizing systems-writing compiler is required at present from IBM; Burroughs has produced such a compiler for their B-6500. Hopefully one will soon be produced, making writing and debugging system programs much easier.

BIBLIOGRAPHY

1 Creech, B. A. "Architecture of the B-6500", Software Engineering, Vol. 1, Julis T. Tou, ed., 1970, Academic Press, New York, pp 29-43.

2 David, E. E. "Design Strategies and Techniques." Software Engineering, P. Naur and B. Randell, ed., Jan. 1969, Nato Science Committee, Garmish, Germany, pp 55-56, conversation.

3 Hassitt, A. Computer Programming and Computer Systems. 1967, Academic Press, New York.

4 Kelso J. H. An Interpreter for the CPU Machine-Code Instructions of the CDC 6400/6500/6600/, A Thesis, Faculty of Science, University of Manitoba, 1972.

5 Knuth, D. E. The Art of Computer Programming, Vol. 1 Addison-wesley, 1968.

6 Lang, C. A. "Languages for Writing System Programs." Software Engineering Techniques, J. N. Buxton and B. Randel, ed., April 1970, Nato Science Committee, Rome, Italy, PP 101-106.

7 McClure, R. M. "Design Strategies and Techniques." Software Engineering, P. Naur and B. Randell, ed., Jan. 1969, Nato Science Committee, Garmish, Germany, p. 58,

conversation.

8 Perlis, A. J. "Software Flexibility."
Software Engineering Techniques, J. N. Buxton and B. Randel, ed., April 1970, Nato Science Committee, Rome, Italy, P 28, conversation.

9 Rosen, Saul. Programming Systems and Languages.
1967, McGraw-Hill Book Company, New York.

10 Thornton, J. E. Design of a Computer, The Control Data 6600. 1970, Scott, Foresman and Company, Glenview, Illinois.

11 Control Data 6400/6500/6600/ Computer Systems Scope Reference Manual. Control Data Corp. Documentation Dept. Palo Alto, California.

12 Control Data 6400/6500/6600/ Computer Systems Reference Manual. No.60100000, Control Data Corp. Technical Publications Dept. St. Paul, Minnesota.

13 Control Data 7600 Computer System, Compass Preliminary Reference Manual. No. 60279900, Control Data Corp. Software Documentation, St. Paul, Minnesota.

14 IBM System/360 Operating System Fortran IV (H) Compiler, Program Logic Manual, No. GY28-6642-4, IBM Corp. Programming Publications, New York.

15 An Introduction to the Spectre Computer.

Department of Applied Analysis and Computer Science, University of Waterloo, Waterloo, Ontario.

A P P E N D I C E S

APPENDIX A

READ/WRITE OPCODES

Since the load-and-go assembler was primarily designed as a teaching aid input/output was introduced in a straight forward manner. "New" opcodes and mnemonics were introduced for read and write. A number may be read from the card reader in any form that is an acceptable literal but without the equal sign.

One number may be punched anywhere on a data card.

A memory location may be printed on the line printer in the following ways:

i) octal

ii) integer (base 10)

iii) real (base 10)

iv) character

v) hexadecimal

General form is 01ijkK

where i - indicates the type of I/O operation

j - Number of consecutive memory locations to be printed

k - the index register Bk

K - a 15-bit address expressiion

PA	K	Print K (character)	(30)	012j0K
PA	K+Bk	Print K+Bk (character)	(30)	012jkK
PI	K	Print K (integer base 10)	(30)	013j0K
PI	K+Bk	Print K+Bk	(30)	013jkK
PD	K	Print K (real base 10)	(30)	014j0K
PD	K+Bk	Print K+Bk (real base 10)	(30)	014jkK
PO	K	Print K (octal)	(30)	015j0K
PO	K+Bk	Print K+Bk (octal)	(30)	015jkK
PH	K	Print K (hexadecimal)	(30)	016j0K
PH	K+Bk	Print K+Bk (hexadecimal)	(30)	016jkK

A card may be read from the card reader using the following instructions:

IN	K	Read into K	(30)	01100K
IN	K+Bk	Read into K+Bk	(30)	0110kK

APPENDIX B

The Pseudo-Opcodes

The following is a list of the pseudo-opcodes that were implemented in the load-and-go assembler. They include two that were introduced to make debugging programs easier for the student (These are TRACEON and TRACEOFF).

PSEUDO-OPCODE	PURPOSE
END	Ends a program and indicates the starting location for execution.
IDENT	Indicates the start of a program.
SET	Sets its label to a specified value.
BSS	Saves storage locations.
TRACEON	Sets trace flag, statements that follow are traced at execution time.
TRACEOFF	Turns off the trace flag.
ORG	Resets the location counter to a specified value.

APPENDIX C

OPERATION CODES

The following is the list of CDC central processor instructions that have been implemented in this load-and-go assembler.

A,B,X	register symbols
i,j,k	register number
K	address expression (18) bits

OCTAL	MNEMONIC	VARIABLE FIELD	LENGTH(bits)
00	PS		30
0100K	RJ	K	30
02i0K	JP	Bi+K	30
030jK	ZR	Xj,K	30
031jK	NZ	Xj,K	30
032jK	PL	Xj,K	30
033jK	NG	Xj,K	30
034jK	IR	Xj,K	30
035jK	OR	Xj,K	30
036jK	DF	Xj,K	30
037jK	ID	Xj,K	30
0400K	ZR	K	30
0400K	EQ	K	30

04i0K	EQ	Bi,K	30
04i0K	ZR	Bi,K	30
04ijK	EQ	Bi,Bj,K	30
05i0K	NZ	Bi,K	30
05i0K	NE	Bi,K	30
05ijK	NE	Bi,Bj,K	30
060jK	LE	Bj,K	30
06i0K	PL	Bi,K	30
06i0K	GE	Bi,K	30
06ijK	GE	Bi,Bj,K	30
06ijK	LE	Bj,Bi,K	30
070jK	GT	Bj,K	30
07i0K	LT	Bi,K	30
07i0K	NG	Bi,K	30
07ijK	LT	Bi,Bj,K	30
07ijK	GT	Bj,Bi,K	30
10ijj	BXi	Xj	15
11ijk	BXi	Xj*Xk	15
12ijk	BXi	Xj+Xk	15
13ijk	BXi	Xj-Xk	15
14ikk	BXi	-Xk	15
15ijk	BXi	-Xk*Xj	15
16ijk	BXi	-Xk+Xj	15
17ijk	BXi	-Xk-Xj	15
20ijk	LXi	jk	15

21ijk	AXi	jk	15
22i0k	LXi	Xk	15
22ijk	LXi	Bj,Xk	15
23ijk	AXi	Bj,Xk	15
24i0k	NXi	Xk	15
24ijk	NXi	Bj,Xk	15
25i0k	ZXi	Xk	15
25ijk	ZXi	Bj,Xk	15
26i0k	UXi	Xk	15
26ijk	UXi	Bj,Xk	15
27ijk	PXi	Bj,Xk	15
30ijk	FXi	Xj+Xk	15
31ijk	FXi	Xj-Xk	15
32ijk	DXi	Xj+Xk	15
33ijk	DXi	Xj-Xk	15
34ijk	RXi	Xj+Xk	15
35ijk	RXi	Xj-Xk	15
36ijk	IXi	Xj+Xk	15
37ijk	IXi	Xj-Xk	15
40ijk	FXi	Xj*Xk	15
41ijk	RXi	Xj*Xk	15
42ijk	DXi	Xj*Xk	15
43ijk	MXi	jk	15
44ijk	FXi	Xj/Xk	15
45ijk	RXi	Xj/Xk	15

46000	NO		15
47ikk	CXi	Xk	15
50ijK	SAi	Aj+K or Aj-K	30
51i0K	SAi	K	30
51ijK	SAi	Bj+K or Bj-K	30
52ijK	SAi	Xj+K or Xj-K	30
53ij0	SAi	Xj	15
53ijk	SAi	Xj+Bk	15
54ij0	SAi	Aj	15
54ijk	SAi	Aj+Bk	15
55ijk	SAi	Aj-Bk	15
56ij0	SAi	Bj	15
56ijk	SAi	Bj+Bk	15
57i0k	SAi	-Bk	15
57ijk	SAi	Bj-Bk	15
60ijK	SBi	Aj+K or Aj-K	30
61i0K	SBi	K	30
61ijK	SBi	Bj+K or Bj-K	30
62ijK	SBi	Xj+K or Xj-K	30
63ij0	SBi	Xj	15
63ijk	SBi	Xj+Bk	15
64ij0	SBi	Aj	15
64ijk	SBi	Aj+Bk	15
65ijk	SBi	Aj-Bk	15
66ij0	SBi	Bj	15

66ijk	SBi	Bj+Bk	15
67i0k	SBi	-Bk	15
67ijk	SBi	Bj-Bk	15
70ijK	SXi	Aj+K or Aj-K	30
71i0K	SXi	K	30
71ijK	SXi	Bj+K or Bj-K	30
72ijK	SXi	Xj+K or Xj-K	30
73ij0	SXi	Xj	15
73ijk	SXi	Xj+Bk	15
74ij0	SXi	Aj	15
74ijk	SXi	Aj+Bk	15
75ijk	SXi	Aj-Bk	15
76ij0	SXi	Bj	15
76i0k	SXi	-Bk	15
76ijk	SXi	Bj+Bk	15
77ijk	SXi	Bj-Bk	15

APPENDIX D

The Assembler I/O Routines

A set of assembler programs is used to do the I/O for the load-and-go assembler. A detailed document (BUFIR - A Set of FORTRAN I/O Routines Using the Basic Sequential Access Method, Gerald Neufeld, April 1971) is available on the use of these routines from the University of Manitoba Computer Centre. The routines have been changed slightly to work in the manner that was required. The entry points used in these routines are OPEN and CLOSE for opening and closing the files used for doing I/O, and BUFIN for reading a card of input, and BUFOUT for printing a line of output.

APPENDIX E

Sample Programs

The programs that follow are CDC Compass language programs with few changes. They give examples of the output produced by the load-and-go assembler. They are some of the test programs used to see how well the assembler program worked. The error in each program was introduced to make sure that none of the programs would execute when they were timed using the PROGLOOK program. This gives a better indication of where the CPU time is being used in the load-and-go assembler.

```
                *  THE LARGEST REAL NUMBER
1                  IDENT
2                  NO

ERROR$$$--- NO OPCODE
3                  I SB1 =377677777777777777777B
4                    PD B1
5                    PS
6                    END I


ATION       CONTENTS                    EXECUTABLE SOURCE STATEMENTS
00     6110C00002                          I SB1 =377677777777777777777B
                  0140100000               PD B1
01     0000000000                          PS
02     377677777777777777777


OMPILE TIME =      0.06 SECS.


CUTION TIME =      0.00 SECS.
```

```
                *    THE LOGICAL OPERATIONS ARE TESTED
1                  IDENT
2                NO

RROR$$$--- NO OPCODE
3                  S  SB1  =10496
4                     SB2  =37452
5                     PO2  B1
6                     SA1  B1
7                     SA2  B2
8                     BX6  X1*X2
9                     SA6  S
0                     PO   S
1                     BX6  X1+X2
2                     SA6  S
3                     PO   S
4                     BX6  X1-X2
5                     SA6  S
6                     PO   S
7                     BX6  -X1
8                     SA6  S
9                     PO   S
0                     BX7  X6
1                     SA7  S
2                     PO   S
3                     BX6   -X1*X2
4                     SA6  S
5                     PO   S
6                     BX6  -X1+X2
7                     SA6  S
8                     PO   S
9                     BX6  -X1-X2
0                     SA6  S
1                     PO   S
2                     PS
3                     END  S


TION       CONTENTS                    EXECUTABLE SOURCE STATEMENTS
0    6110000017                      S  SB1  =10496
               6120000020                SB2  =37452
1    01511C0000                        PO2  B1
               56110                     SA1  B1
                    56220                SA2  B2
2    11612                             BX6  X1*X2
          5160000000                     SA6  S
3    0150000000                        PO   S
               12612                     BX6  X1+X2
4    5160000000                        SA6  S
               0150000000                PO   S
5    13612                             BX6  X1-X2
          5160000000                     SA6  S
6    0150000000                        PO   S
               14611                     BX6  -X1
7    5160000300                        SA6  S
               0150000000                PO   S
0    10766                             BX7  X6
          5170000000                     SA7  S
1    0150000000                        PO   S
               15621                     BX6   -X1*X2
2    5160000000                        SA6  S
               0150000000                PO   S
3    16621                             BX6  -X1+X2
          5160000000                     SA6  S
4    0150000000                        PO   S
```

```
5160CC0000                                  SA6 S
          0150000000                        PO S
0000C00000                                  PS
000000CCCC0000024400
0000CC000C00000111114
```

PILE TIME = 0.41 SECS.

TION TIME = 0.00 SECS.

```
1                IDENT
2                NO

ERROR$$$--- NO OPCODE
3             S  SB1  =6.3
4                SB2  =4.9
5                PD2  B1
6                SA1  B1
7                SA2  B2
8                RX6  X1+X2
9                FX7  X1+X2
0                SA6  S
1                SA7  S+1
2                PD2  S
3                PD2  S
4                DX6  X1+X2
5                SA6  S
6                PD   S
7                PS
8                END  S
```

```
TION      CONTENTS                EXECUTABLE SOURCE STATEMENTS
0    611C000007                      S  SB1  =6.3
          6120000010                    SB2  =4.9
1    0141100000                      PD2  B1
          56110                      SA1  B1
               56220                 SA2  B2
2    34612                           RX6  X1+X2
          30712                      FX7  X1+X2
          5160C00000                 SA6  S
3    5170000001                      SA7  S+1
          0151000000                 PD2  S
4    0141000000                      PD2  S
          32512                      DX6  X1+X2
5    5160CC0000                      SA6  S
          0140000000                 PD   S
6    0000000000                      PS
7    1722623146314631
0    1722471463146314
```

```
MPILE TIME =      0.21 SECS.

UTION TIME =      0.00 SECS.
```

```
                    *   ID AND DF ARE TESTED
1                       IDENT
2                   NO

RROR$$$--- NO OPCODE
3                   S SB1 =377777777777777777778
4                     SB2 =400000000000000000008
5                     SB3 =123456765432123000048
6                     PO3 B1
7                     SA1 B1
8                     SA2 B2
9                     SA3 B3
0                     IR X1,A
1                     JP B
2                   A PA =H/ERROR IR/
3                     JP J
4                   B PA =H/OKAY IR/
5                   J OR X1,C
6                     PA =H/ERROR OR/
7                     JP D
8                   C PA =H/OKAY OR/
9                   D OR X2,E
0                     PA =H/ERROR OR/
1                     JP F
2                   E PA =H/OKAY OR/
3                   F IR X3,G
4                     PA =H/ERROR IR/
5                     JP H
6                   G PA =H/OKAY IR/
7                   H OR X3,I
8                     PA =H/OKAY OR/
9                     JP K
0                   I PA =H/ERROR OR/
1                   K SA4 =177777777777777777778
2                     SA5 =600000000000000000008
3                     SA3 =123456701234567012348
4                     SB1 A4
5                     PO3 B1
6                     ID X4,Z
7                     PA =H/ERROR ID/
8                     JP Y
9                   Z PA =H/OKAY ID/
0                   Y ID X5,X
1                     PA =H/ERROR ID/
2                     JP W
3                   X PA =H/OKAY ID/
4                   W DF X3,V
5                     PA =H/ERROR DF/
6                     JP U
7                   V PA =H/OKAY DF/
8                   U DF X4,T
9                     PA =H/OKAY DF/
0                     JP R
1                   T PA =H/ERROR DF/
2                   R PS
3                     END S

TION      CONTENTS                EXECUTABLE SOURCE STATEMENTS
0     611000000041                      S SB1 =377777777777777777778
            6120000042          66        SB2 =400000000000000000008
1     6130000043                        SB3 =123456765432123000048
            0152100000                  PO3 B1
2     56110                             SA1 B1
          55220                         SA2 B2
```

```
003     0341000004
                0200000005
004     0120000044
                0200000006
005     0120000045
006     0351000010
                0120000046
007     0200000011
010     0120000047
011     0352000013
                0120000046
012     0200000014
013     0120000047
014     0343000016
                0120000044
015     0200000017
016     0120000045
017     0353000021
                0120000047
020     0200000022
021     0120000046
022     5140000050
                5150000051
023     5130000052
                54140
024     0152100000
                0374000026
025     0120000053
                0200000027
026     0120000054
027     0375000031
                0120000053
030     0200000032
031     0120000054
032     0363000034
                0120000055
033     0200000035
034     0120000056
035     0364000037
                0120000056
036     0200000040
037     0120000055
040     0000000000
041     3777777777777777777
042     4000000000000000000
043     1234567654321200004
044     0522221722551122555
045     1713013155112255555
046     0522221722551722555
047     1713013155172255555
050     1777777777777777777
051     6000000000000000000
052     1234567012345670123
053     0522221722551104555
054     1713013155110455555
055     0522221722550406555
056     1713013155040655555
```

```
        IR X1,A
        JP B
A       PA =H/ERROR IR/
        JP J
B       PA =H/OKAY IR/
J       OR X1,C
        PA =H/ERROR OR/
        JP D
C       PA =H/OKAY OR/
D       OR X2,E
        PA =H/ERROR OR/
        JP F
E       PA =H/OKAY OR/
F       IR X3,G
        PA =H/ERROR IR/
        JP H
G       PA =H/OKAY IR/
H       OR X3,I
        PA =H/OKAY OR/
        JP K
I       PA =H/ERROR OR/
K       SA4 =1777777777777777778
        SA5 =6000000000000000008
        SA3 =1234567012345670123
        SB1 A4
        PO3 B1
        ID X4,Z
        PA =H/ERROR ID/
        JP Y
Z       PA =H/OKAY ID/
Y       ID X5,X
        PA =H/ERROR ID/
        JP W
X       PA =H/OKAY ID/
W       DF X3,V
        PA =H/ERROR DF/
        JP U
V       PA =H/OKAY DF/
U       DF X4,T
        PA =H/OKAY DF/
        JP R
T       PA =H/ERROR DF/
R       PS
```


OMPILE TIME = 0.58 SECS.

