1977-09

The development of COBOL "calculator" for high-performance, bit-slice microprocessors

Conley, Elizabeth K.; Modes, Ronald W.

Monterey, California. Naval Postgraduate School

THE DEVELOPMENT OF A COBOL "CALCULATOR" FOR HIGH-PERFORMANCE BIT-SLICE MICROPROCESSORS

Elizabeth K. Conley
and
Ronald W. Modes

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

The development of a COBOL "Calculator" for High-performance Bit-slice Microprocessors

by

Elizabeth K. Conley
and
Ronald W. Modes

September 1977

Thesis Advisor: G. A. Kildall

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): The development of a COBOL "Calculator" for High-performance Bit-slice Microprocessors	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; September 1977
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Elizabeth K. Conley Ronald W. Modes	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93940	12. REPORT DATE: September 1977 13. NUMBER OF PAGES: 111
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Naval Postgraduate School Monterey, California 93940	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

COBOL Interpreter
Bit-slice Microprocessors

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design of a COBOL editor and interpreter is explored which uses simple, straightforward algorithms similar to the approach found in programmable calculators. The algorithms

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

are designed to be implemented using the microinstructions of a high-performance bit-slice microprocessor. A possible machine design using a family of microprogrammable four-bit-slice bipolar circuits is outlined.

Approved for public release; distribution unlimited

The Development of a COBOL "Calculator" for High-performance, Bit-slice Microprocessors

by

Elizabeth K. Conley
Lieutenant, United States Navy
B.A., Manhattanville College, 1971

and

Ronald W. Modes
Lieutenant, United States Navy
B.S., University of Utah, 1971

Submitted in partial fulfullment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September, 1977

ABSTRACT

The design of a COBOL editor and interpreter is explored which uses simple, straightforward algorithms similar to the approach found in programmable calculators. The algorithms are designed to be implemented using the microinstructions of a high-performance bit-slice microprocessor. A possible machine design using a family of microprogrammable four-bit-slice bipolar circuits is outlined.

Table of Contents

I. INTRODUCTION

The objective of this thesis was to investigate the feasibility of a specialized microcomputer: a "COBOL calculator." This COBOL calculator would interpret only COBOL programs and was to be designed using simple algorithms, similar to a programmable calculator, which would be implemented in microcode on a high-performance microprocessor.

COBOL was chosen as the source language because it is used throughout the Navy and the business world. The version of COBOL used for this machine is MICRO-COBOL, which was developed by LT A. S. Craig during the course of his thesis work at the Naval Postgraduate School (see Reference 1). MICRO-COBOL is an extension of the U.S. Navy defined Hypo-COBOL, which represents the minimum acceptable working subset of COBOL within the Navy.

The use of a pure interpreter to execute COBOL programs can be considered feasible since COBOL is heavily I/O dependent. Thus the time required to interpret the source code is small in comparison to the I/O delays.

The preliminary specification of the machine hardware was completed using the Advanced Micro Devices Am2900 family of microprogramable high-performance circuits. This family is centered around a bipolar, four-bit-slice microprocessor LSI chip. Figure 7 contains a system design

using this family of circuits. In addition, the use of a highly flexible "meta-assembler," which simplifies the designers' task of creating microinstructions, is introduced.

The last major goal was the design of the algorithms to be used to interpret COBOL in the machine microcode. This was accomplished using the language C, which is available on the UNIX operating system at the Naval Postgraduate School. The software which was implemented includes a working editor and interpreter. The design of the editor and the interpreter stressed simplicity rather than efficiency. Concepts such as replacing pointers with a linear search, ASCII character arithmetic, and storage of variable values within the source code were used. These algorithms must now be converted to microcode and run on a development system to determine if the execution times are fast enough to make the COBOL calculator concept operationally feasible.

II. SOFTWARE

The COBOL machine described here consists of the following software modules: a monitor, an editor, a pure interpreter, and a debugger. Of these, the editor and the interpreter were designed and implemented as part of this thesis. Both the editor and the interpreter were written in the systems programming language 'C' and were compiled and tested on the UNIX operating system. The skeleton monitor, which was necessary for testing purposes, has not been included. The debugger will be briefly described as a simple extension of the interpreter.

The editor and the interpreter were designed for simplest implementation. In general, each module consists of a string of subroutines which are as independent as possible. The objective was to develop simple, straightforward, linear program modules that would edit and interpret COBOL and be short enough to be stored in Read Only Memory (ROM).

A. THE EDITOR--COED

The editor is the user's means of communicating with the COBOL machine. Aside from accepting COBOL programs line-by-line from the console and providing the standard edit functions, the editor performs some functions normally performed by an interpreter/compiler. These functions include creating and maintaining a symbol table and replacing the COBOL source words in the procedure division with token numbers. This greatly simplies the interpretation phase and does not create a substantial overhead during editing.

When invoked, the editor either creates a new file, or opens an already existing file by loading it into memory from a random access device. Each COBOL statement to be entered into this file must begin with a five character line index. The first character of this index depends upon the division of the COBOL program which is being edited (see APPENDIX C).

The Identification and Environment divisions are stored as they are entered from the user's console in lower case ASCII code (extension to allow upper case will be discussed later) with the UNIX newline character (CR) being replaced with a special 'EOL' character for ease of detection.

The Data division is also stored in lower case ASCII code. Whenever an identifier is declared, however, an entry is made in the symbol table. The symbol table is

simply a list of printnames, each a maximum of 16 characters long. The printnames of all COBOL reserved words are entries in the symbol table. For a completed COBOL program, there exists a one-to-one correspondence between COBOL reserved words, acceptable punctuation, all program identifiers, literals and symbol table entries. The relative address of a printname within the symbol table is the entry's token number, allowing simple conversion between token numbers and printnames.

The Procedure division is converted entirely to token numbers by the editor except for the five character line number that must precede every COBOL statement. All identifiers must be declared in the Data division, and, aside from comments, only reserved words and literals are allowed in the Procedure division. Thus, any undeclared identifiers, or typing and spelling errors are caught here in the edit phase where they can be easily corrected. It is mandatory, therefore, to enter the entire Data division before entering the Procedure division.

The editor operates in memory in the following way. When invoked, the editor initializes two memory pointers, cp1 and cp2 (see Figure 1). Cp1 is the pointer in low memory which always addresses the last byte accessed. This byte was either newly added, modified, or just examined by the user. The pointer in high memory is cp2, which addresses the remaining COBOL program, if it exists, or the last available byte in memory (see Figure 2). Memory is thus divided into three areas: area I in low memory, area II in

high memory, and the work space in the middle. Console input characters are queued in a line buffer until a period and a carriage return character are entered. At this time, the entire line is written into memory, beginning with the last accessed byte (the one addressed by col). Characters (bytes) are constantly being moved up or down in memory as necessary to open the work space at the point in user's COBOL program where editing is taking place (see Figure 3). The symbol table is stored in a temporary data area during the edit phase.

When the user terminates the edit phase, the work space is closed by moving the entire program down into low memory. The symbol table is then written at the end of the COBOL program, directly following its end of file (EOF).

The edited COBOL program consists of Identification, Environment, and Data divisions in their original source code, the tokenized Procedure division, and the symbol table. In this form, it can either be stored on some random access device for subsequent interpreting, or be interpreted directly following the termination of edit.

MEMORY DIAGRAMS

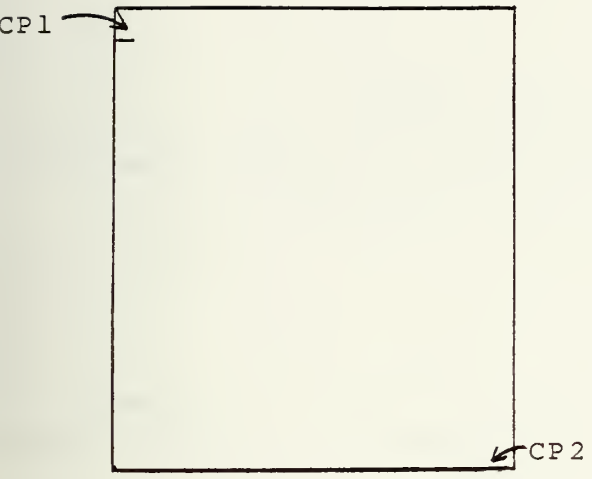

Figure 1.

Initially, all of memory is viewed as work space.

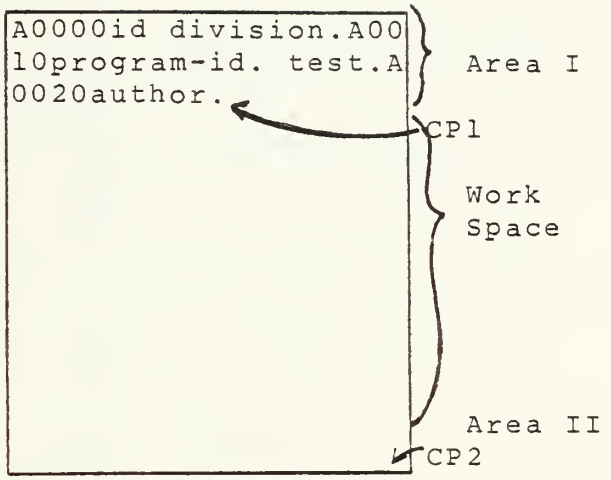

Figure 2.

CP1 is the byte offset of the last character entered by the user. CP2 stays at its initialized value until the first search.

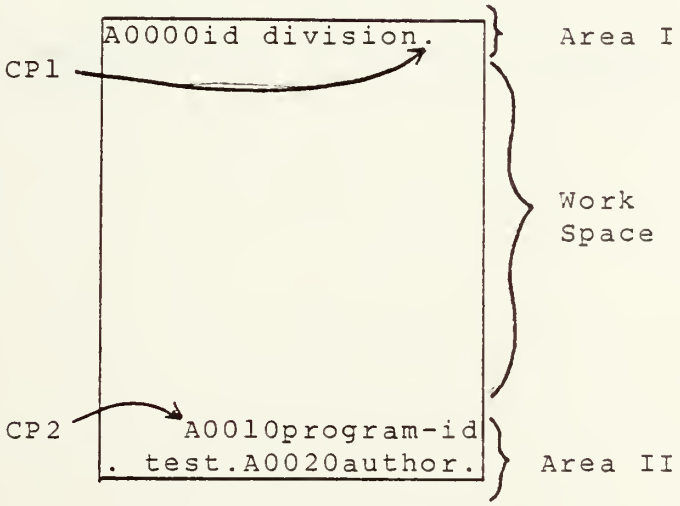

Figure 3.

When the editor must access a particular line, in this case A0010, bytes are moved between Area I and Area II until the line is found.

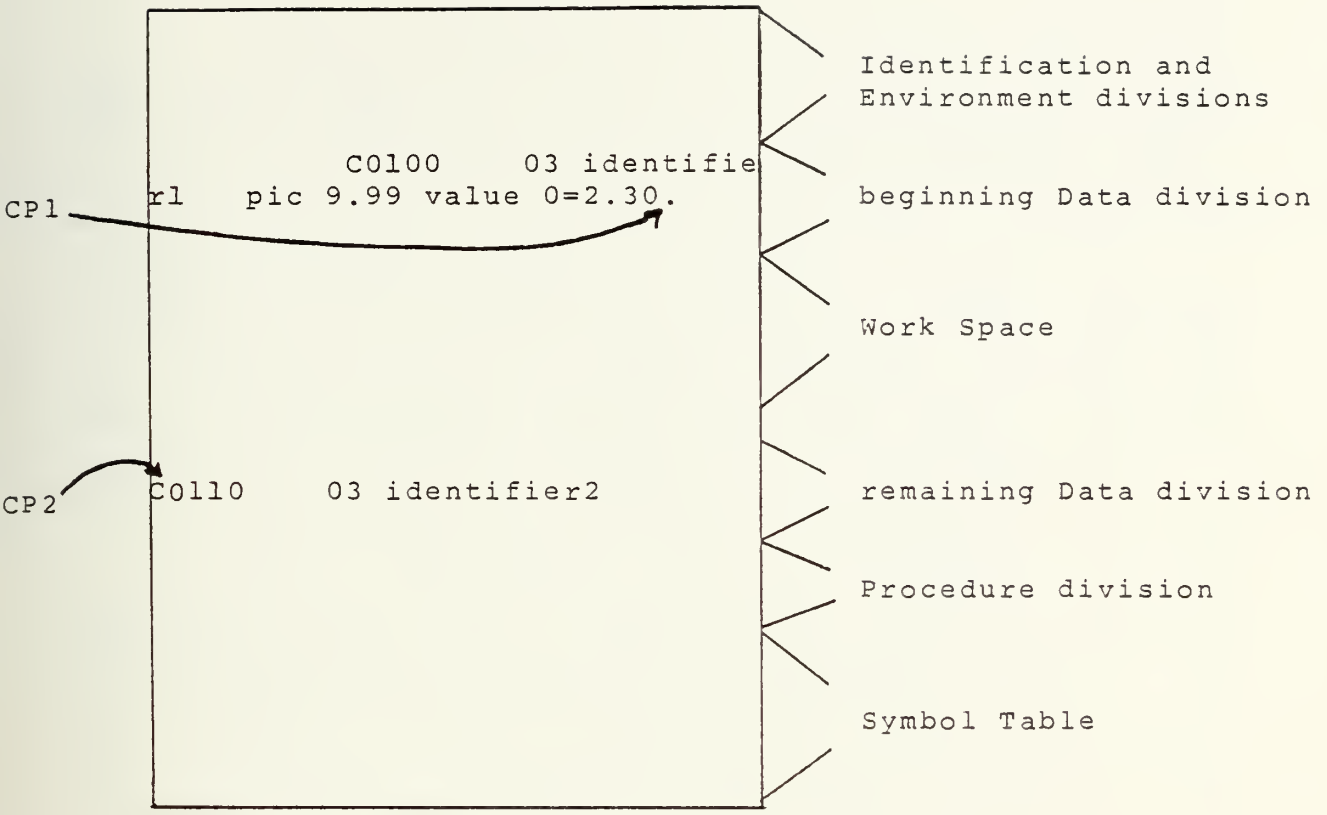

Figure 4.

Memory as viewed by the interpreter
after storing a value into identifierl.

B. THE INTERPRETER--COIN

When the interpreter is invoked, if the edited COBOL program does not already exist in memory, it is loaded one byte at a time beginning with relative address 0. Next, the beginning of the procedure division must be located. As the search proceeds, bytes are moved up in memory starting with the last byte of the symbol table until the first line number of the Procedure division has been moved. The work space in the middle of the memory array is now defined.

The interpreter then initializes all the identifiers whose initial value has been defined by the user with the VALUE option. Identifier values are stored in the Data division following its PIC and/or VALUE clause. Bytes are moved up in memory until the work space begins at the end of the particular declaration. Any value not exceeding the length of the work space can be stored here. All identifier values are stored in the Data division in this manner. The interpreter writes the desired value into the work space, thereby making it a part of the Data division (see Figure 4).

A program counter is initialized to the relative address of the beginning of the Procedure division. This counter is incremented byte by byte as the division is scanned for grammatical correctness. Each complete COBOL statement is executed as it is scanned.

