ASSEMBLER SOFTWARE FOR THE ATHENA COMPUTER

A Thesis

Presented to

the Faculty of the Department of Electrical Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Thomas C. Minter, Jr.

December 1971

ASSEMBLER SOFTWARE FOR THE ATHENA COMPUTER

An Abstract

of a Thesis

Presented to

the Faculty of the Department of Electrical Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Thomas C. Minter, Jr.

December 1971

ASSEMBLER SOFTWARE FOR THE ATHENA COMPUTER

ABSTRACT

The Athena Computer lacks essential software for general computational purposes. All programming on the Athena is performed using numeric machine instructions.

Software was prepared to eliminate this problem. An assembler program was prepared to allow Athena users to program the Athena Computer in assembly language. The assembler program translates the user's assembly language program into numeric machine code for the Athena Computer. The assembler has features which allow the user to refer to storage locations using symbolic names and write operation codes in symbolic form. The assembler also allows the user to specify constants in the form which the user thinks of it, such as 1.0 for a floating-point constant. The assembler language programmer can write calls on closed subroutines and the assembler will generate the necessary linkages. Calls on open subroutines (macros) may be made. The assembler will automatically insert the macro code in-line following the macro call. To aid the user in debugging his program, numerous checks for errors are made and a error message is printed out if an error is detected. An Athena assembly language programmer's guide was prepared to aid users in programming the Athena. A systems programmer's guide was

provided to aid in modifying the assembler program itself as requirements change.

TABLE OF CONTENTS

Example

```
1     00 0040     017004     DD     4
1     00 0041     340042     WP     S+1
```

'DD', opcode 017, with a mode of 4, causes the digital printer to space. The 'Wait Partial' instruction halts program execution until the printer has completed spacing.

3.11 Programming Errors

As a programming aid, the assembler provides error detection facilities for the programmer. Diagnostic messages are printed above the offending instruction line in the output listing. An error number is given for use in diagnosing the problem. An explanation of the meaning of the error numbers is given in Appendix B.

3.12 Executing a Program Across Drum Group Boundaries

A significant problem with the Athena is doing program execution across a drum group boundary. In this discussion, the problem will be defined, and then the solution will be presented. The drum on the Athena consists of eight groups, of 1024 words each. The groups are numbered 0-7, and the words 0-1023. If a program is executing on say group zero, and arrives at the last word of group zero (location 1023), execution will not continue with the first word of group one but will start over again at location zero of group zero.

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

The Athena computer was designed as a special purpose on-ground guidance control computer for the Titan 1 ICBM. Input to the machine consisted of radar tracking data as well as other parameters such as weather information. This data was analyzed and compared to a pre-entered target data. Steering and acceleration corrections were generated and transmitted to the in-flight missile. The computer also enabled the transmission of aiming instructions for the missile warhead when certain criteria was satisfied.

The Department of Electrical Engineering, University of Houston, acquired the Athena as a piece of government surplus. The Athena has been modified to a general purpose digital computer. Indexing capabilities have been added and the magnetic drum has been modified from a read-only memory to a random access memory. The word size on the drum was increased from seventeen bits to eighteen bits per word. Problems which made sequential program tedious when extended sequence instructions, such as multiply, divide, and shift were used, have been eliminated. A typewriter has been interfaced with the Athena and a set of input/output operation codes created to allow input and output of information on the typewriter.

The Athena had no software when it arrived at the University of Houston. All programming was in machine language. The purpose of this thesis is to provide assembly language programming capabilities for the Athena computer.

To provide the Athena computer with assembly language programming capabilities, an assembler program was prepared. The assembler program was written in Fortran V for execution on the Univac 1108 computer. The assembler program will normally be executed from a teletype terminal. Input to the assembler program is an Athena assembly language source program. The assembler program assembles the source program and prints out the results. An object program for the Athena is also punched out on the teletype terminal paper tape punch. The paper tape is then placed on the Athena's paper tape reader and read into the Athena's drum memory and executed.

Chapter III presents an "Athena Assembly Language Programming Guide" for users of the assembler program. The guide covers all aspects of assembly language programming and presents many examples to illustrate assembly language programming techniques.

In Chapter IV the theory and structure of the assembler program is described. A syntactical definition for the Athena assembly language is presented. The assembler program

processes Athena assembly language instructions based on these syntactical definitions. In processing a source program, the assembler makes two complete passes over the source program.

The documentation on the assembler program is presented in Appendix A. Appendix A presents documentation on the individual subroutines and functions which comprise the assembler program. Deck setups for executing the assembler program are described.

Appendix B gives a listing of error message numbers and describes their meaning.

Appendix C is a reference for assembler users. It presents a list of all alphanumeric characters used by the Athena computer and their internal binary representation in the Athena computer.

Appendix D presents a listing of macro programs used to perform floating-point arithmetic on the Athena. Macro programs are discussed in Chapter III.

Appendix E is a reference table of operation codes for the Athena. It gives the name of each operation code recognized by the assembler program and briefly describes the operation code. Included on the list of operation codes are several new mnemonic operation which duplicate the functions

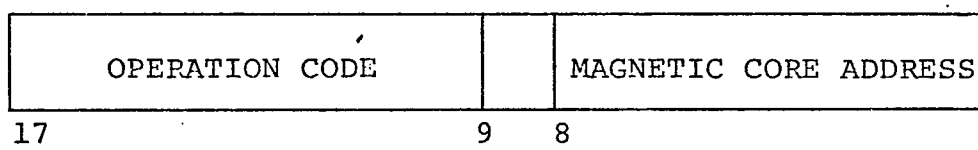
of original mnemonic operation codes. The mnemonic meaning of several of the original operation have become obscure and are no longer meaningful. For example, mnemonic operation code "TP" is an original operation code which clears the accumulator and loads the accumulator with the content of a core memory location. The original meaning of "TP" is obscure. Another mnemonic operation code, "LA", for "Load Accumulator", which duplicates the functions of "TP", was made available to Athena assembly language programmers.

CHAPTER II

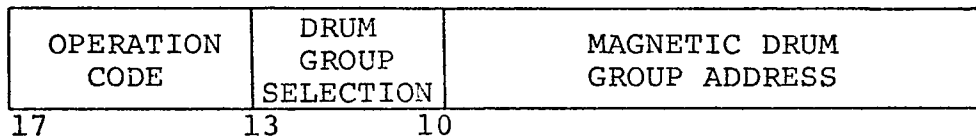
BASIC OPERATION OF THE ATHENA COMPUTER

The Athena has two separate and distinct memories, a magnetic core memory of 256 twenty-four bit words and a magnetic drum of 8192 eighteen bit words. The addressing of these two memories requires specifications of different length. This difference is reflected in the instruction word format (see Figure 2-1). In an instruction affecting core memory, the lower eight bits are used as an address and the upper nine are the operation code. Bit eight is ignored. A drum address requires a thirteen bit specification which is divided into two parts. The upper three bits determine which of the eight groups of the drum is desired, and the lower ten bits determine the specific location within that group. Thus, in an instruction affecting drum memory, the lower thirteen bits are used for an address and the upper five bits are the operation code (see Figure 2-1).

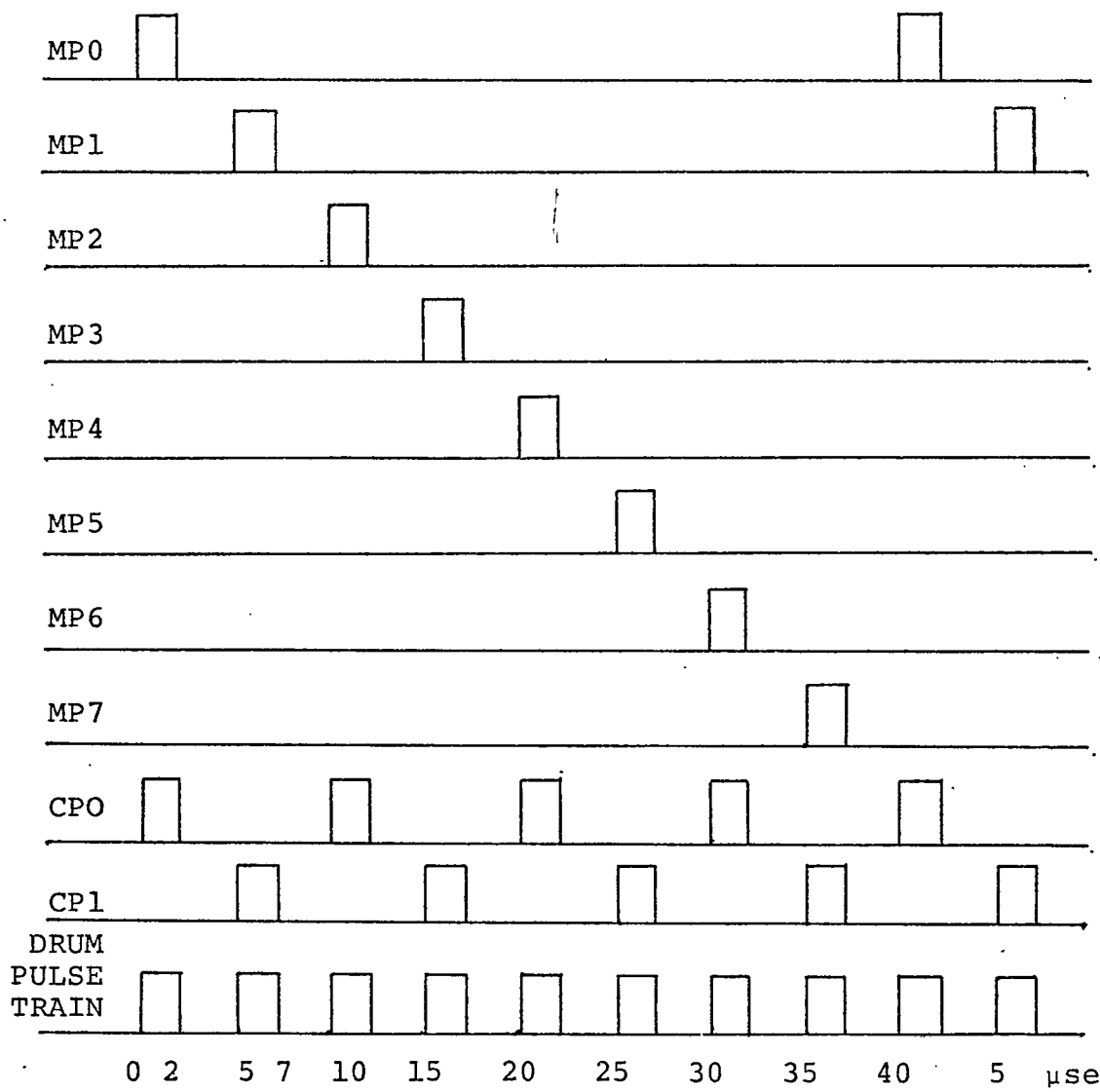
A timing track on the drum is the basis for all timing signals in the computer. A 200 KHz sine wave on the timing track is converted into a 200 KHz pulse train (Figure 2-1). This pulse train itself is not used for timing in the machine but is divided into two 100 KHz timing pulse trains, CP0 and CP1. The CP0 and CP1 pulses are used in the Main Pulse



MAGNETIC CORE INSTRUCTION WORD



INSTRUCTION WORD FORMAT

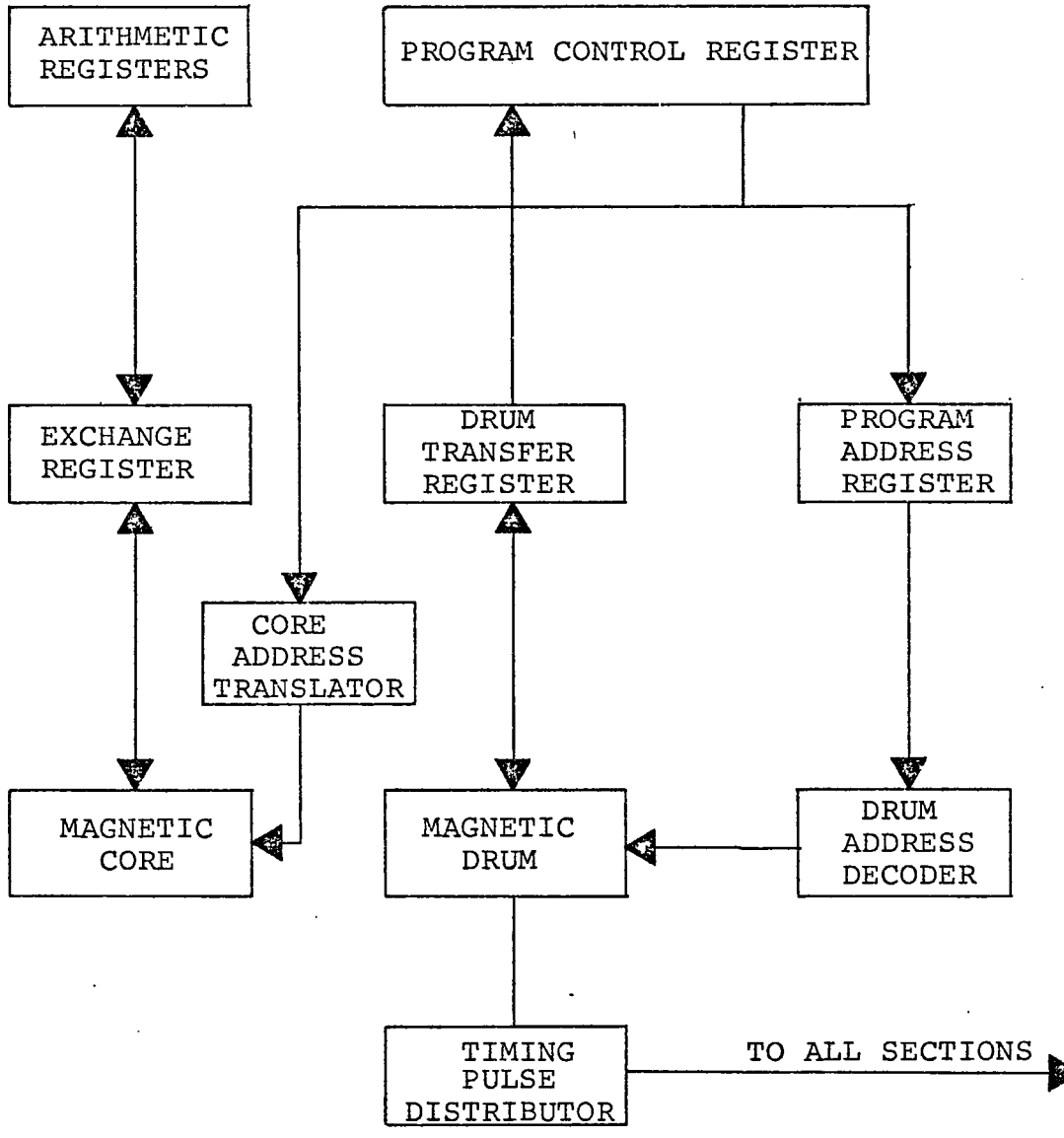


TIMING PULSES

Fig. 2-1

Distributer (MPD) to generate the eight Main Pulses, MP0 through MP7. From Figure 2-1, it is seen that odd numbered Main Pulses occur coincidentally with CP1 pulses, and even numbered Main Pulses, with CP0 pulses. Some operation codes require only Main Pulses to complete their operation while others require both Main Pulses and CP0 and CP1. To stop the computer, the Main Pulse Distributer is inhibited from generating Main Pulses. The basic cycle time, 40 microseconds, for the computer is the time required to generate a complete set of Main Pulses (MP0 through MP7).