CUTION TIME = 0.00 SECS.

```
1                       IDENT
2                     NO

RROR$$$--- NO OPCODE
3                     S SA1 =1234567
4                       SA2 =7654321
5                       SB6 A1
6                       PI2 B6
7                       SA3 A1+1
8                       SB6 A3
9                       PI B6
0                       SB3 A1+1
1                       PI B3
2                       SB2 B3-1
3                       PI2 B2
4                       PA =H/STEP1/
5                       SB4 1
6                       SB2 B3-B4
7                       PI B2
8                       SB5 B2+B4
9                       PI B5
0                       SB5 A2-B4
1                       PI B5
2                       SB5 A1+B4
3                       PI B5
4                       SA5 A1+B4
5                       SB6 A5
6                       PI B6
7                       SA5 A2-B4
8                       SB6 A5
9                       PI B6
0                       PA =H/STEP2/
1                       SA5 B3-B4
2                       SB6 A5
3                       PI B6
4                       SA5 B2+B4
5                       SB6 A5
6                       PI B6
7                       SA4 B2+1
8                       SB6 A4
9                       PI B6
0                       SX6 A1+1
1                       SX7 B2+1
2                       SA6 S
3                       SA7 S+1
4                       PI2 S
5                       SB5 X6-1
6                       PI B5
7                       SX6 A2-B4
8                       SA6 S
9                       PI S
0                       PA =H/STEP3/
1                       SA5 X6+B4
2                       SB6 A5
3                       PI B6
4                       SA4 X6+1
5                       SB6 A4
6                       PI B6
7                       SB5 X6+B4
8                       PI B5
9                       SX6 X6+B4
0                       SX7 B3-B4
1                       SA6 S
2                       SA7 S+1
```

```
64          SX7  B2+B4
65          SX6  X6-1
66          SX6  A1+B4
67          SA6  S
68          SA7  S+1
69          PI2  S
70          PS
71          END  S
```

```
ATION      CONTENTS                EXECUTABLE SOURCE STATEMENTS
00    5110000037                  S SA1  =1234567
            5120000040              SA2  =7654321
01    64610                         SB6  A1
            0131600000              PI2  B6
02    5031000001                    SA3  A1+1
            64630                   SB6  A3
03    0130600000                    PI   B6
            6031000001              SB3  A1+1
04    0130300000                    PI   B3
            6123777776              SB2  B3-1
05    0131200000                    PI2  B2
            0120000041              PA   =H/STEP1/
06    6140000001                    SB4  1
            67234                   SB2  B3-B4
07    0130200000                    PI   B2
            66524                   SB5  B2+B4
10    0130500000                    PI   B5
            65524                   SB5  A2-B4
11    0130500000                    PI   B5
            64514                   SB5  A1+B4
12    0130500000                    PI   B5
            54514                   SA5  A1+B4
               64650                SB6  A5
13    0130600000                    PI   B6
            55524                   SA5  A2-B4
               64650                SB6  A5
14    0130600000                    PI   B6
            0120000042              PA   =H/STEP2/
15    57534                         SA5  B3-B4
         64650                      SB6  A5
            0130600000              PI   B6
16    56524                         SA5  B2+B4
         64650                      SB6  A5
            0130600000              PI   B6
17    5142000001                    SA4  B2+1
            64640                   SB6  A4
20    0130600000                    PI   B6
            7061000001              SX6  A1+1
21    7172000001                    SX7  B2+1
            5160000000              SA6  S
22    5170000001                    SA7  S+1
            0131000000              PI2  S
23    6256777775                    SB5  X6-1
            0130500000              PI   B5
24    75624                         SX6  A2-B4
         5160000000                 SA6  S
25    0130000000                    PI   S
            0120000043              PA   =H/STEP3/
26    53564                   69    SA5  X6+B4
         64650                      SB6  A5
            0130600000              PI   B6
27    5246000001                    SA4  X6+1
            64640                   SB6  A4
30    0130600000                    PI   B6
```

```
31      0130500000                      PI  B5
                73664                   SX6 X6+B4
                    77734               SX7 B3-B4
32      5160000000                      SA6 S
                    5170000001          SA7 S+1
33      0131000000                      PI2 S
                73724                   SX7 B2+B4
34      7266777776                      SX6 X6-1
                74614                   SX6 A1+B4
35      5160000000                      SA6 S
                    5170000001          SA7 S+1
36      0131000000                      PI2 S
                0000000000              PS
37      0000000000004553207
40      0000000000003514561
41      2324052034555555555
42      2324052035555555555
43      2324052036555555555

UMPILE TIME =      0.83 SECS.

CUTION TIME =      0.00 SECS.
```