The first byte after each line number is the token for a keyword that determines the type of COBOL statement

which follows. Based upon this token, the interpreter calls a subroutine to handle the statement form. The subroutine scans the statement (token chain) for acceptibility until the 'EOL' character, inserted by the editor, and performs the desired action. All embedded blank, tab, and comma tokens are accepted by the interpreter.

If a grammatical error (misplaced or unexpected token) is scanned, an error flag is set that causes execution to stop with an appropriate error message sent to the user. At this point, a debugger could be entered. Since all identifier values are available in the Data division and the current line number is always accessible, an error could be corrected and scanning/execution could resume at the statement. Normal execution halts when the token for STOP is scanned.

When a jump is required to execute a particular statement, the program counter is reset to the first token of the Procedure division and the first token of each line is scanned until the desired destination is found. Execution of the PERFORM verb involves a return jump and therefore requires the return token to be saved. At present, an eight level software stack is implemented which allows PERFORM statements to be nested eight deep. This could be changed, however, since overall machine memory size is the only restriction.

Each time an identifier value is fetched or stored, the following steps are executed:

1) The token number is mapped to its printname

(the token number is the index of the printname in the symbol table).

2) Bytes are moved sequentially up or down in memory between Area I and Area II until a printname match is found.

3) Scan continues from here until an equal sign or 'EOL' character.

4) The value is then either written from a scratch buffer into the work space at this point or written from the Data division into a scratch buffer for further action. If a READ or WRITE statement is being processed, the PIC clause determines the format in which the data value is written. Otherwise, it is simply copied byte by byte as it appears in the buffer or memory.

The arithmetic routines operate upon the ASCII characters stored in the Data division if the operand is a variable, or upon the symbol table in the case of a literal. Each routine uses look-up tables to perform its operation of addition, subtraction, multiplication, or division. The operands are loaded into two variable length buffers, called x and y, and operated upon sequentially one pair of bytes at a time beginning with the rightmost byte of each buffer. As defined by MICRO-COBOL, the result is written into the Data division area corresponding to the second operand. It should be noted that as the LSI circuits are developed, the table lookup method can easily be replaced by using four-bit adder/subtractor and multiplier chips.

I. THE TARGET MACHINE

One major development which makes a COBOL machine feasible is the wide availibility of high performance bit-slice microprocessors on LSI chips. A bit-slice microprocessor is a circuit which has all the required inputs and outputs of a basic processor function, such as the ALU, but is only two or four bits wide. Any number of these two or four bit-slices are connected in parallel to allow processing of data words of any desired width. Thus, the system designer is able to specify a processor which meets particular requirements. This flexibility is highly desirable in many applications where the optimum word size is greater than the common eight bits available on most fixed architectured microprocessors.

The majority of bit-slice microprocessors achieve their high performance using bipolar technology. The commercial development of bipolar ITL, ECL, and Schottky bipolar DTL chips represent a major step in increasing performance. Bipolar switching circuitry is ten to a hundred times faster than similar MOS switching circuitry.

The disadvantage of bipolar circuitry is that it cannot yet be packed as densely as the MOS equivalent, thus requiring more chips for a given number of circuits. In many bit-slice applications this is not a significant disadvantage when compared with the increase in speed.

The second major development which effects the COBOL "calulator" is the introduction of bit-slice microprocessor families which are microprogrammable. Microprogram logic is placed in Read Only Memory (ROM) and replaces the usual hardware random logic circuits. The advantages are found in the more ordered approach to function implementation and the ease of replacing the logic. The logic to define a function is stored in Programmable Read-Only Memory (PROM) as a block of microinstructions and, if a change is desired, a new PROM is programmed with different microinstructions. A further advantage is the ability to define a powerful instruction set, called macroinstructions for an architecture where each instruction is actually implemented with several microinstructions. This increases the performance of the machine significantly since the access time for ROM is two to ten times faster than Random Access Memory (RAM) where macroinstructions are stored.

The most common uses for bit-slice microprogrammable machines include digital filters and emulators. Bit slice processors are particularly well suited for emulation, since a given macroinstruction can be mapped to a set of microinstructions which execute the desired function.

Although emulation is effective, the COBOL calculator does not emulate a machine language. The necessary logic to interpret a MICRO-COBOL program is in microcode in the PROM. By implementing the COBOL editor and interpreter in microcode, the machine becomes a COBOL "calculator" and thus it will only process MICRO-COBOL. The interpreter is

implemented in microcode to increase performance, and, as noted previously, the algorithms were extremely simple. With simple algorithms, balanced by a high performance processor, it is possible to keep the overall length of the program small enough to write in microcode, while retaining a feasible execution time.

A. THE Am2900 FAMILY

The COBOL calculator was designed within the limitations of a widely available microprogrammable four-bit-slice bipolar microprocessor family, the Am2900 series, developed by Advanced Micro Devices (see Reference 2).

1. Am2901

The Am2901 is four bit-slice bipolar microprocessor chip which can be easily cascaded to any number of chips. Figure 5 is the block diagram of this chip. The 2901 consists of 16 working registers, a Q register, shift multiplexers, and an eight-function ALU. It features simultaneous access to any two registers, and left and right shift operations independent of the ALU. The machine cycle times, based upon a two register add function, is 110 nanoseconds for the Am2901 and 55 nanoseconds for the new Am2900A.

2. Am2909 and Am2911

The Am2909 is a four bit cascadable microprogram sequencer. Its block diagram is shown in Figure 6. The 2909 controls the execution sequence of a series of microinstructions. The address of the next microinstruction to be executed can come from the program counter register, the stack, the internal address register, or a direct input. The latter two sources allow for an n-way branch at any point in the microprogram. The stack allows four levels of subroutine calls. The internal address register (IAR) can be connected to the pipeline register or the mapping PROM as shown in

Figure 7. The output can be set to the first microword (0000) using the zero input. Each output bit can be "OR'ed" for conditional instructions.

The Am2911 is similar, except the direct input is connected to the internal register and the OR inputs are removed.

3. Additional Circuits

The Am2902 is a high-speed look-ahead carry generator providing a look-ahead carry for up to four Am2901 chips. The Am2905 and Am2906 are quad two-input open-collector bus transceivers. Additional chips which are available include four-bit registers, counters, multiplexers, adders, and multipliers. Read-Only Memory (RAM) and Programmed Read-Only Memory (PROM) are available in several sizes and access speeds. Chips which will soon become available include a Direct Memory Access. Reference 2 contains a detailed explanation of these and other circuits.

Am2901
MICROPROCESSOR SLICE BLOCK DIAGRAM

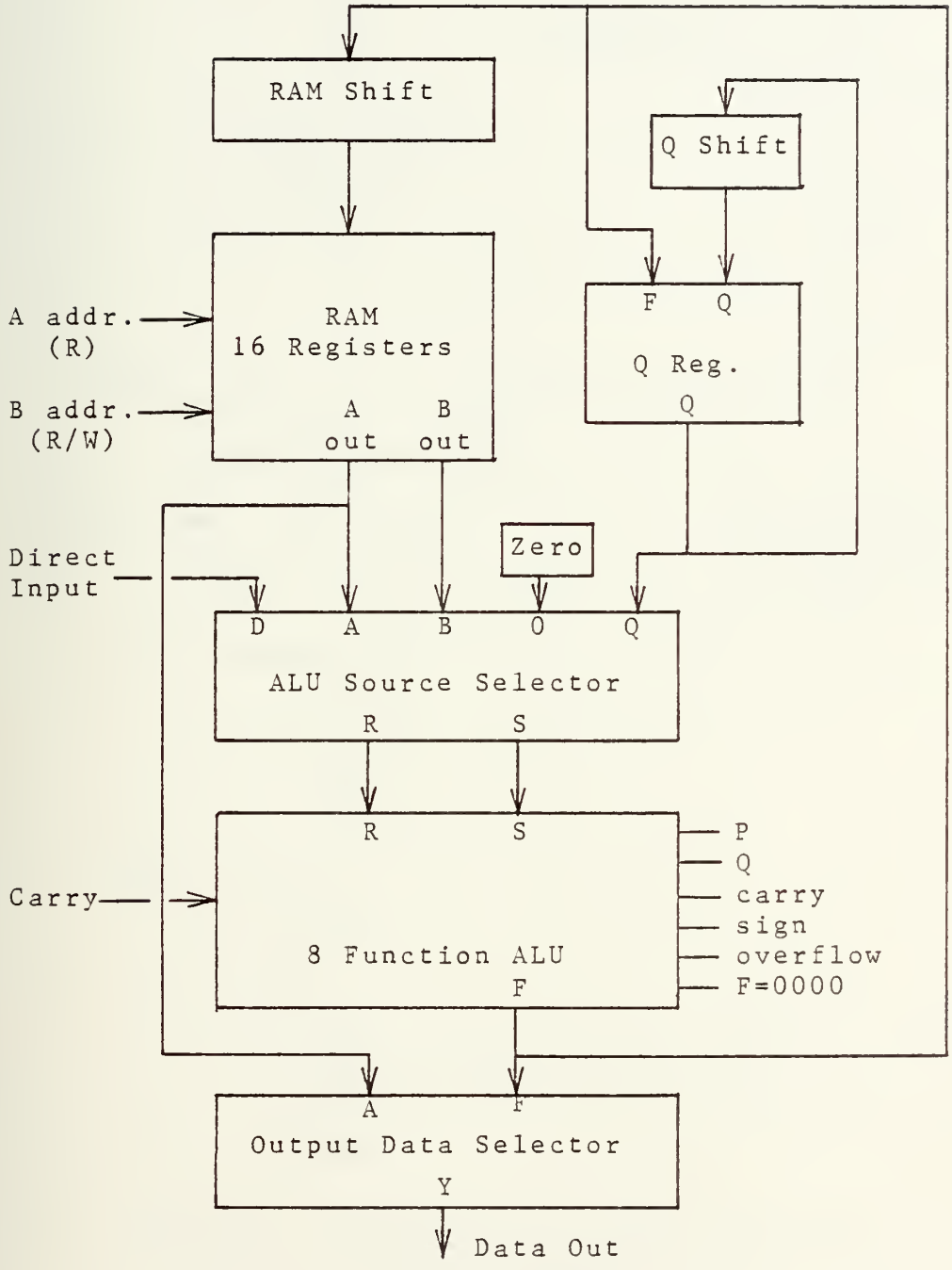

Figure 5.

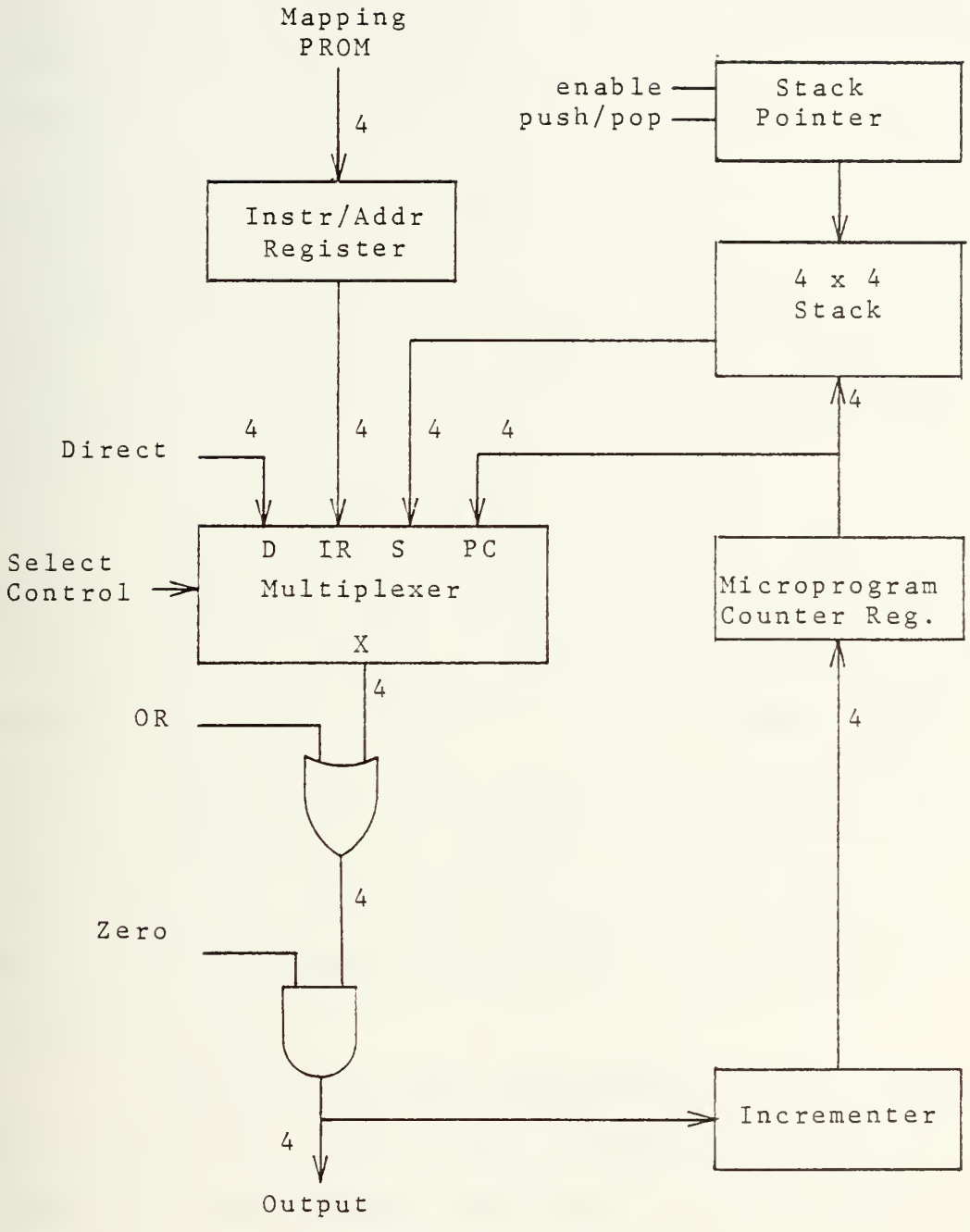

Figure 6. Am2909 Microprogram Sequencer
Block Diagram

B. MACHINE DESIGN

Figure 7 is a detailed block diagram of the proposed COBOL calculator. The machine is designed around several basic blocks: a 4096 by 28-bit microprogram PROM, a 16-bit ALU, an 8-bit data bus, and a pipeline register.

The pipeline register holds the next instruction to be executed. While an microinstruction is being executed, the next instruction is moved into the pipeline register. Within one machine cycle, the sequencer fields are decoded and the next instruction address is available to PROM. This fetch is concurrent with any ALU operations.

The size of the microprogram PROM was determined by the estimated size of the editor and the interpreter. This estimate was made by examining the size of the PDP-11 assembly language version of these software modules (3000 lines of assembly code and tables). While the correspondence is not one-to-one, any increase in size due to microcoding will be partially offset by eliminating routines and statements which were inserted to permit execution under the UNIX operating system.

The data bus size was determined by the nature of the interpreter. Only ASCII characters and 8-bit token numbers are manipulated. Thus, only an 8-bit data bus is required.

The ALU handles two types of data. First, the majority of operations will involve characters and token numbers of eight bits. The second type is memory address da-

ta, such as the program counter. This requires 16 bits for a 64K byte memory and 20 bits for a one megabyte memory. Random Access Memory of 64K should be sufficient, but if the larger memory is required another Am2901 and Am2902 can easily be connected.