The basic operation, disregarding Input/Output, occurs in the following manner (see Figure 2-2). On MP0, execution of the instruction in the Program Control Register (PCR) is begun and continues through MP6. During this period, the next instruction to be executed is read from a drum location which is specified by the Program Address Register (PAR) and is placed in the Drum Transfer Register (DTR). At the same time the DTR is loaded, the PAR is incremented for the next instruction. On completion of execution, a MD Resume signal is sent to the Main Pulse Distributer to allow generation of MP7. On MP7 the instruction in DTR is transferred into the PCR, and on the following MP0 the cycle is reinitiated. On branch and extended sequence instructions, such as the multiply, divide, and shift instructions, an extended sequence flip



BASIC COMPUTER OPERATION

Fig. 2-2

flop is set and the MD Resume signal is delayed or altered until completion of the operation. Interlacing is used in the drum address decode network to prevent delay in reading the next instruction.

CHAPTER III

ATHENA ASSEMBLY LANGUAGE PROGRAMMING GUIDE

3.1 Introduction

An assembly language is a symbolic form of machine language. While machine language is numeric, assembly language allows alphabetic names for operation codes and storage locations. A program called an assembler translates a program written in assembly language into machine language which can be executed by the computer.

This chapter is a guide to programming in assembly language on the Athena computer. The general topic of assembly language will first be discussed and then a detailed description will be given of Athena Assembly Language Programming.

3.2 A Look at Assembly Language

To write programs directly in machine language, several clerical activities must be performed. The machine language program must keep track of exactly what locations are used for which instructions, data areas and constants, so that he may refer to these locations correctly later in the program and also that the program does not erase data, instructions, or constants by inadvertently using the same space for another purpose. Tables are referred to in writing the numerical

operation codes required by the computer. Constants must be expressed in binary form.

Until the early 1950's all programming was done directly in machine language. It was tedious, and the performance of clerical jobs by programmers resulted in many clerical errors. Assembly languages evolved as symbolic ways of writing, machine language. The clerical tasks are, as far as possible, delegated to the assembler, which is a program that translates programs written in assembly language into machine language. The assembler keeps track of the storage locations used; the programmer refers to them symbolically, using names suggestive of the actual meaning in the problem oriented program. The assembly language programmer is also allowed to write operation codes in symbolic form; SA, for instance, may be written for "Store Accumulator" and the assembler will translate SA to the numerical operation code 004.

An assembler also allows the programmer to specify constants and data areas symbolically. The constant may be given in the form in which the programmer thinks of it, such as 1.0 for a floating point constant. The length of the storage area to be assigned to the constant may be specified, as well as a symbolic name by which the programmer will refer to the constant. The assembler will do the necessary conversions of the constants to binary representations, allocate storage, and keep a table

of names and corresponding addresses. The same storage allocations and symbol table procedures are used for data areas.

The assembly language programmer can conveniently write calls on open and closed subroutines. The instructions that make up an open subroutine are placed in line at each place where the execution may be desired. For closed subroutines the assembler generates the necessary linkages between the calling and called subroutine.

Modifications of programs written in assembly language is much easier than modification of programs written directly in machine language. Insertion of instructions, for example, changes the allocation of all storage addresses after the point of insertion. In a program written directly in machine language the programmer would have to change the addresses of all affected instructions and constants, and all references to them, with the strong likelihood of error. Changing the size of a data area has similar hazards. To change an assembler language program, however, it is usually necessary only to change or insert the directly affected assembly language instruction; the assembler will reassign all addresses and references to instructions, constants and data areas according to the current structure of the program. Thus the assembler helps to avoid many potential mistakes in changing of a program.

The assembler produces a printed listing of the program,

showing the assembly language, the corresponding machine language generated, and diagnostic error messages. Many programming errors are caught in the assembly process, and do not have to be found, one by one, through debugging execution runs.

3.3 Information Formats for the Athena Assembly Language

In this section, the various categories of information that are pertinent to the Athena will be introduced. This information includes instruction and data. In each case, both symbolic and actual representations will be of interest.

3.3.1 Statement Formats

A statement is the basic element of an assembler source program. A statement may be the symbolic counterpart of a computer instruction; it may be an instruction to the assembler, or it may be expository information (a comment).

3.3.1.1 Label Field

The label field begins in Column 1 and is terminated by a blank column. A label is a symbol used by the programmer to identify a statement. Normally the use of a label is optional.

3.3.1.2 Command Field

The command field begins in the first nonblank column following the label field or in the first nonblank

column following Column 1 if the label field is omitted. The entry in a command is a mnemonic operation code and, as such, is mandatory for every statement (except comments).

3.3.1.3 Operand Field

The operand field begins in the first nonblank column following the command field and it is terminated by a blank, or upon reaching Column 72; which ever comes first. Normally an operand is required in the operand field but there are exceptions.

3.3.1.4 Comment Field

The comment field begins in the first nonblank column following the operand field. If no operand field is present, the comment field cannot be used. The comment field is terminated at Column 80. Use of this field is optional and its use has no effect on the assembly.

3.3.1.5 Sequence Field

The sequence field (Columns 73-80) is provided as an aid to the programmer in keeping the source statements in order. Use of this entry is optional.

3.3.1.6 Continuation Lines

Continuation lines are not permitted.

3.3.1.7 Comment Lines

If an asterisk (*) appears in Column 1 of a statement, then the entire line is considered to be comment. All valid characters may be used and there is no effect on the assembly.

3.3.1.8 Output Listings

The following is an example of the format used for output listings.

CARD	GRP	LOC	MACHINE CODE	COL. 1	ASSEMBLER INSTRUCTION		
1	00	0040	014732	EXPL1	DATA	Ø'14732'	EXAMPLE 1
2				*A COMMENT CARD			EXAMPLE 2
3		000040		EXPL3	EQU	EXPL1	EXAMPLE 3
4	00	0041	000012	EXPL4	DATA	10	EXAMPLE 4
5	00	0042	000000	EXPL5	DATA,2	12	EXAMPLE 5
	00	0043	000014				
6		000031		EXPL6	EQU	25	EXAMPLE 6
				***DIAGNOSTIC MESSAGE 28			
7	00	0044	424716	EXPL7	DATA	Ø'7424716'	EXAMPLE 7

The first integer under the column labeled "CARD" gives the card count. This count is begun at the beginning of the assembly process and is incremented by one for each source statement (e.g., Examples 1-8).

The number under the column labeled "GRP LOC" is a six digit octal number that gives the address that has been assigned

to the binary code generated by the particular source statement.

The first two digits indicate the drum group in which the code will be located. The Athena drum has eight groups, therefore, this will be a number between zero and seven. The last four digits indicate location of the code within the drum group.

The code generated, if any, appears under the column labeled "MACHINE CODE" (e.g., Examples 1, 4, and 5). In certain cases, the source statement does not result in object code and thus only a value (e.g., Example 6) or a location, '000040' (e.g., Example 3) is indicated. All code generated is printed in octal form.

An exact copy of the source image appears under the heading "ASSEMBLER INSTRUCTION."

When an error is detected by the assembler, the message ***DIAGNOSTIC***ERROR MESSAGE (number) is printed above the erroneous assembler instruction and a number is printed indicating the error number. See Appendix B for the list of error messages.

Comment cards are printed out as card images.

In all further examples of output listing, the heading labels will be omitted.

3.3.2 Basic Directives

Directives are assembler instructions that do not have counterparts in the instruction repertoire of the computer. Instead, directives supply information to the assembler and/or invoke certain actions by it.

3.3.2.1 Data

The assembler instruction

```
LABEL          DATA, W        A
```

places the value associated with the symbol 'A' into an area of drum whose size is 'w' words. If a label is present, it is assigned the value of the location (or instruction) counter. The operand field may contain only one entry. If 'w' is omitted, a field size of one word is assumed. The value of 'w' must not exceed 100.

Examples

```
1   000040   000003   LB    DATA,2     Ø'3474120'
    000041   474120
2   000042   000040         DATA       LB
3   000043   000000         DATA,3     31
    000044   000000
    000045   000037
                  ***DIAGNOSTIC***  MESSAGE 30
4   000046   000000         DATA
```

In Line 1, above, the value in the operand field, Ø'3474120', is stored in two consecutive words starting at

location 40_8. The label 'LB' is assigned the value of 40_8. Note that the value of the octal constant, Ø'3474120', the decimal constant, 31, and the value of the symbol, LB, are all stored right justified within the field with leading zeroes. If the operand field is blank, as in Line 4 above, error message 30 is printed and zeroes are entered in the field.

3.3.2.2 RES

The directive 'RES' is used to reserve drum storage locations.

label RES n

'RES' reserves a data field 'n' words long. If 'n' is omitted a diagnostic is printed and one word is reserved for the symbol in the label field.

3.3.2.3 EQU

When the assembler encounters 'EQU' in the command field, the symbol in the label field is given the value of the expression in the operand field. Several examples are shown below.

```
1    000040    000000    A       RES    10
2              000040    DELTA   EQU    A
3              000041    A128    EQU    A+1
4              000037    AAA     EQU    A-1
```

3.3.2.4 LOC

'LOC' sets the location counter to the value in the operand field as shown in examples below. The location counter is printed out in the second column of the output listing under the column heading of 'GRP LOC.' It is the counter by which the assembler maintains a record of where information being generated will be stored at execution time.

```
1    00 0040     000012     A     DATA 10
2                                 LOC 38
3    00 0046     000014     B     DATA Ø'14'
4                                 LOC Ø'53'
5    00 0053     000040     C     DATA A
```

The 'LOC' operand field may contain either an integer number or an octal number inside an octal declaration.

The location counter is initially set to 40_8 by the assembler unless a 'LOC' instruction is used to set another value initially. The Athena will automatically start execution at location 40_8. It is possible though to start execution at another location by manual means.

3.3.2.5 END

The 'END' directive terminates an assembler program. It establishes conditions for the assembly of a new assembler program where the listed symbols have a quite distinct value from any that they had in the preceeding assembler program.

The following program will illustrate this.

```
CARD   GRD LOC      MACHINE CODE  ASSEMBLER INSTRUCTION
  1    00 0040          000012    PHI     DATA    10
  2             000040            BETA    EQU     PHI
  3                                       END
  4    00 0041          001437    PHI     DATA    Ø'1437'
  5             000041            BETA    EQU     PHI
  6                                       END
```

BETA = 40_8 in Line 2. The 'END' directive then terminates the first program. In the second program BETA = 41_8. In the second program the symbols have different values which are independent of their previous values.

3.3.2.6 DEF

If a symbol occurs in the operand field of a 'DEF' directive, the value of the symbol is made available to other programs that are assembled along with the program in which the 'DEF' appears. In other programs if a symbol appears in the operand field of a statement but nowhere appears in the label field of a statement, the symbol is assumed to have been defined in a 'DEF' statement in another program. An example is shown on the following page.

```
 1    00 0040      000012          AA        DATA      10
 2    00 0041      000014          BB        DATA      12
 3    00 0042      000017          CC        DATA      15
 4                                           DEF       AA
 5                                           DEF       BB
 6                                           END
 7    00 0043      000147          BB        DATA      Ø'147'
 8          000040                ALPHA     EQU       AA
 9          000043                BETA      EQU       BB
10                          ***DIAGNOSTIC**MESSAGE    23
11                          ***DIAGNOSTIC**MESSAGE     8
12                                GAMMA     EQU       CC
13                                          END
```

AA is assigned a value of 40_8 in Line 1. AA is defined as available to other programs by a 'DEF' statement in Line 4. In Line 8, the value of AA is assigned to ALPHA. The result is that ALPHA is given a value of 40_8.

BB is assigned a value of 41_8 in Line 2. BB is also defined as available to other programs by a 'DEF' statement in Line 5. BB is then assigned a value of 43_8 in Line 7. When BETA is equated with BB in Line 9, BETA is set equal to 43_8 which is the value assigned BB in the second program. This illustrates that the value assigned a symbol in the present program takes precedent over values for symbols made

available to other programs through 'DEF' statements.

CC is assigned a value of 42_8 in Line 3. The value of CC was not made available to other programs by a 'DEF' statement in the first program. In the second program GAMMA is equated with CC in Line 10. Diagnostic messages result. CC has not been assigned a value in the present program, nor made available through a 'DEF' statement in the first program. CC has no value.

3.3.3 Symbols

A symbol may contain 1-15 characters. The following are legitimate characters; A-Z, 0-9. A symbol must have at least one non-numeric character. A blank is used to terminate fields and hence cannot be used as a symbol character.

The dollar sign '$' is used to indicate the current value of the location counter. The following are examples.

```
1    00 0040      000001  THISISASYMBOL          DATA 1
                          ***DIAGNOSTIC***MESSAGE  14
2            000007       THISSYMBOLISTOOLONG  EQU  7
                          ***DIAGNOSTIC***MESSAGE   2
3            000001       1845       EQU       1
                          ***DIAGNOSTIC***MESSAGE   2
4            000003       ABC%%      EQU       3
5            000004       14A8       EQU       4
6            000005       BFS4       EQU       4
```

In the preceding examples, Lines 1, 5, and 6 have legal symbols appearing in their label fields. These three lines have symbols less than 15 characters long and consisting of letters and digits only or letters only. Lines 2, 3, and 4 have illegal symbols in their label field. Line 2 has a symbol more than 15 characters long, Line 3 has a symbol consisting of only digits, and Line 4 has an illegal character appearing in the symbol. A diagnostic error message appears above each erroneous line. Error message number 14 declares the symbol in the label field to be too long. Error message number 2 declares the symbol to contain only digits or a symbol other than a letter or digit.

A symbol may be defined only once in a given program. A symbol is defined within a program by its appearance in the label field of an assembly language instruction or directive. In the case of all instructions and most directives (e.g., RES, DATA) the assembler assigns to the symbol the current value of the location counter. In an 'EQU' statement, the symbol is assigned the value in the operand field.

3.3.4 Constants

The assembler language allows for the introduction of a wide variety of constants into the computer program. These values include character string constants, octal constants, decimal constants, and floating point constants.

3.3.4.1 Character String Constants

Character string constants are the octal representation for alphanumeric characters. See Appendix C for a list of alphanumeric characters and their octal representations.

A character string appearing inside a character string declaration will be converted to its octal representation and stored in the data field provided. To declare a string of characters to be a character string constant, the character string is enclosed in quote marks and preceded by a "C". After conversion of the characters to their octal representation, the characters are stored left justified in the data field. Trailing blanks are provided to fill out the data field. The character string can include any character including blanks (i.e., A-Z, 0-9,;, &, ', -, =, _, ,, ., :, /). A single quote mark is represented by two quote marks inside the character string declaration. No more than 24 characters may appear in a character string declaration.

Since on the Athena flexowriter, a character is incompletely specified unless case is given (i.e., "A" can be an upper case "A", or lower case "a"), provisions are made to specify the character case along with the character in the converted character string. Inserted before the octal representation for the characters is another octal number identifying the case of the character. Inserted in front of upper case

characters is the octal number 74_8. Inserted in front of lower case characters is the octal number 72_8. These two octal numbers are commands to the flexowriter latching it into a mode where only upper case characters are printed out thereafter if the command is 74_8, and only lower case characters are printed out thereafter if the command is 72_8. If only upper case characters appear in the character string then only a single 74_8 upper case command has to be inserted in front of all these characters to identify all of them as upper case characters and also insure that they will all be printed out in upper case on the flexowriter. If upper case and lower case characters appear interspersed in a character string, upper and lower case flexowriter commands (i.e., 74_8 and 72_8) are placed before each occurrence of a one or more upper or lower case characters. The following are examples of character string constants.

```
1   000040   746162   A1   DATA,2   C'ABC'
    000041   631414
2   000042   746140   A2   DATA,2   C'A"C'
    000043   631414
3   000044   746114   A3   DATA,2   C'A B'
    000045   621414
4   000046   746172   A4   DATA,2   C'A-B'
    000047   407462
```

In the preceding examples of character string constants, Line 1 shows a typical character string conversion to its octal representation. The conversion is the characters 'ABC'. The first two octal numbers (74_8) of the generated machine code is a command to the flexowriter, placing it in the upper case mode. The octal representation for A, B, and C follow, where A = 61_8, B = 62_8, and C = 63_8. Trailing blanks are used to fill out the rest of the second data word. The octal representation for a blank is 14_8.