APPENDIX F

The Load-and-Go Assembler Program Listing

The following is a listing of the source statements for the load-and-go assembler

```
      LOGICAL*1 GOFL
      CALL MAIN1(GOFL,&20)
10    CALL MACODE
      CALL ZAPAR(&20)
      GO TO 10
20    STOP
      END
```

```
      BLOCK DATA
C THIS ROUTINE ENABLES YOU TO CHANGE THE SIZE OF TABLES IN ANY ROUTINE
C   BY CHANGING VALUES IN THIS ROUTINE ONLY
C     TABLE,LOCA ARE SYMBOL TABLE VECTORS LENGTH FINAL=LSYM
C LIT LNUM ARE LITERAL TABLE VECTORS LENGTH LITESS=LLIT
C SAVE IS A VECTOR USED TO HOLD VALUES OF SYMBOLS TO BE DEFINED
C                    LENGTH L200
C  IVEC15 IS A VECTOR  WHICH POINTS TO THE SECOND HALF OF 30-BIT
C  OPCODES THAT BEGIN IN  A WORD AFTER A 15-BIT OPCODE LENGTH L50
C  IPOINT IS A VECTOR THAT POINTS TO AN ENTRY IN THE SYMBOL OR LITERAL
C         TABLE LENGTH L1000
C MEMORY IS A VECTOR CONTAINING THE OUTPUT IN CDC FORM LENGTH L1000
C   PRINTO IS A VECTOR CONTAINING THE INPUT STATEMENTS LENGTH L100
C VEC VECTOR HOLDING OPCODE TYPE OF INSTRUCTIONS IN PRINTO VECTOR LENGTH
C     FINAL - LENGTH OF THE SYMBOL TABLE
C     LITESS - LENGTH OF THE LITERAL TABLE
C     Q - - VALUE OF UNDEFINED SYMBOLS IN THE SYMBOL TABLE
C     INS - LENGTH L500
C     I30 - LENGTH OF IVEC15 I30=L50
C     ISAVE - POINTER TO THE END OF MEMORY
C     ISEEK - SEQUENCE NUMBER OF THE INPUT STATEMENTS
C       L1000 - LENGTH OF MEMORY      DIVISIBLE BY 8
C       L500 - =L1000/2          NUMBER OF CDC MEMORY LOCATIONS
C       L200 - LENGTH OF SAVE TABLE
C       L100 -  LENGTH OF INPUT STATEMENT SAVE AREA
C       L50 - LENGTH OF IVEC15
C       LLIT - LENGTH OF LITERAL TABLE
C       LSYM - LENGTH OF SYMBOL TABLE
      COMPLEX*16 PRINTO
      REAL*8 TABLE
      INTEGER SAVE,FINAL,Q
      INTEGER*2 VEC,LNUM,IVEC15,IPOINT
      LOGICAL*1 CHARS(64)
      COMMON /PAGE/PRINTO(5,300)
      COMMON /PAGE2/ISEEK,VEC(100)/AR1/TABLE(100)/CHAR/CHARS/AR2/SAVE(
     &200)/AR3/LOCA(100)
      COMMON /LIMITS/L1000,L500,L200,L100,L50,LLIT,LSYM/AR5/LNUM(200)/
     &AR6/IPOINT(1024)/AR7/IVEC15(50)/LITERL/LITEND,ISAVE,LIT(200)/DATA1
     &/FINAL,LITESS,Q/DATA2/INS,I30
      COMMON MEMORY(1024)
      DATA FINAL/100/,LITESS/200/,Q/-1/
      DATA L1000,L500,L200,L100,L50,LLIT,LSYM/1024,512,200,300,50,200,
     &100/
      DATA INS/512/,I30/50/,CHARS/':','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','0','1','2','3','4','5','6','7','8','9','+','-','*','/','(',')
     &','$','=',' ',''''','.',' ','#',' ','%',',',',''','|','_',' ','?','<'
     &,'>',' ','&','¬',';'/
      END
```

```
      SUBROUTINE MAIN1(GOFL,*)
C-----------------------------------------------------------------------
C
C     CDC COMPASS ASSEMBLER SIMULATOR
C
C
C THIS PROGRAM TRANSLATES A SUBSET OF  CDC COMPASS ASSEMBLER INTO
C     MACHINE CODE WHICH RESEMBLES THE CDC MACHINE CODE
C
C
C  INPUT -- VECTOR OF INPUT CARD IMAGE
C  A$ -- NUMERIC REPRESENTATION VECTORS AND FLAGS
C  ISEEK -- SEQUENCE NUMBER
C  ISLOC -- NUMBER OF WORDS TO BE SAVED
C  JPOINT -- LOCATION OF UNDEFINED SYMBOL IN SYMBOL TABLE
C  IPOINT -- VECTOR CONTAINEING ALL THE VALUES OF JPOINT
C  INTR -- TEMPORARY LOCATION OF LITERALS
C  TABLE -- SYMBOL TABLE
C  LIT -- LITERAL TABLE
C  LOCA -- LOCATION OR VALUES OF SYMBOLS
C  ATTS -- HOLDS VALUES OF ISLOC
C  LNUM -- POINTER FOR LITERAL TABLE
C  SAVE -- FINAL VALUE FOR SYMBOLS
C  ILOC -- MEMORY LOCATION COUNTER
C  INIT -- POINTER TO SYMBOL TABLE LOCATION
C  OUTPUT -- OUTPUT VECTOR
C IBSS -- VALUE FOR BSS STATEMENTS
C
C FLAGS USED IN THIS PROGRAM ARE AS FOLLOWS
C
C     FZAP -- MISSING END STATEMENT
C     TRACEF -- TRACE FLAG
C     LABEL -- LABEL FLAG
C     HWFLG -- QUARTER WORD FLAG
C     STATR -- IDENT STATEMENT FLAG
C     STOPFL -- STOP FLAG
C     ORG -- ORG STATEMENT FLAG
C     BTH -- FLAGS SERIES OF QUARTER WORD OPCODES
C     ENDFLG -- END FLAG
C     FLAG1 -- SET STATEMENT FLAG
C     ERFLG -- FLAGS ERRORS TO STOP EXECUTION
C     IFL1 -- FLAG SYMBOLS FROM LABEL FIELD
C
C-----------------------------------------------------------------------
      COMPLEX*16 PRINTO,SHORT(7),LBL
      REAL*8 BREAL/' '/,MCODE,A$NUMZ,PARAM(3)
      REAL BLANKX,B/' '/,Y(5)
      INTEGER A$VMOD(3)
      INTEGER*2 A$SIGN,A$SMOD(3),VEC
      LOGICAL*1 ENDFLG,INPUT(80),FZAP,A$MOD(3),A$LIT,A$FLAG,A$TYPE,
     &TRACEF,HWFLG,LABEL,START,BL/' '/,BUFFER(133),NCODE(12),$JSWT,SWT(
     &),GOFL,XLOG,'2','3','4','5','6','7','8','9','+','-','*','/','(','
      DATA LBL/' '/,PARAM/'NOSOURCE','NOWARN','NOLIST'/
      EQUIVALENCE (INPUT(1),SHORT(1),BUFFER(21)),(NCODE(1),MCODE),(NCODE
     &(9),BLANKX),(Y(1),BUFFER(1))
      COMMON /NBLT/SWT,XLOG
      COMMON /BLOCK1/A$NUMZ,A$VMOD,A$SMOD,A$SIGN,A$MOD,A$TYPE,A$FLAG,
     &A$LIT/NBL1/START,TRACEF,ENDFLG,FZAP,T/NBL2/HWFLG,LABEL/PT/JPOINT
      COMMON /ALIGN/B7,BUFFER
      COMMON /LIMITS/L1000,L500,L200,L100,L50,LLIT,LSYM
```

```
      COMMON /PAGE/PRINTO(5,1)
      COMMON /PAGE2/ISEEK,VEC(1)
C-----------------------------------------------------------------
C  TABLES AND VARIABLES ARE INITIALIZED FOR RUNNING A PROGRAM
C-----------------------------------------------------------------
      IFTOUT = 6
      IFTIN = 5
      CALL OPEN(IFTIN,0,80)
      CALL OPEN(IFTOUT,1,133)
      FZAP = .FALSE.
      BUFFER(133) = BL
      DO 10 I = 1,7
10    SHORT(I) = LBL
20    CALL STCODE
      CALL $TRTM(TIME)
      T = TIME
      DO 30 I = 1,3
30    SWT(I) = .FALSE.
      GOFL = .FALSE.
      TRACEF = .FALSE.
      ISEEK = 0
      CALL SYMTAB(MCODE,ISEEK,FZAP,BLANKX,&40)
      START = .FALSE.
      $JSWT = .FALSE.
      IF (FZAP) GO TO 50
      ENTRY READ(GOFL,*)
40    CALL BUFIN(IFTIN,INPUT,&320)
C-----------------------------------------------------------------
C   VARIABLES ARE INITIALIZED FOR EACH STATEMENT READ
C-----------------------------------------------------------------
50    IF (INPUT(1) .EQ. 91) GO TO 250
      IF ( .NOT. $JSWT) GO TO 40
60    JPOINT = 0
      IF (GOFL) RETURN
      XLOG = .FALSE.
      HWFLG = .FALSE.
      FZAP = .FALSE.
      LABEL = .FALSE.
      A$LIT = .FALSE.
      A$FLAG = .FALSE.
      A$TYPE = .TRUE.
      DO 70 I = 1,3
      A$VMOD(I) = 0
      A$MOD(I) = .FALSE.
70    A$SMOD(I) = -1
      A$VMOD(2) = -1
      A$NUMZ = 0.0C000000D0
      A$SIGN = -1
      BLANKX = B
      BUFFER(1) = BL
C-----------------------------------------------------------------
C  CHECK FOR COMMENT CARDS
C-----------------------------------------------------------------
      IF (INPUT(1) .EQ. 92) GO TO 220
C-----------------------------------------------------------------
C  STATEMENT AND SEQUENCE NUMBERS ARE PRINTED
C-----------------------------------------------------------------
      ISEEK = ISEEK+1
      IF (SWT(1)) GO TO 100
      J = ISEEK
```

```
      DO 80 I = 1,8
      BUFFER(7-I) = J-J/10*10+240
      J = J/10
      IF (J .EQ. 0) GO TO 90
80    CONTINUE
90    CALL BUFOUT(IFTOUT,BUFFER)
100   IF (ISEEK .GT. L100) CALL ERROR1(13,&20)
      DO 110 I = 1,5
110   PRINTO(I,ISEEK) = SHORT(I)
      VEC(ISEEK) = 0
      N = 1
      I = 3
      N8 = 9
      M = 72
      MCODE = BREAL
      IF (INPUT(1) .EQ. BL .AND. INPUT(2) .EQ. BL) GO TO 150
      IF (INPUT(1) .EQ. 78 .AND. INPUT(2) .EQ. BL) GO TO 190
      IF (INPUT(1) .EQ. BL) N = 2
C-----------------------------------------------------------------
C  SCAN LOCATION FIELD
C-----------------------------------------------------------------
      LABEL = .TRUE.
120   J = 1
      MCODE = BREAL
      DO 140 I = N,M
      IF (INPUT(I) .EQ. BL) GO TO 150
      IF (J .GE. N8) CALL ERRORT(MCODE,&140)
      NCODE(J) = INPUT(I)
      IF (INPUT(I) .EQ. 75) GO TO 130
      IF (INPUT(I) .LT. 193 .OR. (J .EQ. 1 .AND. INPUT(I) .GT. 233))
     &CALL ERROR(MCODE,7)
130   J = J+1
140   CONTINUE
C-----------------------------------------------------------------
C ENTER SYMBOL IN THE SYMBOL TABLE
C-----------------------------------------------------------------
150   DO 160 J = I,M
      IF (INPUT(J) .NE. BL) GO TO 170
160   CONTINUE
      I = M
170   I = J
      IF (I .GT. 35 .AND. N8 .EQ. 9) CALL ERROR1(26,&40)
      IF (N8 .GE. 13) GO TO 200
      IF ( .NOT. LABEL .OR. INPUT(1) .EQ. 78) GO TO 180
      CALL SYMS(MCODE,-1.0,LABEL)
      IF ( .NOT. LABEL) A$NUMZ = MCODE
      IF ( .NOT. LABEL) A$LIT = .TRUE.
C-----------------------------------------------------------------
C    SCAN THE OPERATION FIELD
C-----------------------------------------------------------------
180   N8 = 13
      N = I
      GO TO 120
190   LABEL = .TRUE.
      GO TO 150
C-----------------------------------------------------------------
C  DECODE THE OPCODE AND ENTER IN THE PROPER FIELDS OF THE OUTPUT
C-----------------------------------------------------------------
200   CALL OPTAB(NCODE,MCODE,&40)
      IF ( .NOT. FZAP .AND. A$LIT) CALL ERROR(A$NUMZ,9)
```

```
      A$LIT = .FALSE.
      A$NUMZ = 0.0D0
      FZAP = .FALSE.
      J = I
      IF (J .GT. 35) GO TO 210
C-----------------------------------------------------------------
C SCAN THE VARIABLE FIELD
C-----------------------------------------------------------------
      IF (INPUT(J) .EQ. 126) CALL FIRST(INPUT,J,M,&40,&240)
      CALL SCAN2(INPUT,J,M,&40)
210   IF (START .AND. ENDFLG) FZAP = .TRUE.
      IF (FZAP) GO TO 20
C-----------------------------------------------------------------
C THIS ROUTINE PUTS OUT THE MACHINE CODE
C-----------------------------------------------------------------
      RETURN
      ENTRY ZAPAR(*)
      IF ( .NOT. START) GO TO 20
      GO TO 40
220   IF (SWT(1) .AND. ICC .EQ. 40) GO TO 40
      DO 230 I = 1,5
230   Y(I) = B
      BUFFER(1) = ICC
      ICC = 40
      CALL BUFOUT(IFTOUT,BUFFER(1))
      GO TO 40
240   HWFLG = .FALSE.
      GO TO 210
250   IF (INPUT(2) .EQ. 226) GO TO 320
      IF ($JSWT) GO TO 300
      J = 1
      MCODE = BREAL
      DO 280 I = 16,80
      IF (INPUT(I) .EQ. BL .OR. INPUT(I) .EQ. 107 .OR. J .EQ. 9) GO TO
     &260
      NCODE(J) = INPUT(I)
      J = J+1
      GO TO 280
260   DO 270 I1 = 1,3
      IF (MCODE .EQ. PARAM(I1)) SWT(I1) = .TRUE.
270   CONTINUE
      MCODE = BREAL
      J = 1
      IF (INPUT(I) .EQ. BL) GO TO 290
280   CONTINUE
290   INPUT(1) = 92
      $JSWT = .TRUE.
      ICC = 241
      GO TO 60
300   IF (GOFL) GO TO 310
      CALL $JOB(&310)
310   FZAP = .TRUE.
      IF (GOFL) RETURN 1
      GO TO 20
320   CALL CLOSE(IFTIN)
      CALL CLOSE(IFTOUT)
      RETURN 1
      END
```

```
      SUBROUTINE TRANS1(N,INPUT,JTYPE,NUMZ,IY)
C-----------------------------------------------------------------
C
C THIS ROUTINE TRANSLATES CHARACTER STRINGS INTO NUMERIC REPRESENTATION
C
C-----------------------------------------------------------------
      REAL*8 A$NUMZ
      REAL RY/Z3FFFFFFF/
      INTEGER ZERO/':::::'/,BLANK/' '/,OCTAL,VAL,INTR(8),A$VMOD(3),MASK/
     &Z0FFFFFFF/,X
      INTEGER*2 A$SIGN,A$SMOD(3),TEMP,T
      LOGICAL*1 INPUT(80),VALUE(40),IY,A$MOD(3),A$FLAG,A$LIT,A$TYPE,
     &CHARS(64)
      EQUIVALENCE (RX,IXX),(TEMP,INTR(2))
      COMMON /BLOCK2/VALUE,I$/CHAR/CHARS/BLOCK4/INTR/XXXXY/IXX/BLOCK1/
     &A$NUMZ,A$VMOD,A$SMOD,A$SIGN,A$MOD,A$TYPE,A$FLAG,A$LIT
C-----------------------------------------------------------------
C
      ENTRY TRANS(N,INPUT,JTYPE,NUMZ,IY)
      DO 10 IO = 1,8
10    INTR(IO) = 0
      IF (IY) GO TO 20
      L = 1
C-----------------------------------------------------------------
CHECK FOR VALUE IN A$NUMZ AND VECTOR VALUE
C-----------------------------------------------------------------
      IF (A$FLAG) GO TO 190
      K = I$-1
      NUMZ = K
      GO TO 40
20    L = N+1
      K = N+NUMZ
C-----------------------------------------------------------------
C PUT THE CHARACTER STRING INTO VECTOR VALUE
C-----------------------------------------------------------------
      DO 30 I = L,K
30    VALUE(I-L+1) = INPUT(I)
      K = K-L+1
      L = 1
40    KN = (K+9)/10*10
      IF (K/10*10 .EQ. K) GO TO 60
      VAL = ZERO
C-----------------------------------------------------------------
C CHECK THE TYPE OF THE STRING
C-----------------------------------------------------------------
      IF (JTYPE .EQ. 4 .OR. JTYPE .EQ. 5) VAL = BLANK
      IF (JTYPE .GT. 4) GO TO 140
      K1 = K+1
      DO 50 I = K1,KN
50    VALUE(I) = VAL
60    K = KN
      DO 90 I = L,K
      VAL = VALUE(I)
C-----------------------------------------------------------------
C BUILD THE INTEGER REPRESENTATION OF THE STRING
C-----------------------------------------------------------------
      DO 70 J = 1,64
      IF (VAL .EQ. CHARS(J)) GO TO 80
70    CONTINUE
80    INTR((I-L)/5+1) = INTR((I-L)/5+1)*64+J-1
```

```
 90     CONTINUE
C-------------------------------------------------------------------
C TEST AND SET A$ SYMBOLS
C-------------------------------------------------------------------
100     IF (A$SIGN .EQ. 1) GO TO 170
110     A$TYPE = .TRUE.
        A$VMOD(1) = (K-L)/5+1
        IF ( .NOT. A$LIT) GO TO 120
        IXX = -1
        RETURN
C-------------------------------------------------------------------
C GET A VALUE FOR FUTURE PLACEMENT IN THE SYMBOL TABLE
C-------------------------------------------------------------------
120     INTR(2) = INTR(2)*4096
        IF (NUMZ .GT. 3) CALL ERROR0(15,&130)
130     RX = AND(RY,INTR(2))
        IF (JTYPE .LE. 4) IXX = INTR(1)
        IXX = IXX/4096
        RETURN
C-------------------------------------------------------------------
CRIGHT JUSTIFIED CHARACTER STRINGS
C-------------------------------------------------------------------
140     DO 150 I = 1,K
150     VALUE(KN-I+1) = VALUE(K-I+1)
        KN1 = KN-K
        DO 160 I = 1,KN1
160     VALUE(I) = VAL
        GO TO 60
170     KN = (K-L)/5+1
        DO 180 I = 1,KN
180     INTR(I) = 1073741823-INTR(I)
        GO TO 110
C-------------------------------------------------------------------
C VALUE OF INTEGER LITERALS
C-------------------------------------------------------------------
190     K = 10
        IF ( .NOT. A$TYPE .AND. .NOT. A$MOD(1) .AND. .NOT. A$MOD(3)) GO TO
      &310
        I = 1
        M = 8
CHECK   TYPE FOR BASE
        IF (A$TYPE) M = 10
200     N = 1
210     T = TEMP/2**12
        IF (T .EQ. 0) GO TO (220,230,250,280),N
        INTR(1) = INTR(1)+T
        INTR(2) = LAND(MASK,INTR(2))
        GO TO (220,230,250,280),N
C    MULTIPLY BY BASE
220     INTR(2) = INTR(2)*M
        INTR(1) = INTR(1)*M
        N = 2
        GO TO 210
C     ADD NEW DIGIT FOR VALUE
230     INTR(2) = INTR(2)+VALUE(I)-240
        IF ( .NOT. A$TYPE .AND. (VALUE(I)-240 .GT. 7 .OR. VALUE(I) .LT. 0)
      &) CALL ERROR0(25,&240)
240     N = 3
        GO TO 210
C     INCREASE COUNTER I AND CHECK IF ANY DIGITS REMAIN IN VALUE
```

```
250     I = I+1
        IF (I .LT. I$) GO TO 200
        N1 = 3
        M1 = 1
        M = 10
260     IF (A$VMOD(N1) .EQ. 0) GO TO (290,300),M1
        I = 1
        N = 4
C    MULTIPLY BY THE EXPONENT
270     INTR(2) = INTR(2)*M
        INTR(1) = INTR(1)*M
        GO TO 210
CHECK   IF MODIFIER IS FINISHED
280     I = I+1
        IF (I .LE. A$VMOD(N1)) GO TO 270
        GO TO (290,300),M1
290     M1 = 2
        N1 = 1
        M = 2
        GO TO 260
C       FINAL FIXUP OF INTEGER LITERALS
300     X = INTR(1)/4
        INTR(2) = (INTR(1)-X*4)*2**28+INTR(2)
        INTR(1) = X
        INTR(1) = LAND(RY,INTR(1))
        INTR(2) = LAND(RY,INTR(2))
        GO TO 100
310     I$1 = I$-1
        IF (I$1 .GT. 10) I$1 = 10
        M = 1
        DO 320 I = 1,I$1
        INTR(2) = INTR(2)+(VALUE(I$-I)-240)*M
        IF (I$-10-I .GT. 0) INTR(1) = INTR(1)+(VALUE(I$-10-I)-240)*M
320     M = M*8