In the block diagram, Figure 7, the suggested LSI chips for the major blocks are noted along with the number required for the function in parenthesis. Table 1 lists the total circuits which are required to implement the COPUL machine. The logic diagrams and detailed description of these circuits are in Reference 2.

Three blocks are not discussed in detail: Direct Memory Access (DMA), Interrupt Control Unit (ICU), and the Sync and Control Logic. Suggested architecture may be found in Reference 2 and a listing of the required circuits is included in Table 1.

One family of blocks detailed in the machine diagram are the bus tranceivers which interface between the various blocks and the data and address busses. These may be a combination of Am2906 and Am2907 or Am26S10 chips.

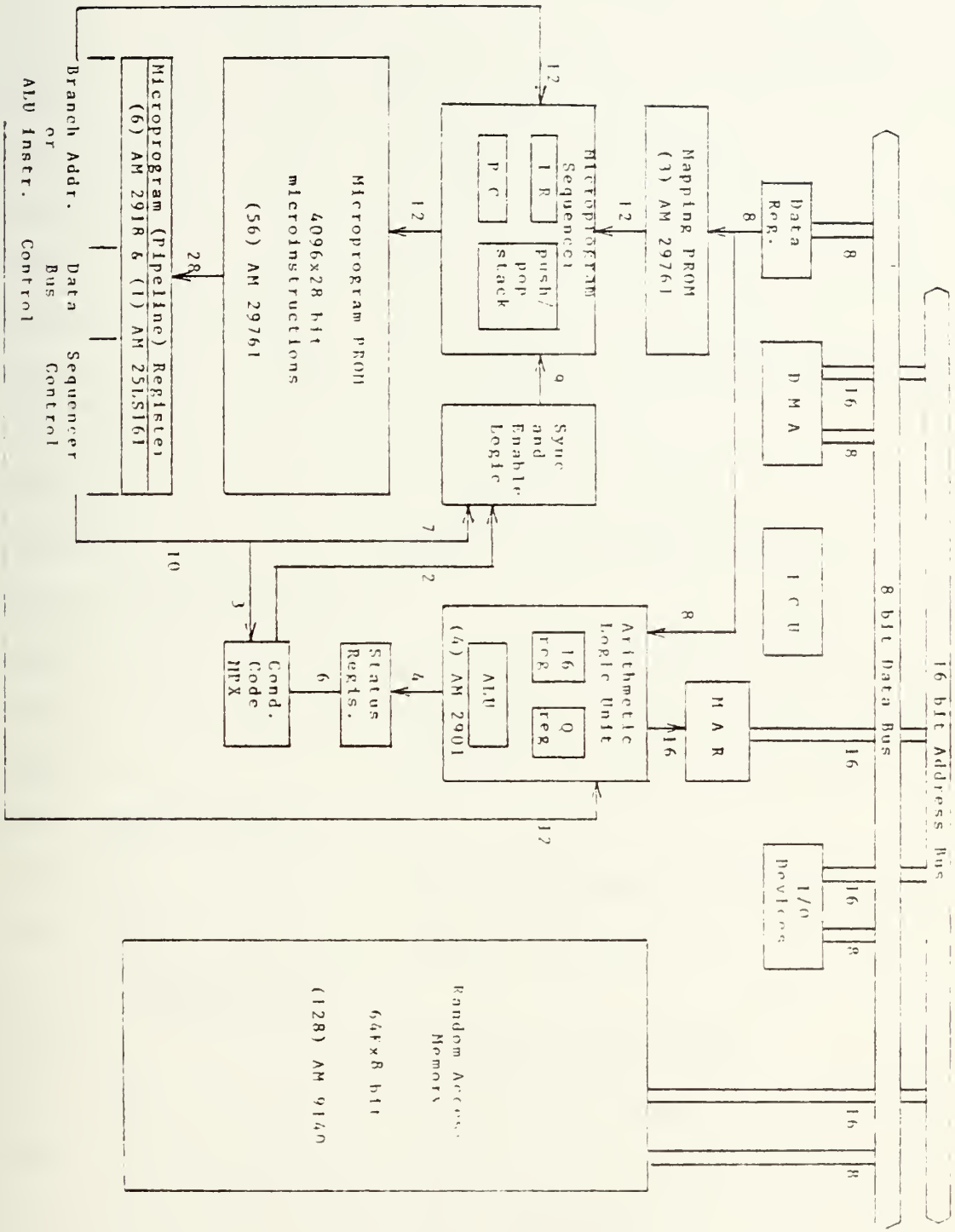

Figure 7. COBOL Calculator
Block Diagram

Table 1. LSI Chips required for the COBOL Calculator

CHIP	NUMBER	DISCRIPTION
Am2901	4	bipolar 4-bit microprocessor slice
Am2902	1	look ahead carry
Am2905/6/7	12	quad bus transceiver
Am2909	3	microprogram sequencer
Am2918	14	four bit register
Am29761	59	256 word x 4 bit PROM
Am9140	128	4096 word x 1 bit static R/W RAM
Am25LS07	1	6 bit register
Am25LS151	1	8 input multiplexer
Am25LS161	1	binary hexadecimal counter
Interrupt Control Unit		
Am2913	1	priority interrupt expander
Am2914	2	vectored priority interrupt encoder
Am29705	1	16 word x 4 bit two-port RAM
Direct Memory Access		
Am25LS161	8	binary hexadecimal counters
Am25LS2541	2	octal buffer
Sync and Enable Logic		
Am29750	1	32 word x 8 bit PROM
Am25LS157	1	2 input multiplexer
Am555	1	timer
Am74LS112	1	dual flip-flop
Am74LS124	1	clock

C. AMDASM / 80

The next step in the development of the COBOL calculator is to convert the algorithms to microcode using the "meta-assembler" AMDASM/80. AMDASM/80 is a microprogram assembler which operates on the INTEL INTELLEC MDS-DOS system under the ISIS-II operating system. A meta-assembler differs from a conventional assembler in that the user must define not only labels and symbols but also word lengths and formats. Very little information is predefined in a meta-assembler, allowing the user a great flexibility in matching microprograms to a hardware configuration. AMDASM/80 is sufficiently general and powerful to be used for nearly any microprogrammable machine. However, it was designed especially for the Am2900 family.

The assembler operates in two phases: the Definition phase and the Assembly phase. The Definition phase, which is executed first, establishes the tables to map each user defined format and constant names to the corresponding bit patterns. The length of each microinstruction word can be defined from 1 to 128 bits. A microinstruction word may consist of one format, or several overlapping formats. A format defines the fields of a microinstruction and their usage. Fields may contain specified numeric bit patterns: either hexidecimal, decimal, octal, or binary. The field may also contain a variable field to be filled-in during execution, or a "don't care" field to be ignored, which is usually a filler between other fields. The "don't care" fields

are used when formats are overlapped. That is, formats can be overlapped if the defined fields of one format correspond to "don't care" fields of the other format.

The Assembly phase uses the output phase, and operates similar to a more conventional assembler. This phase reads a symbolic program, performs the common assembler functions and produces a binary output, listings, and tables, which are retained for execution or post-processing.

The Post-processing phase uses the Assembly phase output to create paper tapes suitable for use in programming PROM's. The user may define the organization of the PROM matrix and then create a tape for any particular row or column. This will allow the user to organize the microprogram memory as desired. Further, it is relatively easy to change a particular PROM by creating a new tape for the desired column or row.

The advantages of a meta-assembler approach, such as AMDASM/80, are flexibility and ease of use. The user may define multiple formats with overlapping fields and link meaningful mneumonics with various bit patterns. Thus, the assembler creates the bit strings for microinstructions. The user may also write programs using strings of 1's and 0's , with a short Definition phase.

D. TIMING ANALYSIS

Practicality of the Cobol calculator depends partially upon the time which is required to execute a program. The actual timing of the machine performance will occur when it is run in microcode on the Intellec MDS-DOS system. However, several important factors can be assessed by using manual methods and estimating the required cycles and number of references to Random Access Memory (RAM).

Table 2 contains a list of the sequential processing steps of the ADD verb. The size of the data and procedure divisions are assumed to be 2000 bytes each, which allows storage of a reasonably complex program. A RAM access time of 300 nanoseconds was assumed for these calculations, and any processing which could be done simultaneously with the RAM access was not included. The time of 1.336 milliseconds for the fetch, add, and store is quite long. This value includes 4050 access to memory, most of which were used in moving data division bytes through the working area. The time used in moving these bytes represents 1.215 milliseconds or about 91 percent of the total execution time.

Another example of a large amount of time spent manipulating data is the initialization of the program in memory. The interpreter, COIN, must find the top of the procedure division and then locate the top of the data division. For the simple example program in Appendix A, the initialization requires 800 RAM accesses and 1000 machine cycles. Execution of this program will require about 2500

accesses to memory and 5500 machine cycles. Thus, execution time is about 1.05 milliseconds, of which 71 percent is absorbed in memory access. Again this is quite long for a simple program.

It must be remembered that these estimates are very rough. Estimation errors will be found in counting the number of machine cycles which correspond to a given line of assembly code.

Using these estimates as a rough guide, it is evident that the reference time to Random Access Memory is a major factor in the speed of program execution. Due to this problem, two possible changes to any future design should be considered. First, Random Access Memory with an access time of under 200 nanoseconds should be used. Memory with this access speed is available, although it is more expensive than the slower RAM. This change could improve execution time by one-fourth. The second suggestion is to develop a pipeline technique for bytes access, much like the technique used in the sequencer. This could be done in microcode; it would consist of loading the next sequential byte during the processing of a given byte. Since a majority of RAM accesses occur during a linear search of the user's program, this simple technique could cut the execution time by 25 percent.

TABLE 2. SAMPLE TIMING FOR AN ADD STATEMENT

TIME	ACTION
410ns	map token number to symbol table entry
660us	search data division for a variable printname (assuming 1000 character compares)
355ns	fetch next token number--does it equal 'TO'?
300ns	fetch next token number
660us	search data division for second variable printname (assuming 1000 character compares)
1.2us	align decimal points--assume two zeros
4.5us	access arithmetic tables--assume longest operand has 10 digits * .45 us/digit
3us	transfer data from buffer to memory a cpl

1.336 ms/add	

I. RECOMMENDATIONS

This thesis represents only the first half of the implementation of a COBOL calculator. The next phase includes the final machine design and conversion of the 'C' language program modules to microcode. AMDASM/80 would be an appropriate language.

The following are items which can be considered if an expansion of the COBOL calculator is desired. Each of these items was considered during the initial design of the algorithms; all should be relatively straightforward to implement.

First, the arithmetic package could be expanded to handle exponentials. As presently written, the routines are independent of decimal point position once the location is known. A routine to detect and manipulate the exponents can be added to both the multiply and divide routines. For the add/subtract routine, the section which aligns the decimal points must be expanded to adjust the exponential values.

Second, expansion to allow for both upper and lower case ASCII characters has some value since it would improve readibility. The editor, COED, will presently allow upper case characters anywhere in the Identification and Environment divisions. If the token routine is modified to detect upper case letters when looking for a symbol name in the Data division, all user defined symbols can contain ei-

ther upper or lower case characters. In the Procedure division, however, all reserved words must contain only lower case characters because they are entered in the symbol table in lower case. This can be easily changed to upper case, or a mixture of the two, by changing the symbol table entries. However, it may not be not practical to allow the user any mixture of upper and lower case characters for the reserved words, due to the limited size of the symbol table.

A third item of expansion could be a character editing ability. This could be accomplished relatively easily since the line to be modified is in the line buffer. A significant number of revisions to the modify and input routines would be required, however, and it was not implemented at this time.

The fourth area of improvement is the inclusion of a debug facility. This is probably the most important area of improvement. The debug routine would have the value of all declared variables available since they remain with the program until reinitialized. Thus if a fatal error occurs during the interpretation phase, the error routine needs to insure that the subroutine stack and the program counter are saved. The debugger would use the various search, match, and find routines which have already been implemented to determine the value of any desired variable. A restart could be easily accomplished by passing the interpreter's initialization routine.

In conclusion, the first phase is complete. In the course of this thesis, several important concepts were

implemented. It was shown that a COBOL Calculator using simple algorithms to interpret a basic version of COBOL is technically feasible. It was also shown that there are some unresolved questions as to the operational feasibility of this design.

APPENDIX A. SAMPLE TERMINAL SESSION

```
COED:Cobol Editor.  Version 1.1
*A0000 Identification Division.
*A0010  Program name.  test.
*A0020  Author.
            Conley.
*B0000 Environment Division.
*B0010  Configuration Section.
            Source computer.
                PDP-11/50.
*B0015  This line will be deleted later.
*B0015 d
*C0000 Data Division.
*C0010  File Section.
*C0020  fd  file-in
            data record is emp-in.
*C0030  01  emp-in.
*C0040          03  name-in      pic  x(10).
*C0050          03  hrs-in       pic  99.
*C0060          03  rate-in      pic  999.99.
*C0070  fd  file-out
            data record is emp-out.
*C0080  01  emp-out.
*C0090          03  name-out     pic  x(10).
*C0100          03  filler       pic  x(10).
*C0110          03  pay          pic  9.99.
*C0110 m
*C0110          03  pay          pic  999.99.
*C0120          03  filler       pic  x(10).
*C0130          03  total        pic  $9,999.99  value 0.
*D0000 Procedure Division.
*D0010  010-main.
*D0020   read emp-in.
*D0030   if name-in numeric  go 020-endtest.
*D0035   perform 030-process.
*D0040   020-endtest.
*D0045   stop.
*D0050  030-process.
*D0055   move name-in to name-out.
*D0060   multiply hrs-in by rate-in.
*D0065   move rate-in to pay.
*D0070   add pay to total.
*D0080   write emp-out.
*D0090  030-exit.   exit.
*A0030
*B0020
*C0140
*A0025 remarks.  This is a test program to
```

```
        calculate the pay of an employee
        based on his pay rate and the
        hours worked.
*D0005 * this is an example comment,
        which can be inserted anywhere.
*A0000,E
A0000 Identification Division.
A0010   Program name.  test.
A0020   Author.
            Conley.
A0025 remarks.  This is a test program to
        calculate the pay of an employee
        based on his pay rate and the
        hours worked.
A0030
B0000 Environment Division.
B0010   Configuration Section.
            Source computer.
                PDP-11/50.
B0020
C0000 Data Division.
C0010   File Section.
C0020   fd  file-in
            data record is emp-in.
C0030   01  emp-in.
C0040       03  name-in      pic  x(10).
C0050       03  hrs-in       pic  99.
C0060       03  rate-in      pic  999.99.
C0070   fd  file-out
            data record is emp-out.
C0080   01  emp-out.
C0090       03  name-out     pic  x(10).
C0100       03  filler       pic  x(10).
C0110       03  pay          pic  999.99.
C0120       03  filler       pic  x(10).
C0130       03  total        pic  $9,999.99  value 0.
C0140
D0000 Procedure Division.
D0005 * this is an example comment,
        which can be inserted anywhere.
D0010  010-main.
D0020    read emp-in.
D0030    if name-in numeric  go 020-endtest.
D0035    perform 030-process.
D0040   020-endtest.
D0045    stop.
D0050  030-process.
D0055    move name-in to name-out.
D0060    multiply hrs-in by rate-in.
D0065    move rate-in to pay.
D0070    add pay to total.
D0080    write emp-out.
D0090  030-exit.   exit.
EOF
*quit
```

APPENDIX B. GRAMMAR RULES IN BNF FOR MICRO-COBOL

```
 1  <program> ::= <id-div> <e-div> <d-div> <p-div>

 2  <id-div> ::= IDENTIFICATION DIVISION. PROGRAM-ID.
                    <comment> . <auth> <date> <sec>

 3  <auth> ::= AUTHOR . <comment> .

 4           | <empty>

 5  <date> ::= DATE-WRITTEN . <comment> .

 6           | <empty>

 7  <sec> ::= SECURITY . <comment> .

 8          | <empty>

 9  <comment> ::= <inout>

10              | <comment> <inout>

11  <e-div> ::= ENVIRONMENT DIVISION . CONFIGURATION SECTION.
                   <scr-obj> <i-o>

12  <src-obj> ::= SOURCE-COMPUTER . <comment> <debug> .
                    OBJECT-COMPUTER . <comment> .

13  <debug> ::= DEBUGGING MODE
```

```
14              | <empty>
15  <i-o> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .
                <file-control-list> <io>
16            | <empty>
17  <file-control-list> ::= <file-control-entry>
18                        | <file-control-list> <file-control-entry>
19  <file-control-entry> ::= SELECT <id> <attribute-list> .
20  <attribute-list> ::= <one attrib>
21                     | <attribute-list> <one attrib>
22  <one-attrib> ::= ORGANIZATION <org-type>
23                 | ACCESS <acc-type> <relative>
24                 | ASSIGN <input>
25  <org-type> ::= SEQUENTIAL
26               | RELATIVE
```

The relative attribute is saved for production 19.

```
27  <acc-type> ::= SEQUENTIAL
```

This is the default.

```
28               | RANDOM
```

The random access mode needs to be saved for produc-

tion 19.