Two consecutive quotes represent a single quote in a character string. Line 2 illustrates a character string containing two consecutive quotes.

Line 3 illustrates a character string containing a blank.

Line 4 illustrates a character string containing both upper case and lower case characters. The assembler places an upper case (74_8) at the head of the string to insure that the flexowriter is in the upper mode when it types the 'A' (octal representation 61_8) which follows. The minus sign (-) following the A is a lower case character. The assembler inserts a lower case mode command (72_8) in front of the octal representations for a minus sign (40_8). The 'B' following the minus sign is an upper case character. The assembler inserts an upper case mode command (74_8) for the flexowriter and then a 62_8 for the 'B'.

3.3.4.2 Octal Constants

The assembler accepts octal constants. Octal constants which appear inside an octal constant format, are placed in the data field right justified. An octal constant declaration consists of an octal number enclosed in quote marks and preceded by a "Ø".

Octal constant declarations may be preceded by a plus "+", or minus "-" sign. An octal constant may not be more than twelve digits long. The following are examples of octal constants.

```
1     000040    000012     DATA      Ø'12'
2     000041    112345     DATA,2    Ø'11234567741'
      000042    677741
3     000043    000000     DATA,3    Ø'1473450000'
                001473
                450000
4     000044    777746     DATA      -Ø'31'
```

3.3.4.3 Decimal Integer Constants

Decimal integer constants appearing in the operand field of a statement are converted to octal and stored right justified in the data field provided. A decimal integer constant requires no declaration. A decimal integer constant may be preceded by a plus or minus sign. A decimal integer

may not be more than six digits in size. The following are examples of decimal integer constants.

```
1       00 0040     000012     DATA          10
2       00 0041     777765     DATA         -10
3       00 0042     000000     DATA,2         4
        00 0043     000004
```

3.3.4.4 Floating-Point Numbers

The Athena computer has no hardware facilities to perform floating-point arithmetic. However, software programs may be used to perform floating-point arithmetic. The assembler converts real numbers to a special floating-point format for doing software floating-point arithmetic.

To declare a real number to be a floating-point number, a floating-point declaration is used. The floating-point declaration consists of enclosing the real number in quote marks and preceding it by a "F". The real number inside the quote marks may consist of no more than six digits and a mandatory decimal point or no more than six digits and a mandatory decimal point followed by an "E", followed by an optional plus "+" or "-" sign, followed by a two digit integer not greater than 38.

Real numbers are converted to floating-point numbers and stored in two consecutive drum words. The first word

contains the fraction in fractional form (decimal point to left of the leftmost bit) and the second word contains the exponent in integer form (decimal point to right of rightmost bit). If the real number is positive the leftmost bit of the fraction will be zero; if negative, the leftmost bit is one. The fraction is normalized so that the second bit from the left is one if the real number is positive; zero if the real number is negative. The exponent is stored right justified in integer form in the second word. A positive exponent will be stored as a positive integer; a negative exponent as a negative integer.

The value of a positive floating-point number (N) is determined as follows.

$$N = F \times 2^{EXP}$$

Example

```
1     00 0040     200000     AA     DATA,2     F'4.0'
      00 0041     000004
```

$$N = 2/8 \times 2^{4} = 1/4 \times 16 = 4$$

To convert negative floating-point numbers, the fraction must be complemented first.

The 'Normal' or 'True Zero' has a fraction of zero and an exponent of zero.

3.3.5 Expressions

In assembler language, a value may be indicated via a simple arithmetic expression. The assembler will handle simple arithmetic expressions in the operand field of the form,

symbol ± decimal integer
octal constant ± decimal integer
decimal integer ± decimal integer
dollar sign ($) ± decimal integer
± decimal integer
± octal constant

The following are some examples of expressions

```
1    00 0040          777774   AA   DATA   4-7
2    00 0042          000003        DATA   Ø'12'-7
3                000010        BB   EQU    6+2
4                000013        CC   EQU    BB+3
5                000014        DD   EQU    Ø'17'-3
6                000044        EE   EQU    $+1
7    00 0043     000041             DATA   AA+1
```

3.3.6 Literals

A literal is a constant preceded by an equal sign (=) or enclosed in a literal declaration (L 'constant'). The constant may be one of four types, a character string constant,

octal constant, integer constant, or a floating-point constant. The value of the constant is enclosed in single quotes, with a prefix designating the type of constant. When a literal appears in an assembler statement, the assembler stores the binary value of the constant in a drum location following the rest of the program. The address where the constant is stored is the value used in assembling the instruction in which the literal was encountered. When the assembler completes printing the assembled source program, the last items to be printed will always be the value of the literals and their addresses. The following are examples of literals.

```
1     00 0040    060044     LA      = 10
2     00 0041    060045     LA      L'Ø'744"
3     00 0042    060046     LA      = F'4.0'
4     00 0043    060050     LA      = C'AB'
      00 0044    000012
      00 0045    000744
      00 0046    200000
      00 0047    000004
      00 0050    746162
```

3.4 Addressing

Having surveyed the large variety of information types that can be effectively handled by the assembler, we will now turn our attention to the methods which the programmer

may use to reference information that is stored in the core memory and on the drum of the Athena. For the most part, these methods are direct reflections of hardware facilities (e.g., the accumulator and index register) but certain ones are simply notational convention provided by the assembler (e.g., arithmetic expressions that must be evaluated to obtain an address component).

3.4.1 Immediate Addressing

The value of interest is explicitly mentioned as part of the instruction and thus no address is used. Consider the following example.

```
1    00 0040    140017    A    CA    15
2    00 0041    140044    AA   CA    Ø'44'
```

CA loads the rightmost 12 bits of the instruction word into the X register at bit positions X_{12-23}, clears the accumulator and adds the quantity in the X register to the accumulator. Decimal 15 and Ø'44' is used, because it is in the appropriate part of the instruction at the time of execution.

3.4.2 Direct Addressing

The desired address is given in the reference address field of the instruction.

```
1    00 0040    060157    LA    Ø'157'
2    00 0041    060100    LA    64
```

The first 'Load Accumulator' loads the contents of core location Ø'157' into the accumulator. Ø'157' is a direct address. The second address is another example of a direct address using a decimal number (i.e., 64).

3.4.3 Relative Addressing

In relative addressing the desired address is expressed symbolically.

```
1     00 0040     001773     ALPHA  DATA    Ø'1773'
2     00 0041     000144            DATA    Ø'144'
3     00 0042     060040            LA      ALPHA
4     00 0043     060041            LA      ALPHA + 1
```

In Line 3 above, the instruction 'Loads the Accumulator' with the contents of location ALPHA, ALPHA is the symbolic name given to location 40_8. In Line 4 the source of the 'Load Accumulative is one word after the address ALPHA.

3.4.4 Indirect Addressing

In indirect addressing, the desired address is at a remote location and the reference address field is used to point to that location. The Athena does not have hardware to perform indirect addressing. A short softward program is given below which will perform indirect addressing off the drum.

```
          RW      ALPHA
          CX      0
          AC      Ø'76'
          WW      BETA
BETA      RES     1
```

The program below does an indirect address off core memory.

```
          LA      ALPHA
          CX      0
          AC      Ø'06'
          SA      BETA
BETA      RES     1
```

The first program does an indirect address off the drum. The address of the desired quantity is stored in location ALPHA. Line 1 places the content of ALPHA in the accumulator. Line 2 is an instruction (i.e., the CX instruction) which clears and loads the lower 12 bits of the exchange register with the value in the operand field. Line 2 places zeroes in the lower 12 bits of the exchange register. The instruction in Line 3 (i.e., the AC instruction) loads the upper 12 bits of the exchange register with the value in its operand field and then adds all 24 bits of the exchange register to all 24 bits of the accumulator. Line 3 adds the

opcode for a 'Read Word' instruction onto the front of address obtained from ALPHA. The accumulator now contains a machine code to do a 'Read Word' the contents of the address contained in ALPHA. Line 4 stores the machine code into location BETA. This machine code will be executed next. ALPHA and BETA are assumed to be drum address and ALPHA is assumed to contain a drum address.

The second program does an indirect address off core memory on the Athena. The description of the process is similar to the indirect address off the drum. ALPHA and BETA are assumed to be core address and the content of ALPHA is a core address.

3.4.5 Indexing

In indexing the contents of the index register is added to the contents of the address field and the resulting address is used to obtain the operand of the instruction. The following is an example of indexing.

```
1     00 0040     000000      AA     DATA,3      16
      00 0041     000000
      00 0042     000020
2     00 0043     520003             LIRI        2
3     00 0044     024001             INDEX
4     00 0045     760040             RW          AA
```

Line 2 does a 'Load' Index Register Immediate' with two. Line 3 causes the contents of the index register to be added to the address field of the following instruction. When Line 4, a 'Read Word' instruction, is executed, its address field has been modified to a value of $40_8 + 2_8 = 42_8$. The contents of location 42_8 on the drum is read into the accumulator.

The thirteen bits of the index register allow every address on the drum or core memory to be accessed by indexing. The indexing instructions will be discussed in detail in a later section of this chapter.

3.5 Movement of Information

In this section the instruction repertoire of the Athena will be discussed. The instructions discussed will be limited to those that are concerned primarily with movement of information within the confines of the CPU, drum, and core memory. In subsequent sections, the remainder of the instruction repertoire will be considered.

The most general form of each instruction is assembled along with a comment that gives the operation code and a description of the instruction function.

3.5.1 Clear and Load the Exchange (X) Register Immediate

```
                                CX      V
Example
00 0040      120014             CX      12
```

CX, opcode 12_8, clears the exchange register and places the rightmost 12 bits of the instruction word in the exchange (X) register at bit positions X_{00-11}. 'V' is an integer or octal constant, not more than $2^{12}-1$ in size. This is an immediate instruction.

3.5.2 Clear and Load the Accumulator Via the Exchange Register Immediate

CA Value

Example

```
1      00 0040      140017      CA      15
```

'CA', opcode 14, transmits the rightmost 12 bits of the instruction word to the exchange (X) register at bit positions X_{12-23}, clears the accumulator and adds the quantity in X to the accumulator. 'Value' is an integer or octal constant not greater than $2^{12}-1$ in size. This is an immediate instruction.

The use of a 'CX' instruction preceding a 'CA' instruction will result in a complete 24 bit number being loaded into the accumulator. Example:

```
00 0040      120777      CX      '777'
00 0041      140777      CA      '777'
```

Executions of the two instructions above will cause the number 777777_8 to be loaded into the accumulator.

3.5.3 Load Accumulator

LA Core Address

or TP Core Address

Example

```
1              000104            AA      EQU     70
2      00 0040       060104              LA      AA
3      00 0041       060104              TP      AA
```

A 'Load Accumulator', opcode 060 or TP, opcode 060, clears the accumulator and adds the content of the magnetic core storage address to the accumulator. Note that an 'EQU' statement was used in Line 1 to specify the location 70 (104_8) for the core address intended when the symbol AA is used.

3.5.4 Load Complement Accumulator

LCA Core Address

or TN Core Address

Example

```
1              000103            AA      EQU     67
2      00 0040       062103              LCA     AA
3      00 0041       062103              TN      AA
```

A 'Load Complement Accumulator', opcode 062 or TN, opcode 062, clears the accumulator and subtracts the content of the magnetic core storage address from the accumulator.

3.5.5 Store Accumulator

SA Core Address

Example

```
1              000103          AA      EQU     67
2     00 0040      004103              SA      AA
```

Store Accumulator, opcode 004, stores the contents of the accumulator in core storage.

3.5.6 Read Word

RW Drum Address

Example

```
1     00 0040     777776      CC      DATA    -1
2     00 0041     760040              RW      CC
```

'Read Word', opcode 76, clears the accumulator and loads an 18 bit word from the drum into the lower 18 bits of the accumulator. If the leftmost bit of the drum word is one, the upper six bits of the 24 bit accumulator are set to one to provide sign extension. The content of the accumulator after execution of the preceding example would be; 77777776_8.

3.5.7 Read Upper

RU Drum Address

Example

```
1     00 0040     112233       AA      DATA    Ø'112233'
2     00 0041     700040              RU      AA
```

'Read Upper', opcode 70, transmits twelve bits of drum word to the upper twelve bits of the accumulator. If the content of the drum word is 112233_8 and the content of the accumulator is 44556677_8. After execution of a 'Read Upper' instruction the content of the accumulator is 33116677_8. Note that the lower six bits of the drum word (33_8) were transmitted to the upper six bits of the accumulator. The upper six bits of the drum word (11_8) were transmitted to the next six bits of the accumulator. The lower twelve bits of the accumulator were unchanged.

3.5.8 Read Lower

RL Drum Address

Example

```
1     00 0040     112233      AA     DATA    Ø'112233'
2     00 0041     660040             RL      AA
```

'Read Lower', opcode 66, transmits the lower twelve bits of a drum word to the lower twelve bits of the accumulator. If the content of the drum word is 112233_8 and the content of the accumulator is 44556677_8, after execution of a 'Read Lower' the content of the accumulator is 44552233_8.

3.5.9 Write Word

WW Drum Address

Example

```
1     00 0040     000000     AA     RES     1
2     00 0041     620040            WW      AA
```

'Write Word', opcode 62, stores the lower eighteen bits of the accumulator into a drum location. If the content of the accumulator is 11223344_8, and a 'Write Word' is executed, the content of the drum location where the information was stored would be 223344_8.

3.5.10 Write Upper

WU Drum Address

Example

```
1     00 0040     000000     AA     RES     1
2     00 0041     640040            WU      AA
```

'Write Upper', opcode 64, stores the upper twelve bits of the accumulator into a drum location. If the content of the accumulator is 11223344_8, and a 'Write Upper' instruction is executed, the content of the drum location where the information is stored would be 117722_8. Note that the upper six bits of the accumulator (11_8) are stored in the upper six bits of the drum word and the next six bits of the accumulator (22_8) are stored in the lower six bits of the drum word. The middle six bits of the drum word are set to all one.

3.5.11 Write Lower

WL Drum Address

Example

```
1     00 0040     000000     AA     RES     1
2     00 0041     600040            WL      AA
```

'Write Lower', opcode 60, stores the lower twelve bits of the accumulator into a drum location. If the content of the accumulator is 11223344_8, and a 'Write Lower' instruction is executed, the content of the drum location where the information is stored is 443377_8. Note that the lower six bits of the accumulator (44_8) were stored in the upper six bits of the drum word, and the next most significant six bits of the accumulator (33_8) were stored in the middle six bits of the drum location. The lower six bits of the drum word were set to all ones.

3.5.12 Load Index Register

LIR Core Address

or LV Core Address

Example

1	00 0040	026050	LIR	40
2	00 0041	026050	LV	40

'Load Index Register', opcode 026, or 'LV', opcode 026, loads the lower thirteen bits of the contents of the core address into the index register. The index register will a accept a number no larger than $2^{13}-1$.