        GO TO 100
        END
```

```
      SUBROUTINE SYMTAB(TEST,VALUE,IFL1,XVAT,*)
C-----------------------------------------------------------------
C
C  SYMTAB IS A SYMBOL TABLE ROUTINE CALLED ONCE FOR INITIALIZATION
C
C-----------------------------------------------------------------
      REAL*8 TABLE,TEST,A$NUMZ
      REAL B/' '/
      INTEGER VALUE,FINAL,INTR(8),LIT,SAVE,LOCA,IB/' '/,OCTAL,A$VMOD(3)
     &Q
      INTEGER*2 LNUM,A$SMOD(3),A$SIGN
      LOGICAL*1 FIRCHR,IFL1,A$MOD(3),A$FLAG,A$TYPE,A$LIT
      COMMON /BLOCK4/INTR/PRINTX/ILOC,IPLOC/BLOCK1/A$NUMZ,A$VMOD,A$SMOD
     &A$SIGN,A$MOD,A$TYPE,A$FLAG,A$LIT/PT/JPOINT/AR1/TABLE(1)/AR2/SAVE(
     &1)/AR3/LOCA(1)/AR5/LNUM(1)/LITERL/LITEND,ISAVE,LIT(1)
      COMMON /LIMITS/L1000,L500,L200,L100,L50,LLIT,LSYM/DATA1/FINAL,
     &LITESS,Q
      EQUIVALENCE (TEST,FIRCHR)
C-----------------------------------------------------------------
C
C INITIALIZE TABLES AND POINTERS
C
C-----------------------------------------------------------------
      DO 10 I = 1,FINAL
      TABLE(I) = B
10    LOCA(I) = Q
      DO 20 I = 1,LITESS
      LIT(I) = IB
20    LNUM(I) = I+1
      LITEND = 1
      LNUM(LLIT) = 1
      XVAT = -1.0
      FINAL = 1
      RETURN
C-----------------------------------------------------------------
C  SYMTRA IS THE ENTRY POINT FOR LOADING SYMBOLS AND ITS VALUE
C-----------------------------------------------------------------
      ENTRY SYMTRA(TEST,XVAT)
C-----------------------------------------------------------------
C SYMVAL IS THE ENTRY POINT FOR FINDING THE VALUE OF A SYMBOL
C-----------------------------------------------------------------
      ENTRY SYMVAL(TEST,VALUE)
      IFL1 = .FALSE.
C-----------------------------------------------------------------
C ENTRY POINT FOR LABELS
C-----------------------------------------------------------------
      ENTRY SYMS(TEST,XVAT,IFL1)
      INIT = 1
30    IF (TABLE(INIT) .EQ. TEST) GO TO 40
      INIT = INIT+1
      IF (INIT .LT. FINAL) GO TO 30
C-----------------------------------------------------------------
C    ENTER SYMBOL
C-----------------------------------------------------------------
      TABLE(FINAL) = TEST
C-----------------------------------------------------------------
C   ENTER VALUE
C-----------------------------------------------------------------
      LOCA(FINAL) = XVAT
      INIT = FINAL
```

```
C-----------------------------------------------------------------
C   UPDATE POINTERS
C-----------------------------------------------------------------
2         FINAL = FINAL+1
3         IF (FINAL .GT. LSYM) CALL ERROR1(13,&70)
C-----------------------------------------------------------------
C  OBTAIN VALUE FROM TABLES
C-----------------------------------------------------------------
4     40  INA = ILOC+IPLOC-2
5         IF (IFL1 .AND. LOCA(INIT) .NE. Q) IFL1 = .FALSE.
6         IF (IFL1 .AND. LOCA(INIT) .EQ. Q) LOCA(INIT) = INA
7         VALUE = LOCA(INIT)
8         XVAT = -1.0
9         IF (LOCA(INIT) .EQ. Q .AND. (FIRCHR .GT. 240 .OR. FIRCHR .LT. 193
        &) CALL ERROR(TEST,7)
0         IF (LOCA(INIT) .EQ. Q) JPOINT = INIT
1         RETURN
C-----------------------------------------------------------------
C ENTRY POINT FOR SET STATEMENT VALUES
C-----------------------------------------------------------------
2         ENTRY SYMSET(XVAT,*)
3         LOCA(INIT) = XVAT
4         RETURN 1
C-----------------------------------------------------------------
C ENTRY POINT  FOR LITERALS
C-----------------------------------------------------------------
5         ENTRY SYMLIT
6         J = 1
7         I8 = A$VMOD(1)
8         IF (I8 .EQ. 0) RETURN
9         I27 = 1
0         JPOINT = (I27+1)/2+L500
1     50  IF (LIT(I27) .EQ. INTR(J)) GO TO 80
2         J = 1
3         I27 = LNUM(I27)
4         JPOINT = (I27+1)/2+L500
5         IF (I27 .LT. LITEND) GO TO 50
6         IF (LITEND/2*2 .EQ. LITEND) LITEND = LITEND+1
7         IF (LITEND+I8 .GT. LLIT) CALL ERROR1(13,&70)
C-----------------------------------------------------------------
C  PUT LITERAL IN THE TABLE 'LIT'
C-----------------------------------------------------------------
8         DO 60 I7 = 1,I8
9     60  LIT(I7+LITEND-1) = INTR(I7)
0         LNUM(LITEND) = LITEND+I8
1         JPOINT = (LITEND+1)/2+L500
2         LITEND = LITEND+I8
3     70  RETURN
4     80  IF (J .GE. I8) GO TO 70
5         J = J+1
6         I27 = I27+1
7         GO TO 50
C-----------------------------------------------------------------
C
C ENTRY POINT FOR DEFINING SYMBOLS
C
C-----------------------------------------------------------------
8         ENTRY SYMDEF(J)
9         DO 120 I = 1,J
0         IF (SAVE(I) .LT. L500) GO TO 100
```

```
1           SAVE(I) = ISAVE+SAVE(I)-L500-2
2           SAVE(I) = OCTAL(SAVE(I))
3           GO TO 120
          C-------------------------------------------------------------
          C CHECK FOR THE UNDEFINED SYMBOLS
          C-------------------------------------------------------------
4         90    CALL ERROR(TABLE(SAVE(I)),5)
5           SAVE(I) = 0
6           GO TO 120
7         100   IF (LOCA(SAVE(I)) .EQ. Q) GO TO 90
8         110   SAVE(I) = OCTAL(LOCA(SAVE(I)))
9         120   CONTINUE
0           LITESS = LITEND
1           RETURN
2           END
```

```
      SUBROUTINE SCAN2(INPUT,N,M,*)
C-----------------------------------------------------------------
C
C THE SCAN2 ROUTINE SCANS THE VARIABLE FIELD
C
C  ISIGN -- HOLDS SIGN OF SYMBOL
C  SAVE -- HOLDS SYMBOLS USED IN THE STATEMENT BEING SCANNED
C  ABX -- REGISTER NAMES
C  WREG -- HOLDS REGISTERS
C IVAL8  -- CHARACTERS OF THE PRESENT SYMBOL
C  IVAL -- HOLDS SYMBOL
C  IXS -- + OR - SIGN BEFORE REGISTER OR SYMBOL
C  IXX -- VALUE FOR SYMBOL
C
COUNTERS
C
C     LP -- NUMBER OF SYMBOLS IN THE STATEMENT
C     ID -- NUMBER OF CHARACTERS IN SYMBOL
C     IMEM -- MEMORY LOCATION COUNTER
C
C FLAGS
C
C     REG -- REGISTER FLAG
C     DREG -- REGISTER FLAG
C
C-----------------------------------------------------------------
      REAL*8 SAVE(4),IVAL,BLA/' '/,A$NUMZ
      INTEGER A$VMOD(3),MASK/ZC0000007/,OCTAL
      INTEGER*2 ISIGN(4),OUTPUT(10),A$SMOD(3),A$SIGN
      LOGICAL*1 LABEL,FIVE,NUSWT
      LOGICAL*1 DEL(10)/'+','-','.',',',' ','/','*','(',')','='/,HWFLG,
     &ABX(3)/'A','B','X'/,DREG,FLAG1,B/' '/,IVAL8(8),INPUT(80),A$MOD(3),
     &A$FLAG,A$TYPE,A$LIT
      EQUIVALENCE (IVAL8(1),IVAL),(RI,IDR)
      COMMON /NBL3/OUTPUT,IDK,FLAG1/NBL2/HWFLG,LABEL/XXXXY/IXX/PRINTX/
     &IMEM,ILMEM/BLOCK1/A$NUMZ,A$VMOD,A$SMOD,A$SIGN,A$MOD,A$TYPE,A$FLAG,
     &A$LIT
C-----------------------------------------------------------------
C VARIABLES USED IN THIS ROUTINE ARE INITIALIZED
C-----------------------------------------------------------------
      DO 10 I = 1,4
10    SAVE(I) = BLA
      IVAL = BLA
      I = 0
      IXS = 0
      IXX = -1
      LP = 0
      JID = 0
      ID = 1
      L = 1
      K = 2
      I10 = 10
      NUSWT = .FALSE.
      FIVE = .FALSE.
      DREG = .FALSE.
      DO 20 J = N,M
      IF (INPUT(J) .NE. B) GO TO 30
20    CONTINUE
30    N = J
C-----------------------------------------------------------------
```

```
C SCAN THE FIELD FOR CERTAIN SYMBOLS
C-----------------------------------------------------------------------
      IF (N .GT. 35) RETURN
      DO 440 J = N,M
      IF (J .LT. JID) GO TO 440
      IF (ID .EQ. 1 .AND. INPUT(J) .GE. 240) GO TO 90
      INPUT1 = INPUT(J)
      DO 40 I = 1,10
      IF (INPUT1 .EQ. DEL(I)) GO TO (60,60,220,70,70,70,70,70,70,70),I
40    CONTINUE
50    IF (ID .GT. 8) CALL ERRORT(IVAL,&430)
      IVAL8(ID) = INPUT1
      ID = ID+1
      GO TO 430
C-----------------------------------------------------------------------
C SIGN OF OPERAND
C-----------------------------------------------------------------------
60    A$SIGN = I-1
70    IF (ID .EQ. 3) GO TO 230
      IF (NUSWT) GO TO 210
      IF (INPUT(J-1) .EQ. 64 .AND. I .NE. 7) GO TO 430
80    IF (IVAL8(1) .EQ. DEL(5) .AND. I .EQ. 5 .AND. INPUT(J-1) .NE. 92)
     &CALL ERROR1(10,&450)
      IF (IVAL8(1) .EQ. DEL(5)) GO TO 180
C
C CHECK FOR SYMBOL OR NUMBER
      IF (FLAG1) CALL ERROR1(10,&540)
      IF (IVAL8(1) .LT. 240) GO TO 120
90    JID = J-ID+1
C-----------------------------------------------------------------------
C FIND THE VALUE OF THE NUMBER
C-----------------------------------------------------------------------
      NUSWT = .TRUE.
      CALL FIRST(INPUT,JID,M,&100,&110)
C-----------------------------------------------------------------------
C TRANSLATE THE NUMBER INTO BASE 8 FORM
C-----------------------------------------------------------------------
100   IF (A$FLAG) CALL FIX(XZVAL)
      IXX = XZVAL
C  ENTER VALUE OF SYMBOL FROM SET STATEMENTS IN THE SYMBOL TABLE
C-----------------------------------------------------------------------
C CHANGE THE TYPE OF SYMBOL TABLE ENTRY TO A NUMBER RATHER THAN A SYMBOL
C PLACE THE SYMBOL IN THE TABLE
C-----------------------------------------------------------------------
110   IVAL = IXX
120   XZVAL = IXX
130   IF (FLAG1) CALL SYMSET(XZVAL,&140)
      CALL SYMTRA(IVAL,XZVAL)
      IF ( .NOT. DREG) GO TO 150
C-----------------------------------------------------------------------
C FIND THE VALUE OF THE SYMBOL
C-----------------------------------------------------------------------
      CALL SYMVAL(IVAL,IDREG)
      DREG = .FALSE.
C-----------------------------------------------------------------------
C DETERMINE WHICH REGISTER IS DESIGNATED
C-----------------------------------------------------------------------
      RI = AND(IDREG,MASK)
      IVAL8(2) = IDR+240
      GO TO 260
```

```
  140   RETURN 1
  150   IF (LP .GT. 3) GO TO 470
C-------------------------------------------------------------------
C SAVE THE SYMBOL
C-------------------------------------------------------------------
      IF (INPUT(J-ID) .EQ. 78 .OR. INPUT(J-ID) .EQ. 96 .OR. INPUT(J-ID)
     &.EQ. 64 .OR. INPUT(J-ID) .EQ. 107) GO TO 160
      CALL ERROR1(10,&540)
  160 LP = LP+1
      ISIGN(LP) = 1
      IF (INPUT(J-ID) .EQ. DEL(2)) ISIGN(LP) = -1
      SAVE(LP) = IVAL
C-------------------------------------------------------------------
C CHECK FOR END OF FIELD
C-------------------------------------------------------------------
  170 IF (I .EQ. 5) GO TO 450
      GO TO 420
  180 IF (INPUT(J-1) .EQ. 92 .AND. INPUT(J-2) .LT. 193) GO TO 190
      IF (I .NE. 7) CALL ERROR1(10,&540)
      IF (OUTPUT(1) .NE. 0 .OR. INPUT(J+1) .NE. DEL(5)) GO TO 440
  190 IF (LP .GT. 3) GO TO 470
      XZVAL = IMEM-1
      IF (FLAG1) GO TO 130
      LP = LP+1
      ISIGN(LP) = 1
      IF (INPUT(J-2) .EQ. 96) ISIGN(LP) = -1
      SAVE(LP) = 92
      XZVAL = 0
      CALL SYMTRA(SAVE(LP),XZVAL)
      XZVAL = IMEM-1
      CALL SYMSET(XZVAL,&200)
  200 IF (I .EQ. 7) GO TO 470
  210 NUSWT = .FALSE.
      GO TO 430
C-------------------------------------------------------------------
C CHECK FOR REGISTERS
C-------------------------------------------------------------------
  220 IF (ID .NE. 2) GO TO 50
  230 DO 240 IVAT8 = 1,3
      IF (IVAL8(1) .EQ. ABX(IVAT8)) GO TO 250
  240 CONTINUE
      IF (I .EQ. 3) GO TO 50
      GO TO 80
  250 IF (ID .EQ. 2) GO TO 410
      IF (IVAL8(2) .GE. 240 .AND. IVAL8(2) .LE. 247) GO TO 260
      GO TO 80
C-------------------------------------------------------------------
C SAVE REGISTER
C-------------------------------------------------------------------
  260 IF (L .GT. 3) CALL ERROR1(10,&540)
  270 IF (INPUT(J-ID) .EQ. DEL(2)) IXS = 1
      L = L+1
      IF (OUTPUT(1) .EQ. 1) GO TO 300
      IF (OUTPUT(1) .EQ. 2) GO TO 320
      IF (OUTPUT(1) .EQ. 4 .OR. OUTPUT(1) .EQ. 3) GO TO 310
      IF (OUTPUT(1) .GE. 5) GO TO 350
      IF ((I .NE. 4 .AND. I .NE. 5) .AND. OUTPUT(2) .GT. 2) CALL ERROR1
     &10,&420)
      IF (OUTPUT(1) .EQ. 0 .AND. OUTPUT(2) .EQ. 3 .AND. IVAT8 .EQ. 2)
     &GO TO 280
```

```
      IF (OUTPUT(1)*10+OUTPUT(2) .EQ. 3) GO TO 290
      IF (OUTPUT(1) .EQ. 0) GO TO 400
C      OPCODE 03I
280   OUTPUT(2) = OUTPUT(3)+4
      IF (OUTPUT(3) .GE. 4) CALL ERROR1(10,&540)
      OUTPUT(3) = IVAL8(2)-240
      GO TO 420
C      OPCODE 03I
290   OUTPUT(OUTPUT(2)+1) = IVAL8(2)-240
      GO TO 420
C      OPCODE 1I
300   IF (L .EQ. 3) GO TO 310
      IF (I .NE. 5) OUTPUT(2) = I+1
      IF (I .EQ. 7) OUTPUT(2) = 1
      IF (IXS .EQ. 1) OUTPUT(2) = OUTPUT(2)+4
      OUTPUT(5) = IVAL8(2)-240
C      OPCODE 3I AND 4I
310   IF (IXS .EQ. 1 .AND. L .EQ. 3 .AND. OUTPUT(1) .EQ. 1 .AND. (INPUT(
     &J-ID) .NE. DEL(2))) K = 1
      IF (OUTPUT(1) .EQ. 4 .AND. OUTPUT(2) .EQ. 7) K = 3
      OUTPUT(L+K) = IVAL8(2)-240
      K = 2
      IF (OUTPUT(1) .GE. 3) GO TO 330
      GO TO 420
C      OPCODE 2I
320   OUTPUT(IVAT8+2) = IVAL8(2)-240
      IF (OUTPUT(2) .LE. 1) OUTPUT(2) = OUTPUT(2)+2
      GO TO 420
330   IF (I .LE. 2) OUTPUT(2) = OUTPUT(2)+I-1
      IF (I .GE. 6) GO TO 340
      IF (I .GT. 2 .AND. I .LT. 5) CALL ERROR1(10,&540)
      GO TO 420
C      OPCODE 4I
340   OUTPUT(1) = 4
      IF (I .GT. 7) CALL ERROR1(10,&540)
      IF (OUTPUT(2) .EQ. 4) OUTPUT(2) = 1
      IF (I .EQ. 6) OUTPUT(2) = OUTPUT(2)+4
      GO TO 420
C      OPCODE 5I AND 6I AND 7I
350   IF (IVAT8 .NE. 2) GO TO 390
      OUTPUT(L+2) = IVAL8(2)-240
      IF (L .EQ. 2) OUTPUT(2) = 1
      IF (I .EQ. 5 .AND. LP .LT. 1) GO TO 360
      IF (L .EQ. 2) GO TO 420
360   IF (OUTPUT(2) .EQ. 0) OUTPUT(2) = 4+IXS
      IF (OUTPUT(2) .EQ. 2) OUTPUT(2) = 3
      IF (OUTPUT(2) .EQ. 1) OUTPUT(2) = 6+IXS
      IF (OUTPUT(2) .EQ. 5) GO TO 420
      IF (INPUT(J-ID) .NE. DEL(2) .AND. IXS .EQ. 1) GO TO 370
      GO TO 420
370   IF (OUTPUT(2) .EQ. 7) GO TO 380
      CALL ERROR1(16,&460)
380   OUTPUT(6) = OUTPUT(4)
      OUTPUT(4) = OUTPUT(5)
      OUTPUT(5) = OUTPUT(6)
      OUTPUT(6) = 0
      GO TO 420
390   IF (L .NE. 2) OUTPUT(5) = OUTPUT(4)
      IF (FIVE) CALL ERROR1(10,&420)
      FIVE = .TRUE.
```

```
      OUTPUT(4) = IVAL8(2)-240
      OUTPUT(2) = IVAT8-1
      IF (L .EQ. 3 .OR. (I .EQ. 5 .AND. LP .LT. 1)) GO TO 360
      GO TO 420
400   OUTPUT(L+1+IDK) = IVAL8(2)-240
      IF (IDK .EQ. 1) IDK = -1
      IF (IVAT8 .NE. 2) CALL ERRORD(10,&420)
      GO TO 420
410   IF (I .EQ. 5) GO TO 80
      DREG = .TRUE.
420   ID = 1
      IVAL = BLA
430   IF (I .EQ. 5) GO TO 450
440   CONTINUE
450   IF (LP .LT. 1) HWFLG = .TRUE.
      IF (FLAG1) CALL ERROR1(10,&140)
      IF (OUTPUT(1) .EQ. 0) HWFLG = .FALSE.
      IF (LP .GE. 1) GO TO 470
460   RETURN
C-------------------------------------------------------------------
C
C COMPLETE OUTPUT VECTOR
C
C-------------------------------------------------------------------
470   IF (LP .LT. 1) GO TO 570
      IF (OUTPUT(1) .EQ. 1) CALL ERROR1(10,&540)
      IF (OUTPUT(1) .EQ. 3) CALL ERROR1(10,&540)
      IF (OUTPUT(2) .EQ. 3 .OR. OUTPUT(2) .EQ. 6) GO TO 480
      IF (OUTPUT(1) .EQ. 4) CALL ERROR1(10,&540)
C-------------------------------------------------------------------
C FIND VALUE OF SYMBOL FOR OUTPUT VECTOR
C-------------------------------------------------------------------
480   IZVAL = 0
C-------------------------------------------------------------------
C TOTAL VALUE OF SYMBOLS
C-------------------------------------------------------------------
      JUNDEF = 0
      DO 490 I = 1,LP
      IF (SAVE(I) .NE. BLA) CALL SYMVAL(SAVE(I),IZV1)
      IF (IZV1 .LT. 0 .AND. ISIGN(I) .EQ. -1) CALL ERROR1(10,&540)
      IF (IZV1 .LT. 0) JUNDEF = JUNDEF+1
      IF (IZV1 .LT. 0) IZV1 = 0
490   IZVAL = IZVAL+IZV1*ISIGN(I)
      IF (JUNDEF .GT. 1) CALL ERROR1(10,&540)
      IF (OUTPUT(1) .GE. 2 .AND. OUTPUT(1) .LE. 4) GO TO 560