29 <relative> ::= RELATIVE <id>

30 | <empty>

31 <ic> ::= I-O-CONTROL . <same-list>

32 | <empty>

33 <same-list> ::= <same-element>

34 | <same-list> <same-element>

35 <same-element> ::= SAME <id-string> .

36 <id-string> ::= <id>

37 | <id-string> <id>

38 <d-div> ::= DATA DIVISION . <file-section> <work> <link>

39 <file-section> ::= FILE SECTION . <file-list>

40 | <empty>

41 <file-list> ::= <file-element>

42 | <file-list> <file-element>

43 <files> ::= FD <ic> <file-control> . <record-description>

44 <file-control> ::= <file-list>

45 | <empty>

46 <file-list> ::= <file-element>

```
47                  ! <file-list> <file-element>
48  <file-element> ::= BLOCK <integer> RECORDS
49                     ! RECORD <rec-count>
      The record length can be saved for comparison with the
      calculated length from the picture clauses.
50                     ! LABEL RECORDS STANDARD
51                     ! LABEL RECORDS OMITTED
52                     ! VALUE OF <id-string>
53  <rec-count> ::= <integer>
54                  ! <integer> TO <integer>
55  <work> ::= WORKING-STORAGE SECTION . <record-description>
56             ! <empty>
57  <link> ::= LINKAGE SECTION . <record-description>
58             ! <empty>
59  <record-description> ::= <level-entry>
60                           ! <record-description> <level-entry>
61  <level-entry> ::= <integer> <data-id> <redefines>
                      <data-type> .
62  <data-id> ::= <id>
```

63 | FILLER

64 <redefines> ::= REDEFINES <id>

The redefines option gives new attributes to a previously defined record area.

65 | <empty>

66 <data-type> ::= <prop-list>

67 | <empty>

68 <prop-list> ::= <data-element>

69 | <prop-list> <data-element>

70 <data-element> ::= PIC <input>

The <input> at this point is the character string that defines the record field.

71 | USAGE COMP

The field is defined to be a packed numeric field.

72 | USAGE DISPLAY

The DISPLAY format is the default, and thus no special action occurs.

73 | SIGN LEADING <separate>

This production indicates the presence of a sign in a

numeric field. The sign will be in a leading position. If the <separate> indicator is true, then the length will be one longer than the picture clause, and the type will be changed.

74 ! SIGN TRAILING <separate>

The same information required by production 73 must be recorded, but in this case the sign is trailing rather than leading.

75 ! OCCURS <integer>

The type must be set to indicate multiple occurrences.

76 ! SYNC <direction>

77 ! VALUE <literal>

The field being defined will be assigned an initial value determined by the value of the literal.

78 <direction> ::= LEFT

79 ! RIGHT

80 ! <empty>

81 <separate> ::= SEPARATE

```
82                  ! <empty>
83 <literal> ::= <input>
84                  ! <lit>
      This literal is a quoted string.
85                  ! ZERO
86                  ! SPACE
87                  ! QUOTE
88  <integer> ::= <input>
89  <id> ::= <input>
90  <p-div> ::= PROCEDURE DIVISION <using> .
                  <proc-body> END .
91  <using> ::= USING <id-string>
92                ! <empty>
93  <id-string> ::= <id>
94                    ! <id-string> <id>
95  <proc-body> ::= <paragraph>
96                    ! <proc-body> <paragraph>
97  <paragraph> ::= <id> . <sentence-list>
98                    ! <id> SECTION .
```

```
99  <sentence-list> ::= <sentence>
100                   | <sentence-list> <sentence> .
101  <sentence> ::= <imperative>
102               | <conditional>
103               | ENTER <id> <opt-id>
```

This construct is not implemented. An ENTER allows statements from another language to inserted in the source code.

```
104  <imperative> ::= ACCEPT <subid>
105                 | <arithmetic>
106                 | CALL <lit> <using>
```

This is not implemented.

```
107                 | CLOSE <id>
108                 | <file-act>
109                 | DISPLAY <lit/id> <opt-lit/id>
110                 | EXIT <program-id>
111                 | GO <id>
112                 | GO <id-string> DEPENDING <id>
113                 | MOVE <lit-id> TO <subid>
```

```
114                 ! OPEN <type-action> <id>
115                 ! PERFORM <id> <thru> <finish>
116                 ! <read-id>
117                 ! STOP <terminate>
118   <conditional> ::= <arithmetic> <size-error> <imperative>
119                   ! <file-act> <invalid> <imperative>
```

A BST operator is output to complete the branch from production 64.

```
120                   ! IF <condition> <action> ELSE <imperative>
121                   ! <read-id> <special> <imperative>
122   <Arithmetic> ::= ADD <l/id> <opt-l/id> TO <subid> <round>
123                 ! DIVIDE <l/id> INTO <subid> <round>
124                 ! MULTIPLY <l/id> BY <subid> <round>
125                 ! SUBTRACT <l/id> <opt-l/id> FROM
                      <subid> <round>
126   <file-act> ::= DELETE <id>
127                ! REWRITE <id>
128                ! WRITE <id> <special-act>
129   <condition> ::= <lit/id> <not> <cond-type>
```

```
130  <cond-type> ::= NUMERIC
131                | ALPHABETIC
132                | <compare> <lit/id>
133  <not> ::= NOT
     NEG
134          | <empty>
135  <compare> ::= GREATER
136              | LESS
137              | EQUAL
138  <ROUND> ::= ROUNDED
139            | <empty>
140  <terminate> ::= <literal>
141                | RUN
142  <special> ::= <invalid>
143              | END
144  <opt-id> ::= <subid>
145             | <empty>
146  <action> ::= <imperative>
147             | NEXT SENTENCE
```

```
148  <thru> ::= THRU <id>
149          | <empty>
150  <finish> ::= <i/id> TIMES
151             | UNTIL <condition>
152             | <empty>
153  <invalid> ::= INVALID
154  <size-error> ::= SIZE ERROR
155  <special-act> ::= <when> ADVANCING <how-many>
156                  | <empty>
157  <when> ::= BEFORE
158           | AFTER
159  <how-many>::= <integer>
160              | PAGE
161  <type-action> ::= INPUT
162                  | OUTPUT
163                  | I-O
164  <subid> ::= <subscript>
165            | <id>
166  <integer> ::= <input>
```

```
167  <id> ::= <inout>
168  <l/id> ::= <inout>
169         | <subscript>
170         | ZERO
171  <subscript> ::= <id> ( <inout> )
172  <opt-l/id> ::= <l/id>
173              | <empty>
174  <nn-lit> ::= <lit>
175             | SPACE
176             | QUOTE
177  <literal> ::= <nn-lit>
178              | <inout>
179              | ZERO
180  <lit/id> ::= <l/id>
181             | <nn-lit>
182  <opt-lit/id> ::= <lit/id>
183                | <empty>
184  <program-id> ::= <id>
185                | <empty>
```

186 <read-id> ::= READ <id>

APPENDIX C. COED USER'S MANUAL

This manual describes the facilities of the COBOL editor, COED, which is designed to run on the UNIX operating system at NPS. This editor is the first, and mandatory, step for writing a COBOL program to be executed by COIN, the COBOL interpreter which also run on UNIX at NPS.

The following items apply to programs written using COED:

1) Lower case ASCII characters are used except for line numbers, as noted below.

2) Each COBOL sentence must have a unique line number, which is entered by the user using the formats noted below.

3) Periods, blanks, tabs, and newline characters may be freely inserted in a line to improve readibility. A period followed by a newline character, however, is treated as an end of line indicator.

4) The COBOL sentence must be less than 256 characters.

5) Each COBOL sentence must end with a period and newline character '.(CR)'. No blanks are allowed between these characters.

6) Lines may be entered in any order; the editor places them in ascending numeric order.

7) Four line numbers must be present with the following format:

A0000 identification division.

B0000 environment division.

C0000 data division.

D0000 procedure division.

The following items apply to line numbers:

1) A line number consists of one of the upper case ASCII characters followed by four digits.

2) The ASCII alphabetic characters have special meanings within the program. The character indicates the COBOL division into which the line will be inserted.

AXXXX represents the IDENTIFICATION DIVISION

BXXXX represents the ENVIRONMENT DIVISION

CXXXX represents the DATA DIVISION

DXXXX represents the PROCEDURE DIVISION

3) The four digits indicate the position of the line within the division.

When the editor is called it will respond with an asterisk, '*'; it is then ready to accept a new line. This asterisk must always be followed by a line number.

ELEMENT:

ADD A NEW LINE TO THE PROGRAM

FORMAT:

<line number> <COBOL sentence> .(CR)

DESCRIPTION:

The editor will find the proper location corresponding to the line number. The line number and the COBOL sentence up to the EOL indicator are entered into the memory area. If the sentence is in the Procedure division it is converted to token numbers before being entered. Otherwise, the ASCII characters are inserted directly.

EXAMPLES:

A0010 program-id.(CR) example.

C0040 01 data-in pic x(80).(CR)

D1900 add x to y.(CR)

ELEMENT:

ADD A BLANK LINE

FORMAT:

<line number>(CR)

<line number> .(CR)

DESCRIPTION:

A blank line is inserted in the program. This is useful to improve readibility and may be used freely. Note that the first option will result in a cleaner printing.

EXAMPLES:

A0100.(CR)

C0040(CR)

ELEMENT:

COMMENT

FORMAT:

<line number> * <ASCII string> .(CR)

DESCRIPTION:

If the line number is followed by a single blank and then an '*', the interpreter will ignore the rest of the line up to the period-newline. Comments may be freely inserted in any division and are always stored as an ASCII string.

EXAMPLES:

D0100 * This is a sample comment. It may be a maximum of 256 characters.

ELEMENT:

DELETE A LINE

FORMAT:

<line number> d(CR)

DESCRIPTION:

The line is found in the program area and all characters or token numbers are deleted. Note that no blanks are allowed between the 'd' and the '(CR)'.

EXAMPLES:

B0200 d(CR)

ELEMENT:

MODIFY AN OLD LINE

FORMAT:

<line number> m(CR)

DESCRIPTION:

The editor will print the line as written, then print the line number and wait for instructions. The user has three options:

1) type <COBOL sentence> to replace the present sentence ending with a '.(CR)',

2) type d(CR) to delete the line,

3) type (CR) to leave the line as written.

Note that no character editing capability is available.

EXAMPLES:

```
*D0040 m(CR)
D0040 add x to y.
*D0040 add z to y.(CR)
*
```

ELEMENT:

PRINT

FORMAT:

<line number>,<line number> (CR)

<line number>,E (CR)

<line number>, (CR)

DESCRIPTION:

The first format causes all lines to be printed starting at the first line number and continuing until the second line number has been printed.
The second format causes all lines to be printed starting at the first line and continuing until the end of file indicator.
The third format causes that line number only to be printed.

EXAMPLES:

A0010,B0000 (CR)

A0000,E (CR)

A0000, (CR)

ELEMENT:

QUIT EDITOR

FORMAT:

q(CR)

quit(CR)

DESCRIPTION:

Edit is terminated. The program is written into low memory, closing up the work space. The symbol table is written directly following the program's EOF, and any deleted entries are set to 'NUL'.

APPENDIX D. COIN USER'S MANUAL

This manual describes the subset of MICRO-COBOL currently accepted by the COBOL interpreter, COIN. The following conventions are used in explaining the formats:

1) Elements enclosed in broken braces < > are complete entities and are described elsewhere in the manual.

2) Elements enclosed in stacks of braces { } are choices, one of which must be chosen.

3) Elements enclosed in brackets [ ] are optional.

4) All elements in capital letters are reserved words and must be spelled exactly so that they can be mapped into the correct token numbers. When using CUED and COIN however, the entire COBOL program is entered in lower case characters.

User defined variables and paragraph names are indicated as lower case. These names are restricted to 16 characters in length. Variable names must begin with an alphabetic character and must be declared in the Data division. Paragraph names must begin with a numeric character.

ELEMENT:

IDENTIFICATION DIVISION Format

FORMAT:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. <comment>.
[AUTHOR. <comment>.]
[DATE-WRITTEN. <comment>.]
[SECURITY.<comment>.]
```

DESCRIPTION:

This division provides information for program identification for the reader.

EXAMPLES:

```
identification division.
program-id. sample.
```

ELEMENT:

ENVIRONMENT DIVISION Format

FORMAT:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
[SOURCE-COMPUTER. <comment>.]
[OBJECT-COMPUTER.<comment>.]
[INPUT-OUTPUT SECTION.
FILE-CONTROL.
<file-control-entry>.]
```

DESCRIPTION:

At present, this information is ignored. Once the COBOL calculator is operational, it will be needed for specifying input and output files.

ELEMENT:

DATA DIVISION Format

FORMAT:

```
DATA DIVISION.
[FILE SECTION.
[FD filename
    [<record-description-entry] ...] ...
```

DESCRIPTION:

This section describes how the data is structured.

ELEMENT:

*<comment>

FORMAT:

* <any string of ASCII characters>

DESCRIPTION:

If a space and an '*' directly follow the line number, all characters until the '.(CR)' are ignored by the interpreter.

EXAMPLES:

D0010 * this is a sample comment.(CR)

ELEMENT:

<data-description-entry> Format

FORMAT:

```
level-number {data-name}          .
             {FILLER}
[PIC character-string
      [VALUE character-string]]
```

DESCRIPTION:

This statement describes that specific attributes of the data. The VALUE clause is used to initialize a variable to a specific value when the program begins execution.