3.5.13 Load Index Register Immediate

LIRI Value

Example

1	00 0040	520050	LIRI	40

'Load Index Register Immediate', opcode 52, loads the value given in the operand field of this instruction, into the index register. Execution of the 'LIRI' instruction in the example above, with the value 40 in the argument field, would result in the value 40 being placed in the index register. The value in the operand field of a 'LIRI' instruction cannot be greater than $2^{13}-1$.

3.5.14 Load Index Register from the Exchange (X) Register

LIRX

Example

```
1     00 0040     000100      AA     DATA    64
2     00 0041     760040             RW      AA
3     00 0042     426000             LIRX
```

'Load Index Register from the Exchange Register', LIRX (426), causes the lower thirteen bits of the exchange (X) register to be loaded into the index register. This instruction is useful since several instructions (such as the 'Read Word' instruction) cause information to pass through the exchange register where it is retained until the next instruction is executed. The preceding example illustrates the use of the 'LIRX' instruction. In studying the example it is seen that Line 1 store 100_8 in locations 40_8 on the drum. The 'Read Word' instruction in Line 2 loads the content of location 40_8 into the accumulator. In the process,

it also stores the content of location 40_8 into the exchange register where it is retained until execution of the next instruction. The 'LIRX' instruction now loads the content of the exchange register into the index register. The net result of this sequence of executions is that the content of a drum location (location 40_8) is loaded into the index register.

3.5.15 Store Index Register

```
     SIR     Core Address
Example
1                000055            AA     EQU     45
2     00 0040      404055                 SIR     AA
```

'Store Index Register', opcode 404, stores the content of the index register into the core address specified by the operand field (i.e., location 45 in above example).

3.5.16 Index

```
     INDEX
or   TD     1
Example
1     00 0040     000000     BB     DATA, 2     20
                  000024
2     00 0042     520001            LIRI         1
3     00 0043     024001            INDEX
4     00 0044     760040            RW           BB
```

'Index', opcode 024001, or a 'TD' instruction, opcode 024, with a 1 in the operand field, adds the content of the index register to the address field of the instruction that follows. Then upon execution of the instruction following the 'Index' instruction, the modified address field is used in the execution.

In the preceding example, the 'LIRI' instruction in Line 2 places a one in the index register. The 'INDEX' instruction in Line 3 causes the content of the index register to be added to the address field of the following instruction (i.e., $40_8 + 1_8 = 41_8$). The machine code instruction of Line 4 has now been modified to the value 760041_8. Execution of Line 4 now causes the content of drum location 41_8 to be loaded into the accumulator. The original 'Read Word' instruction of statement four is not modified by the indexing instruction (i.e., the content of location 44_8 has not changed).

The 'INDEX' instruction can be used with every instruction in the Athena repertoire.

Use of the 'TD' instruction with a one in the operand field will accomplish the same result as the single 'INDEX' instruction.

3.5.17 Increment the Index Register

```
     INC
or   TD     2
```

Example

```
1     00 0040     520005 A     LIRI     5
2     00 0041     024002 A     INC
```

'Increment' the index register, INC (024002) or a 'TD' instruction with a '2' in the operand field will cause one to be added to the content of the index register. In the example, the 'Load Index Register Immediate' instruction of Line 1 causes five to be loaded into the index register. The content of the index register after execution of the 'Increment' (INC) instruction of Line 2 is now six.

3.5.18 Index and Increment the Index Register

```
     INDEX I
or   TD    3
```

Example

```
1     00 0040     000000 A    CC    DATA, 3    Ø'1234567'
      00 0041     000001 A
      00 0043     234567 A
2     00 0044     520000 A          LIRI       1
3     00 0045     024003 A          INDEXI
4     00 0046     760040            RW         CC
```

'Index and Increment' the index register, INDEXI (024003), or the 'TD' instruction, opcode 024, with a '3' in the operand field, adds the content of the index register to the instruction that follows, and then adds one to the content of the index

register. In the example above, the 'Load Index Register Immediate' instruction of Line 2, loads one into the index register. The 'Index and Increment' instruction that follows adds the content of the index register to the address field of the 'Read Word' (RW) instruction of Line 4, and then adds one to the content of the index register (i.e., 1 + 1 = 2). Execution of the modified 'Read Word' instruction (760041) causes the content of location CC + 1 (i.e., location 41_8) to be loaded into the accumulator. The original 'Read Word' instruction of Line 4 is not modified by the indexing instruction (i.e., the contents of location 46_8 is unchanged).

3.5.19 Exercising the Instructions

The following short program moves ten words of information from one table to another table on the drum.

```
 1                                          LOC     Ø'300'
 2   00 0300      000000    FROM    RES     10
 3   00 0312      000000    TO      RES     10
 4          000050          TEMP    EQU     40
 5          000012          NUMB2   EQU     10
 6   00 0324      520000              LIRI     0
 7   00 0325      024001    START     INDEX
 8   00 0326      760300              RW      FROM
 9   00 0327      024003              INDEXI
10   00 0330      020312              WW      TO
11   00 0331      626050              SIR     TEMP
12   00 0332      120012              CX      NUMB2
13   00 0333      140000              CA      0
14   00 0334      066050              SB      TEMP
15   00 0335      240325              BNEZ    START
```

3.6 Arithmetic Operations'

The instruction repertoire of the Athena includes commands for carrying out the basic arithmetic operation on data fields. These operations along with the related 'shift' instructions are the subject of this chapter.

3.6.1 Add to the Accumulator Immediately Via the Exchange Register

AC V

Example

```
00 0040     150010     AC     8
```

AC (15) transmits the rightmost 12 bits of the instruction word to the exchange (X) register at bit positions X_{12-23}. The contents of the exchange register is then added to the contents of the accumulator. V is an integer or octal constant, not greater than $2^{12}-1$ in size. This is an immediate instruction.

The use of a 'CX' instruction preceding an 'AC' instruction will result in a complete 24 bit number being added to the content of the accumulator.

Example:

```
00 0040     127777     CX     Ø'7777'
00 0041     157777     AC     Ø'7777'
```

Execution of the two instructions above will cause the number 77777777_8 to be added to the content of the accumulator.

3.6.2 Add and Subtract

AD Core Address

SB Core Address

Example

```
1              000051              AA      EQU     41
2      00 0040         064051              AD      AA
3      00 0041         066051              SB      AA
```

'Add', AD (064), adds the contents of the accumulator to the quantity in core storage and places the result in the accumulator. A check is made for overflow.

'Subtract', SB (066), subtracts from the contents of the accumulator the quantity in core storage and places the results in the accumulator. A check is made for overflow.

3.6.3 Multiply and Divide

MP Core Address

DV Core Address

Example

```
1      00 0040         000005      AA      DATA    5
2      00 0041         000006      BB      DATA    6
3              000016              CC      EQU     14
4      00 0042         760040              RW      AA
5      00 0043         004016              SA      CC
6      00 0044         760041              RW      BB
7      00 0045         110016              MP      CC
```

8	00 0046	104027	LS	23
9		*		
10	00 0047	760041	RW	BB
11	00 0050	106027	RS	23
12	00 0051	112016	DV	CC
13	00 0052	104027	LS	23
14	00 0053	114000	OC	

'Multiply', opcode 110, multiplies the content of the accumulator by the quantity in core storage and places the result in the accumulator and quotient (AQ) register.

'Divide', opcode 112, divides the content of the AQ register by the quantity in core storage and places the quotient in the Q register and the absolute value of the remainder in the accumulator.

In the example, the content of the drum location AA is stored in core storage at location CC. The content of drum location BB is loaded into the accumulator and then multiplied by the content of core location CC. The result of the multiplication (5 x 6 = 30) is right justified in the AQ register (this being integer arithmetic). In Line 8, a 'Left Shift', LS instruction is executed to left shift the final result into the accumulator for disposition.

Also shown in the example is the use of the division code. In Line 10, the content of BB is placed in the accumulator.

The content of the accumulator is the shifted over into the quotient register. The content of the AQ register is then divided by the content of core location CC. The result (6/5 = 1) is in the quotient register. A 'Left Shift' instruction is executed to left shift the result into the accumulator for disposition.

3.6.4 Right Shift

RS k

Example

```
1     00 0040     106014     RS     12
```

The 'Right Shift' instruction, RS (106), shifts the contents of the accumulator and quotient register to the right by 'k' bits ($0 \leq k \leq 31$). The contents of the accumulator is shifted over into the quotient register. The algebraic sign is retained (ones shifted in from the left if negative, zeroes shifted in from the left if negative).

3.6.5 Right Shift Logical

RSL k

Example

```
00 0040     506022     RSL     18
```

'Right Shift Logical', opcode 506, shifts the contents of the accumulator and quotient register to the right by 'k' bits ($0 \leq k \leq 31$). The compliment of the algebraic sign is retained (zeros shifted in from the left if negative, ones if positive).

3.6.6 Left Shift

LS k

Example

```
1     00 0040     104014 A     LS     12
```

The 'Left Shift' instruction, LS (104), shifts the contents of the accumulator and quotient registers to the left 'k' bits ($0 \leq k \leq 31$). A check is made for a shift overflow during the 'Left Shift' operation. A shift overflow occurs when the sign of a number is changed at any time during the left shift operation. When a shift overflow is detected, the system will stop execution unless the next instruction is an 'Overflow Jump', (OJ), or an 'Overflow Condition', (OC). Execution of either of these instructions will clear the overflow fault. If no overflow is detected, execution will continue uninterrupted.

3.6.7 Transfer the Quotient Register

TQ k

Example

```
1     00 040      102027 A     TQ     23
```

'Transfer Quotient' register, TQ (102), left shifts the content of the accumulator and quotient register 'k' places ($0 \leq k \leq 31$). After shifting the algebraic sign is placed from A_{24} to A_{23}.

3.6.8 Overflow

Overflow is a condition in which a number is too large in magnitude to be expressed in a register. Three types of overflow conditions are detected by the Athena computer. They are addition overflow, subtraction overflow, and shifting overflow.

Addition overflow exists when two numbers are added whose sum is two large in magnitude to be expressed in the register that receives the sum. In order for overflow to be possible during addition, the signs of the two original numbers must be the same. If the signs are the same and the sum of the two numbers is different in sign, then overflow has occurred.

Subtraction overflow is possible when the signs of the two numbers are opposite. If the signs are different, the sign of the final difference should be the same sign as the minuend; that is subtracting a negative number from a positive number, the result should be positive.

Shifting overflow occurs when the sign of a number is changed at any time during the left shift operation.

The Athena provides two instructions for handling overflow conditions during arithmetic operations. These are the 'Overflow Jump' instruction and the 'Overflow Condition' instruction.

3.6.8.1 Overflow Jump

OJ Drum Address

Example

```
1     00 0040     064032     AD     '32'
2     00 0041     26XXXX     OJ     ERROR ROUTINE
```

'Overflow Jump', OJ (26), on detection of an overflow error in the preceding instruction, causes the computer to clear the overflow flip-flop and transfers control to the address given in the operand field. Execution continues from that point. If an overflow condition was not detected in the preceding instruction, no action is taken and execution is continued with the next instruction in line.

3.6.8.2 Overflow Condition

OC (no argument)

Example

```
1     00 0040     066014     SB     12
2     00 0041     114000     OC
```

If an overflow has occurred since the last 'Overflow Condition', 'Overflow Jump' or 'Wait Computation' the 'Overflow Condition', (114), instruction will clear the overflow condition flip-flop and lights the 'COMP FAULT' light on the console. Program execution then continues with the next instruction in line. If no overflow has occurred, no action is taken and execution continues with the next instruction in line.

3.6.9 Floating-Point Arithmetic

The Athena computer has no hardware facilities for floating-point arithmetic. Several macro type programs which perform floating-point add, subtract, multiply, and divide are presented in Section 3.8.1.5 where macro type programs are discussed. These programs are stored permanently in the assembler.

3.6.9.1 Shifting Floating-Point Numbers

There are rules for shifting the fractional part of a floating-point number right or left without disturbing its value. For each right shift of the fractional part of the floating-point number, add one to the value of the exponent. For each left shift of the fractional part subtract one from the exponent.

3.7 Selective Sequencing

The ability to move items of information and perform arithmetic operations is fundamental to the utility of a digital computer; but it is the ability to selectively and repeatedly execute instructions that gives a digital computer its real power. The material in this section is concerned with the facilities of the Athena computer that enable a user to pose questions about items of information and conditionally (or unconditionally) alter the flow of his program.

3.7.1 Unconditional Branch (Jump)

B Drum Address

UJ Drum Address

Example

```
1     00 0040     200042            B      LOAD
2     00 0041     200042            UJ     LOAD
3     00 0042     060062   LOAD     LA     50
```

'Branch', opcode 20, or 'Unconditional Jump', opcode 20, cause the program to branch to the drum address given in the argument field. Program execution continues from that point.

3.7.2 Branch If Less Than (Zero) or Sign Jump

BLZ Drum Address

BL Drum Address

SJ Drum Address

Example

```
1     00 0040     777773       AA     DATA     -4
2     00 0041     760040              RW       AA
3     00 0042     220043              BLZ      BB
4     00 0043                  BB     EQU      $
```

'Branch if Less than (Zero)', opcode 22, or 'Sign Jump', opcode 22, branchs if the content of the accumulator is negative.

3.7.3 Branch If Not Equal (Zero) or Zero Jump

```
        BNEZ    Drum Address
or      BNE     Drum Address
or      ZJ      Drum Address
```

'Branch If Not Equal (Zero)', or 'Zero Jump', opcode 24, causes the program to branch if the content of the accumulator is non-zero.

3.7.4 Branch If Greater Than Zero

Execution of the three instructions below will cause the program to branch if the content of the accumulator is greater than zero.

```
        BLZ     CONTINUE
        BNEZ    Drum Address
CONTINUE EQU    $
```

3.7.5 Branch If Greater Than or Equal to Zero

The following instructions will perform a branch if the content of the accumulator is greater than or equal to zero.

```
        BLZ     CONTINUE
        B       Drum Address
CONTINUE EQU    $
```

3.7.6 Branch If Less Than or Equal to Zero

The following instructions will perform a branch if the content of the accumulator is less than or equal to zero.

```
            BLZ     Drum Address
            BNEZ    CONTINUE
            B       Drum Address
CONTINUE    EQU     $
```

3.7.7 Branch If Equal to Zero

The following instructions will perform a branch if the content of the accumulator is zero.

```
            BNEZ    CONTINUE
            B       Drum Address
CONTINUE    EQU     $
```

3.7.8 Branch (Jump) If Index Register is Equal to Zero

```
     BIZ     Drum Address
or   JIZ     Drum Address
```

Example

```
1     00 0040     5217774              LIRI   -2
2     00 0041     300044      NEXT     BIZ    CONTINUE
3     00 0042     024002               INC
4     00 0043     200041               B      NEXT
5           000044           CONTINUE  EQU    $
```

'Branch If index register Zero, opcode 30, or 'Jump If Index register Zero', opcode 30, cause a branch if the content of the index register is zero. This instruction allows the index register to be used as a loop counter. A simple example is presented above. The example will also illustrate

a problem to be encountered using the index register as a loop counter. In the example, the index register is loaded with a minus two (-2) first. Then the 'BIZ' instruction checks to see if the index register contains a zero, which it doesn't. Then the 'INC', increments the index register by one. The index register now contains a minus one. A branch instruction then branchs the program back up to NEXT. Executions will continue in this loop until the index register is zero, at which time the 'BIZ' instruction will cause a branch to CONTINUE. The program will make "three" passes through the loop before leaving it; not two. The reason is that index register passes through negative zero as it is incremented to zero. This is an extra count. Negative zero is a binary number consisting of all ones. In the example the index register went through the sequence -2, -1, -0, 0.