      IF (IZVAL .GE. 0) GO TO 500
      IZVAL = -IZVAL
      IZVAL = -OCTAL(IZVAL)
      GO TO 510
500   IZVAL = OCTAL(IZVAL)
C-------------------------------------------------------------------
C CHECK FOR A NEGATIVE NUMBER AND FIND ITS COMPLEMENT
C-------------------------------------------------------------------
510   IF (IZVAL .LT. 0) IZVAL = 777777-IABS(IZVAL)
      IF (IZVAL .GE. 777777) GO TO 550
520   DO 530 I = 1,6
      IFIN = IZVAL/10
      OUTPUT(11-I+1) = IZVAL-IFIN*10
      IF (IFIN .EQ. 0) GO TO 540
530   IZVAL = IFIN
```

```
      540   RETURN
      550   CALL ERROR(SAVE(1),2)
            RETURN
      560   HWFLG = .TRUE.
            I10 = 5
            IF (IABS(IZVAL) .GT. 60) CALL ERROR1(18,&540)
            IF (IZVAL .LT. 0) IZVAL = 60+IZVAL
            GO TO 500
      570   IZVAL = IMEM
            HWFLG = .FALSE.
            GO TO 520
            END
```

```
      SUBROUTINE OPTAB(NCODE,MCODE,*)
C-----------------------------------------------------------------
C
C OPTAB  DECODES THE OPERATORS INTO THEIR CODE
C
C  NCODE -- HOLDS CHARACTERS OF OPCODE
C THREE TYPES OF OPCODES CODESA, CODESB AND CODES
C
C-----------------------------------------------------------------
      REAL*8 TEST,CODES(7)/'TRACEON','TRACEOFF','SET     ','END     ','
     &DENT    ','ORG','BSS'/,MCODE,BLANK/' '/
      INTEGER OCTAL,MASK/Z00000007/
      INTEGER*2 ZAR,OUTPUT(10),CODESB(25)/'PS','RJ','JP','ZR','NZ','PL',
     &'NG','IN','PA','PI','PD','PC','PH','NO','EQ','NE','GE','LE','MI',
     &GT','LT','IR','OR','DF','ID'/,IN2C1(25)/0,0,0,0,1,2,3,1,2,3,4,5,6
     &0,0,0,0,0,3,0,0,4,5,6,7/,IN2C(25)/0,1,2,3,3,3,3,1,1,1,1,1,1,6,4,5
     &6,6,3,7,7,3,3,3,3/,INC1(14)/2,2,1,3,0,3,3,3,2,4,2,2,2,4/,INC2(14),
     &0,1,0,6,0,0,2,4,4,7,6,7,5,3/
      LOGICAL*1 CODESA(14)/'L','A','B','I','S','F','D','R','N','C','U',
     &P','Z','M'/,FLAG1,ENDFLG,TRACEF,BL/' '/,NCODE(12),TEST2(8),START,
     &ORG,ERFLG,STOPFL,FZAP
      COMMON /NBL3/OUTPUT,IDK,FLAG1/NBL1/START,TRACEF,ENDFLG,FZAP/NBL4/
     &IBSS,ERFLG,ORG,STOPFL
      EQUIVALENCE (MCODE,ZAR),(TEST2(1),TEST)
C-----------------------------------------------------------------
      FLAG1 = .FALSE.
      ENDFLG = .FALSE.
      IDK = 0
      DO 10 I = 1,8
10    TEST2(I) = NCODE(I+3)
      N = 1
      DO 20 I = 1,10
20    OUTPUT(I) = 0
C-----------------------------------------------------------------
C FIND OPCODE
C-----------------------------------------------------------------
      DO 30 J = 1,14
      IF (NCODE(1) .EQ. CODESA(J)) GO TO 90
30    CONTINUE
40    DO 50 J0 = 1,25
      IF (ZAR .EQ. CODESB(J0)) GO TO 220
50    CONTINUE
60    DO 70 J0 = 1,7
      IF (MCODE .EQ. CODES(J0)) GO TO 280
70    CONTINUE
      IF (MCODE .EQ. BLANK) RETURN
80    CALL ERROR(MCODE,N)
      RETURN 1
C-----------------------------------------------------------------
C CHECK OPCODES
C-----------------------------------------------------------------
90    IF (J .EQ. 5) GO TO 200
      IF (NCODE(2) .NE. 231) GO TO 40
100   IF (NCODE(3) .EQ. 75) GO TO 150
      I = NCODE(3)-240
      IF (I .GE. 0 .AND. I .LE. 7) GO TO 110
      GO TO 80
110   IF (NCODE(4) .NE. BL) GO TO 80
120   IF (J .EQ. 5) GO TO 210
      OUTPUT(1) = INC1(J)
```

```
      OUTPUT(2) = INC2(J)
C-----------------------------------------------------------------
C PUT CODE IN OUTPUT VECTOR
C-----------------------------------------------------------------
130   OUTPUT(3) = I
      IF ( .NOT. START) GO TO 140
      RETURN
C-----------------------------------------------------------------
C  FLAG MISSING IDENT STATEMENTS
C-----------------------------------------------------------------
140   START = .TRUE.
      ERFLG = .FALSE.
      CALL ERRORD(17,&240)
C-----------------------------------------------------------------
C  FIND REGISTER NUMBER
C-----------------------------------------------------------------
150   IF (NCODE(12) .NE. BL) GO TO 80
      IF (NCODE(4) .LT. 240) GO TO 190
      I = 0
      DO 160 JO = 4,11
      IF (NCODE(JO) .LT. 240) GO TO 170
160   I = I*10+NCODE(JO)-240
170   IF (NCODE(JO) .NE. BL) GO TO 80
      I = OCTAL(I)
180   I = LAND(I,MASK)
      GO TO 120
C-----------------------------------------------------------------
C FIND VALUE OF SYMBOLS USED IN THE OPCODE
C-----------------------------------------------------------------
190   CALL SYMVAL(TEST,I)
      GO TO 180
200   IF (NCODE(2) .EQ. 231 .OR. NCODE(2) .EQ. 194 .OR. NCODE(2) .EQ.
     &193) GO TO 100
      GO TO 40
210   OUTPUT(1) = NCODE(2)-188
      IF (OUTPUT(1) .GE. 7) OUTPUT(1) = 7
      OUTPUT(2) = 1
      GO TO 130
220   IF (JO .GE. 8 .AND. JO .LE. 13) GO TO 260
      IF (NCODE(3) .NE. BL) GO TO 60
      OUTPUT(2) = IN2C(JO)
      OUTPUT(3) = IN2C1(JO)
      IF (JO .EQ. 14) GO TO 250
      IF (JO .EQ. 18 .OR. JO .EQ. 20) IDK = 1
230   IF (JO .EQ. 1) STOPFL = .TRUE.
      IF ( .NOT. START) GO TO 140
240   RETURN
250   OUTPUT(1) = 4
      GO TO 230
260   OUTPUT(2) = IN2C(JO)
      IDK = 2
      OUTPUT(3) = IN2C1(JO)
      IF (NCODE(3) .EQ. BL) GO TO 270
      IF (NCODE(3) .LE. 240 .OR. NCODE(3) .GT. 247) GO TO 80
      OUTPUT(4) = NCODE(3)-241
      IF (NCODE(4) .NE. BL) GO TO 80
270   IF ( .NOT. START) GO TO 140
      RETURN
C-----------------------------------------------------------------
C PSUDEO OPCODES  SET UP FLAGS
```

```
C-------------------------------------------------------------------
      280   N = 3
            GO TO (290,300,310,340,320,360,360),J0
      290   TRACEF = .TRUE.
            GO TO 330
      300   TRACEF = .FALSE.
            GO TO 330
      310   FLAG1 = .TRUE.
            FZAP = .TRUE.
            IF ( .NOT. START) GO TO 140
            GO TO 240
            ENTRY $JOB(*)
      320   IF (START) GO TO 350
            START = .TRUE.
            ERFLG = .FALSE.
      330   RETURN 1
      340   IF ( .NOT. START) GO TO 80
            START = .FALSE.
      350   CALL TABFIX
            ENDFLG = .TRUE.
            GO TO 240
      360   ORG = .TRUE.
            IBSS = J0-6
            IF ( .NOT. START) GO TO 140
            GO TO 240
            END
```

```
1           SUBROUTINE E$$$$E(OPCODE,N,*)
      C-------------------------------------------------------------------
      C
      C ERROR ROUTINE
      C
      C-------------------------------------------------------------------
2           COMPLEX*16 ERR(4,26),OUT(9),X,Y,Y1
3           REAL*8 X1(2),OPCODE
4           LOGICAL*1 BUFFER(133),ERFLG,SWT(3),L
5           DATA ERR/' STRING ',' IS NOT AN OPCOD','E. PROBABLE CAUS','E, MIS
           &ING BLANK',' SYMBOL  ',' HAS AN IMPROPER',' VALUE ',' ',' OPCODE
           &,' IS MISPLACED ',2*' ',' ILLEGAL LITERAL',' OR CONSTANT',2*' ','
           &SYMBOL ',' IS UNDEFINED ',2*' ',' OVERFLOW OR UND','ERFLOW CONDIT
           &ON',2*' ',' SYMBOL ',' HAS AN ILLEGAL ','CHARACTER ',' ',' SYMBOL
           &',' IS TRUNCATED ',2*' ',' SYMBOL ','IS ALREADY DEFIN','ED ',' ',
           & ILLEGAL OPERAND',3*' ',' ILLEGAL CONSTAN','T ',2*' ',' ILLEGAL DI
           &CIMAL',' POINT ',2*' ',' TABLE OVERFLOW,',' PROGRAM TOO LAR','GE
           &,' ',' EXPONENT TOO LA','RGE ',2*' ',' CHARACTER STRIN','GS OVER
           & CHARAC','TERS WILL BE TRU','NCATED ',' ILLEGAL SIGN IN',' OPERANI
           & ',2*' ',' MISSING IDENT S','TATEMENT ',2*' ',' ADDRESS OUT OF ',
           &RANGE, OR CONSTA','NT TOO LARGE',' ',' ENTRY POINT IS ','UNDEFINEI
           & ',2*' ',' NO STOP STATEME','NT ',2*' ',' CHARACTER STRIN','G TRU
           &CATED AT 4','0 CHARACTERS ',' ',' SYMBOL    ',' APPEARS TO FOLL','(
           &W A CHARACTER S','TRING DELIMITER ',' SYMBOL    ',' IS ILLEGAL',2*'
           & ',' P CONSTANT IGNO','RED IN INTEGER C','ONSTANTS ',' ','BASE 8 \
           &ALUE','CONTAINS CHARACT','ER OTHER THAN','0-7','NO OPCODE',3*' '/
            DATA Y/'0$$$ERROR$$$--- '/,Y1/'0$$WARNING$$--- '/
            DATA OUT/9*' '/
            DATA IFTOUT/6/
            EQUIVALENCE (X1(1),X),(BUFFER(1),OUT(1))
            COMMON /NBLT/SWT,L/NBL4/IBSS,ERFLG
            ENTRY ERROR(OPCODE,N)
      C-------------------------------------------------------------------
            ERFLG = .TRUE.
            OUT(1) = Y
10          X = ERR(1,N)
            X1(2) = OPCODE
            ERR(1,N) = X
20          DO 30 I = 2,5
30          OUT(I) = ERR(I-1,N)
            CALL BUFOUT(IFTOUT,BUFFER)
            X1(2) = 0.0
            IF (N .LE. 5 .OR. N .EQ. 7 .OR. N .EQ. 9) RETURN
            RETURN 1
            ENTRY ERROR1(N,*)
            ERFLG = .TRUE.
            OUT(1) = Y
            GO TO 20
      C-------------------------------------------------------------------
      C
      C ERRORT IS AN ERROR MESSAGE RCUTINE FOR TRUNCATION
      C
      C-------------------------------------------------------------------
            ENTRY ERRORT(OPCODE,*)
            IF (SWT(2)) RETURN 1
            IF (L) RETURN 1
            L = .TRUE.
            N = 8
            OUT(1) = Y1
            GO TO 10
```

```
      ENTRY ERRORD(N,*)
      IF (SWT(2)) RETURN 1
      IF (L) RETURN
      IF (N .EQ. 25) L = .TRUE.
      OUT(1) = Y1
      GO TO 20
      ENTRY ERROT(OPCODE,N)
      IF (SWT(2)) RETURN
      OUT(1) = Y1
      GO TO 10
      END
```

```
      SUBROUTINE STCODE
C-----------------------------------------------------------------------
C
C STCODE INITIALIZES THE MEMORY FOR OUTPUT OF THE MEMORY WORDS
C
C-----------------------------------------------------------------------
      COMPLEX*16 PRINTO,AD(5),AC(9),LBL,COMPIL,EXECUT
      DATA LBL/' '/,AD/' LOCATION','CONTENTS','          EXECUT','ABLE S
     &OURCE STAT','EMENTS '/
      REAL RT/Z40000000/
      INTEGER CODE,MEMORY,MASK/Z3FFFFFFF/,XX,SAVE,LIT
      INTEGER*2 IPOINT,IVEC15,OUTPUT(10),VEC
      LOGICAL*1 HWFLG,BTH,ENDFLG,TRACEF,LABEL,ORG,START,ERFLG,STOPFL,
     &OUTPUU(5),BUFFER(133),SWT(3),FZAP
      DATA AC/9*' '/,EXECUT/' EXECUTION TIME '/
      DATA IFTOUT/6/,COMPIL/'   COMPILE TIME '/
      COMMON MEMORY(1)
      COMMON /NBL1/SWT
      COMMON /NBL3/OUTPUT/PRINTX/I,IL/NBL1/START,TRACEF,ENDFLG,FZAP,T/
     &NBL2/HWFLG,LABEL/NBL4/IBSS,ERFLG,ORG,STOPFL/PT/JPOINT
      COMMON /LIMITS/L1000,L500,L200,L100,L50,LLIT,LSYM/DATA2/INS,I30/
     &AR2/SAVE(1)/AR6/IPOINT(1)/AR7/IVEC15(1)/LITERL/LITEND,IND,LIT(1)
      COMMON /PAGE/PRINTO(5,1)
      COMMON /PAGE2/ISEEK,VEC(1)
      EQUIVALENCE (CODE,RX),(XX,RR),(AC(1),BUFFER(1))
      DO 10 I2 = 1,INS
      MEMORY(I2) = 0
      MEMORY(I2+L500) = 0
      IPOINT(I2+L500) = 0
10    IPOINT(I2) = 0
      DO 20 I2 = 1,I30
20    IVEC15(I2) = 0
      L99 = L500-1
      I = 1
      I30 = 1
      IL = 1
      ORG = .FALSE.
      HWFLG = .FALSE.
      STOPFL = .FALSE.
      BTH = .FALSE.
      RETURN
C-----------------------------------------------------------------------
C
C MACODE IS CALLED TO CHANGE THE OUTPUT VECTOR INTO ONE 60 BIT WORD
C THIS WORD IS STORED IN 2 MEMORY WORDS 1 AND 501
C
C-----------------------------------------------------------------------
      ENTRY MACODE
      IF (I .GT. L500) CALL ERROR1(13,&230)
      IF (OUTPUT(1) .EQ. 4 .AND. OUTPUT(2) .EQ. 6) HWFLG = .TRUE.
      CODE = 0
C AN OPERATION IS PREFORMED ON THE OUTPUT VECTOR TO GET THE PROPER CODE
      K1 = 1
      KTM = 2**30
30    DO 40 K = K1,10
      KTM = SHFTR(KTM,3)
      IF (OUTPUT(K) .EQ. 0) GO TO 40
      CODE = OUTPUT(K)*KTM+CODE
40    CONTINUE
C-----------------------------------------------------------------------
```

```
C CHECK FLAGS
C-------------------------------------------------------------------
      IF (ORG) GO TO 60
      IF (ENDFLG) GO TO 110
      IF (IL .EQ. 3) GO TO 340
      IU = IL
      IQ = I
      IF (LABEL .AND. IL .EQ. 2) GO TO 90
50    IF (BTH .AND. .NOT. HWFLG .AND. IL .EQ. 2) GO TO 100
      IF (BTH .AND. HWFLG .AND. .NOT. LABEL) GO TO 170
      IF (BTH) MEMORY(IQ-(IU-1)*L99+L99) = MEMORY(IQ-(IU-1)*L99+L99)+
     &19456
      BTH = .FALSE.
      IF (HWFLG) BTH = .TRUE.
      IF (IL .EQ. 2) GO TO 210
C-------------------------------------------------------------------
C  CODE FOR  THE FIRST HALF OPCODES
C-------------------------------------------------------------------
      IF (TRACEF) RX = OR(RT,CODE)
      MEMORY(I) = CODE
      IPOINT(I) = JPOINT
      IL = 2
      VEC(ISEEK) = 1
      IF (HWFLG) VEC(ISEEK) = 2
      GO TO 230
C-------------------------------------------------------------------
C  CODE  FOR ORG AND BSS STATEMENTS
C-------------------------------------------------------------------
60    ORG = .FALSE.
      IF (CODE .LT. 0 .OR. CODE .GE. L500) GO TO 70
      IF (IL .EQ. 2 .AND. BTH) MEMORY(I) = MEMORY(I)+19456
      IF (IL .EQ. 2) MEMORY(I+L500) = 822083584
      IF (IL .EQ. 1 .AND. BTH) MEMORY(I-1+L500) = MEMORY(I-1+L500)+19456
      IF (IL .EQ. 2) I = I+1
      IL = 1
      I = CODE+(I-1)*IBSS+1
      PRINTO(1,ISEEK) = CODE
      VEC(ISEEK) = 8+IBSS
      GO TO 80
70    PRINTO(1,ISEEK) = 0.0
      CALL ERRORD(18,&80)
80    RETURN
C-------------------------------------------------------------------
C  CODE FOR FORCING UPPER
C-------------------------------------------------------------------
90    MEMORY(I+L500) = 822083584
      I = I+1
      IL = 1
      GO TO 50
C-------------------------------------------------------------------
CODING HALF WORD OPCODES FOLLOWING QUARTER WORD CODES
C-------------------------------------------------------------------
100   RR = AND(MASK,CODE)
      VEC(ISEEK) = 7
      CODE = XX
      XX = XX/32768
      MEMORY(I) = MEMORY(I)+XX
      XX = CODE-XX*32768
      XX = XX*32768
      MEMORY(I+L500) = XX
```

```
      IF (JPOINT .NE. 0) IVEC15(I30) = I+L500
      IF (JPOINT .NE. 0) I30 = I30+1
      IF (I30 .GT. L50) CALL ERROR1(13,&230)
      GO TO 220
C-----------------------------------------------------------------
CODE FOR END STATEMENTS
C-----------------------------------------------------------------
110   MEMORY(L1000) = CODE
      IF (CODE .LT. 0) CALL ERROR1(19,&120)
120   IF (ERFLG) CODE = -1
      CALL $TPTM(T)
      AC(1) = COMPIL
130   J = T*100
      DO 140 I = 2,9
140   AC(I) = LBL
      N8 = 27
      DO 150 I = 1,10
      BUFFER(N8-I) = J-J/10*10+240
      IF (I .EQ. 2) N8 = 26
      IF (J .EQ. 0 .AND. I .GE. 3) GO TO 160
150   J = J/10
160   BUFFER(24) = 75
      BUFFER(28) = 226
      BUFFER(29) = 197
      BUFFER(30) = 195
      BUFFER(31) = 226
      BUFFER(32) = 75
      BUFFER(17) = 126
      BUFFER(1) = 96
      CALL BUFOUT(IFTOUT,BUFFER)
      IF (CODE .EQ. -2) RETURN
      CALL $TRTM(TIME)
      T = TIME
      IF (CODE .GE. 0) CALL CDCINT
      CALL $TPTM(T)
      CODE = -2
      AC(1) = EXECUT
      GO TO 130
C-----------------------------------------------------------------
CODING QUARTER WORDS
C-----------------------------------------------------------------
170   CODE = CODE/2**15
      VEC(ISEEK) = 3
      IF (IL-1) 180,180,190
180   JJ = I+L99
      VEC(ISEEK) = 6
      GO TO 200
190   JJ = I
200   MEMORY(JJ) = MEMORY(JJ)+CODE
      BTH = .FALSE.
      GO TO 230
C-----------------------------------------------------------------
CODE SECOND HALF WORD OPCODE
C-----------------------------------------------------------------
210   MEMORY(I+L500) = CODE
      VEC(ISEEK) = 4
      IF (HWFLG) VEC(ISEEK) = 5
220   IPOINT(I+L500) = JPOINT
      IL = 1
      I = I+1
```

```
  230   RETURN
C-------------------------------------------------------------------
C PUT LITERAL TABLE IN MEMORY AFTER END OF PROGRAM AND SET UP ADDRESSES
C-------------------------------------------------------------------