EXAMPLES:

```
01 data-in.
   02 part   pic x(5).
   02 num    pic 99 value 0.
```

ELEMENT:

PROCEDURE DIVISION Format

FORMAT:

```
PROCEDURE DIVISION.
[paragraph-name.] <sentence> [<sentence>...] ...
```

DESCRIPTION:

This division contains all the executable COBOL statements.

ELEMENT:

<sentence>

FORMAT:

<imperative-statement>

<condition-statement>

ELEMENT:

<imperative-statement>

FORMAT:

ACCEPT

ADD

CLOSE

DISPLAY

DIVIDE

GO

MOVE

MULTIPLY

OPEN

PERFORM

READ

STOP

WRITE

DESCRIPTION:

ACCEPT, DISPLAY, OPEN, and CLOSE are currently ignored by the interpreter because they depend on devices external to the machine.

ELEMENT:

<conditional-statement>

FORMAT:

IF

ELEMENT:

ADD

FORMAT:

```
ADD {identifier-1} TO identifier-2
    {literal}
```

DESCRIPTION:

This instruction adds identifier-1/literal to identifier-2 and stores the result in identifier-2.

EXAMPLES:

```
add 10 to total.
```

ELEMENT:

DIVIDE

FORMAT:

DIVIDE {identifier-1} INTO {identifier-2}
{literal}

DESCRIPTION:

This instruction divides identifier-1/literal into identifier-2 and stores the result into identifier-2.

EXAMPLES:

divide 5 into total.

ELEMENT:

GO

FORMAT:

GO paragraph-name

DESCRIPTION:

The GO instruction causes an unconditional branch to the specified paragraph name.

EXAMPLES:

go 10-begin.

ELEMENT:

IF

FORMAT:

IF <condition> {imperative-1} [ELSE imperative-2]

{NEXT}

DESCRIPTION:

If the condition evaluates true either imperative-1 or the next sentence in the program is executed. If the condition is false, either imperative-2 is executed, or the next sentence is skipped.

EXAMPLES:

if a greater b go 10-begin else go 30-end.

if x numeric next.

ELEMENT:

MOVE

FORMAT:

```
MOVE {identifier-1} TO identifier-2
     {literal}
```

DESCRIPTION:

Either the value stored in identifier-1 or the literal value is stored into identifier-2.

EXAMPLES:

```
move 10 to subtotal.
move sub-total to total.
```

ELEMENT:

MULTIPLY

FORMAT:

MULTIPLY {identifier-1} BY identifier-2
{literal}

DESCRIPTION:

The multiply instruction causes identifier-1 to be multiplied by identifier-2. The result must be able to be stored into identifier-2.

EXAMPLES:

multiply 3.5 by 3.99.

ELEMENT:

PERFORM

FORMAT:

1. PERFORM paragraph-name [THRU paragraph-name-2]

2. PERFORM paragraph-name [THRU paragraph-name-2]

{identifier} TIMES

{integer}

3. PERFORM paragraph-name [THRU paragraph-name-2]

UNTIL <condition>

DESCRIPTION:

This instruction causes an unconditional branch to the specified paragraph name. When the return conditions are met, execution resumes at the instruction following the PERFORM statement. At present, PERFORM statements can be nested eight levels deep.

EXAMPLES:

```
perform 10-begin.
perform 10-read thru 20-end-read.
perform 10-read until card-in numeric.
```

ELEMENT:

READ

FORMAT:

READ file-name

DESCRIPTION:

Data is read from some external device into the Data division entries for that file-name.

EXAMPLES:

read data-in.

ELEMENT:

STOP

FORMAT:

STOP [RUN]

DESCRIPTION:

This statement causes the interpreter to halt and return control to the monitor.

EXAMPLES:

```
stop.
```

ELEMENT:

WRITE

FORMAT:

WRITE file-name

DESCRIPTION:

This statement causes the values in the Data division associated with the specified file name to be printed on the line printer.

EXAMPLES:

write total.

ELEMENT:

<condition>

FORMAT:

RELATIONAL CONDITION:

{identifier-1} [NOT] {GREATER} {identifier-2}

{literal-1} {LESS} {literal-2}

{EQUAL}

CLASS CONDITION:

identifier [NOT] {ALPHABETIC}

{NUMERIC}

DESCRIPTION:

Identifiers are tested as specified.

EXAMPLES:

total greater 100.

number not numeric.