3.8 Subroutines

The creation of subroutines extends the instruction repertoire of a computer by providing new functions which can be invoked in much the same manner as an individual instruction. The creation and use of subroutines is normally accomplished by packaging appropriate sets of instructions in a standard manner and employing certain facilities provided by the assembler for this purpose.

There are several motives for using subroutines. The use of subroutines can result in a saving of space, e.g., a set

of instructions may be used many times throughout a program but occupy only a single set of drum locations.

The use of subroutines can result in a saving of both programming and assembly time since a set of instructions can be packaged for extended use. Of perhaps primary importance is the possibility of extending the usefulness of the individual programmer, e.g., the limits imposed by the availability of personal time and knowledge are extended through the use of prepackaged sets of instructions that may be executed through adherence to a standard set of conventions.

3.8.1 Open Subroutines

In an open subroutine the instructions that make up the open subroutine are always placed in line at each place where their execution may be desired.

Special techniques are not used to pass control or information to (or from) an open subroutine. Proper placement establishes all communcation.

Open subroutines are usually short-longer sequences of instructions are normally packaged as closed subroutines in order to conserve space.

3.8.1.1 Macro Instructions

The macro facility of the assembler allows a programmer to give a name to a sequence of instructions that make up an open subroutine. Later, he may ask for the insertion of the

named instructions at any point in his program that he wishes. The programmer may also direct that certain parameter substitutions be made at the point of insertion.

3.8.1.2 Macro Definitions

The coding sequence that defines a macro is delimited by the directives 'MACRO' and 'END'. The macro name appears in the label field of the 'MACRO' directive. The operand field may contain dummy variables. When the macro call is processed, call arguments are substituted for dummy variables wherever they appear in the macro. An example of a macro definition is shown as follows.

```
SUM     MACRO     LABEL,ARG1,ARG2,ARG3
*THIS MACRO SUMS ARG1 AND ARG2, RESULT IN ARG3
LABEL   RW        ARG1
        SA        100
        RW        ARG2
        AD        100
        WW        ARG3
        END
```

In the example SUM is the name of the macro, and appears as such in the label field of the MACRO directive. This simple macro sums two numbers, ARG1, and ARG2, then stores the result in ARG3. A location on the core memory (location

100) is used to hold intermediate results. LABEL, ARG1, ARG2, and ARG3 are dummy arguments. The first dummy argument is always used for label substitution. The other dummy arguments may occur anywhere in the macro. This includes the label field, command field or operand field. The END directive indicates the end of the macro definition.

Comment cards are permitted in macro definition and are stored with the macro definition. The END directive when used on a MACRO definition only serves to indicate the finish of the MACRO definition.

The definition of a particular macro must precede any reference to it.

3.8.1.3 Using the Macro

To call for the use of a macro, its name is placed in the command field of a statement just as if it were an operation code. If a label is attached, it is placed in the appropriate place by the assembler. Arguments in the operand field are substituted also. The number of arguments in the operand field of the macro call and the macro definition must be equal. An example of a call on the macro defined previously would cause the following code to be substituted in line.

```
ALPHA     SUM     X,Y,Z
*THIS MACRO SUMS ARG1 AND ARG2,RESULT IN ARG3
ALPHA     RW      X
          SA      100
          RW      Y
          AD      100
          WW      Z
```

3.8.1.4 Substitution of Arguments

When the macro is used, the arguments of the macro call are substituted for the dummy arguments in the macro definition. The preceding example of a macro call indicates how the argument substitutions are made. The label ALPHA is substituted for the first dummy argument in the macro definition. X is substituted for ARG1, Y for ARG2 and Z for ARG3. There is no restriction on where dummy arguments are used in an instruction. Dummy arguments may appear in the label field, command field, or argument field. A dummy argument may appear more than once in the body of the macro. No more than one dummy argument may appear in the label, command, or argument field. Dummy arguments and call arguments may consist of any character other than a comma or blank. The number of call arguments has to equal the number of dummy arguments, excluding the label. A label is not required in the macro call. The following are examples of macro definitions and macro calls.

```
TEST    MACRO     LABEL, ARG1, ARG2, BRANCH, ADDRESS
LABEL   LA        ARG1
        SB        ARG2
        BRANCH    ADDRESS
        END
```

A macro call is made on the preceding macro definition. The code generated is:

```
TEST     ALPHA, BETA, BLZ, LESS THAN
LA       ALPHA
SB       BETA
BLZ      LESS THAN
```

A second example.

```
SQUARE-INDEX-REG     MACRO     LABEL
LABEL                SIR       100
                     LA        100
                     MP        100
                     LS        23
                     OC
                     SA        100
                     LIR       100
                     END

AA       SQUARE-INDEX-REG
AA       SIR       100
         LA        100
         MP        100
         LS         23
         OC
         LIR       100
```

3.8.1.5 Macro Programs for Floating-Point Arithmetic

The Athena computer has no hardware facilities for floating-point arithmetic. Several macros for performing

floating-point arithmetic are discussed in this section.

These macros are an integral part of the assembler. The discussion that follows will describe how to call these macros. The macro definitions, as they are stored in the assembler, are given in Appendix D. As is evidenced by the length of the floating-point arithmetic macros in Appendix D, floating-point arithmetic requires much more storage (not only for the data but for the instructions themselves) and execution time than integer arithmetic.

3.8.1.5.1 Floating Add and Subtract Macro Program

The call on this macro is:

```
label   FA  L1,L2,L3,L4,L5,L6,L7,A1,A2,B1,B2,C1,C2
```

The floating add macro takes two floating-point numbers A and B, adds them together and stores the result in C. All locations are core addresses. The first number A is stored in core location A1 and A2. B is stored in core locations B1 and B2. The result is stored in core locations C1 and C2. A1, B1, and C1 are assumed to be the locations of the fractional part of the floating-point numbers. A2, B2 and C2 are assumed to be the locations of the exponent of the floating-point numbers. The calling arguments L1, L2, L3, L4, L5, L6, and L7 are labels used internally by the macro. With each call on the floating add macro, new symbols have to be provided for L1, L2, L3, L4, L5, L6, and L7. No storage locations have to be provided for L1 - L7.

Three core locations have to be provided for the macro to store intermediate results. These temporaries are TEMP1, TEMP2 and TEMP3. Two typical call on the floating-point add macro are as follows.

```
TEMP1    EQU    200
TEMP2    EQU    210
TEMP3    EQU    220
AA       EQU    100
BB       EQU    102
CC       EQU    104
XX       EQU     10
YY       EQU     20
ZZ       EQU     30
         FA  P1,P2,P3,P4,P5,P6,P7,AA,AA+1,BB,BB+1,CC,CC+1
         FA  R1,R2,R3,R4,R5,R6,R7,XX,XX+1,YY,YY+1,ZZ,ZZ+1
```

In the above example, the first three statements designate core locations for the temporaries, TEMP1, TEMP2, and TEMP3. The following six statements designate core memory locations for floating-point numbers AA, BB, CC, XX, YY, and ZZ. In each case it is assumed that the fractional part of the floating-point number is stored in the core memory location designated for the symbol (for example, the fractional part of floating-point number AA is stored in core location 100) and the exponent of the floating-point number is stored in the next

location (the exponent of 'AA is stored in core loaction 101). In the above example of calls on the floating add macro, the first call adds floating-point numbers AA and BB and stores the result in CC. Seven symbols, P1, P2, P3, P4, P5, P6, and P7, are provided to the macro for use as internal labels. The second call on the floating add macro, adds floating-point numbers XX and YY and stores the result in ZZ. Seven new symbols, R1, R2, R3, R4, R5, R6 and R7 are provided to the macro for use as internal labels.

Floating subtract is performed by complementing the number to be subtracted and then floating add the complemented number to the other number (i.e., A-B = A+(-B)). To complement a floating-point number, it is only necessary to complement the fractional part of the number. The exponent is left unchanged. The following is an example of a floating subtract using the floating add macro.

```
TEMP1     EQU     100
TEMP2     EQU     101
TEMP3     EQU     102
COMPL     EQU     200
AA        EQU     201
BB        EQU     203
CC        EQU     205
          LCA     BB
          SA      COMPL
          FA      I1,I2,I3,I4,I5,I6,I7,AA,AA+1,COMPL,BB+1,
                  CC,CC+1
```

In the preceding example, the floating subtraction of floating-point number BB from floating-point number AA is performed (i.e., AA - BB). The result is stored in CC. The first seven statements designate core locations for TEMP1, TEMP2, TEMP3, COMPL, AA, BB, and CC. In the two statements preceding the floating add macro call, the factional part of BB complemented and stored into core location COMPL. COMPL is then used as a calling argument in the floating add macro call in place of BB. Again I1, I2, I3, I4, I5, I6 and I7 are provided to the macro for use as internal labels.

3.8.1.5.2 Floating Multiply Macro Program

The call on the floating multiply macro is:

```
label   FM   L1,L2,L3,L4,A1,A2,B1,B2,C1,C2
```

The floating multiply macro takes two floating-point numbers A and B, multiplies them together and stores the result in C. All locations are core addresses. The first number A is stored in core location A1 and A2, and B is stored in core location B1 and B2. A1, B1, and C1 are assumed to be the core addresses for the fractional part of the floating-point number. A2, B2, and C2 are assumed to be the core addresses for the exponents. The calling arguments L1, L2, L3, and L4 are labels used internally by the macro. With each call on the floating multiply macro, new symbols have to be provided. No storage locations have to be provided for L1 - L4. A core location has to be provided for

the macro to store intermediate results. The temporary is called TEMP1. A typical call on the floating multiply macro is shown in the following example.

```
TEMP1    EQU    10
AA       EQU    100
BB       EQU    102
CC       EQU    104
         FM     M1,M2,M3,M4,AA,AA+1,BB,BB+1,CC,CC+1
```

In the preceding example, the first four statements designate core locations for TEMP1, AA, BB, and CC. The fractional parts of AA and BB are assumed to be stored in core locations AA and BB (i.e., 100 and 102). The exponents of AA and BB are assumed to be stored in core locations AA+1 and BB+1 (i.e., 101 and 103). The results of the floating multiply are stored in core location CC and CC+1 (i.e., 104 and 105). Location CC will contain the fractional part and CC+1 will contain the exponent. Four symbols, M1, M2, M3 and M4 were provided to the macro for use as internal labels.

3.8.1.5.3 Floating Divide Macro Program

The call on the floating divide macro is:

```
label  FD  L1,L2,L3,L4,A1,A2,B1,B2,C1,C2
```

The floating divide macro takes two floating-point numbers A and B, divides B into A,(A/B), and stores the result in C. Numbers A and B are stored in core location A1, A2, and

B1, B2 respectively. A1 and B1 are assumed to be the core ad addresses for the fractional parts of A and B. A2 and B2 are assumed to be the core addresses for the exponents of A and B. The floating divide macro stores the result of the division in C1 and C2. C1 will contain the fractional part and C2 the exponent of C. The calling arguments L1, L2, L3 and L4 are labels used internally by the macro. With each call on the floating divide macro, new symbols have to be provided for L1, L2, L3 and L4. No storage locations have to be provided for L1 - L4. A core location has to be provided for the macro to store intermediate results. The temporary is called TEMP1. A typical call on the floating divide macro is shown in the following example.

```
TEMP1       EQU       220
XX          EQU        10
YY          EQU        12
ZZ          EQU        14
            FD        F1,F2,F3,F4,XX,XX+1,YY,YY+1,ZZ,ZZ+1
```

In the preceding example, the first four statements designate core locations for TEMP1, XX, YY, and ZZ. The fractional parts of XX and YY are assumed to be stored in core locations XX and YY (i.e., 10 and 12). The exponents of XX and YY are assumed to be stored in core locations XX+1 and YY+1 (i.e., 11 and 13). The results of the floating divide (XX/YY) are stored in core locations ZZ and ZZ+1 (i.e., 14 and

15). Location ZZ (i.e., 14) will contain the fractional part and ZZ+1 (i.e., 15) will contain the exponent. Four symbols F1, F2, F3 and F4 were provided to the macro for use as internal labels.

3.8.2 Closed Subroutines

Instructions that make up a closed subroutine normally occur only once in a given user's program. In addition to the desired instructions, a number of additional instructions must be added to handle overhead functions such as receiving and returning control and transmitting appropriate information.

3.8.2.1 Transfer of Control

Execution of the following instructions causes control to be transfered to the closed subroutine.

```
                B       LO2
        LO1     DATA    RETURN ADDRESS
        LO2     RW      LO1
                B       subroutine name
RETURN ADDRESS  EQU     $
```

Thus control is passed to the subroutine, with the address to be returned to, stored in the accumulator.

3.8.2.2 Entry into the Subroutine

On entry into a subroutine, the return address, which is stored in the accumulator is stored for subsequent use.

This is accomplished with a store into a temporary location such as:

```
        WW      TEMP
```

3.8.2.3 Transfer of Information

Along with control it is often necessary to pass data to a subroutine and/or receive answers back. On the Athena the best method is to designate a section of core memory or drum storage to serve as a buffer area between the calling program and the subroutine. Information to be passed to the subroutine is placed in the area previous to transfer of control to the subroutine. Information to be passed back to the calling program is contained in this area on return from the subroutine.

3.8.2.4 Return of Control to Calling Program

Return of control to the calling program from the subroutine is accomplished with the following instructions.

```
        RW      TEMP
        CX      0
        AC      Ø'20'
        WW      RETURN
RETURN  RES     1
```

The return address which is stored in TEMP is loaded into the accumulator. The opcode for a 'Branch' instruction is placed onto the front of the address. The 'Branch' opcode with the

return address is stored in the next drum location for execution. Subsequent execution of the 'Branch' instruction causes control to return to the calling program at the point following the call to the subroutine.

3.8.2.5 Example of a Closed Subroutine

The following example illustrates a closed subroutine.

```
**PLACED DATA FOR SUBROUTINE INTO
**BUFFER AREA
                RW      ALPHA
                WW      BUFFER1
                RW      BETA
                WW      BUFFER2
*
**STORE RETURN ADDRESS AND PASS
**CONTROL TO SUBROUTINE
*
                B       LO2
LO1             DATA    RETURNADDRESS
LO2             RW      LO1
                B       SUMSQR
RETURNADDRESS   EQU     $
**
**GET RETURNED DATA FROM BUFFER AREA
                RW      BUFFER3
                WW      GAMMA
                WC
```

```
**SUBROUTINE SUM OF SQUARES
*
*
*STORE RETURN ADDRESS
*
SUMSQR          WW          TEMP
*
*SQUARE FIRST NUMBER
*
                RW          BUFFER1
                SA          100
                MP          100
                LS           23
                OC
                SA          101
*
*SQUARE SECOND NUMBER
*
                RW          BUFFER2
                SA          100
                MP          100
                LS           23
                OC
*
*SUM THE SQUARES
```

```
*
                    AD          101
*
*STORE RESULT IN BUFFER AREA
*
                    WW          BUFFER3
*
*RETURN TO CALLING PROGRAM
*
                    RW          TEMP
                    CX          O
                    AC          Ø'20'
                    WW          RETURN
RETURN              RES         1
*
*
*THE FOLLOWING IS THE
*DATA AREA FOR THE
*PROGRAM
*
ALPHA               DATA        4
BETA                DATA        5
GAMMA               RES         1
BUFFER1             RES         1
BUFFER2             RES         1
BUFFER3             RES         1
```

In the preceding example, a call is made on a short subroutine. The subroutine forms the sum of the squares of two numbers. Data for the subroutine is passed to and from the subroutine through a buffer area. The various features of the example have been explained previously.

3.9 Control Operations

The Athena provides two computer control instructions. These are the 'Wait Partial' command, and 'Wait Computation' command.