      ENTRY TABFIX
      IF (I .GT. L500) CALL ERROR1(13,&230)
      IF ( .NOT. STOPFL) GO TO 390
  240 IND = I
      IF (IL .EQ. 2) IND = I+1
      LITI = LITEND/2+IND
      INS = LITI+2
      IF (LITI .GT. L500) CALL ERROR1(13,&230)
      IF (LITEND .LE. 1) GO TO 260
      DO 250 J = IND,LITI
      MEMORY(J) = LIT(2*J-2*IND+1)
  250 MEMORY(J+L500) = LIT(2*J-2*IND+2)
  260 LAST = 1
      J1 = 1
      I1 = IND-1
C-------------------------------------------------------------------
C FINDING UNDEFINED SYMBOLS
C-------------------------------------------------------------------
  270 DO 300 J = J1,I1
      IF (IPOINT(J) .EQ. 0) GO TO 300
      LAST1 = LAST-1
      IF (LAST1 .GT. L200) CALL ERROR1(13,&230)
      DO 280 N1 = 1,LAST1
      IF (IPOINT(J) .EQ. SAVE(N1)) GO TO 290
  280 CONTINUE
      SAVE(LAST) = IPOINT(J)
      IPOINT(J) = LAST
      LAST = LAST+1
      GO TO 300
  290 IPOINT(J) = N1
      IF (LAST .EQ. N1) LAST = LAST+1
  300 CONTINUE
      IF (J1 .GT. L500) GO TO 310
      J1 = J1+L500
      I1 = I1+L500
      GO TO 270
  310 LAST = LAST-1
      IF (LAST .EQ. 0) GO TO 410
C-------------------------------------------------------------------
C  DEFINE SYMBOLS
C-------------------------------------------------------------------
      CALL SYMDEF(LAST)
      LD = 1
  320 IZVAL = SAVE(LD)
      IF (SAVE(LD) .LT. 8) GO TO 350
      DO 330 I1 = 1,6
      IFIN = IZVAL/10
      OUTPUT(10-I1+1) = IZVAL-IFIN*10
  330 IZVAL = IFIN
      K1 = 5
      KTM = 2**18
      IL = 3
      CODE = 0
      GO TO 30
  340 SAVE(LD) = CODE
  350 LD = LD+1
```

```
      IF (LD .LE. LAST) GO TO 320
      J1 = 1
      I1 = IND-1
360   DO 380 J = J1,I1
      IF (J .EQ. IVEC15(I30)) GO TO 370
      IF (IPOINT(J) .NE. 0) MEMORY(J) = SAVE(IPOINT(J))+MEMORY(J)
      GO TO 380
370   MEMORY(J) = SAVE(IPOINT(J))*32768+MEMORY(J)
      I30 = I30+1
380   CONTINUE
      IF (J1 .GT. L500) GO TO 410
      J1 = J1+L500
      I1 = I1+L500
      I30 = 1
      GO TO 360
390   MEMORY(I+(IL-1)*L500) = 0
      CALL ERROR0(20,&400)
400   IF (IL .NE. 2) I = I+1
      GO TO 240
410   I1 = LITI
      IF (SWT(3)) GO TO 680
C-----------------------------------------------------------------------
C OUTPUT OF MEMORY
C-----------------------------------------------------------------------
      I = 0
      ISN = 0
      DO 420 J = 1,5
420   AC(J) = AD(J)
      BUFFER(1) = 96
430   CALL BUFOUT(IFTOUT,BUFFER)
      DO 440 J = 1,8
440   AC(J) = LBL
450   ISN = ISN+1
      IF (ISN .GE. ISEEK) GO TO 650
      IVRC = VEC(ISN)
      GO TO (460,460,550,620,620,610,550,630,640),IVRC
      GO TO 450
460   I = I+1
      K = MEMORY(I)
      N1 = 21
470   K = LAND(MASK,K)
      DO 480 J = 1,10
      BUFFER(N1-J) = K-K/8*8+240
480   K = K/8
      GO TO (520,490,580,580,600,600,600,660,670),IVRC
490   N5 = 15
500   DO 510 J = 1,5
      OUTPUU(J) = BUFFER(N5+J)
510   BUFFER(N5+J) = 64
      IF (N5 .EQ. 25) GO TO 580
520   ASSIGN 580 TO JUMP
530   I2 = I-1
      DO 540 J = 1,3
      BUFFER(7-J) = I2-I2/8*8+240
540   I2 = I2/8
      GO TO JUMP, (580,460,620)
550   N5 = 15
560   DO 570 J = 1,5
570   BUFFER(N5+J) = OUTPUU(J)
      IF (IVRC .EQ. 7) GO TO 620
```

```
      580   DO 590 J = 4,8
      590   AC(J) = PRINTO(J-3,ISN)
            GO TO 430
      600   N5 = 25
            GO TO 500
      610   N5 = 25
            GO TO 560
      620   K = MEMORY(I+L500)
            N1 = 31
            GO TO 470
      630   I = PRINTO(1,ISN)
            GO TO 450
      640   I = I+PRINTO(1,ISN)
            GO TO 450
      650   IF (LITEND .LE. 1) GO TO 680
            IVRC = 8
            GO TO 460
      660   IVRC = 9
            ASSIGN 620 TO JUMP
            GO TO 530
      670   CALL BUFOUT(IFTOUT,BUFFER(1))
            IF (I .LT. I1-1) GO TO 650
      680   DO 690 IIF = 1,10
      690   OUTPUT(IIF) = 0
      700   IF (START) CALL ERROR1(19,&120)
            RETURN
            END
```

```
      SUBROUTINE FIRST(INPUT,J,M,*,*)
C-----------------------------------------------------------------
C
C FIRST IS ANOTHER SCAN ROUTINE FOR THE VARIABLE FIELD
C CHARACTER STRINGS AND NUMBERS ARE DECODED
C
C  FLAGS
C     SWT(1) -- NUMBER ENCOUNTERED
C     SWT(2) -- CHARACTER MUST BE MOVED TO VECTOR VALUE
C     SWT(3) -- POST RADIX MODIFIER FOUND
C     SWT(4) -- DECIMAL POINT
C
C-----------------------------------------------------------------
      REAL*8 A$NUMZ,CAD
      INTEGER $IN,ZVALUE(10),IBZ/' '/,A$VMOD(3)
      INTEGER*2 A$SMOD(3),A$SIGN
      LOGICAL*1 SWT(4),ZB/' '/,INPUT(80),A$MOD(3),A$TYPE,A$FLAG,A$LIT,
     &$ALL(16)/'S','P','E','+','-','=','C','Z','L','H','A','R','O','B',
     &D','.'/,VALUE(40)
      EQUIVALENCE (VALUE(1),ZVALUE(1))
      COMMON /BLOCK2/VALUE,I$/BLOCK1/A$NUMZ,A$VMOD,A$SMOD,A$SIGN,A$MOD,
     &A$TYPE,A$FLAG,A$LIT
C-----------------------------------------------------------------
C INITIALIZE VARIABLES
C-----------------------------------------------------------------
      A$LIT = .FALSE.
      A$FLAG = .FALSE.
      A$TYPE = .TRUE.
      DO 10 I = 1,3
      A$VMOD(I) = 0
      A$MOD(I) = .FALSE.
10    A$SMOD(I) = -1
      A$VMOD(2) = -1
      A$NUMZ = 0.0000000000
      A$SIGN = -1
      IF (M .EQ. 80) A$LIT = .TRUE.
      DO 20 I = J,M
      IF (INPUT(I) .NE. ZB) GO TO 30
20    CONTINUE
      I = M
30    J = I
40    I$ = 1
      DO 50 I = 1,10
50    ZVALUE(I) = IBZ
      NPCOL = 0
      IXOB = 0
      NUMZ = 0
      DO 60 I = 1,4
60    SWT(I) = .FALSE.
      DO 140 L = J,M
      INPUT1 = INPUT(L)
      IF (INPUT1 .GE. 240) GO TO 110
      DO 70 INT = 1,16
      IF (INPUT1 .EQ. $ALL(INT)) GO TO (280,280,280,90,90,80,150,150,15
     &,150,150,150,120,120,120,100),INT
70    CONTINUE
      IZ$ = L
      IF (M .EQ. 80 .AND. J .EQ. M) INPUT1 = 240
      IF (M .EQ. 80 .AND. J .EQ. M) GO TO 110
      GO TO 380
```

```
C-------------------------------------------------------------------
C SET A$ SYMBOL FLAGS
C-------------------------------------------------------------------
80      IF (A$LIT) CALL ERROR1(10,&390)
        A$LIT = .TRUE.
        GO TO 130
90      IF (SWT(1)) GO TO 250
        A$SIGN = INT-4
        GO TO 130
C-------------------------------------------------------------------
C FLAG DECIMAL POINT
C-------------------------------------------------------------------
100     IF (SWT(4)) GO TO 390
        IF (SWT(3)) CALL ERROR1(10,&390)
        IF ( .NOT. A$LIT) CALL ERROR1(10,&390)
        NPCOL = L
        SWT(4) = .TRUE.
        GO TO 140
110     SWT(1) = .TRUE.
        IF (I$ .GT. 20) CALL ERROR1(11,&250)
        VALUE(I$) = INPUT1
        I$ = I$+1
        A$NUMZ = A$NUMZ*10+INPUT1-240
        IF ( .NOT. A$LIT .AND. A$NUMZ .GT. 777777.0) CALL ERROR1(18,&390)
        GO TO 130
C-------------------------------------------------------------------
CHECK POST RADIX MODIFIER
C-------------------------------------------------------------------
120     IF (SWT(3)) GO TO 390
        IF (SWT(4)) CALL ERROR1(10,&390)
        IF (INT .LE. 14) A$TYPE = .FALSE.
        SWT(3) = .TRUE.
130     A$FLAG = .TRUE.
140     CONTINUE
        GO TO 380
C-------------------------------------------------------------------
C
C CHARACTER STRINGS
C
C-------------------------------------------------------------------
150     JTYPE = INT-6
        A$FLAG = .FALSE.
        NUMZ = A$NUMZ
        IF (SWT(1)) GO TO 180
        IF ( .NOT. A$LIT) GO TO 390
        $IN = INPUT(L+1)
        IP$ = L+2
160     I$ = 1
C-------------------------------------------------------------------
C SEARCH FOR DELIMITER
C-------------------------------------------------------------------
        DO 170 IZ$ = IP$,M
        IF ($IN .EQ. INPUT(IZ$)) GO TO 220
        IF (I$ .GT. 40) GO TO 200
        VALUE(I$) = INPUT(IZ$)
        I$ = I$+1
170     CONTINUE
        RETURN 2
C-------------------------------------------------------------------
C FIND NUMBER OF CHARACTES IN STRING
```

```
C-----------------------------------------------------------------
180   IF (NUMZ .EQ. 0) GO TO 190
      INPUT1 = INPUT(L+NUMZ+1)
      SWT(1) = .FALSE.
      IZ$ = L+NUMZ+1
      SWT(2) = .TRUE.
      GO TO 230
190   IP$ = L+1
      $IN = ZB
      GO TO 160
200   CALL ERRORD(21,&210)
210   INPUT(IZ$+1) = ZB
      IZ$ = IZ$+1
      GO TO 270
220   INPUT1 = INPUT(IZ$+1)
      IZ$ = IZ$+1
230   IF (INPUT1 .EQ. ZB .OR. INPUT1 .EQ. 107) GO TO 270
      CAD = INPUT1
      CALL ERROT(CAD,22)
240   CALL ERROR1(4,&250)
250   J = L
      IF (A$LIT) CALL ERROR1(11,&260)
260   RETURN 1
270   CALL TRANS(L,INPUT,JTYPE,NUMZ,SWT(2))
C-----------------------------------------------------------------
C  PUT LITERALS IN TABLE 'LIT'
C-----------------------------------------------------------------
      IF (A$LIT .AND. M .LT. 80) CALL SYMLIT
      J = IZ$
      IF (INPUT(IZ$) .EQ. ZB .OR. IZ$ .EQ. 80) RETURN 2
      CAD = INPUT(IZ$)
      CALL ERROT(CAD,23)
      CALL ERROR1(11,&250)
C-----------------------------------------------------------------
C
C NUMERIC NOTATIONS
C
C-----------------------------------------------------------------
280   A$MOD(INT) = .TRUE.
      J$23 = INT
      IF (SWT(4)) A$NUMZ = A$NUMZ/10.**(L-NPCOL-1)
      IXOB = J$23
      L = L+1
      DO 340 I = L,M
      INPUT1 = INPUT(I)
      IF (INPUT1 .GE. 240) GO TO 310
      DO 290 J$ = 1,5
      IF (INPUT1 .EQ. $ALL(J$)) GO TO 320
290   CONTINUE
      GO TO 350
C-----------------------------------------------------------------
C ENTER MODIFIER AND VALUE IN A$ VECTOR
C-----------------------------------------------------------------
300   IF (IXOB .GE. 4) CALL ERROR1(11,&250)
      IF (A$SMOD(IXOB) .NE. -1) CALL ERROR1(11,&250)
      A$SMOD(IXOB) = J$/5
      GO TO 330
310   NUMZ = NUMZ*10+INPUT1-240
      J$ = 6
      GO TO 340
```

```
320     IF (J$ .GE. 4) GO TO 300
        A$VMOD(J$23) = NUMZ
        IF (J$23 .EQ. 2 .AND. IXOB .NE. 6) A$VMOD(2) = -1
        NUMZ = A$VMOD(J$)
        IF (A$MOD(J$)) CALL ERROR1(11,&250)
        A$MOD(J$) = .TRUE.
        J$23 = J$
330     A$FLAG = .TRUE.
340     IXOB = J$
350     A$VMOD(J$23) = NUMZ
        IF (SWT(4) .AND. A$LIT) GO TO 370
        IF (A$MOD(2)) CALL ERRORD(24,&360)
360     IF (SWT(4)) CALL ERROR1(12,&250)
        IZ$ = I
        IF (A$LIT) GO TO 270
        GO TO 390
C-------------------------------------------------------------------------
C  FLOATING POINT LITERALS
C-------------------------------------------------------------------------
370     CALL DECMAL
        J = IZ$
        RETURN 2
380     IF (SWT(4)) GO TO 400
        IF (IZ$ .EQ. 80) INPUT1 = ZB
        IF (A$LIT .AND. INPUT1 .EQ. ZB) GO TO 270
        IF (A$LIT .AND. L .EQ. J+1) GO TO 240
        J = IZ$
390     RETURN
400     A$NUMZ = A$NUMZ/10.**(L-NPCOL-1)
        GO TO 370
        END
```

```
      SUBROUTINE FIX1(XVAL,N)
C-----------------------------------------------------------------------
C
C FIX TRANSLATES NUMBERS INTO BASE 8 NOTATION IF REQUIRED
C
C-----------------------------------------------------------------------
      REAL*8 A$NUMZ,D,DLOG10
      INTEGER OCTAL,A$VMOD(3),X,Y
      INTEGER*2 A$SMOD(3),A$SIGN
      LOGICAL*1 A$MOD(3),A$LIT,A$TYPE,A$FLAG
      COMMON /BLOCK1/A$NUMZ,A$VMOD,A$SMOD,A$SIGN,A$MOD,A$TYPE,A$FLAG,
     &A$LIT
C-----------------------------------------------------------------------
C CHECK IF NUMBERS ARE ALREADY IN BASE
C FIND VALUE ACCORDING TO A$ SYMBOLS
C-----------------------------------------------------------------------
      ENTRY FIX2
      ITOT = 0
      IF (A$NUMZ .EQ. 0.0D0) GO TO 10
      IF ( .NOT. A$MOD(3)) GO TO 10
      IS = 1
      IF (A$SMOD(3) .EQ. 1) IS = -1
      D = (DLOG10(A$NUMZ)+A$VMOD(3)*IS)/1.20411998265925D0
      ITOT = D
      A$NUMZ = 16.0D0**(D-ITOT)
10    IF (A$SMOD(1) .EQ. 1) A$VMOD(1) = -A$VMOD(1)
      A$VMOD(1) = ITOT*4+A$VMOD(1)
      RETURN
      ENTRY FIX(XVAL)
      X = 0
      Y = 0
      IF (A$MOD(1)) Y = A$VMOD(1)
      IF (A$SMOD(1) .EQ. 1) Y = -Y
      IF (A$MOD(3)) X = A$VMOD(3)
      IF (A$SMOD(3) .EQ. 1) X = -X
      XVAL = ((A$NUMZ*10**X)*2**Y)
      IF (A$TYPE) GO TO 50
C-----------------------------------------------------------------------
C CHANGE BASE TO 10
C-----------------------------------------------------------------------
      N = XVAL
      K = 0
      L = 1
      DO 30 I = 1,20
      IF (N-N/10*10 .GT. 7) CALL ERRORD(25,&20)
20    K = (N-N/10*10)*L+K
      L = L*8
      N = N/10
      IF (N .EQ. 0) GO TO 40
30    CONTINUE
40    XVAL = K
50    RETURN
      END
```

```
1         SUBROUTINE DECMAL
      C-----------------------------------------------------------------------
      C
      C FLOATING POINT LITERALS
      C
      C-----------------------------------------------------------------------
2         REAL*8 X,A$NUMZ
3         INTEGER MASK/Z00FFFFFF/,MASK1/Z7FFFFFFF/,INTR(8),MASK2/ZFFFC0000/
         &COMP/Z3FFFFFFF/,A$VMOD(3)
4         INTEGER*2 I,A$SMOD(3),A$SIGN
5         LOGICAL*1 T(8),TEST,A$MOD(3),A$TYPE,A$LIT,A$FLAG
6         EQUIVALENCE (I,T(1)),(T(1),K),(T(5),L),(T(1),X),(NR,RN)
7         COMMON /BLOCK4/INTR/BLOCK1/A$NUMZ,A$VMOD,A$SMOD,A$SIGN,A$MOD,
         &A$TYPE,A$FLAG,A$LIT
      C-----------------------------------------------------------------------
8         CALL FIX2
9         IF (A$VMOD(2) .EQ. -1) A$MOD(2) = .FALSE.
0         X = A$NUMZ
1         K1 = 0
2         L1 = 0
3         IF (X .EQ. 0.0) GO TO 80
      C-----------------------------------------------------------------------
      C FIND IBM EXPONENT
      C-----------------------------------------------------------------------
4         J = (I/(16*16)-64)*4+A$VMOD(1)
5         IF (IABS(J) .GT. 1023) CALL ERROR1(6,&90)
6         K = LAND(MASK,K)
      C-----------------------------------------------------------------------
      C CHANGE FRACTION PART TO CDC FORM
      C-----------------------------------------------------------------------
          K1 = K*256
          J1 = 5
          IF (A$SMOD(2) .EQ. 1) A$VMOD(2) = -A$VMOD(2)
          IF (A$MOD(2)) J1 = A$VMOD(2)+J-43
          IF (A$MOD(2)) GO TO 30
10        IF (K1) 30,20,20
20        K1 = K1*2
          J1 = J1+1
          GO TO 10
30        IF (J1 .GT. 11) CALL ERROR1(6,&90)
          IF (J1 .LE. -20) GO TO 100
          K1 = K/2**(11-J1)
          RN = AND(MASK2,K1)
          IF (NR .NE. 0) CALL ERROR1(6,&90)
          K2 = K-K1*2**(11-J1)
          K2 = K2*2**(19+J1)
          TEST = .FALSE.
          IF (L) 40,50,50
40        L = LAND(MASK1,L)
          TEST = .TRUE.
50        IF (J1 .LE. -18) GO TO 60
          L1 = L/2**(13-J1)
          IF (TEST) L1 = L1+2**(18+J1)
60        L1 = L1+K2
70        IF (L1 .EQ. 0 .AND. K1 .EQ. 0) CALL ERROR1(6,&90)
          IF (A$VMOD(2) .LE. 0 .AND. A$MOD(2)) J1 = J1-1
      C-----------------------------------------------------------------------
      C PUT IN CDC EXPONENT
      C-----------------------------------------------------------------------
          K1 = K1+(J-J1+980)*262144
```

```
C-------------------------------------------------------------------
C CHECK FOR NEGATIVE NUMBERS
C-------------------------------------------------------------------
80      IF (A$SIGN .EQ. 1) K1 = COMP-K1
        IF (A$SIGN .EQ. 1) L1 = COMP-L1
        INTR(1) = K1
        INTR(2) = L1
        A$VMOD(1) = 2
C-------------------------------------------------------------------
C ENTER IN LITERAL TABLE
C-------------------------------------------------------------------
        IF (A$LIT) CALL SYMLIT
90      RETURN
100     K1 = 0
        IF (J1 .LE. -50) GO TO 70
        L1 = K/2**(-J1-19)
        GO TO 70
        END
```