character equal beta.

```
#define     EOL         0177            /* end of line character     */
#define     MEMSIZE     4096            /* size of memory area       */
#define     RWORDS      53              /* number of reserved words */
#define     SPACE       0
#define     TAB         1
#define     PERIOD      2
#define     FILLER      32
#define     ERR         0
#define     ADDLINE     1
#define     MODIFY      2
#define     DELETE      3
#define     BLANK       4
#define     PRINT       5
#define     QUIT        6
#define     LESS        3
#define     EQUAL       2
#define     GREATER     1
#define     TRUE        1
#define     FALSE       0
#define     CR          '\r'
#include    "table"


/* The following characters have special significance in COED        */

/*  'A0000' the first line number and mandatory "IDENTIFICATION      */
/*          DIVISION." line entry.                                   */
/*  'B0000' mandatory "ENVIRONMENT DIVISION." line entry             */
/*  'C0000' mandatory "DATA DIVISION."        line entry             */
/*  'D0000' mandatory "PROCEDURE DIVISION."   line entry             */

/*  For the following items 'CR' is the newline character for UNIX   */
/*      and 'XXXXX' is a correctly formed line number.               */

/*  ".CR"                               end of line                  */
/*  "XXXXX dCR"                         delete this line             */
/*  "XXXXX mCR"                         modify this line             */
/*  "XXXXX * <ASCII string>.CR "        comment line                 */
/*  "q CR" or "quit CR"                 quit the editor              */

/*  NOTE: LOWER CASE ASCII characters will be used by the programmer */
/*          except for comments and the 1st char in the line number.*/

/*  This version will generate sequential line numbers.  If the      */
/*  user wishes to change the line number, type the new line number  */
/*  immediately following the 1st number.  If the user wishes to     */
/* increment the numbers by one. Instead of 10, type a line number   */
/*  with a least significant digit of 1 thru 9, i.e. not 0.          */

char    lb[256];                        /* input line buffer        */
char    tb[256];                        /* token buffer             */
char    st[4096];                       /* print name table         */
char    cln[5];                         /* current line # buff      */
char    temp;                           /* temporary char buff      */
char    mem[MEMSIZE];                   /*  memory size             */

int     lbp;                            /* line buffer pointer      */
int     tbp;                            /* token buffer pointer     */
int     sbp;                            /* 1st char of sym in lb    */
int     cp1;                            /* top of area 1            */
int     cp2;                            /* bottom of area 2         */
int     action;                         /* editor action            */
int     ea;                             /* top of symbol table      */
int     em;                             /* top of memory            */
int     del;                            /* delete flag              */
int     cnt     0;                      /* line # increment         */

                        /* ERROR */
```

```
error(s)
    char *s;  {

    /* When an error is detected, a '^' is inserted at the point    */
    /* that the error was detected and the appropriate message is   */
    /* printed.                                                     */

    int n;
    action = ERR;
    for (n=0; n<lbp; n++)                       /* insert an ^ at error */
        putchar (' ');
    putchar ('^');
    putchar (CR);
    printf ("%s CR",s);                         /* print the error msg  */
}

                            /* INPUT */
input (mod)
    int mod;  {

    /* This routine accepts characters from the console and moves   */
    /*  them to the input buffer, LP.  It then will analyse the     */
    /*  input to determine what action the editor will take.        */

    /* The parameter MOD indicates if the action involves a new     */
    /*  line or a modification of an old line.                      */

int  next,digit,number,i;
putchar ('*');                                  /* input request char   */
action = ADDLINE;

if (mod == 0)  {                                /* input a new line     */
    lbp = 0;
    if (lb[0] >= 'A' && lb[0] <= 'D')  {        /* line # in line buff  */
        if (lb[4] != '0')
            /* if the last digit of the line # is not '0' increment
               the line number by 10 otherwise by 1.                */
            digit = 4;
        else
            digit = 3;
        next = TRUE;
        while (next)  {
            next = FALSE;
            number = lb[digit]+1;               /* increment line #     */
            if (number>'9' && digit!=0)  {      /* check for overflow   */
                number = '0';
                next = TRUE;
            }
            lb[digit] = number;                 /* put number in buff   */
            digit = digit-1;
        }
        for (lbp=0; lbp<5; lbp++)               /* print the new #      */
            putchar (lb[lbp]);
    }
    else  {                                     /* print curr line #    */
        for (lbp=0; lbp<5; lbp++)
            putchar (cin[lbp]);
        }

    lb[lbp] = getchar();                        /* read the 1st char    */

    if (lb[lbp]>='A' && lb[lbp] <='D')  {       /* check if a line #    */
        lb[0] = lb[lbp];                        /* write over 1st #     */
        for ( lbp=1; lbp<5; lbp++)  {           /* read the rest of #   */
            lb[lbp] = getchar();
            if (lb[lbp]<'0' || lb[lbp]>'9')
                error("INVALID LINE NUMBER");
        }
        lb[lbp] = getchar();
        while (lb[lbp] == ' ')                  /* skip blanks          */
            lb[++lbp] = getchar();
    }
}
```

```
else  {
    for (lbp=0; lbp<5; lbp++)                   /* print the line c      */
        putchar (lb[lbp]);
    lb[lbp] = getchar();
}

if (action == ADDLINE)  {
    /* the first characters were a line number -- no errors             */

    while (lb[lbp]==' ' || lb[lbp]=='')
        lb[++lbp] = getchar();                  /* wait for 1st non-blank   */

    switch (lb[lbp])  {                         /* depends on 1st non-blank */

        case CR:                                /* enter a BLANK LINE       */
            action = BLANK;
            break;

        case '.':                               /* PRINT a block of lines   */
            action = PRINT;
            while (lb[lbp] != CR)                   /* read 2nd line number */
                lb[++lbp] = getchar();
            break;

        case 'd':                                   /* DELETE a line        */
            lb[++lbp] = getchar();
            if (lb[lbp] == CR)                  /* next char must = newline */
                action = DELETE;
            break;

        case 'm':                               /* print and MODIFY a line  */
            lb[++lbp] = getchar();
            if (lb[lbp] == CR)                  /* next char must = newline */
                action = MODIFY;
            break;

        case 'q':                                   /* QUIT the editor      */
            action = QUIT;
            lb[0] = 'E';
            for (lbp=1; lbp<5; lbp++)
                lb[lbp] = '0';                      /* put 'E0000' in LB    */
            lb[lbp] = getchar();                    /* get next char        */
    }

    if (action == ADDLINE)  {                       /* ADD a new line       */
        while (1)  {
            lb[++lbp] = getchar();                  /* read the console     */
            if (lbp > 254)  {
                error("LINE GREATER THAN 256 CHARACTERS");
                return;
            }
            if (lb[lbp] == '.')  {                  /* if '.' is read, check
                                                       if it is the EOL     */
                lb[++lbp] = getchar();
                if (lb[lbp] == CR)                  /* a ".(cr)" is EOL     */
                    break;
            }
        }
    }
}
while (lb[lbp] != CR)                               /* wait for newline     */
    lb[lbp] = getchar();
lb[lbp] = EOL;
return;
}

                    /* COMPARE */
compare ()  {

    /* compares the current line number buffer, CLN, with the line  */
    /* number in the 1st five char in the input line buffer, LB     */
    /*       if  CLN > LB    then    C = 1                          */
    /*           CLN = LB            C = 2                          */
```

```
    /*          CLN < LB            C = 3                                   */

    int c,k;
    for (k=0; k<5; k++)  {
        if (cln[k] == lb[k])                    /* char are equal: cont */
            c = EQUAL;
        else if (cln[k] < lb[k])  {             /* not equal: halt      */
            c = LESS;
            break;
        }
        else if (cln[k] > lb[k])  {             /* not equal: halt      */
            c = GREATER;
            break;
        }
    }
    return(c);
}

                    /* UPLINE */
upline()  {

    /* Finds the next larger line number and moves the current line */
    /*  to the low memory area.                                     */

    int j;
    temp = '0';
    while (temp != EOL)  {                  /* move char until end  */
        temp = mem[++cp1] = mem[cp2++];         /* of the line      */
    }
    for (j=0; j<5; j++)                     /* move next line # to  */
        cln[j] = mem[cp2+j];                    /* curr line buff   */
    return;
}


                    /* DOWNLINE */
downline()  {

    /*  Finds the next smaller line number and makes it the current */
    /*  line in the high core area.  The pointer cp2 indicates the  */
    /*  begining address of the current line.                       */

    int j;
    mem[--cp2] = mem[cp1--];                    /* move EOL char    */
    temp = mem[cp1];
    while (temp != EOL)  {                  /* move char until EOL  */
        mem[--cp2] = temp;
        temp = mem[--cp1];
    }
    for (j=0; j<5; j++)                     /* move line # to curr  */
        cln[j] = mem[cp2+j];                /*   line number buff   */
    return;
}

                    /* SEARCH */
search()  {

    /*  Searches the memory area until the line number is found.    */
    /*  ( or the next larger line number for an add request).       */
    /*  This line is made the current line number and pointer cp2   */
    /*  indicates the first address.                                */

    int c;
    if (action != ERR)  {           /* no error has been detected   */
        c = compare();              /* compare New and Current LN's */

        switch (c)  {

            case GREATER:                       /* NLN < CLN        */
                while (c == GREATER)  {         /* move down in memory */
                    downline();
                    c = compare();
                }
```

```
                if (c == LESS)                  /* passed line move up 1*/
                    upline();
                break;

            case EQUAL:                         /* NLN = CLN   halt     */
                break;

            case LESS:                          /* NLN > CLN  move up   */
                while (c == LESS)  {
                    upline();
                    c = compare();
                }
        }

        if ((action==ADDLINE || action==BLANK) && c==EQUAL)
                            /* duplicate line number for a new line */
            error("DUPLICATE LINE NUMBER");
        else
            if ((action==MODIFY || action==DELETE || action==PRINT)
                    && c != EQUAL)
                    /* a modify, delete, or print request and the
                       line is not found.                          */
                error("LINE NUMBER NOT FOUND");
    }
    return;
}

                        /* GETSYM */
getsym()  {

    /* Finds the length of the symbol print name in the line buff.  */
    /*      SBP indicates the 1st character of the symbol           */
    /*      LBP indicates the 1st blank, tab, or period             */

    int l;
    sbp = lbp;

    for (l=0; l<17; l++)  {
        if (lb[lbp]==' ' || lb[lbp]=='' || lb[lbp]==',')
            return(l);
        else
            lbp++;
    }
    return (l);                                 /* return the length    */
}

                        /* MATCH */
match (l)
    int l;  {

    /* Compares the symbol print name indicated by the SBP to all   */
    /* the print names in the symbol table until a match is found.  */
    /* The parameter "l" is the length of the print name.           */

    int addr,eq,k,tn;

    addr = -16;                                 /* begining of sym tab  */
    del = 0;                                    /* delete is false      */
    eq = 0;                                     /* equal is false       */

    while (eq == 0 && addr < ea)  {
    /* compare until equal is true or the end of the symbol table   */
        addr =+ 16;                             /* next address in S.T. */
        eq = 1;
        if (st[addr] < 0)  {                    * the delete flag is set*/
            st[addr] =& 0177;
            del = 1;
        }
        for (k=0; k<l; k++)                     * compare all char     */
            if (lb[sbp+k] != st[addr+k])
                eq = 0;
        if (eq == 1)
        /* insure that the next  char in sym tab is ' ' (end of name)*/
```

```
            if (k<15 && st[addr+k+1] != '/')
                eq = 0;

        if (del == 1 && eq == 0)  {
        /* found a deleted name and it doesn't match: restore flag  */
            del = 0;
            st[addr] =| 0200;
        }
    }

    if (eq == 1)  {
                        /* name matched, return the token number     */
        tn = addr >> 4;
        return (tn);
    }
    else
        return (-1);
}

                        /* ENTERPN */
enterpn(l)
    int l;  {

    /* Copies the symbol print name from the line buffer, located    */
    /* at SBP with length 'l', into the next empty symbol table      */
    /* location.                                                     */

    int a,k;
    a = 0;
    while ( st[a] != '/')  {
        /* Find the 1st entry where the 1st char is '/', i.e. empty */
        a =+ 16;
        if (a >= ea)
            return(-1);
    }
    for (k=0; k<l; k++)                 /* Move the print name       */
        st[a++] = lb[sbp++];
    st[a] = '/';                        /* Insert end of name char   */
    return (a>>4);                      /* Return the token number   */
}

                        /* TOKENIZE */
tokenize()  {

    /* Converts a line in the Procedure Division to token numbers    */
    /* and puts them in the token buffer, TB.  The TB is then moved  */
    /* to the memory area.  If the line is in the Data Division an   */
    /* entry in the symbol table is made.  Lastly, all lines in the  */
    /* Identification, Environment, and Data Divisions are copied    */
    /* directly from the line buffer into the memory working area    */
    /* (i.e. in ASCII).                                              */

    int j,tn,length;
    char t;

    if (lb[0]=='D' && (lb[1]!='0' || lb[2]!='0' || lb[3]!='0' ||
        lb[4]!='0') && lb[6] != '*')  {
        /* line is in the Proc. Div and not a comment or the header */
        tbp = lbp = 0;
        while (tbp < 5)                         /* Move the line # to TB*/
            tb[tbp++] = lb[lbp++];
        while (lb[lbp] != EOL)  {               /* find tokens until EOL*/
            switch (lb[lbp])  {
                /* switch is based on the next character in LB       */

                case ' ':
                    tb[tbp++] = SPACE;          /* Token # for blank     */
                    lbp++;
                    break;

                case '':
                    tb[tbp++] = TAB;            /* Token # for TAB       */
                    lbp++;
```

```
                break;

            case '.':
                tb[tbp++] = PERIOD;         /* token # for PERIOD   */
                lbp++;
                break;

            case '0':                       /* Symbol is a LITERAL: */
            case '1':                       /* check the sym table  */
            case '2':                       /* for a match else     */
            case '3':                       /* make a new entry     */
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
            case '+':
            case '-':
                length = getsym();
                if (length == 17)
                /* literals can be more than 16 char: reset lbp */
                    lbp--;
                tn = match(length);
                if (tn < 0)  {              /* no match found   */
                    tn = enterpn(length);
                    if (tn < 0)  {
                        error("SYMBOL TABLE FULL");
                        return;
                    }
                }
                t = tn;
                tb[tbp++] = t;
                break;

            default:                        /* a symbol print name   */
                length = getsym();
                if (length == 17)  {
                    error("NAME GREATER THAN 16 CHAR");
                    return;
                }
                else  {
                    tn = match(length);
                    if (tn < 0)  {
                        error("NAME NOT IN SYMBOL TABLE");
                        return;
                    }
                    else  {
                        t = tn;             /* put token in TB  */
                        tb[tbp++] = t;
                    }
                }
        }
    }
    tb[tbp++] = EOL;                        /* insert EOL in TB */
    if ((cp2-cp1) > tbp)
        /* Insure there is room in memory for the buffer        */
        for (j=0; j<tbp; j++)
            mem[++cp1] = tb[j];
    else
        error("MEMORY FULL");
}
else  {
    if (lb[0] == 'C')  {
                                    /* line is in the Data Div  */
        lbp = 5;                            /* search fm line # */
        while (lb[lbp]== ' ' || lb[lbp]== '')
            lbp++;
        if (lb[lbp] == '0' || lb[lbp] == '7')  {
                                    /* entry is a data definition   */

            while ((lb[lbp]<'a' || lb[lbp]>'z') && lb[lbp]!='.')
                                    skip to the 1st alphabetic char or EOL   */
```

```
                lbp++;
            j = getsym();
            if (j == 17)  {                     /* name is too long */
                error("NAME GREATER THAN 16 CHAR");
                return;
            }
            else if (j == 0)  {
                error("INCORRECT DATA DIVISION ENTRY");
                return;
            }
            else  {
                tn = match(j);
                if (tn == -1)  {                /* NEW print name   */
                    tn = enterpn(j);
                    if (tn < 0)  {
                        error("SYMBOL TABLE FULL");
                        return;
                    }
                }
                else if (tn>=0 && tn != FILLER)  {
                    /* found a match and not the name "filler"  */
                    error("DUPLICATE SYMBOL NAME");
                    return;
                }
            }
        }
        while (lb[lbp] != EOL)                  /*  LBP to end of line  */
            lbp++;
        if ((cp2-cp1) > lbp)                    /* Insure room in memory*/
            for (j=0; j<=lbp; j++)              /* Move LB to memory    */
                mem[++cp1] = lb[j];
        else
            error("MEMORY FULL");
    }
return;
}

                        /* DECODE */
decode(t)
    char t;  {

    /* finds the address of token number "t" and copies the print   */
    /* name into the line buffer. LB.                               */

    int addr,m;
    addr = t << 4;                              /* address of the token */
    m = addr+16;
    while (st[addr] != '/' && addr < m)         /* copy name until end  */
        lb[lbp++] = st[addr++];
    return;
}

                        /* BLANK */
blank()  {

    /* Inserts a blank line into any division.  */

    int j;
    for ( j=0; j<3; j++)
        mem[++cp1] = lb[j];                     /* Move the line number */
    mem[++cp1] = EOL;                           * Insert end of line   */
    return;
}

                        /* PRINT */
print()  {

    /* Prints the current line on console.  If the line is in the   */
    /* Proc. Div., it is decoded before printing.  This line is     */
    /* the current line, i.e. both the CL# and CP2 are unchanged.   */

    int j;
```

```
    if (cln[0]=='D' && (cln[1]!='0' || cln[2]!='0' || cln[3]!='0' ||
            cln[4]!='0') && mem[cp2+6]!='*')  {
    /* line is in the Proc Div and not a comment or the header line */
        sbp = cp2;
        lbp = 5;
        for (j=0; j<5; j++)                     /* print the line num   */
            putchar (mem[sbp++]);
        temp = mem[sbp++];
        while ( temp != EOL)  {                 /* Decode token number  */
            decode(temp);
            temp = mem[sbp++];
        }
        for (j=5; j<lbp; j++)                   /* print the LB         */
            putchar (lb[j]);
    }
    else  {
        sbp = cp2;
        temp = mem[sbp++];
        while ( temp != EOL)  {                 /* print line fm memory */