3.9.1 Wait Partial

	WP	Drum Address		
Example				
1	00 0040	017000	DD	O
2	00 0041	340042	WP	S+1

'Wait Partial', opcode 34, causes the computer to hault and wait for the Partial Cycle Sync pulse. When it comes, the program branch to the address given in the operand field of the 'Wait Partial' instruction. One use of the 'Wait Partial' instruction is after output instructions. On execution of an output instruction, program execution is halted while the output device types. The output device on completion of its activities, initiates a Partial Cycle Sync pulse. The 'Wait Partial' instruction starts program execution again

at the address given in the operand field on receipt of this pulse.

3.9.2 Wait Cycle

	WC	Drum Address			
1	00 0040	000410 A	WC	Ø'410'	

'Wait Cycle' opcode 00, causes the computer to wait for the Computation Cycle Sync pulse. When it arrives, it takes the address in the Program Control Register for the next instruction; however, only group zero can be referenced with this instruction. If this sync pulse should occur unexpectedly, that is, during the execution of an instruction, the Program Control Register is cleared at the completion of the current instruction; the SYNC FAULT indicator is lit; the address zero, group zero, is taken as the address of the next instruction.

3.10 Output

The Athena computer provides six instructions for outputting information. Four instructions are provided for outputting to the flexowriter and two instructions for outputting onto the digital printer. Information may be outputted in decimal, octal or alphanumeric format.

3.10.1 Print Octal

	PRTØ	
or	DD	3

Example

```
1      00 0040      017003      PRTØ
2      00 0041      340042      WP        S+1
```

'Print Octal', opcode 017003, or 'DD', opcode 017, with a 3 in the operand field, prints the content of the accumulator in octal form on the digital printer (8 digits). The 'Wait Partial' instruction halts execution until the digital printer is completed.

3.10.2 Print Decimal

```
     PRTD
or   DD        2
```

Example

```
1      00 0040      017002      PRTD
2      00 0041      340042      WP        S+1
```

'Print Decimal', opcode 017002, or 'DD', opcode 017, with a 2 in the operand field, prints the 32 least significant bits of the accumulator and quotient register in decimal form on the digital printer (sign and seven digits). The 'Wait Partial' instruction halts execution until the printer is completed.

3.10.3 Type Octal Short

```
     TYPEØS
or   DD         O
```

Example

```
1     00 0040     017000     TYPEØS
2     00 0041     340042     WP          S+1
```

'Type Octal Short', opcode 017000, or 'DD', opcode 017, with a zero in the operand field, types the 18 least significant bits of the accumulator in octal form on the flexowriter (6 octal digits). The flexowriter then does an automatic space. The 'Wait Partial' instruction in the example halts execution until the flexowriter is completed.

3.10.4 Type Octal Long

```
     TYPEØL
or   DA          3
```

Example

```
1     00 0040     417003     TYPEØL
2     00 0041     340042     WP          S+1
```

'Type Octal Long', opcode 417003, or 'DA', opcode 417, with a 3 in the operand field, types the content of the accumulator in octal form on the flexowriter and then does an automatic return. The 'Wait Partial' instruction in the example halts execution until the flexowriter has completed typing.

3.10.5 Type Decimal

```
     TYPED
or   DA          2
```

'Type Decimal' opcode 417002, or 'DA', opcode 417 with a 2 in the operand field, types the content of bits 8-0 of the accumulator and 23-0 of the quotient register in decimal (BCD) form on the flexowriter (sign and 7 digits). If the leftmost digits is 8_{10}, it will space over one for a plus sign. If the leftmost digit is 9_{10} it will type (") for a minus sign. After the decimal number, the flexowriter does an automatic carriage return. The 'Wait Partial' instruction in the example, halts program execution until the flexowriter has completed typing.

3.10.6 Type Alphanumeric

```
     TYPEA
or   DD        1
```

Example

```
1     00 0040     017001     TYPEA
2     00 0042     340042     WP         S+1
```

'Type Alphanumeric', opcode 017001, or 'DD', opcode 017, with a 1 in the operand field, types the 18 least significant bits of the accumulator in alphanumeric form on the flexowriter (3 characters). The 'Wait Partial' instruction halts program execution until the flexowriter has completed typing.

3.10.7 Space Digital Printer

```
     DD     4
```

Example

```
1     00 0040     017004     DD     4
1     00 0041     340042     WP     S+1
```

'DD', opcode 017, with a mode of 4, causes the digital printer to space. The 'Wait Partial' instruction halts program execution until the printer has completed spacing.

3.11 Programming Errors

As a programming aid, the assembler provides error detection facilities for the programmer. Diagnostic messages are printed above the offending instruction line in the output listing. An error number is given for use in diagnosing the problem. An explanation of the meaning of the error numbers is given in Appendix B.

3.12 Executing a Program Across Drum Group Boundaries

A significant problem with the Athena is doing program execution across a drum group boundary. In this discussion, the problem will be defined, and then the solution will be presented. The drum on the Athena consists of eight groups, of 1024 words each. The groups are numbered 0-7, and the words 0-1023. If a program is executing on say group zero, and arrives at the last word of group zero (location 1023), execution will not continue with the first word of group one but will start over again at location zero of group zero.

The program address register which contains the address of the next executable statement, does not automatically increment the group address portion of its address when it arrives at a group boundary. It retains the group address portion (the upper 3 bits) and starts the word address portion (the lower 10 bits) over at zero. The group address portion may be set to a new group number by executing a branch instruction.

To insure that program execution continues into the next group, a branch instruction is inserted into the assembly language program whenever a group boundary is about the be crossed. The branch instruction contains the address of the first word of the next group in its operand field. The branch instruction will increment the group portion (the upper 3 bits) of the program address register by one. The following example illustrates what has been said.

The programmer inputs the following program to the assembler.

```
LOC    1021
LA      100
SA      102
MP      100
LS       23
OC
```

The assembler output is the following.

```
1                                    LOC     1021
2      00 1775      060144           LA       100
3      00 1776      004146           SA       102
       00 1777      202000           B        S+1
4      01 0000      110144           MP       100
5      01 0001      104027           LS        23
6      01 0002      114000           OC
```

In the example above, the assembler has inserted a 'Branch' statement after Line 3. This 'Branch' statement causes execution to continue with Line 4 which is on the next group of the drum (note the location counter value for Line 4; group 1, location 0000).

There are three variations to this theme. The above method will not work if the 'Branch' instruction were accidentally inserted following an 'Index' or 'Index and Increment' instruction. The index instructions would modify the address portion of the branch instructions and the results would be unpredictable. The solution to this problem is to check for the presence of an indexing instruction at the time of insertion of a 'Branch to next group' instruction into the program. If an indexing instruction is present, a 'Branch to next group' instruction is inserted in its place and the indexing instruction is moved to first word of the next group. The following example illustrates this method.

The programmer inputs the following program to the assembler.

```
LOC       Ø'1775'
LIRI       10
INDEX     100
LA         40
SA
```

The assembler output is as follows.

```
1                                    LOC      Ø'1775'
2      00 1775     520012            LIRI      10
       00 1776     202000            B         S+2
       00 1777     202000            B         S+1
3      01 0000     024001            INDEX
4      01 0001     060144            LA        100
5      01 0002     004050            SA         40
```

The second variation involves the 'Wait Partial' instruction. The 'Wait Partial' instruction accomplishes the same result as a unconditional 'Branch' instruction when it appears as the last executable instructior in a drum group. If the last instruction in a drum group is a 'Wait Partial' instruction, a 'Branch to next group' instruction is not inserted in the program.

The third variation involves data storage areas. A 'Branch to next group' instruction is not inserted among data

statements when the location counter crosses a drum group boundary for obvious reasons.

CHAPTER IV

4.1 Introduction

Broadly speaking, an assembler is a program which produces a machine language object program from a symbolic assembly language source program. In other words, it translates a program in assembly language into a program in machine language.

In this chapter the internal structure of the assembler will be described. After a brief description of the assembly process, the tasks an assembler must perform will be described. The gross structure of the assembler will be described and interfaces identified. A detailed discussion of the assembler then follows.

4.2 Athena Assembly Language

There are several disadvantages to using direct machine language. It is difficult to work with octal numbers and the numeric operations codes are difficult to remember. In addition, use of numeric addresses make it difficult to modify a program. Instructions must be stored in sequence. Inserting new instructions or deleting old ones cause the location of many of the remaining instructions to change, resulting in a change in all instructions which refer to them.

Symbolic assembly language attempts to overcome these disadvantages by permitting the use of symbols for operation codes, addresses and other items in the instructions. A symbol is a group of letters and digits. Symbols act as names or labels and usually are chosen to have some mnemonic significance. The symbols for the operation codes are fixed. They are frequently an abbreviation of their name. The symbols used for addresses are selected by the writer of the program. When a symbol is used for the address of an instruction or datum, that instruction or datum must be labeled with that symbol. Finally, in assembly language numbers are usually written in decimal notation.

To describe the Athena assembly language, syntactical notation will be used. Syntactical notation allows a precise description of an assembly language. The following is a syntactical definition of the Athena assembly language.

machine instruction: = [label] - operation code - machine operand

data directive: = [label] - DATA [, decimal integer] - literal
|machine operand | constant

reserve directive: = [label] - RES - decimal integer

equivalence directive: = label - EQU - machine operand

set location counter directive: = LOC - decimal integer | octal constant

define symbol directive: = DEF - symbol

end of program directive: = END

macro definition directive: = operation code - MACRO - [dummy argument]$_{0}^{19}$ {,dummy arguments}

macro call: = [label] - operation code - [call argument] $_{0}^{19}${, call arguments}

machine operand: = expression | octal constant | symbol | decimal integer | dollar sign

literal: = = constant | L'constant'

constant: = decimal integer | octal constant | floating-point constant | character string constant

expression: = {octal constant | symbol | decimal integer|$} {+|-} decimal integer} | {+|-} {octal constant | decimal integer}

label: = symbol

operation code: = $_{1}^{15}${any character except a comma or a blank}

symbol: = $_{1}^{15}${letters or digits, at least one letter and no blanks}

dummy argument: $_{1}^{30}${any character except a comma or blank}

call argument: = dummy argument

decimal integer: = $_{1}^{6}${digits}

octal constant: = Ø'$_{1}^{12}${octal numbers}'

floating-point constant: = F'$_{0}^{6}${digits}$_{n}$ $_{0}^{6-n}${digits}$_{m=6}$ [E[+|-]$_{1}^{2}${digits}]'

character string constant: = C' $_{1}^{24}${any character, with a single quote being represented by two quotes}'

dollar sign: = $

digit: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

letter: = A | B | C - - - X | Y | Z

octal number: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

The above syntactical definition for a machine instruction can be interpreted as follows: A machine instruction is defined as consisting of an optional label, followed by a blank, followed by an operation code, followed by a machine operand. The brackets around the label indicate that it is optional. The bar symbol (-) is used to indicate a blank.

The syntactical definition for a data directive reads; a data directive is defined as consisting of an optional label, followed by a blank, followed by DATA, followed by an optional comma, and an optional decimal integer, followed by a blank, followed by a literal, or a machine operand or constant. The vertical slash (|) is a symbol for "or" as in literal or machine operand (literal | machine operand).

As a final example, the syntactical definition for an octal constant reads; an octal constant is defined as consisting of a Ø followed by a quote ('), followed by one to twelve octal numbers, followed by a quote ('). The parenthesis with the lower one and upper twelve indicate the lower and upper limits of the contents of the parenthesis.

4.3 The Assembler

Since assembly language can not be executed directly by the computer, it must first be translated into machine language. That is the function of the assembler. The input to the assembler then is a symbolic assembly language program containing symbolic machine instructions and directive instructions.

The assembler usually produces a single numeric machine language instruction for each symbolic machine instruction. The assembler must act upon the information given in each of the directives. The assembler has two major tasks. First it must process symbolic machine instructions. In so doing, it must translate a symbolic operation code into a numeric operation code. In addition, it must also find the value of the operand field in order to determine the numeric value of the rightmost bits of the machine language instruction. Any label preceding the symbolic machine language instruction will have to be defined. The second major task of the assembler is to process the directives. These directives to the assembler define the values of symbols which do not appear as labels, define data, reserve space, set the location counter, segregate programs and link programs.

4.4 Identification of Tasks, Gross Structure, and Interfaces

In this section the assembly process will be discussed in more detail. Two things must be done in order to generate a

machine language instruction. First the symbolic operation code must be converted into a numeric operation code. Second the operand field must be evaluated. The conversion of symbolic operation codes to numeric codes can be accomplished in a straightforward manner by table lookup. The evaluation of the operand field is considerably more difficult.

The operand field may contain a symbol, a literal, a constant, or an expression. An expression is defined as a symbol or constant added to or subtracted from an integer number. In evaluating an expression, the address (location) of a symbol is used rather than the contents. For example, if the address of A is 42, then the value of A + 1 is 43 and not 1 plus the contents of memory location 42. The same is true for literals. The address where the literal is stored is used instead of the value of the literal. Thus, in order to evaluate the operand field, the values (locations) for all the symbols and literals which appear in operand fields are needed. This indicates that the values of symbols must be defined.

Values can be defined in two ways. If a symbol appears as a label in a machine instruction, its value is defined to be the relative location of that instruction. The EQU directive also defines the value of a symbol. A symbol defined in this way has a value equal to the value of the expression in the operand field of the EQU directive. To assign the relative

location of an instruction or datum to be the value for a label, a relative location is assigned to each instruction or datum. Instructions and data are assigned locations on the drum sequentially in the order in which they appear in the source language program.

An additional task arises since a symbol need not be defined before it is used. That is, an instruction such as a transfer may use in its operand field a symbol which does not appear as a label until later in the source program. The simplest solution to this problem is to make two passes over the source program. One complete pass is made for the purpose of defining all of the symbols which occur in the program. Each symbol and its value are stored in a table called the symbol table.

Additional tasks are performed on the first pass. It is advantageous to eliminate any duplicate literals which may appear in the program. During the first pass all literals are collected into a single table. Duplication of entries in the literal table is not allowed. Literals are assigned locations on the drum following the body, or text, of the program. Thus, their location can be assigned only after the length of the text is known, that is, only after all instructions and data have been assigned locations.

A further task to be performed is the processing of the set location counter (LOC) directive, the program end (END) directive and the define symbol (DEF) directive.

In processing the set location counter (LOC) directive, the location counter is set equal to the value in the operand field.

The program end (END) directive terminates an existing program and establishes a new program where the symbols have a different value from any that may have existed before.

The define (DEF) directive is used to establish linkages between programs. The value of the symbol in the operand field is made available to other programs.

A final task is the recognition of a special symbol (a dollar sign - $) indicating the end of the source program. Recognition of the special symbol terminates Pass I and results in a call on Interpass and then Pass II

Hence, the assembler is divided into two main routines, each of which passes over the source program once. In addition, some tasks which are required at the end of the first pass are incorporated into a third routine which is called Interpass processing. These three routines perform the following tasks.

A. First Pass: Pass I

1. Define the Symbols. Locations are assigned to all the instructions and data. EQU, DEF, LOC, and END directives are processed. Data and storage definitions specified by DATA and RES directives are processed. Symbols and their addresses are entered

into the symbol table.

2. Literals. All literals are entered into the literal table with no duplicate entries.

B. Interpass.

In the interpass, locations are assigned to all literals in the literal table.

C. Second Pass.

Everything else is done in Pass II. This consists principally of generating the instructions and consequently evaluating the operand field. In addition, the constants appearing in the operand field of DATA directives are converted into machine language form. END and LOC directives are processed again.

The interfaces between the three major routines are given in the following table.