APPENDIX G

Timing of Algorithms

The interpreter was run using a monitoring program called PROGLOOK (available from COSMIC, Barrow Hall, University of Georgia, Athens, Georgia). The program gives timings of the percent of CPU time spent in each section of the interpreter. This points out the algorithms that are using the most time and therefore the ones that need to be optimized to the fullest extent. The programs used in the timing of the interpreter were the ones listed in APPENDIX E. The linkage editor listing for the interpreter is listed here so that the different modules can be distinguished in the timing plots. There are examples of both the FORTRAN I/O routines and the assembly language I/O routines for the same set of input. The histograms have been labelled with the names of the subroutines, that the code timed is located in.

The PROGPLOT for the FORTRAN I/O Routines

MODULE MAP

CONTROL SECTION

NAME	ORIGIN	LENGTH
OPEN	00	C8
CLOSE	C8	B0
BUFIN	178	13C
BUFOUT	2A8	10A
IHCECOMH	3B8	F41
IHCCOMH2	1300	65D
IHCFCVTH	1960	119D
IHCFATH	2B00	512
IHCFFICS	3C18	1378
IHCUOPT	4390	350
IHCERRM	46F0	58C
IHCLATBL	4CA0	198
IHCETRCH	4E38	28E
MAIN	50C8	116
CHAR	51E0	40
LIMITS	5220	1C
DATA1	5240	C
DATA2	5250	8
MAIN1	5258	C4C
TRANS1	5FA8	8F6
SYMTAB	67A0	A2C
SCAN2	7100	1376
OPTAB	8543	894
E$$$$E	8DE0	C6C
STCODE	9A50	174C
FIRST	B1A0	C72

ENTRY

NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
IBCOM#	3B8	FDICCS#	474	INTSWTCH	120E		
SEQCASE	1678						
AUCON#	1960	FCVAOUTP	1A0A	FCVLOUTP	1A9A	FCVZOUTP	1BEA
FCVICUTP	1F98	FCVEOUTP	249A	FCVCOUTP	2684	INT6SWCH	299B
ARITH#	2B00	ADJSWTCH	2E6C				
FICCS#	3018	FIOCSBEP	301E				
ERRMCN	46E0	IHCERRE	46F8				
IHCTRCF	4E38	ERRTRA	4E40				
READ	5E04	ZAPAR	5E60				
TRANS	6708						
SYMTRA	6F6C	SYMVAL	6FDC	SYMS	704C	SYMSET	70D0
SYMLIT	712C	SYMDEF	7170				
$JOB	8D98						
ERROR	987C	ERROR1	98E0	ERRORT	993C	ERRORD	998C
ERROT	99F8						
MACODE	B10C	TABFIX	B154				

CDCINT	C948	57DA
CCTAL	12128	130
JHK	12258	A88

MPASN	12258	MPASNB	12282	DOSHFT	122DA	MPCNVD	1244C
MPMUL	124CC	MPADD	12554	MPCOM	1250A	MPSUB	1264A
MPDIVA	12686	MPADDA	12776	MPMULA	1287A	AND	1290C
CR	12920	ECR	12934	MPDIV	1294A	MPDIVW	12814

$TPTM	12086	$ETMR	12DCA
DLOG10	12F48	DLOG	12F60
FDXPD#	13148		
FIXPI#	132E8		
FRXPI#	13438		
DEXP	13580		
EXIT	13808		

$TRTM *	12CE0	268
IHCLLOG *	12F48	200
IHCFDXPD*	13148	1A0
IHCFIXPI*	132E8	14F
IHCFRXPI*	13438	141
IHCLEXP *	13580	288
IHCFEXIT*	13808	1C
PAGE	13828	5DC0
PAGE2	19SE8	CC
AR1	19688	320
AR2	199D8	320
AR3	19CF8	190
AR5	19E88	190
AR6	1A018	80C
AR7	1A818	64
LITERL	1A880	328
	1ABA8	1000
NBLT	1BBA8	4
BLOCK1	1BBB0	22
NBL1	1BBD8	8
NBL2	1BBE0	2
PT	1BBE8	4
ALIGN	1BBF0	89
PRINTX	1BC80	8
BLOCK2	1BC88	2C
BLOCK4	1BCB8	20
XXXXY	1BCD8	4
NBL4	1BCE0	7
NBL3	1BCE8	19
$PRIVATE	1BD08	00

ENTRY ADDRESS 5CC8
TOTAL LENGTH 1BCC8

```
PROGRAM TIMELINE


TCB=01BA40 NAME= LOWSYS  SEGMENT=  0
TCB=01BA4C NAME=CDC      SEGMENT=  1


END DATA AFTER 11913 RECORDS


 11129 RUN SAMPLES DISCARDED BECAUSE MEASURED SUBTASK WAS DISPATCHABLE
       BUT NOT ACTUALLY RUNNING WHEN SAMPLED.
       IF THIS NUMBER IS LARGE COMPARED TO TOTAL RUN SAMPLES ACCEPTED,
       YOUR SUBTASK WAS HELD UP BY OTHER JOBS OF HIGHER DISPATCHING PRIORITY.

ENTITIES ENCOUNTERED =   3

      1 TCB=  00001BA40 NAME= LOWSYS  HIGHEST SEGMENT=  0 LOAD POINT=  00000000
      2 TCB=  00001BA4C NAME=CDC      HIGHEST SEGMENT=  1 LOAD POINT=  0003DAF8
      3 TCB=  00001BA40 NAME= HIGHSYS HIGHEST SEGMENT=  0 LOAD POINT=  00000000

  91 BUCKETS TO BE SORTED


 ENTITY      IDENTIFIER          NUMBER  NUMBER  TOTAL    RUNNING  WAITING
 NUMBER   TCB     NAME    SEG WAITING RUNNING PERCENT  PERCENT  PERCENT
    1   C1BA4C  LOWSYS     0      56       4    0.50     0.57    65.48
    2   C1BA4C  CDC        1      21     587    5.10    83.86    25.00
    3   C1BA40  HIGHSYS     0       6     109    0.98    15.57     9.52
                           84     700


TOTAL PERCENT OF TIME SUBTASK WAS RUNNING 89.29
```

ABS.	REL.				NUMBER	
03DD40	CCC248	*			1	
03DE40	CCC348	*		BUFOUT	2	
03DE80	CCC388	*			1	
03DFC0	CCC4C8	*			2	
03EC00	CCC508	**			3	
03E040	CCC548	*			1	
03E080	CCC588	*			1	
03E280	CCC788	*			1	
03E340	CCCE48	*		FDIOCS#	1	
03E380	CCCE88	***			2	
03E400	CCC9C8	*			5	
03E440	CCC948	**********		FORTRAN	1	
03E480	CCC988	**********		I/O	88	
03E4C0	CCC9C8	**********		ROUTINES	36	
03F540	CCCA48	*			120	
03F48C	CC1588	*****			1	
03F4C0	CC19C8	**********		ADCON#	9	
03F500	CC1AC8	**********			43	
040B00	CC3CC8	*			172	FCVAOUTP
040B40	CC3C48	*		ADJSWTCH	1	
040D40	CC3248	*			2	
040EC0	CC33C8	*			1	
040E40	CC3348	***			2	
040E80	CC33C8	*			5	
041040	CC3548	*		FIOCSBEP	1	
041100	CC3E08	*			1	
041140	CC3E48	****			1	
041180	CC3688	**			7	
041240	CC3743	*			3	
0414C0	CC39C8	*			1	
041900	CC3EC8	*			1	
041A40	CC3F48	*			1	
041A8C	CC3F08	*			1	
043100	CC5EC8	*			1	
043240	CC5743	*			2	
0432C0	CC57C8	*			1	
043300	CC58C8	**			1	
043340	CC5E40	*		MAIN1	3	
0433C0	CC58C8	*			1	
043540	CC5B43	*			2	
043680	CC5888	*			1	
043300	CC5CC8	*			1	
043B00	CC6CC8	*			1	
043CC0	CC61C8	*		TRANS1	1	
043D00	CC62C8	*			2	
044000	CC65C8	*			2	
044540	CC6A48	*			1	
0446C0	CC6FC8	*		SYMTAB	1	
044A30	CC6F88	*			1	
045040	CC7548	*		SYMTRA	2	
0451C0	CC7EC8	*			1	
045140	CC7E48	**			1	
045180	CC7E86	***			3	
045680	CC7F38	*			5	
045EC0	CC83C8	*			1	
045FC0	CC84C8	*		SCAN2	1	
045FC0	CC84C8	*			1	
046CCC	CC85C8	*			1	
046340	CC8F48	*			1	

0482C0 CCA7C8	*			1
0486C0 CCABC8	*			1
048840 CCAE40	*			2
04888C CCAD88	*		STCODE	2
0489C0 CCADC8	*			1
048900 00AEC8	***			1
048940 CCAE48	*			5
048930 CCAE38	*			1
048CC0 CCB1C8	*			2
049C80 CCB5C8	*			1
049100 CCBEC8	*		FIRST	1
0493C0 CCB8C8	*			1
04FD40 C12248	*		OCTAL	1

The PROGPLOT for the Assembly Language I/O Routines

MODULE MAP

CONTROL SECTION

NAME	ORIGIN	LENGTH
MAIN	00	116
CHAR	118	40
LIMITS	158	1C
DATA1	178	C
DATA2	188	8
MAIN1	190	C4C
TRANS1	CE0	8F6
SYMTAB	16D8	A2C
SCAN2	2108	1376
OPTAB	3480	894
E$$$$E	3D18	C6C
STCODE	4988	174C
FIRST	6DD8	D72
FIX1	6E50	530
DECMAL	7380	500
CDCINT	7880	57DA
CCTAL	DC60	130
JHK	D190	A8B
OPEN	DC18	B16
$TRTM *	E730	268
IHCLLOG *	E998	200
IHCFDXPD*	E898	1A0
IHCFIXPI*	EC38	14F
IHCFRXPI*	EE88	141

ENTRY

NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
READ	D3C	ZAPAR	D98				
TRANS	1640						
SYMTRA	1EA4	SYMVAL	1F14	SYMS	1F84	SYMSET	2008
SYMLIT	2064	SYMDEF	20A8				
$JOB	3CD0						
ERROR	4784	ERROR1	4818	ERRORT	4874	ERRORD	48C4
ERROT	4920						
MACODE	6044	TABFIX	608C				
FIX2	72E0	FIX	7324				
MPASN	D190	MPASNB	D1BA	DOSHFT	D212	MPCNVD	D384
MPMUL	D404	MPADD	D48C	MPCOM	D512	MPSUB	D582
MPDIVA	D5EF	MPADDA	D6AF	MPMULA	D782	AND	D844
OR	D858	ECR	D86C	MPDIV	D882	MPDIVW	DA4C
CLOSE	DEF0	WAIT	DF50	CHECK	E01C	BUFOUT	E078
BUFIN	E228						
$TPTM	F7D6	$ETMR	E81A				
DLOG10	E998	DLOG	E980				
FDXPD#	E898						
FIXPI#	EC38						

IHCCMF2*	FF18	65C					
IHCFCVTH*	10578	119D					
IHCEFNTH*	11718	512					
IHCLEXP *	11C30	2B8					
IHCFEXIT*	11EB8	1C					
IHCEFICS*	11ED8	1378					
IHCERM *	13250	58C					
IHCUOPT *	13810	350					
IHCETRCH*	13860	28E					
IHCUATBL*	13DF0	198					
PAGE	13F88	5DC0					
PAGE2	19C48	CC					
AR1	19E18	320					
AR2	1A138	320					
AR3	1A458	190					
AR5	1A5F8	190					
AR6	1A778	800					
AR7	1AF78	64					
LITERL	1AFE0	328					
	1B3C8	1000					
NBLT	1C3C8	4					
BLOCK1	1C310	22					
NBL1	1C338	8					
NBL2	1C340	2					
PT	1C348	4					
ALIGN	1C350	89					
PRINTX	1C3F0	8					
BLOCK2	1C3E8	2C					
BLOCK4	1C418	2C					
XXXXY	1C438	4					
NBL4	1C440	7					
NBL3	1C448	19					
$PRIVATE	1C468	00					

IHCCMF2*	FF18	FCBC#	F08C	INTSWTCH	FEF6		
SECCASE	10290						
ADCCN#	10578	FCVAOUTP	10622	FCVLOUTP	106B2	FCVZOUTP	10802
FCVICUTP	10880	FCVEOUTP	10B02	FCVCOUTP	112CC	INT6SWCH	115B3
ARITH#	11718	ADJSWTCH	11A84				
DEXP	11C30						
EXIT	11EB8						
FICCS#	11ED8	FIOCSBEP	11EDE				
ERRMCN	13250	IHCERRE	13268				
IHCTRCH	13860	ERRTRA	13B68				

ENTRY ADDRESS 00
TOTAL LENGTH 1C468

****CDC . DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET

PCCT INCREMENT = 000040 HEX

PROGRAM TIMELINE

TCB=010090 NAME= LOWSYS SEGMENT= 0
TCB=010090 NAME=CDC   SEGMENT= 1

END DATA AFTER 681 RECORDS

357 RUN SAMPLES DISCARDED BECAUSE MEASURED SUBTASK WAS DISPATCHABLE
BUT NOT ACTUALLY RUNNING WHEN SAMPLED.
IF THIS NUMBER IS LARGE COMPARED TO TOTAL RUN SAMPLES ACCEPTED,
YOUR SUBTASK WAS HELD UP BY OTHER JOBS OF HIGHER DISPATCHING PRIORITY.