            putchar (temp);
            temp = mem[sbp++];
        }
    }
    putchar(CR);                                /* decode the EOL char  */
    return;
}

                        /* DELETE */
delete(mod)
    int  mod;  {

    /* Deletes current line from memory by moving CP2.  If the line */
    /* is in Data Div, the delete flag is set for the appropriate   */
    /* symbol table entry.  If MOD is true, DELETE was called from  */
    /* MODIFY routine and the line buffer, LB, must be saved.       */

    int tn,j,length;
    if (lb[0] != 'C')                           /* line is not in Data  */
        while (mem[cp2++] != EOL);              /* so move CP2 to end   */
    else  {
        lbp = 0;
        if (mod)  {                             /* save the line buff   */
            tbp = 0;
            while (lb[lbp] != EOL)
                tb[tbp++] = lb[lbp++];
            tb[tbp] =lb[lbp];
            lbp = 0;
        }
        lb[0] = mem[cp2++];
        while (lb[lbp] != EOL)                  /* move old line to LB  */
            lb[++lbp] = mem[cp2++];
        lbp = 5;
        while (lb[lbp] == ' ' || lb[lbp] == '')
            lbp++;                              /* skip blank char      */
        if (lb[lbp] == '0' || lb[lbp] == '7')  {
                    /* not an FD entry, a comment, or a blank line  */
            while (lb[lbp] < 'a' || lb[lbp] > 'z')
                lbp++;                          /* skip to 1st name     */
            length = getsym();
            tn = match (length);                /* find symbol name     */
            if (tn >= RWORDS)  {                /* not a reserver word  */
                j = tn<<4;
                st[j] |= 0200;                  /* set delete flag      */
            }
        }
        lbp = 0;
        if (mod)  {                             /* restore line buff    */
            tbp = 0;
            while (tb[tbp] != EOL)
                lb[lbp++] = tb[tbp++];
            lb[lbp++] = tb[tbp];
            lbp = 0;
        }
```

```
    }
    for (j=0; j<5; j++)                         /* set CLN to next line */
        cln[j] = mem[cp2+j];
    return;
}

                        /* MODIFY */
modify()  {

    /* calls the routines to allow the modification of a line        */

    print();
    input(1);

    switch (action)  {

        case ADDLINE:                       /* Replace old line with new*/
            delete(1);
            tokenize();
            break;

        case DELETE:                        /* delete the old line       */
            delete(0);
            break;

        case BLANK:                    /* do not change the old line    */
            lbp = 0;
            break;

        default:
            error("ILLEGAL ACTION FOR LINE MODIFICATION");
    }
    return;
}

                        /* PRINT BLOCK */
printblk()  {

    /* Starts at the 1st line number in the LB and prints all lines */
    /* until the current line matches the 2nd line number in the LB.*/
    /* If the 1st char following the comma is a CR, one line is      */
    /* printed.  If the 1st char following the comma is a 'E', all   */
    /* lines are printed until the End Of File is encountered.       */
    /* Allowable forms:     XXXXX,XXXXX CR
                            XXXXX,E CR
                            XXXXX,CR                                 */

    int c,i;
    if (lb[6] != EOL)
        /* if not a newline, write the 2nd line number over the 1st */
        for (i=0; i<5; i++)  {
            lb[i] = lb[i+6];
            if (lb[0] == 'E')
                break;
        }
    c = compare();
    while (c != GREATER)  {
    /* print lines until the 2nd line number is found or exceeded    */
        print();
        upline();
        c = compare();
        if (cln[0] == 'E' && lb[0] == 'E')  {
            print();                            /* print the EOF line    */
            break;
        }
    }
    return;
}

                        /* QUIT */
quit()  {

    /* The file has been moved to the low area of memory and cpl     */
```

```
    /* points to the last address of the file.  The symbol table    */
    /* must be cleaned up by setting all deleted p-names to zeros.  */

    int a, j;
    for (j=0; j<6; j++)                     /* move End Of File line    */
        mem[++cp1] = mem[cp2++];
    a = 0;
    while (a < ea)  {                       /* Check entire S.T.        */
        if (st[a] < 0)  {                   /* If the delete flag is set*/
            for (j=0; j<16; j++)            /* set the entry to '/'     */
                st[a+j] = '/';
        }
        a =+ 16;                            /* next p-name entry        */
    }
    return;
}

                        /* INIT */
init()  {

    /* Checks if file is new.  If so, the top of file and end of    */
    /* file lines are inserted.                                     */
    /* If old or new, the EOF is made the current line number       */

    int j,k,l;
    l = 0;
    for (j=0; j<RWORDS; j++)            /* initialize the symbol table  */
        for (k=0; k<16; k++)
            st[l++] = sym[j][k];

    ea = 2048;                          /* set the length of sym tab    */
    j = RWORDS*16;
    while (j < ea)  {
        st[j++] = '/';
    }
    cp2 = MEMSIZE;                      /* CP2 starts at top of memory  */
    if (cp1 == -1)  {                   /* a new file, i.e. no chars    */
        mem[++cp1] = EOL;
        mem[++cp1] = 'A' - 1;               /* Set top of file          */
        mem[++cp1] = EOL;

        mem[++cp1] = 'E';                   /* set end of file line     */
        mem[++cp1] = 'O';
        mem[++cp1] = 'F';
        mem[++cp1] = ' ';
        mem[++cp1] = ' ';
        mem[++cp1] = EOL;
    }
    downline();                         /* makes EOF the current line   */
    return;
}

                        /* HEADING */
heading()  {

    printf ("COED: Cobol Editor.  Version 1.0 CR");
}

                        /* MAIN */
main()  {

    int h;
    cp1 = -1;                           /* for TEST only                */

    heading();                          /* prints heading line          */
    init();                             /* initializing routine         */

    action = ERR;
    while (action != QUIT)  {           /* until user sez "quit"        */
        input(0);                       /* read the console             */
        search();                       /* find the desired line        */

        switch (action)  {              /* "action" is set in input     */
```

```
            case ADDLINE:                   /* ADD a new line           */
                 tokenize();
                 break;

            case MODIFY:                    /* MODIFY an old line       */
                 modify();
                 break;

            case DELETE:                    /* DELETE an old line       */
                 delete(0);
                 break;

            case BLANK:                     /* insert a BLANK LINE      */
                 blank();
                 break;

            case PRINT:                     /* print a BLOCK of lines   */
                 printblk();
                 break;

            case QUIT:                      /* QUIT the editor          */
                 quit();
        }
    }

for (h=0; h<=cpl; h++)                      /* move mem area to file    */
    putchar(mem[h]);
for (h=0; h<ea; h++)
    putchar (st[h]);

}
```

```
#define     SPACE               0
#define     TAB                 1
#define     PERIOD              2
#define     NEWLINE             3
#define     COMMA               4
#define     COMMENT             5
#define     ACCEPT              6
#define     ADD                 7
#define     TO                  8
#define     ROUNDED             9
#define     SIZE-ERROR          10
#define     CALL                11
#define     CLOSE               12
#define     DELETE              13
#define     INVALID             14
#define     DISPLAY             15
#define     DIVIDE              16
#define     INTO                17
#define     ENTER               18
#define     EXIT                19
#define     GO                  20
#define     DEPENDING           21
#define     IF                  22
#define     NEXT                23
#define     ELSE                24
#define     MOVE                25
#define     MULTIPLY            26
#define     BY                  27
#define     OPEN                28
#define     INPUT               29
#define     OUTPUT              30
#define     PERFORM             31
#define     THRU                32
#define     TIMES               33
#define     UNTIL               34
#define     READ                35
#define     END                 36
#define     STOP                37
#define     RUN                 38
#define     SUBTRACT            39
#define     FROM                40
#define     WRITE               41
#define     BEFORE              42
#define     AFTER               43
#define     ADVANCING           44
#define     PAGE                45
#define     NOT                 46
#define     GREATER             47
#define     LESS                48
#define     EQUAL               49
#define     NUMERIC             50
#define     ALPHABETIC          51
#define     EOL                 0177
#define     MEMSIZE             4096
#define     MAX                 0256
#define     ON                  0001
#define     OFF                 0000
#define     TRUE                0000
#define     FALSE               0001
#define     NUL                 '\0'
#define     EOF                 '0'
#define     add                 0
#define     sub                 1

                /*  COBOL  INTERPRETER -- COIN  */

/* The following tables are used for the arithmetic routines.  In  */
```

```
/* the implimentation of this interpreter on a microcomputer, these */
/* tables would be in ROM.  Thus the access time would be approx-   */
/* imately equal to the memory cycle time.                          */

char *addtab1 [10]  {                       /* addition sum table   */
     "0123456789",
     "1234567890",
     "2345678901",
     "3456789012",
     "4567890123",
     "5678901234",
     "6789012345",
     "7890123456",
     "8901234567",
     "9012345678",
     0
};

char *addtab2 [10]  {                       /* addition carry table */
     "0000000000",
     "0000000001",
     "0000000011",
     "0000000111",
     "0000001111",
     "0000011111",
     "0000111111",
     "0001111111",
     "0011111111",
     "0111111111",
     0
};

char *subtab1 [10]  {                       /* subtraction table    */
     "9987654321",                          /* this table assumes   */
     "1098765432",                          /* that subtab [x] [y]  */
     "2109876543",                          /* means x - y          */
     "3210987654",
     "4321098765",
     "5432109876",
     "6543210987",
     "7654321098",
     "8765432109",
     "9876543210",
     0
};

char *subtab2 [10]  {                       /* borrow table         */
     "0111111111",
     "0011111111",
     "0001111111",
     "0000111111",
     "0000011111",
     "0000001111",
     "0000000111",
     "0000000011",
     "0000000001",
     "0000000000",
     0
};

char *multab1 [10]  {                       /* LSD of product       */
     "0000000000",
     "0123456789",
     "0246802468",
     "0369258147",
     "0482604826",
     "0505050505",
     "0628406284",
     "0741852963",
     "0864208642",
     "0987654321",
     0
};
```

```
char *multab2 [10]  {                          /* MSD of product       */
     "0000000000",
     "0000000000",
     "0000011111",
     "0000111222",
     "0001122233",
     "0011223344",
     "0011233445",
     "0012234456",
     "0012344567",
     "0012345678",
     0
};

int length  0;
int errflg  OFF;
char save   ' ';
char mem[MEMSIZE];                  /* byte accessible memory array     */
int pc;                             /* program cntr--byte offset in mem */
int sptr;                           /* search buffer pntr--byte offset  */
char sbuf[16];                      /* search buffer  */
char xs,ys;                         /* sign bytes for x and y buffers   */
int x1,x2,x3,y1,y2,y3,n;
char xb[32],yb[32],temp[32],c[32];/* operand buff for math routines */
int sp  -1  ;                       /* stack pointer for PERFORM verb   */
int cntr[8] ;                       /* counter stack for PERFORM verb   */
int rtn[8]  ;                       /* rtn addr stack for PERFORM verb  */
int ext[8]  ;                       /* exit cond stack for PERFORM verb */
char psave;
int cp1,cp2;                        /* memory pointers--byte offsets    */
int begproc;          /* byte offset--init to beginning of proc div   */
int begdata;          /* byte offset--init to beginning of data div   */
int base;             /* byte offset--init to beginning symbol table  */
int condptr;                        /* condition putr for IF verb   */
char lb[256];                       /* line buffer                  */
int lbp     0;                      /* line buff pntr--byte offset  */
int cln[5];                         /* current line number          */
int x       0;
int y       1;


                    /*  READIN  */

readin()    {   /* This routine initiallzes 2 memory pointers--one  */
                /*  at the beginning and one at the end--and reads  */
                /*  in a predefined input file until eof or the 2   */
                /*  pointers meet.                                  */
    cp1 = -1;
    cp2 = MEMSIZE;
    save = getchar();
    while (save  != EOF && cp1 != cp2)  {
        mem[++cp1] = save;
        save = getchar();
    }
    if (cp1 != cp2) {
        mem[++cp1] = save;                  /* write EOF into memory */
        length = cp1;
    }
    else    {
        printf("PROGRAM EXCEEDS AVAILABLE MEMORY");
        errflg = ON;
    }
    return;
}       /*  end readin  */

                    /*  WRITEOUT  */

writeout()  {
    while ( cp1 != length)
        mem[++cp1] = mem[cp2++];
    cp1 = 0;
    loop4:
```

```
        while(cp1 != length && mem[cp1] != EOL)
            putchar(mem[cp1++]);
        if (cp1 != length)  {
            putchar('*');
            goto loop4;
        }
    return;
}       /* end writeout  */

                    /*  COMPARE  */

compare()   {   /* This routine compares the current line number    */
                /*  buffer, cln, with the first 5 characters in the */
                /*  line buffer, lb.                                */
                /*  If cln > lb a 1 is returned;                    */
                /*  If cln = lb a 2 is returned;                    */
                /*  If cln < lb a 3 is returned.                    */

    int comp,k;
    for (k=0; k<5; k++)  {
        if (cln[k] == lb[k])                /* char are equal: cont */
            comp = EQUAL;
        else if (cln[k] < lb[k])  {         /* not equal: halt      */
            comp = LESS;
            break;
        }
        else if (cln[k] > lb[k])  {         /* not equal: halt      */
            comp = GREATER;
            break;
        }
    }
    return(comp);
}       /*  end compare  */

                    /*  UPLINE  */

upline()  { /* This routine finds the next larger line number and  */
            /*  moves the current line to low memory area.        */

    int j, scratch;
    scratch = '0';
    while (scratch != EOL)  {               /* move char until end  */
        scratch = mem[++cp1] = mem[cp2++];      /* of the line      */
    }
    for (j=0; j<5; j++)                     /* move next line # to  */
    cln[j] = mem[cp2+j];                        /* curr line buff   */
    return;
}       /*  end upline  */

                    /*  DOWNLINE  */

downline()  {   /* This routine finds the next smaller line number  */
                /* and makes it the current line in high core--cp2  */
                /* indicates the beginning addr of the current line */

    int j, scratch;
    mem[--cp2] = mem[cp1--];                /* move EOL char        */
    scratch = mem[cp1];
    while (scratch != EOL)  {               /* move char until EOL  */
        mem[--cp2] = scratch;
        scratch = mem[--cp1];
    }
    for (j=0; j<5; j++)                     /* move line # to curr  */
        cln[j] = mem[cp2+j];                /*   line number buff   */
    return;
}       /*  end downline  */

                    /*  TOKEN  */

 token()    {       /* This routine returns the next token
                       number scanned.  */
    while(mem[pc++] < COMMENT)
                    /* skip periods, commas, tabs, and blanks   */
```

```
        pc++;
    return(mem[pc]);
}       /*  end token  */

                    /*  FIND  */

find()  {   /*  This routine searches the data division until it
                finds a match with the name in sbuf.  */
    int i;
    loop1:
        i = 0;
        while (i < 16 && sbuf[i] != NUL && sbuf[i] == mem[cp2]) {
            i++;
            mem[++cp1] = mem[cp2++];
        }
    if (i != 16 && sbuf[i] != NUL)  {
        i = 0;
        mem[++cp1] = mem[cp2++];
        goto loop1;
    }
    /*  cp1 now points to the last character of the identifier  */
    for (i=0; i<16; i++)                    /* zero out sbuf  */
        sbuf[i] = NUL;
    return;
}   /*  end find  */

                    /*  INITVAL  */

initval()   {   /* This routine initializes variables in the data
                    division that have the 'value' clause.  An '='
                    sign and the initial value is inserted directly
                    following the clause.   */
    int i;
    lb[0] = 'C';            /* fill lb with line # of data division */
    for(i=0; i < 5; i++)
        lb[i] = '0';
    while (compare() != EQUAL)
                    /* move lines to low core til top of data div   */
        downline();
    begdata = cp1 +1;   /* byte offset of first line # of data div */
    sbuf[0] = 'v';
    sbuf[1] = 'a';
    sbuf[2] = 'l';
    sbuf[3] = 'u';
    sbuf[4] = 'e';
    while ( cp2 != begproc) {
        find();             /* finds each occurance of VALUE clause */
        while(!(mem[cp1] >= '0' && mem[cp1] <= '9'))
            mem[++cp1] = mem[cp2++];
        sptr = cp1;
        while (mem[cp1] != EOL)
            mem[++cp1] = mem[cp2++];
        mem[cp1] = '=';
        while (mem[sptr] != '=')
            mem[++cp1] = mem[sptr++];
        mem[++cp1] = EOL;
    }
    return;
}   /*  end initval  */

                    /*  LOAD  */

load(r)         /* This routine loads xb or yb, depending on
                    the value of r, with the value correspond-
                    ing with the token number in save.  */
int r;      {
    int i;
    sptr = save * 16 + base;
    if ((mem[sptr] < '0') || (mem[sptr] > '9')) {
        if (mem[sptr] == '-')   {       /*  negative literal  */
            xb = '-';
            sptr++;
        }
```

```
        else    {
            if (mem[sptr] == '+')   {   /*  positive literal  */
                xs = '+';
                sptr++;
            }
            else    {   /* alphabetic--find value in data div */
                while (cp1 != begdata)
                    mem[--cp1] = mem[cp2--];
                for (i=0; i < 16; i++)
                    sbuf[i] = mem[sptr++];
                find();
                while(mem[cp1] != '=')
                    mem[++cp1] = mem[cp2++];
                sptr = cp2;
            }
        }
        while((mem[sptr]!='.')&&((mem[sptr]!=EOL)&&(mem[sptr]!=NUL))) {
            xb[i] = mem[sptr];
            i++;
            sptr++;
        }
        if (mem[sptr] != '.')   {
            x1 = i;
            x2 = y1;
            x3 = 0;
        }
        else    {
            x1 = i;
            sptr++;
            while ((mem[sptr] != EOL) && (mem[sptr] != 0))  {
                xb[i] = mem[sptr];
                i++;
                sptr++;
            }
            x2 = y1 + i;
            x3 = 1;
        }
        if (r == y) {
            for(i=0; i < x2; i++)
                yb[i] = xb[i];
            y1 = x1;
            y2 = x2;
            y3 = x3;
            ys = xs;
            cp2 = cp2 + y2; /* value will be overwritten by result */
        }
    }
    return;
}       /* end load  */

                    /*  RELOAD  */

reload()    {       /* This routine reloads a value from yb into
                       the data division.  */
    int i;
    i = 0;
    if (ys == '-')
        mem[++cp1] = ys;
    if (yb[i] == '0' && y1 > 1)
        i++;
    while (i != y1)
        mem[++cp1] = yb[i++];
    if (y3 != 0)    {
        mem[++cp1] = '.';
        while (i <= y2)
            mem[++cp1] = yb[i++];
    }
    mem[++cp1] = EOL;
    return;
}       /* end reload  */

                    /*  NEXTLINE  */
```

```
nextline()  {    /* This routine skips to the first token of
                        the next line.    */
     int i;
     if (save != EOL)
          while (token() != EOL);
          for (i=0; i < 5; i++)
               cln[i] = mem[++pc];    /* load cln and skip line # */
}         /*  end nextline  */

                    /*  KEYWD  */

keywd() {    /* This routine calls routines by cobol key words. */
     switch(save)     {
          case COMMENT :              /*  comment  */
               nextline();
               break;

          case ACCEPT :               /*  accept statement  */
               accept();
               break;

          case ADD :                  /*  add statement  */
               compute(add);
               break;