4.5 Pass I: Symbol Definition

The major task of Pass I is to define all of the symbols used in the program. Symbols are defined by their appearance in the label field of a machine instruction, data (DATA) directive, reserve storage (RES) directive, or an equivalence (EQU) directive. Whenever definition information is encountered

TABLE 4-1

INTERFACES BETWEEN THE PASS I, INTERPASS, AND PASS II MODULES

ROUTINE	INPUT	OUTPUT
Pass I	Source program	Symbol table Literal table Segment length Source program
Interpass	Literal table Segment length	Literal table
Pass II	Symbol table Source program Literal table	Object program Listing

it is entered into the symbol table. Each entry contains the symbol and its value. To define the value of a symbol appearing in the label field of a machine instruction, DATA directive, or RES directive, the assembler must assign drum locations to each machine instruction, to the storage reserved by a RES directive and to the data words reserved by a DATA directive.

Storage locations are assigned sequentially. In order to keep track of the location to be assigned to the instructions and data, the assembler needs the length of the storage locations used by each DATA or RES directive. The next drum location available for assignment is remembered in a location counter (LOC). The value of LC is the displacement, relative to the base of the segment, of the first unassigned word of drum storage. The location counter is incremented for each instruction.

To process a RES or DATA directive the assembler must determine the number of words reserved by these directives. The location counter is then incremented by the value of this number. When a label is encountered on a machine instruction, DATA directive or RES directive, the label is entered into the symbol table along with the value of the location counter. The definition of a symbol appearing in the label field of an EQU directive is somewhat more complicated. The expression in the operand field is evaluated and this value is entered into the symbol

table along with the symbol. In order to evaluate the expression in the operand field all symbols used in it must have been previously defined, i.e., the symbol and its value must appear in the symbol table.

An additional task of Pass I is to collect all of the literals and eliminate any duplications. In order to do this, each operand field must be scanned for the occurrence of literals. If a literal is found it is converted into its binary representation and entered into the literal table along with its length. If a duplicate entry already appears in the literal table, no additional entry is made.

Two additional tasks are required in order for Pass I to carry out its major functions. Before any processing of an input card can be achieved, the various elements in a machine instruction or directive must be isolated. The elements of a machine instruction are the label, the operation code, and the operand field. The location and isolation of these three elements of an instruction is called parsing. The final task that must be performed is that of distinguishing machine instructions from directives. This is achieved by putting a flag in the operation table which distinguishes between the two cases. All machine instructions and directives are stored in the operation table. This table gives the correspondence between the symbolic operation code and the numeric operation code as well as flags

distinguishing machine instructions from directives.

An additional task performed by Pass I is the processing of the set location counter (LOC) directive, the end of program (END) directive, and the define symbol (DEF) directive.

In processing the set location counter (LOC) directive, the location counter is set equal to the value in the operand field. The operand field contains an integer or octal constant. This directive allows the programmer to set the value of the location counter.

The end of program (END) directive isolates programs from one another. The END directive terminates an existing program and establishes a new program where the symbols have a quite distinct value from any that may have existed before. The isolation of one program from another is accomplished by attaching a qualifier (called the program number - PRGNUM) to each symbol as it is entered into the symbol table. The qualifier (i.e., PRGNUM) is an integer number whose value is the same for all the symbols of a given program. The value of the qualifier is incremented by one by the occurrance of an END directive. This task is performed again in Pass II.

The define symbol (DEF) directive is used to provide linkage between programs. If a symbol occurs in the operand field of a DEF directive, the value of the symbol is made available to other programs that are assembled along with the

program in which the DEF directive appears. To make use of the value of such a symbol, other programs must have the particular symbol appearing in the operand field of a machine instruction, DATA, or EQU directive. The particular symbol must not also be defined in the same program; i.e., the symbol must not appear in the label field of a machine instruction, DATA directive, or EQU directive in the program referencing it. The symbol appearing in the operand field of the define symbol (DEF) directive is made available to other programs by being entered into the defined symbol table.

The source program is needed again in Pass II, therefore a copy is made of the source program in Pass I for use in Pass II.

Figure 4.1 is the flow chart for Pass I. On the left side of the chart is the main loop for processing a machine instruction. It begins by reading the next line of the source program. The input line is parsed and the operation type is identified. If there is a label, it is entered into the symbol table. Any literals appearing in the operand field of the instruction are processed. The instruction counter is updated and the original source line is copied into an internal file for use in Pass II. On the right side of the flow chart is the processing of the directives (EQU, RES, DATA, LOC, END, and DEF). The two circles γ, α and β will be referred to later in this chapter when macros are discussed.

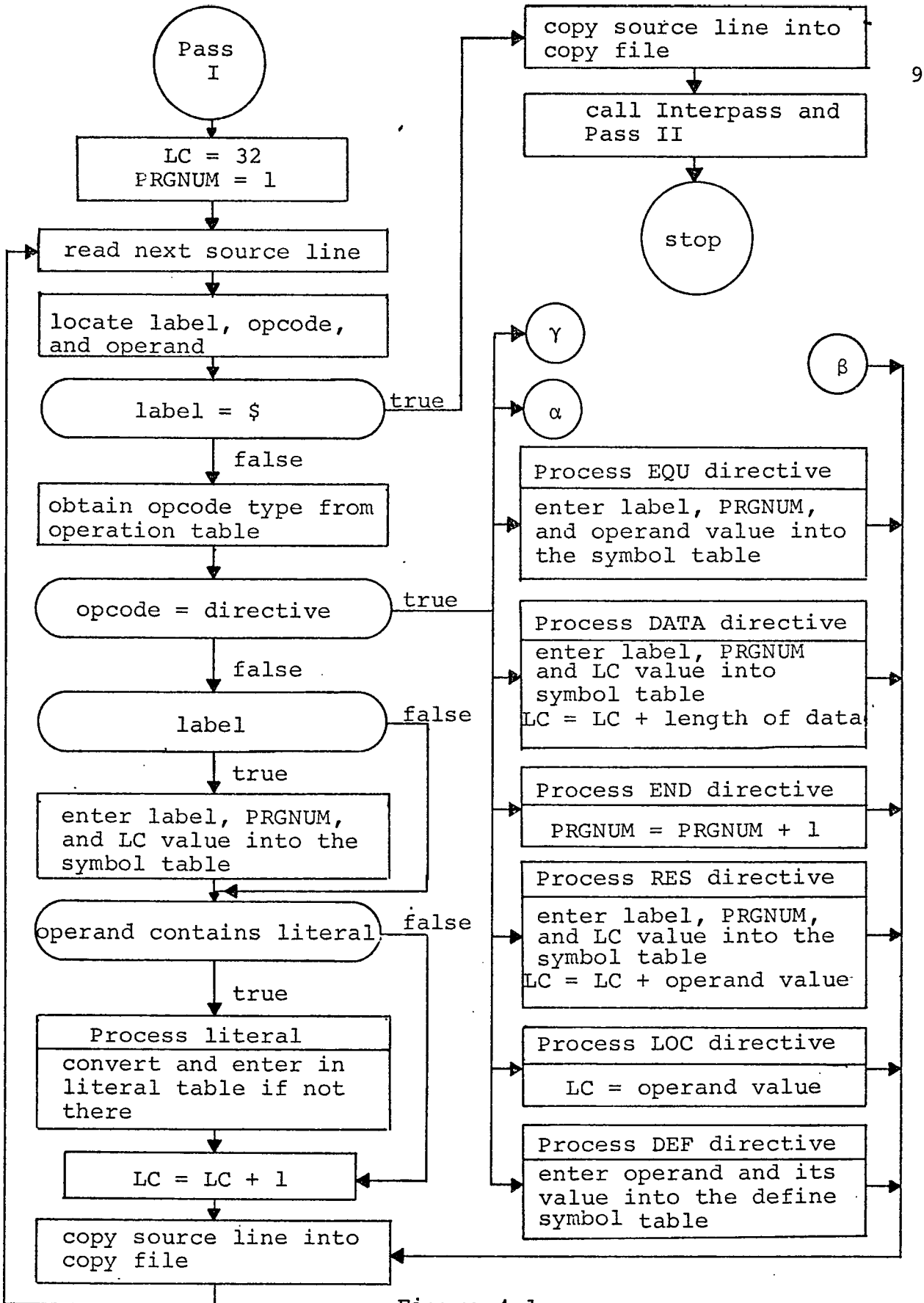


Figure 4.1

PASS I

4.6 Interpass

At the end of Pass I, the literal table contains all of the literals which appear in the program and there are no duplicate entries in the table. Each literal must be assigned a drum location. Literals are assigned locations beginning with a displacement equal to the current segment length which is in fact the value of the location counter at the end of Pass I. It is the number of words occupied by the instructions, data, and reserved words generated from the machine instructions, DATA, and RES directives in the source program. As locations are assigned to the literals, the relative displacement is added to the corresponding entry in the literal table. These literal addresses are needed in Pass II in order to evaluate operand fields.

The data base for Interpass consists of the literal table and the current segment length which is the input data. The output from Interpass is the literal table.

4.7 Pass II: Instruction Generation

The principal task of Pass II is to generate the numeric machine instructions. In order to do this the following functions must be performed. A symbolic operation code must be converted to a numeric operation code. This is achieved by finding the symbolic operation code in the operation table. The associated table entry contains the numeric operation code.

The operand field must be evaluated. The instruction can then be assembled. To assemble the instruction its format must be known. Again this information is found in the operation table. Knowing the format, the numeric operation code and the value of the argument field can be combined into a numeric machine instruction.

The generation of the data items defined by a DATA directive is similar to the generation of numeric machine instructions. The argument in the operand field of a DATA directive is converted into their binary form and assembled.

Reserve (RES) directives are processed again in the same manner as in Pass I. The operand field is evaluated and the location counter incremented by the value of the integer in the operand field.

The set location counter (LOC) directive is evaluated again. The integer or octal constant in the operand field is evaluated. The location counter is set equal to the value in the operand field. The end of program (END) directive is processed again in the same manner as in Pass I. In addition, Pass II generates an object tape and output listing. To generate the object tape each machine instruction must be punched onto the paper tape in a format the Athena computer can handle. In addition, the data generated from DATA directives is punched onto the paper tape. Drum words reserved by RES directives are

punched onto the paper tape with zeroes in them. In order to determine the appropriate drum location for each instruction and datum, Pass II keeps a location counter in exactly the same fashion as the location counter kept by Pass I. The output listing includes the original symbolic source instruction, the generated numeric instruction and data printed in octal and any error information which has been collected during the assembly process. This information consists of flags indicating such errors as undefined symbols and unknown operation codes.

The data base for Pass II consists of the following.

Input	Internal	Output
Source program	Operation table	Paper tape
Symbol table	Location counter	Output listing
Literal table	Partially assembled instructions	
	Error flags	

Figure 4.2 is the flow chart for Pass II. On the left side of the flow chart is the main loop for processing a machine operation. The sequence is very similar to Pass I. The source line is read, the type of instruction identified, the symbolic operation code looked up and the numeric operation obtained, the argument field is processed, the assembled instruction is sent to the tape punch file, the instruction, location, and the symbolic source line is printed out, and finally, the

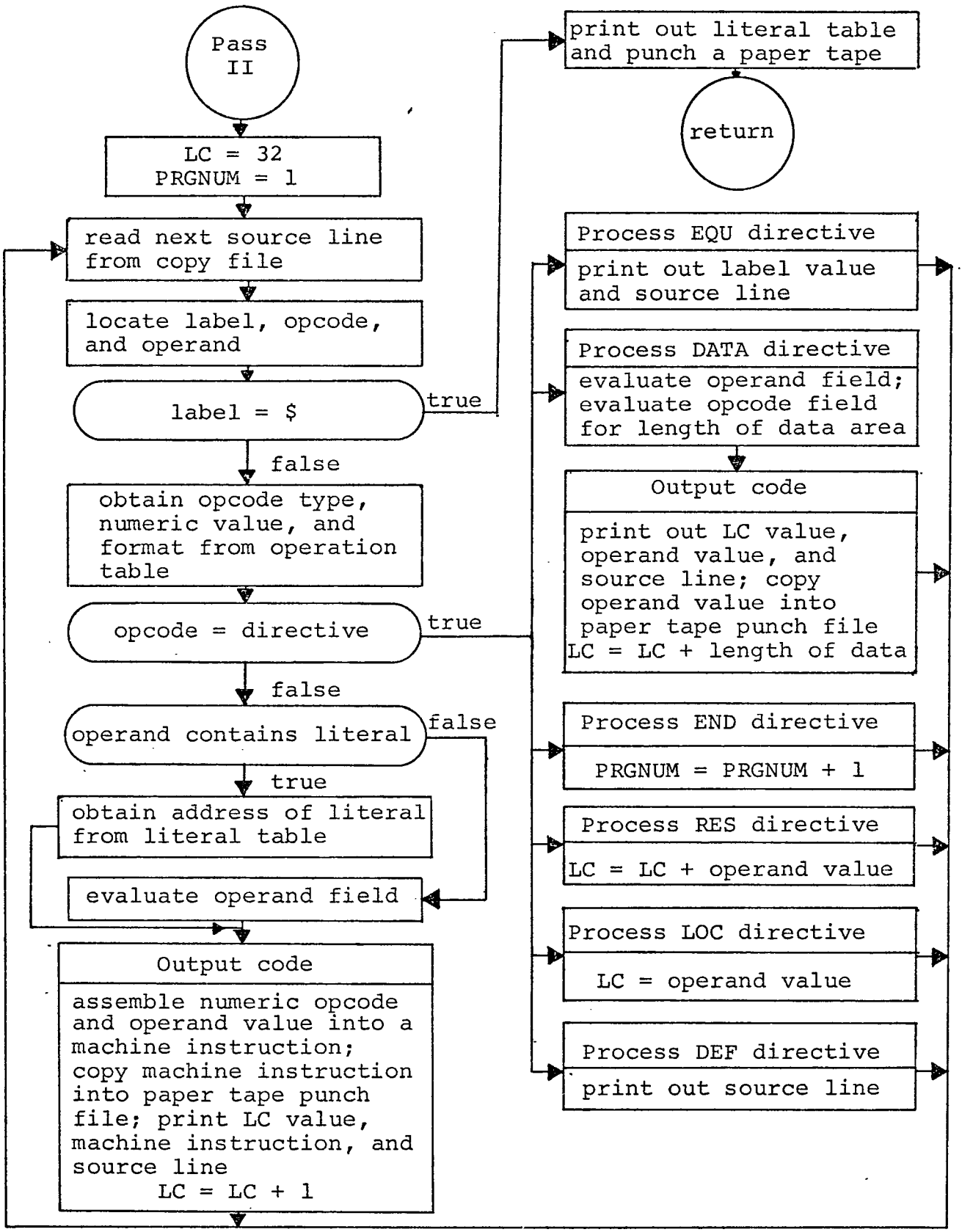


Figure 4.2

PASS II

location counter is updated by the appropriate amount.

The right side of the flow chart shows the processing for the EQU, DATA, RES, END, and LOC directives. The EQU directive involves no processing in Pass II since its only function was to define the value of a symbol in Pass I. Similarly, the DEF directive involves no processing. The DEF directive completed its function in Pass I by defining a symbol as being available to other programs.

4.8 Details of the Assembly Process

The following discussions consist of comments on the particular steps involved in the assembly process.

4.8.1 Locate Label, Operation Code, and Argument

This subroutine scans the source line and locates the label, operation code, and operand. The algorithm for isolating the fields of an instruction is straightforward. The fields are separated by one or more blanks. Starting with the first character, each character in turn is examined to determine if it is a blank. If it is not a blank it is counted and stored. A blank indicates the termination of the field. All following blanks, up to the next non-blank are skipped. Storing and counting them starts on the next field. If the first character of the source line is a blank, then there is no label.