ENTITIES ENCOUNTERED = 3

1 TCB= 00010090 NAME= LOWSYS HIGHEST SEGMENT= 0 LOAD POINT= 00000000
2 TCB= 00010090 NAME=CDC   HIGHEST SEGMENT= 1 LOAD POINT= 00079B98
3 TCB= 00010090 NAME= HIGHSYS HIGHEST SEGMENT= 0 LOAD POINT= 00000000

96 BUCKETS TO BE SORTED

ENTITY NUMBER	IDENTIFIER TCB	NAME	SEG	NUMBER WAITING	NUMBER RUNNING	TOTAL PERCENT	RUNNING PERCENT	WAITING PERCENT
1	010090	LOWSYS	0	63	3	9.69	1.17	94.03
2	010090	CDC	1	3	120	18.06	46.69	4.48
3	010090	HIGHSYS	0	1	134	19.82	52.14	1.49
				67	257			

TOTAL PERCENT OF TIME SUBTASK WAS RUNNING 79.32

```
                (  0)          (  1)          (  3)          (  6)          (  7)
 ABS.    REL.   ---------------------------------------------------------------------- NUMBER
07A140  0005A8  ********************************                                        2
07A180  0005E8  ****************                                                        1
07A1C0  000628  ****************                                                        1
07A240  0006A8  ****************                                                        1
07A280  0006E8  ****************                                                        1
07A2C0  000728  ************************************************                        1
07A300  000768  ********************************                                        3
07A3C0  000828  ****************                                                        2
07A440  0008A8  ********************************                                        1
07A480  0008E8  ****************                                                MAIN1   2
07A4C0  000928  ********************************************************                1
07A500  000968  ****************                                                        3
07A540  0009A8  ****************                                                        1
07A5C0  000A28  ****************                                                        1
07A940  000DA8  ****************                                                        1
07A8C0  001028  ****************                                                ZAPAR   1
07AC40  0010A8  ****************                                                        1
07AC80  0010E8  ********************************************************************    1
07ACC0  001128  ****************                                                        5
07AEC0  001328  ****************                                                        1
07AFC0  001368  ****************                                                TRANS1  1
07B0C0  001528  ****************                                                        1
07B240  0016E8  ****************                                                        1
07B500  001968  ************************************************                        1
07B640  001AA8  ****************                                                        3
07B680  001AE8  ****************                                                SYMTAB  1
07BA00  001E68  ****************                                                        1
07BA80  001EE8  ****************                                                        1
07C000  002468  ****************                                                SYMTRA  1
07C080  0024E8  ****************                                                        1
07C100  002568  ****************                                                        1
07C140  0025A8  ********************************************************************    1
07C180  0025E8  ****************                                                        4
07C1C0  002628  ****************                                                        1
07C340  0027A8  ****************                                                        1
07C400  002868  ****************                                                        1
07C700  002B68  ****************                                                        1
07CC00  003068  ****************                                                SCAN2   1
07CD00  003168  ****************                                                        1
07CE00  003268  ****************                                                        1
07CE80  0032E8  ****************                                                        1
07CFC0  003428  ****************                                                        1
07D000  003468  ****************                                                        1
07D340  0037A8  ********************************                                        1
07D380  0037E8  ****************                                                        2
07D480  0038E8  ****************                                                OPTAB   1
07D600  003A68  ********************************                                        1
07D9C0  003E28  ****************                                                        2
07EA00  004E68  ****************                                                        1
07EA80  004EE8  ****************                                                        1
07EB00  004F68  ****************                                                        1
07EB40  004FA8  ********************************************************************    1
07EC40  0050A8  ****************                                                        4
07ECC0  005128  ****************                                                STCODE  1
07F600  005A68  ****************                                                        1
07F640  005AA8  ****************                                                        1
07F800  005C68  ****************                                                        1
07F880  005CE8  ****************                                                        1
07F8C0  005D28  ************************************************************************1
```

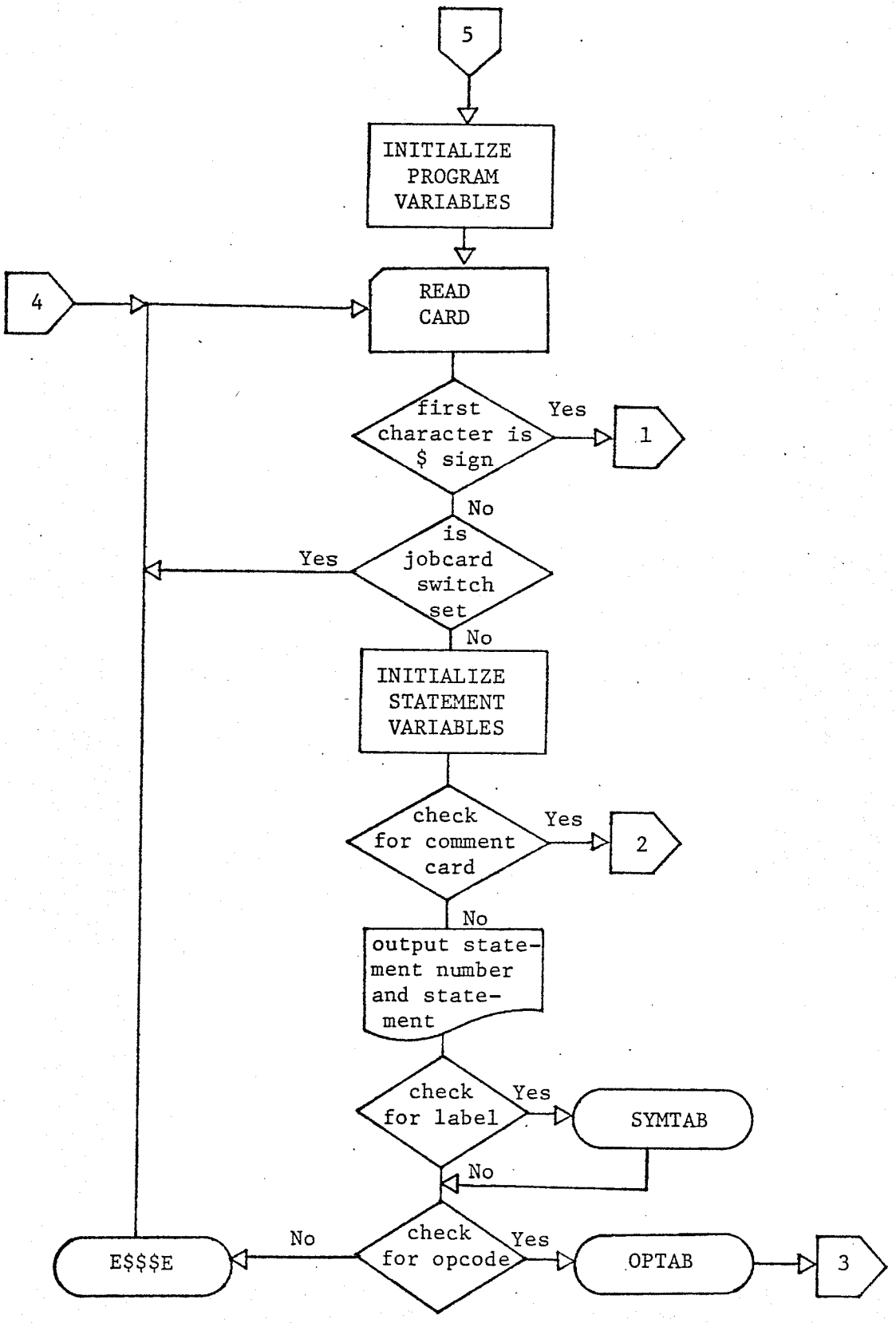
080000	006528				3	
0801C0	006628		FIRST		1	
080330	0067L8				2	
080980	006DE8				1	
086CC0	000128		OCTAL		1	
087AC0	000D28		OPEN	ASSEMBLER I/O ROUTINES	1	
087AC0	000F28		WAIT		1	
087C00	001065		CHECK		1	
087C40	001DA8				1	
087000	00F168		BUFOUT		2	
087F40	001AA8				7	BUFIN
088380	00L7L8		$TPTM		1	
088900	00FL68		FIXPI#		1	
08C3C0	012828				2	
08C840	012CA8				1	
08CCC0	013128		FIOCSBEP		1	

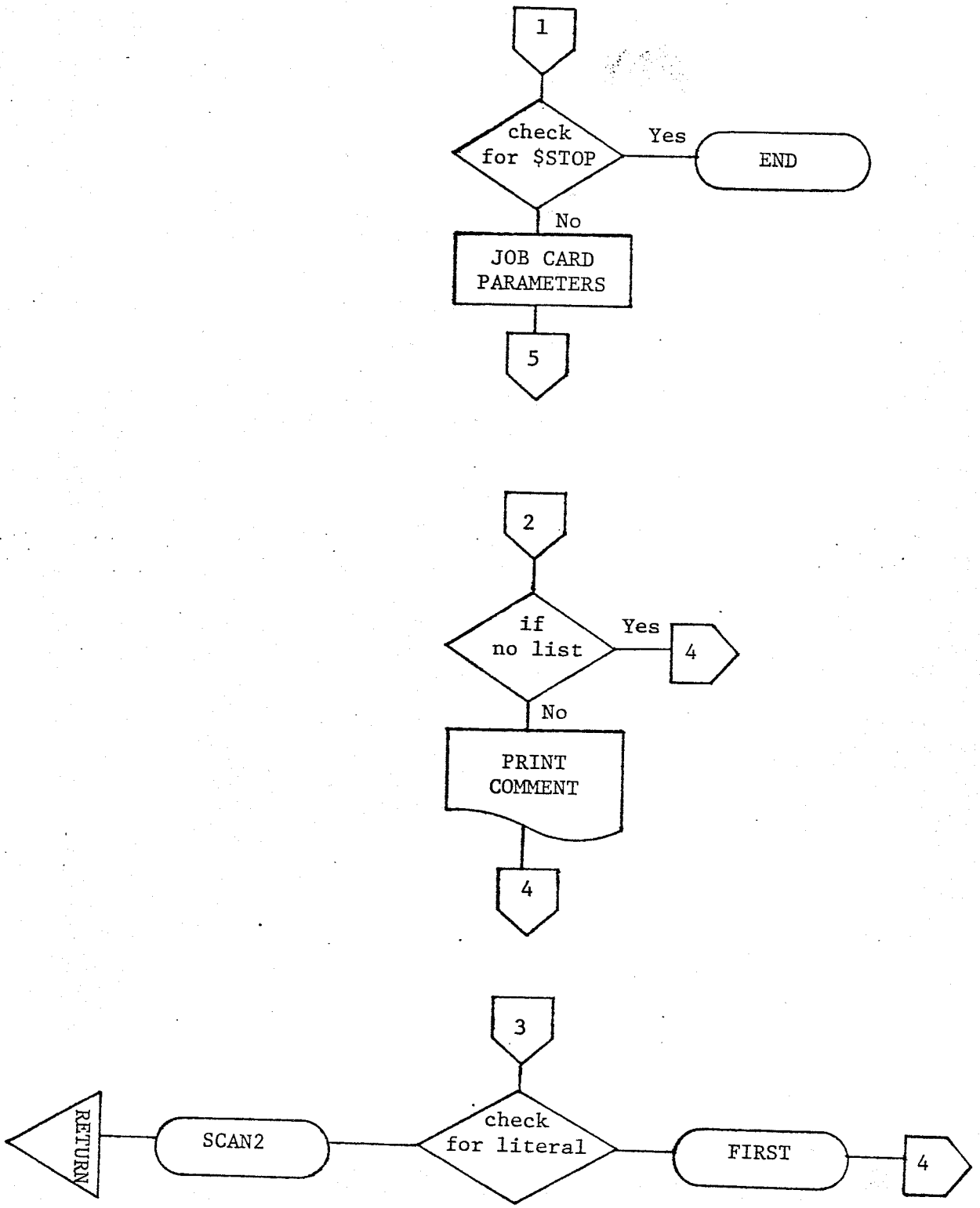
APPENDIX H

Flowcharts

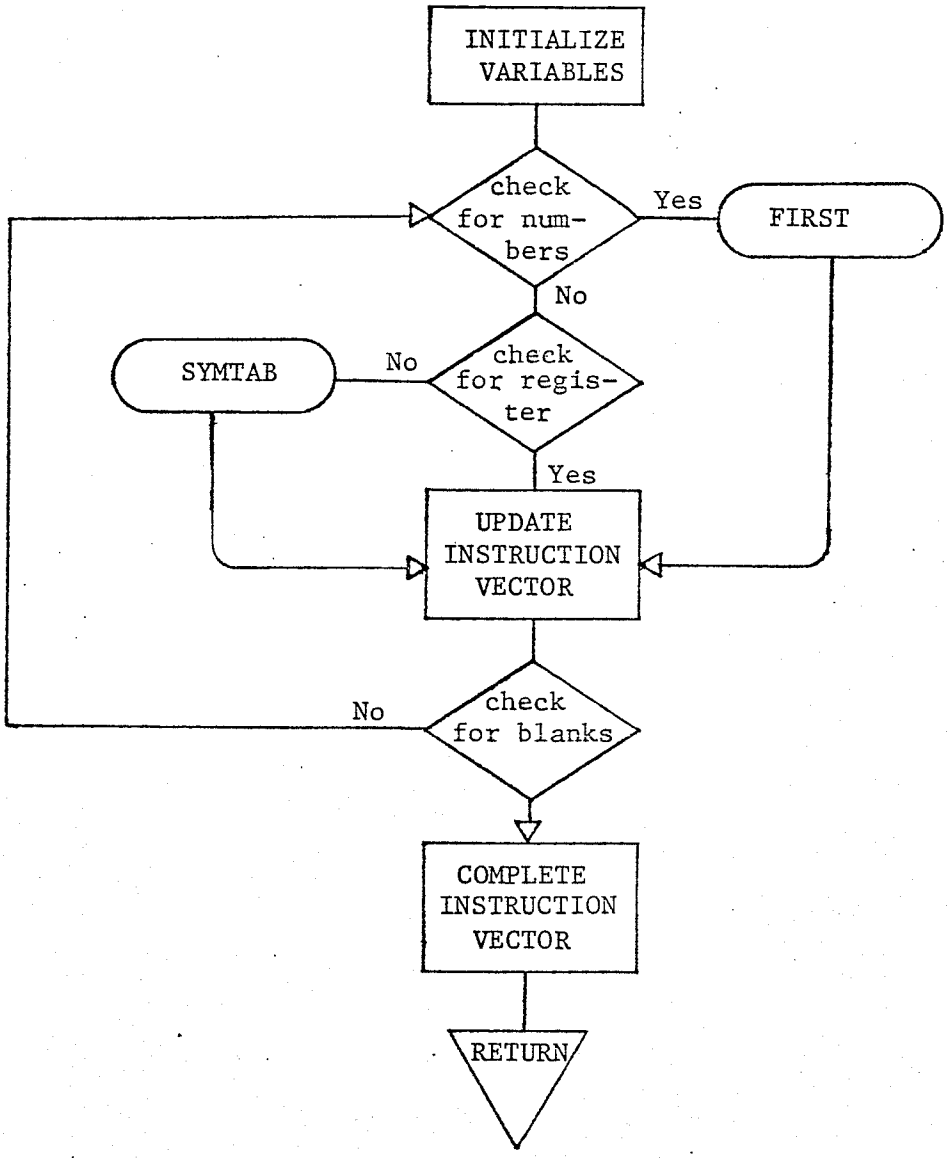
The following flowcharts are for some of the subroutines used in the load-and-go assembler.

Subroutine MAIN1




1
check for $STOP
Yes
END
No
JOB CARD PARAMETERS
5
2
if no list
Yes
4
No
PRINT COMMENT
4
3
check for literal
FIRST
4
SCAN2
RETURN

Subroutine SCAN2



Subroutine STCODE