          case CLOSE :                /*  close statement  */
               close();
               break;

          case DISPLAY :              /*  display statement  */
               display();
               break;

          case DIVIDE :               /*  divide statement  */
               divide();
               break;

          case GO :                   /*  goto statement  */
               go();
               break;

          case MOVE :                 /*  move statement */
               move();
               break;

          case MULTIPLY :             /*  multiply statement  */
               mult();
               break;

          case OPEN :                 /*  open statement  */
               open();
               break;

          case PERFORM :              /*  perform statement  */
               perform();
               break;

          case READ :                 /*  read statement  */
               read();
               break;

          case SUBTRACT :             /*  subtract statement  */
               compute(sub);
               break;

          case WRITE :                /*  write statement  */
               write();
               break;

          default:                         /* syntax error */
               errflg = ON;
               errmsg("INCORRECT KEYWORD FOLLOWING CONDITIONAL OR ELSE");
```

```
    }       /* end switch statement */
    return;
}

                    /*  COND  */

cond()  {   /* This routine evaluates a conditional phrase and
               returns true or false.  */
    char temp1,temp2;
    save = token();
    if((temp2 = token()) == NOT)    {
        temp1 = NOT;
        temp2 = token();
    }

    switch(temp2)   {

        case NUMERIC :                  /*  numeric?  */
            sptr = save * 16 + base;
            if (mem[sptr] <= '9')
                temp2 = TRUE;
            else
                temp2 = FALSE;
            break;

        case ALPHABETIC :               /*  alphabetic?  */
            sptr = save * 16 + base;
            if (mem[sptr] > '9')
                temp2 = TRUE;
            else
                temp2 = FALSE;
            break;

        case GREATER    :               /*  xb > yb?  */
            load(x);
            save = token();
            load(y);
            if (size() == GREATER)
                temp2 = TRUE;
            else
                temp2 = FALSE;
            break;

        case LESS :                     /*  xb < yb?  */
            load(x);
            save = token();
            load(y);
            if (size() == LESS)
                temp2 = TRUE;
            else
                temp2 = FALSE;
            break;

        case EQUAL :                    /*  xb = yb?  */
            load(x);
            save = token();
            load(y);
            if ( size() == EQUAL)
                temp2 = TRUE;
            else
                temp2 = FALSE;
            break;

        default:                        /* syntax error  */
            errmsg("INCORRECT WORD IN CONDITIONAL PHRASE");
            errflg = ON;

    }       /*  end switch statement  */
    if (temp1 == NOT)   {
        if (temp2 == TRUE)
            temp2 = FALSE;
        else
            temp2 = TRUE;
```

```
    }
    return(temp2);
}       /*  end cond  */

accept()    {
    nextline();
    return;
}   /*  end accept  */

close() {
    nextline();
    return;
}   /*  end close  */

display()   {
    nextline();
    return;
}   /*  end display  */

                    /*  GO  */

go()    {   /* This routine searches the procedure division
                from the beginning for a token match.  */
    char scratch;
    save = token();
    pc = begproc;
    while((scratch = token()) != save && scratch != STOP)
        nextline();
    if (scratch == STOP)    {
        errflg = ON;
        errmsg("DESTINATION FOR GO STATEMENT NOT FOUND");
    }
    else
        nextline();
    return;
}       /*  end go  */

                    /*  IFS  */

ifs()   {   /* This routine determines program flow by testing
                a condition--if true, the imperative directly
                following the condition or NEXT SENTENCE is per-
                formed; if false, the imperative following the
                ELSE clause is performed, if present.  */
    if (cond())     {
        save = token();
        if (save != NEXT)
            keywd();
        else
            nextline;
    }
    else
        while(((save = token()) != ELSE)||(save != EOL));
        if (save == ELSE)   {
            save = token();
            keywd();
        }
    return;
}       /* end ifs  */

                    /*  MOVE  */

move()  {   /* This routine moves the value in the first identi-
                fier or literal into the second identifier. */
    int i;
    save = token();
    load(x);
    if (token() != TO)  {
        errmsg("TO EXPECTED AFTER IDENTIFIER IN MOVE STATEMENT");
        errflg = ON;
    }
    save = token();
    sptr = save * 16 + base;
```

```
    for (i=0; i < 16; i++)
        sbuf[i] = mem[sptr++];
    find();
    reload();       /* store value from yb into data division */
    nextline();
    return;
}       /* end move  */

                    /*  NAME  */

name() {    /* This routine handles the processing of paragraph and
               section names.  */
    if (sp != -1)   {
                    /* then processing within PERFORM statement */
        if ((ext[sp] == save)||(ext[sp] == i))  {
            if(cntr[sp] == 0)   {
                ext[sp] = 0;
                pc = rtn[sp];   /* set pc to instr after PERFORM */
                sp--;
                nextline();
            }
            else    {
                if (cntr[sp] != MAX)    {
                    cntr[sp] = cntr[sp] - 1;
                                /* perform proc a # of times    */
                    pc = begproc;
                    while(token() != psave)
                        nextline();
                    nextline();
                }
                else    {       /* UNTIL condition must be evaluated */
                    if (cond()) {
                        pc = rtn[sp];
                        sp--;
                        nextline();
                    }
                    else    {
                        pc = begproc;
                        while (token() != psave)
                            nextline();
                        nextline();
                    }
                }
            }
        }
    }
    else
        nextline();
    return;
}       /*  end name  */

open()  {
nextline();
    return;
}   /* end open  */

                    /*  PERFORM  */

perform()   {       /* This routine causes program flow to jump to a
                       particular procedure and return to the state-
                       ment following the perform statement.  */
    int i;
    i = 0;
    rtn[++sp] = pc;     /* save current program counter */
    cntr[sp] = 0;
    ext[sp] = 1;    /* default value--exit implied at next procname */
    psave = token();
    if (( save = token()) != EOL)   {
        if (save == THRU)   {
            ext[sp] = token();
            save = token();
        }
        if (save == UNTIL)  {
```

```
            cntr[sp] = MAX;
            condptr = pc;
        }
        if (save != EOL)    {
            load(x);
            cntr[sp] = xb[0] - '0';
            if (y2 == 2)
                cntr[sp] = (cntr[sp] * 10) + (xb[1] - '0');
            if (y2 > 2)     {
                errmsg("PROCEDURE CANNOT BE EXECUTED > 100 TIMES");
                errflg = ON;
            }
            if (token() != TIMES)   {
                errmsg("TIMES EXPECTED IN THIS PERFORM STATEMENT");
                errflg = ON;
            }
        }
    }
    if (save == EOL)    {
        pc = begproc;       /* search for proc to be performed */
        while (token() != psave)
            nextline();
        nextline();
    }
    return;
}       /*  end perform  */

                    /*  ERRMSG  */

errmsg(err)     /* This routine prints an error message corresponding
                    to errnum which is set at the time the error
                    occurred.  */
char *err;  {
    int i;
    printf("ERROR OCCURRED AT LINE ");
    for(i=0;i < 5;i++)
        putchar(cln[i]);
    printf("* %s *",err);
    return;
}   /*  end errmsg  */


                    /*  SIZE  */

size()  {       /* This routine returns GREATER if xb > yb
                    EQUAL if xb = yb and LESS if xb < yb.  */
    int siz,i;
    if (x1 > y1)    /* x1, y1 contain the # of significant digits */
        siz = GREATER;
    else    {
        if (x1 < y1)
            siz = LESS;
        else    {
            i = 0;
            while ((xb[i] > yb[i]) && (++i < y2));
            if (i == y2)
                siz = GREATER;
            else    {
                i = 0;
                while((xb[i] == yb[i])&&(++i < y2));
                if (i == y2)
                    siz = EQUAL;
                else
                    siz = LESS;
            }
        }
    }
    return(siz);
}       /*  end size  */


                    /*  FILEIO  */
```

```
filbuf()    {
    if (y2 == 0)    {
        while (x2 > 0)  {
            temp[n--] = xb[--x2];
            c[n] = '0';
        }
    }
    else    {
        while (y2 > 0)  {
            temp[n--] = yb[--y2];
            c[n] = '0';
        }
    }
    return;
}       /*  end filbuf  */

                    /*  COMPUTE  */

compute(op)     /* This routine adds or subtracts 2 values
                   depending on the parameter op.  */
int op; {
    int a,b,m,i,dif;
    save = token();
    load(x);            /* loads first operand into buffer xb */
    save = token();
    if (op == sub)  {
        if (save != FROM)       {
            errflg = ON;
            errmsg("FROM REQUIRED HERE IN SUBTRACT STATEMENT");
        }
    }
    else    {
        if (save != TO) {
            errflg = ON;
            errmsg("TO REQUIRED HERE IN ADD STATEMENT");
        }
    }
    save = token();
    load(y);                    /* loads second operand into buffer yb */
    if (x3 != y3)   {           /* x3, y3 contain the number of digits to
                                   the right of the decimal point   */
        if (x3 > y3)    {       /* zero fill buffer with shortest
                                   mantissa to align decimal points */
            dif = x3 - y3;
            for (i=1; i <= dif; i++)
                yb[y2++] = 0;
            y3 = x3;
        }
        else    {
            dif = y3 - x3;
            for (i=1; i <= dif; i++)
                xb[x2++] = 0;
            x3 = y3;
        }
    }
    if (x2 >= y2)               /* x2, y2 contain the total number
                                   of digits in each operand */
        m = x2;
    else
        m = y2;
    n = m;
    c[n] = '0';
    if (op == sub)  {           /* to subtract, change sign of the
                                   first operand and add */
        if (xs == '+')
            xs = '-';
        else
            xs = '+';
    }

    if (xs == ys)   {
        c[n] = '0';
        while ((y2 > 0) && (x2 > 0))    {
```

```
            a = xb[--x2] - '0';
            b = yb[--y2] - '0';
            temp[n--] = addtab1[a][b];
            c[n] = addtab2[a][b];
        }
        filbuf();
        n = m;
        while (n > 0)   {
            a = temp[n] - '0';
            b = c[n] - '0';
            yb[n--] = addtab1[a][b];
        }
    }

    else    {
        switch(size())  {

            case EQUAL:                         /*  xb = yb  */
                ys = xs = '+';
                c[0] = '0';
                y2 = 1;
                y3 = 0;
                break;

            case GREATER:                       /*  xb > yb  */
                ys = xs;
                while (y2 > 0)  {
                    a = xb[--x2] - '0';
                    b = yb[--y2] - '0';
                    temp[n--] = subtab1[a][b];
                    c[n] = subtab2[a][b];
                }
                filbuf();
                break;

            case LESS:                          /*  xb < yb  */
                xs = ys;
                while (x2 > 0)  {
                    a = xb[--x2] - '0';
                    b = yb[--y2] - '0';
                    temp[n--] = subtab1[b][a];
                    c[n] = subtab2[b][a];
                }
                filbuf();
                break;
        }       /*  end switch statement  */

        n = m;
        while (n > 0)   {
            a = temp[n] - '0';
            b = c[n] - '0';
            yb[n--] = subtab1[a][b];
        }
    }
    yb[0] = c[0];
    y2 = m;
    y1 = y2 - y3;
    reload();
    return;
}       /* end add */

                    /* MULTIPLY */

mult ()  {  /* This routine multiplies the 1st operand by the 2nd and
            stores the result in the second operand.    */

char p,c1,c2,c3,c4;
int  a,b,t1,t2,z1,z2,i;

save = token();
load(x);
if (token() != BY)  {
    errflw = ON;
```

```
    errmsg("BY REQUIRED HERE IN MULTIPLY STATEMENT");
}
save = token();
load(y);
z1 = x1+y1;                         /* location of decimal pt in temp   */
z2 = x2+y2;                         /* number of digits in product      */
t2 = z2;
for (a=0; a<36; a++)                /* set the output buff to zero      */
    temp[a] = '0';

for( y1=y2-1; y1>=0; y1-- )  {      /* for each digit in multiplier     */
    c2 = c4 = '0';                  /* set carry to ascii zero          */
    t1 = t2;

    for( x1=x2-1; x1>=0; x1-- )  {  /* for each digit in multiplican    */
        a = xb[x1]-'0'; b = yb[y1]-'0';
        p =multab1 [a] [b];         /* product of 2 digits              */
        c1=multab2 [a] [b];         /* save carry                       */
        a = p-'0'; b = c2-'0';
        p =addtab1 [a] [b];         /* add previous step carry          */
        c2=addtab2 [a] [b];         /* save carry                       */
        a = c2-'0'; b = c1-'0';
        c2=addtab1 [a] [b];         /* add carry for next step          */
        a = temp[t1]-'0'; b = p-'0';
        p =addtab1 [a] [b];         /* add result to output             */
        c3=addtab2 [a] [b];         /* save carry                       */
        a = p-'0'; b = c4-'0';
        temp[t1--] =addtab1 [a] [b];/* add previous carry, save         */
        c4=addtab2 [a] [b];         /* save carry                       */
        a = c4-'0'; b = c3-'0';
        c4=addtab1 [a] [b];         /* add carry for next step          */
    }
    t2--;                           /* decrement temp counter           */
}
a = c2-'0'; b = c4-'0';
temp[t1] =addtab1 [a] [b];          /* carry is MSD of output           */
y1 = z1;
y2 = z2;
y3 =y2 - y1;
for (i=0; i <= y2; i++)
    yb[i] = temp[i];
if (xs != ys)
    ys = '-';
else
    ys = '+';
reload();
    return;
}       /*  end multiply  */

                    /*  DIVIDE  */

divide()  {
        /*This routine uses the non-restoring technique to calculate
          the quotient--xb is divided into yb and the quotient is
          stored in yb.    */

int  a,j,lp,sc,z1,z2;
char zs;

save = token();                                 * load operands *
load(x);
if (token() != INTO)     {
    errflg = ON;
    errmsg("INTO REQUIRED HERE IN DIVIDE STATEMENT");
}
save = token();
load(y);
sc = 0;                             /* division step count          */
dp = 1;                             /* digit position in temp       */
z1 = x1-y1+1.                               * digits preceeding . *  */

for (j=0; j<32; j++)                        * set temp to zero       */
    temp[j] = '0';
```

```
while (xb[0] == '0')  {                 /* if MSD is zero, shift     */
    for (j=0; j<x2; j++)
        xb[j] = xb[j+1];
    if (x1 != 0)
        x1--;
    x2--;
}

while (dp <= x2 && a < 16)  {
    /* while the number of digits in the output is less than in the */
    /* dividend and the remainder (xb) is not equal to zero         */

    switch (xs)  {                          /* switch on sign of remain.*/

        case '+':
            /* if the remainder is positive subtract the divisor    */
            /* until the sign of the remainder changes              */
            sc = '0'-1;
            while (xs == '+')  {
                compute(sub);
                sc++;
            }
            temp[dp++] = sc;            /* store ASCII value in temp    */
            break;

        case '-':
            /* if the remainder is negative, add the divisor until  */
            /* the sign of the remainder changes                    */
            sc = '9'+1;
            while (xs == '-')  {
                compute(add);
                sc--;
            }
            temp[dp++] = sc;                /* store ASCII value in temp*/
            break;
    }                                       /* end of SWITCH stmt       */

    for (j=15; j>=0; j--)                   /* shift divisor to right   */
        yb[j+1] = yb[j];                    /* for next cycle           */
    y1 = x1;                                /* adjust char counters     */
                                            /* y2++ ???                 */

    for (a=0; a<16; a++)                    /* test remainder (xb) equal*/
        if (xb[a] != '0')                   /* to zero                  */
            break;
}                                           /* end of WHILE stmt        */

    if (xs != ys)                           /* set sign of result       */
        ys = '-';
    else
        ys = '+';
    y2 = dp + 1;                            /* save length of output    */
    y1 = z1;
    y3 = y2 - y1;
    for(j=0; j < y2; j++)
        yb[j] = temp[j];
    reload();
    return;
}       /*  end divide  */

                        /*  MAIN  */

main()  {

/* This routine reads the editor output file into a char array
   called mem in memory; initializes variable values in the data
   division; scans the procedure division token by token (byte by
   byte) for grammatical correctness and executes each complete
   statement as it encounters it; main halts on all errors except
   faulty punctuation or when it scans the token for 'stop'.    */
```

```
int i;
printf("COBOL INTERPRETER, Version 1.0 =");
readin();
if(errflg == ON)
    goto endmain;
downline();                 /* find last EOL--beg of symbol table */
base = cp2 + 5;             /* set base to byte offset of sym table */
lb[0] = 'D';                /* load lb with 1st line # of proc div  */
for (i=1; i < COMMENT; i ++)
    lb[i] = '0';
while (compare() != EQUAL)      /* move proc div into high core */
    downline();
pc = cp2;
nextline();
begproc = pc;   /* pc and begproc now set to 1st executable stat */
for (i=0; i <= 16; i++)
    sbuf[i] = NUL;          /* zero fill the search buffer */
initval();                  /* initialize identifier values */

while (((save = token()) != STOP) && (errflg == OFF))   {
                        /* scan until token for stop or error  */

    switch(save)    {
        case COMMENT :          /*  comment  */
            nextline();
            break;

        case ACCEPT :           /*  accept statement  */
            accept();
            break;

        case ADD :              /*  add statement  */
            compute(add);
            break;

        case CLOSE :            /*  close statement  */
            close();
            break;

        case DISPLAY :          /*  display statement  */
            display();
            break;

        case DIVIDE :           /*  divide statement  */
            divide();
            break;

        case GO :               /*  goto statement  */
            go();
            break;

        case IF :               /*  if statement  */
            ifs();
            break;

        case MOVE :             /*  move statement */
            move();
            break;

        case MULTIPLY :         /*  multiply statement  */
            mult();
            break;

        case OPEN :             /*  open statement  */
            open();
            break;

        case PERFORM :          /*  perform statement  */
            perform();
            break;

        case READ :             /*  read statement  */
            read();
```

```
                break;

            case SUBTRACT :          /*  subtract statement  */
                compute(sub);
                break;

            case WRITE :             /*  write statement  */
                write();
                break;

            default:         /* label proc name,section name or error */
                name();

        }       /* end switch statement */
    }
    endmain:
        writeout();
}   /*  end main--return to monitor  */
```

LIST OF REFERENCES

1. Craig, Alan S. MICRO-COBOL An Implementation of Navy Standard Hypo-Cobol for a Microprocessor-Based Computer System, Masters Thesis, Naval Postgraduate School, March 1977.

2. Advanced Micro Devices, The Am 2900 Family Data Book, Sunnyvale, California 94086, 1976.

3. Advanced Micro Devices, AMDASM/80 Reference Manual, Sunnyvale, California 94086, 1977.

4. Advanced Micro Devices, A Microprogrammed 16-Bit Computer, Sunnyvale, California 94086,1976.

5. Software Development Division, ADPE Selection Office, Department of the Navy, HYPO-COBOL, April 1975.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4. Assoc Professor G. A. Kildall, Code 52 Kd (thesis advisor) Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5. LT Ronald W. Modes, USN (student) 6545 650 West Oak Harbor, Washington 98277	1
6. LT Elizabeth K. Conley, USN (student) 1216 Darwin Street Seaside, California 93955	1