4.8.2 Process Literal

The input to this function is the content of the operand field, and the its character count. The function processes the string of characters starting with the first character, until it has a complete literal. The function converts the string of characters representing the constant into a binary value. This value is returned as an argument. The literal table is probed to determine whether or not the table already contains the constant value. If not already there, the constant value and drum storage requirement count are stored in the literal table.

4.8.3 Process Data Directive

The input to this routine is the content of the label field, operation code field, and operand field, and the count on the number of characters in these fields.

On Pass I, the label is checked for a legal symbol. The label with PRGNUM as a qualifier is entered into symbol along with the value of the location counter. On Pass I and Pass II, the operation code field character string is scanned for an integer number. This integer specifies the number of data words received by the data directive. The integer, if there, is converted from its character representation to a value. On Pass I, the location counter is then incremented by the integer value. If an integer is not found, one data

word is reserved for the data directive. The location counter is increment by one.

On Pass I, only the operand field is evaluated for a literal. If a literal is found, it is converted to a value and entered into the literal table if not already there.

On Pass II, the operand field is evaluated and the machine code is generated. A check is made of the operand field for each of the following: a literal, an expression, a symbol, a decimal integer, an octal constant, a floating-point constant, a character string constant, and a dollar sign. If one of the preceding arguments is found, it is evaluated. The value of the operand field is assembled into the data words reserved by this data directive. A call is made to the output routine. The location counter value, the assembled data, the count of data words, and the source line are given to the output routine as input. The output routine copies the data words into the paper tape punch file, then prints out the location counter, the data words, and the source line. The location counter is incremented by the count on the number of data words reserved by the data directive.

4.8.4 Process Reserve Directive

The input to this routine is the content of the label field and operand field, and the count on the number of characters in these fields.

On Pass I, the label field is checked for a legal symbol. The label with PRGNUM as a qualifier is entered into the symbol table along with the value of the location counter.

On Pass I and Pass II, the operand field is scanned for a decimal integer. This integer specifies the number of drum words being reserved by the reserve directive. The integer is converted from its character representation to an integer value. The location counter is incremented by the value of the integer.

4.8.5 Output Code

The input to this routine is the value of the location counter, the numeric value of the operation code, the format for the operation code, the value of the operand field, and the source line. The output routine combines the numeric operation code with the operand value into a machine instruction with the aid of the format for the particular operation code. The format is a mask. The operand is first masked off and then added to the numeric operation code. The masking of the operand eliminates any higher order bits in the operand which might overlap into the numeric operation code part of the machine instruction. An error message is printed if truncation of the operand results. The machine instruction is copied into the paper tape punch file. The location counter value, machine code, and source line are pointed out. The location counter is incremented.

4.8.6 Evaluate Operand Field

Input to this routine is the content of the operand field. The content of the operand field is scanned. Evaluation is stopped when all the characters in the field have been scanned. If nothing is found in the operand field, the value returned is zero. An expression appearing in the operand field may be a combination of a symbol or a decimal integer or an octal constant or dollar sign separated by a +, or - sign from a decimal integer. An expression may also consist simply of a + or - sign before a decimal integer or octal constant. The routine computes the value of such an expression using the value of the symbol which is found in the symbol table (which is usually an address) and the value of any constant in the expression combined according to the arithmetic operation in the expression.

If only a symbol, decimal integer, octal constant, or dollar sign appears in the operand field, it is evaluated and a value is returned. The value of a symbol is obtained from the symbol table. Decimal integers and octal constants are evaluated directly and their value returned. The value returned for a dollar sign ($) is the current value of the location counter.

4.9 Table Maintenance and Data Structures

The assembler has four tables, the operation table, the symbol table, the literal table, and the define (program linking)

table. In this section the techniques used in constructing these tables will be discussed. First, similarities will be examined and some general properties of the tables and table maintenance will be discussed.

Each of the tables is composed of a number of entries which are related to each other and associated with a key. For example, an operation table entry consists of a symbolic operation code which is the key, a directive flag, a numeric operation code, and an instruction format. A symbol table entry consists of a symbol which is the key, its value, and a multiple definition flag. A literal table entry consists of a literal which is the key, its location, and its length. A linkage table (defined symbol) entry consists of a symbol which is the key, and its value. A table is a collection of such entries. The table can be visualized as consisting of a number of rows, one for each entry. The elements of an entry are the columns of a matrix. A table is not really a matrix because, while in a matrix all elements are homogeneous, an entry in a table may consist of items of different types, e.g., binary integers and character strings. The term data structure implies a collection of related elements of possibly different data types. The relations which exist between the elements in a structure vary from structure to structure. The table is an example of data structure with a very simple relationship between its elements.

Several functions (operations) are needed in order to manage a table. The following are the most basic and useful functions. Examples of its use in the assembler are given for each function.

1. Copy information from an entry: e.g., check directive flag, get value of symbol.
2. Add a new entry: e.g., process (enter) label, process new literal.
3. Modify existing entry: e.g., insert location in literal table, change multi-definition flag in symbol table.
4. Delete existing entry: this function is not used in the assembler.
5. Locate an entry: this is needed for all of the previous functions, even in (2) if avoidance of duplication is desired.

In managing a table, two aspects must be distinguished. The relationship between items of an entry and the representation of the table in memory must be distinguished. The relationship between items determines the way the items can be accessed, e.g., the entries can be referenced sequentially by a numerical index or they can be referenced by specifying the key. Representation in memory also determines the number of words each item occupies and the mapping of the items into

memory, e.g., sequential items may not be located sequentially in memory.

One of the more important functions is that of locating an entry. In order to locate the entry corresponding to a given key in the table, the table must be searched, i.e., the key in each entry of the table must be examined in some sequence until the desired entry is found. There are two requirements that must be met in order to do this:

1. It must be possible to compare the key being sought with the key part of an entry and detect when they match.
2. At some point in the search it must be possible to determine that no match will ever occur, i.e., the entry being sought is not in the table.

To illustrate these points, two different algorithms for locating an entry in a table will be looked at.

The first algorithm is a linear search. In this example the table is composed on N entries which are unordered. All the entries in the table are always stored in the first N places in the table, t_1 - - - t_N. A place is conceptually a row in the table viewed as a matrix. At any given time not all places necessarily contain entries. The search algorithm is straightforward. Each of the entries, t_1, t_2, - is examined in turn until the desired entry is found, or until t_N is reached. Then if t_N does not match, the desired entry is not

in the table. This method has the advantage of being easy to implement and is used in the defined symbol (program linking) table. This table will generally have few entries and low usage.

The second algorithm is a random probe. This method has the advantage of increased efficiency at the cost of a more complicated search algorithm. The random probe method is used in implementing the symbol table, operation table, and literal table. The entries in these tables are stored in the first N places of a table, t_1, t_2 - - - t_N. At any given time not all places necessarily contain entries. Access to the table is through another table called a scatter index table. The scatter index table contains pointers to the entries in the table. The pointers are scattered through the scatter index table in a nonorder fashion. The size of the scatter index table is always fixed at, say, M places. The algorithm for searching for an entry, t_p, in the table uses a randomizing (often called hashing) function which is applied to the key to calculate the index to the place in the scatter index table where the pointer to entry t_p is located. This randomying function, R, is a function from the domain of keys to a range consisting of the integers from 1 to M, i.e.,

$$R(\text{Key}) \rightarrow [1, M]$$

This mapping is usually many to one, i.e., several different

keys may result in the same index to the scatter index table upon application of the randomizing function. Keys having the same index are chained together in the table.

Let R(key) → i be the calculated index to the scatter index table. The search algorithm is then:

1. If location i of the scatter index table is empty, the key is not in the table.
2. If location i of the scatter index table is not empty, get the pointer p in location i and go to place t_p in the table. If t_p is equal to the key, the search is ended. If t_p is not equal to the key, check to see if another entry is chained to t_p. Search down the chain until the key is found. If the key is not found on the chain, the key is not in the table.

The randomizing function used in the symbol table, operation table, and literal table is:

i = [sum of characters in key] mod M + 1

The hollerith representations for the characters in the key name are summed and then modulo arithmetic is performed of the result. The value of i is always 1 to M.

4.10 Macros and Macro Processing

In this section the macro facility of the assembler will be discussed. An overview of macro processing will be given.

Then macro definitions and macro expansions will be discussed.

The simplest form of a macro is an abbreviation for a sequence of symbolic assembly language instructions. For example,

```
SUM ≡ { RW    AA
        SA    100
        RW    BB
        AD    100
        SA    100
        RW    CC
        AD    100
        WW    DD
```

This sequence of instructions sums the variables AA, BB, and CC, then stores the result in DD. To use this macro, SUM is written in the program in the operation field of an instruction. The assembler will substitute for this single instruction the eight instructions written in the right-hand column. An instruction in which SUM appears in the operation field is called a macro instruction (or macro call).

The assembler's macro facilities are much more extensive than this simple abbreviation capability. A macro instruction in its full generality is the invocation of some rule for generating a sequence of assembly language instructions.

The sequence of instructions may be different each time the macro instruction is used. To achieve this generality, the macro instruction includes arguments and the corresponding macro definition includes dummy variables which are substituted for by the arguments when the macro instruction is processed. For example, the definition of SUM might be:

```
SUM ARG1,ARG2,ARG3,ARG4  =  RW    ARG1
                            SA    100
                            RW    ARG2
                            AD    100
                            SA    100
                            RW    ARG3
                            AD    100
                            WW    ARG4
```

When the macro instruction SUM is used, four variables are written in the operand field. These are the call arguments which are substituted for dummy variables ARG1, ARG2, ARG3, and ARG4 in the definition. For example, the macro call,

```
SUM     X,Y,Z,W
```

will be replaced by the four instructions

```
RW      X
SA      100
RW      Y
AD      100
SA      100
RW      Z
AD      100
WW      W
```

4.11 Overview of Macro Processing

In order to do macro processing, the assembler must recognize two types of constructs: a macro definition and a macro call. To process the definition of a macro, the assembler needs to store the name of the macro and the corresponding definition for use later whenever a call for that macro appears. The macro definition begins with a macro definition directive. The action taken in processing a macro definition consists of adding the name of the macro to the operation table and adding the definition of the macro to the macro definition table.

In a macro call the macro name is written in the operation field. This is recognized as a macro call by finding the name in the operation table where it was placed when the definition was processed. The major task in processing a macro call is to make a copy of the definition, substituting the arguments in the call for the dummy variables in the

definition. This copy is inserted into the source program in place of the macro call. The instructions in the copy must also be processed by the assembler in the same way as all the ordinary instructions since the inserted instructions may contain literals which need to be inserted in the literal table and directives, such as EQU, which define symbols.

4.12 Macro Definitions

The format for a macro definition is as follows:

```
name    MACRO    P1, - - - PN
        - - -
        - - -
        END
```

The definition begins with the MACRO directive which gives the name of the macro and the dummy variables. This is followed by a sequence of instructions which is terminated by an END directive. The sequence of instructions defines the macro. The END directive when used on a macro definition simply indicates the finish of the macro definition.

Figure 4.3 shows the processing of the macro definition that takes place when the MACRO directive is encountered. This flow chart is a patch which is to be inserted between points α and β in Figure 4.1. The first action is to insert an entry in the operation table. The entry added to the operation table

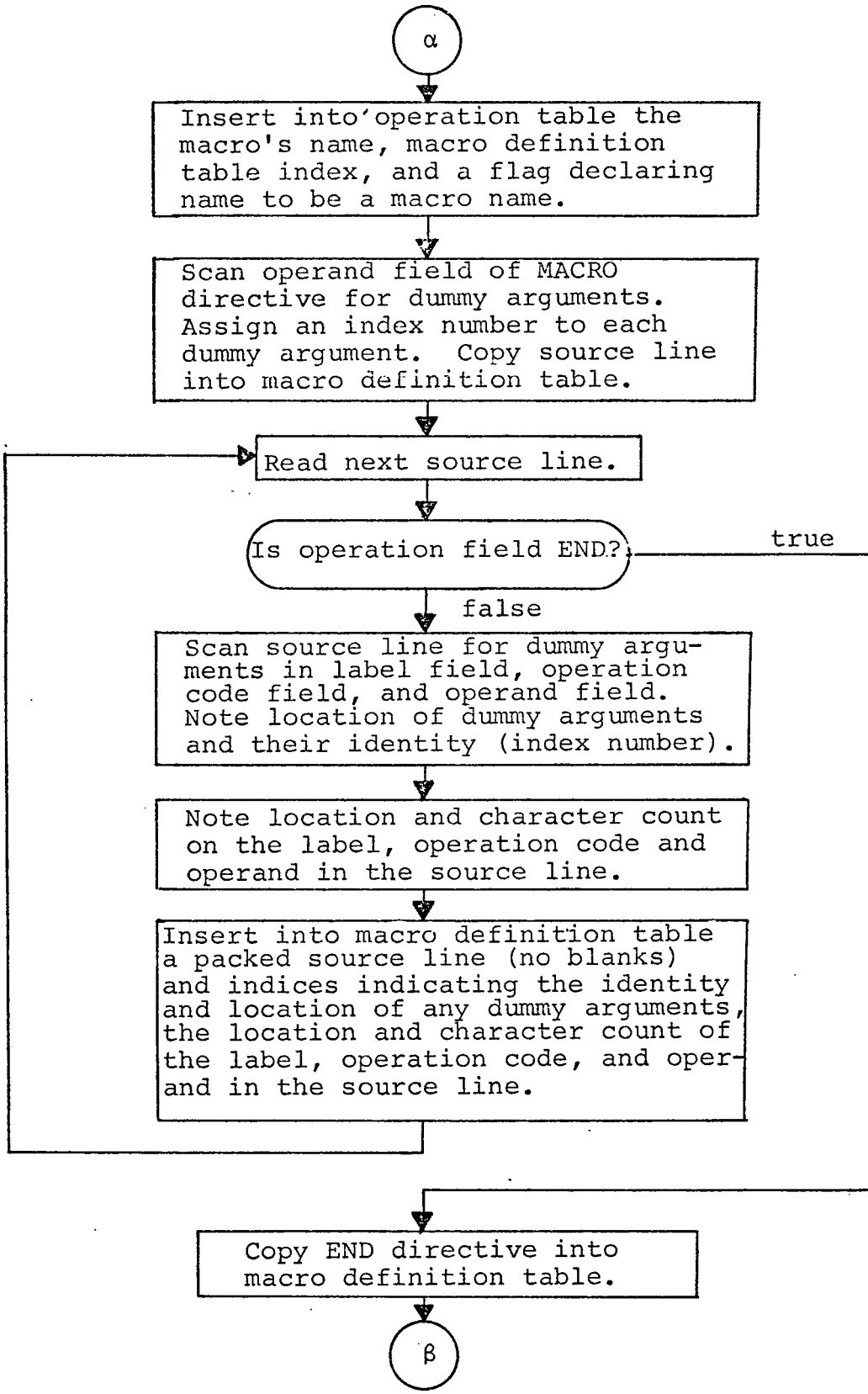


Figure 4.3

PROCESSING OF MACRO DEFINITIONS

contains the name of the macro operation. This corresponds to the symbolic name of the instruction for machine instructions. The flag field in the operation table is set to indicate that the operation code is a macro name. Also an index is stored in the operation table of where in the macro definition table the macro definition can be found.

The next step is to scan the operand field of the MACRO directive for dummy arguments. Each dummy argument is assigned an index number identifying its place in the operand field. This index number will be used later to identify the dummy arguments appearing in the instruction lines which define the macro.

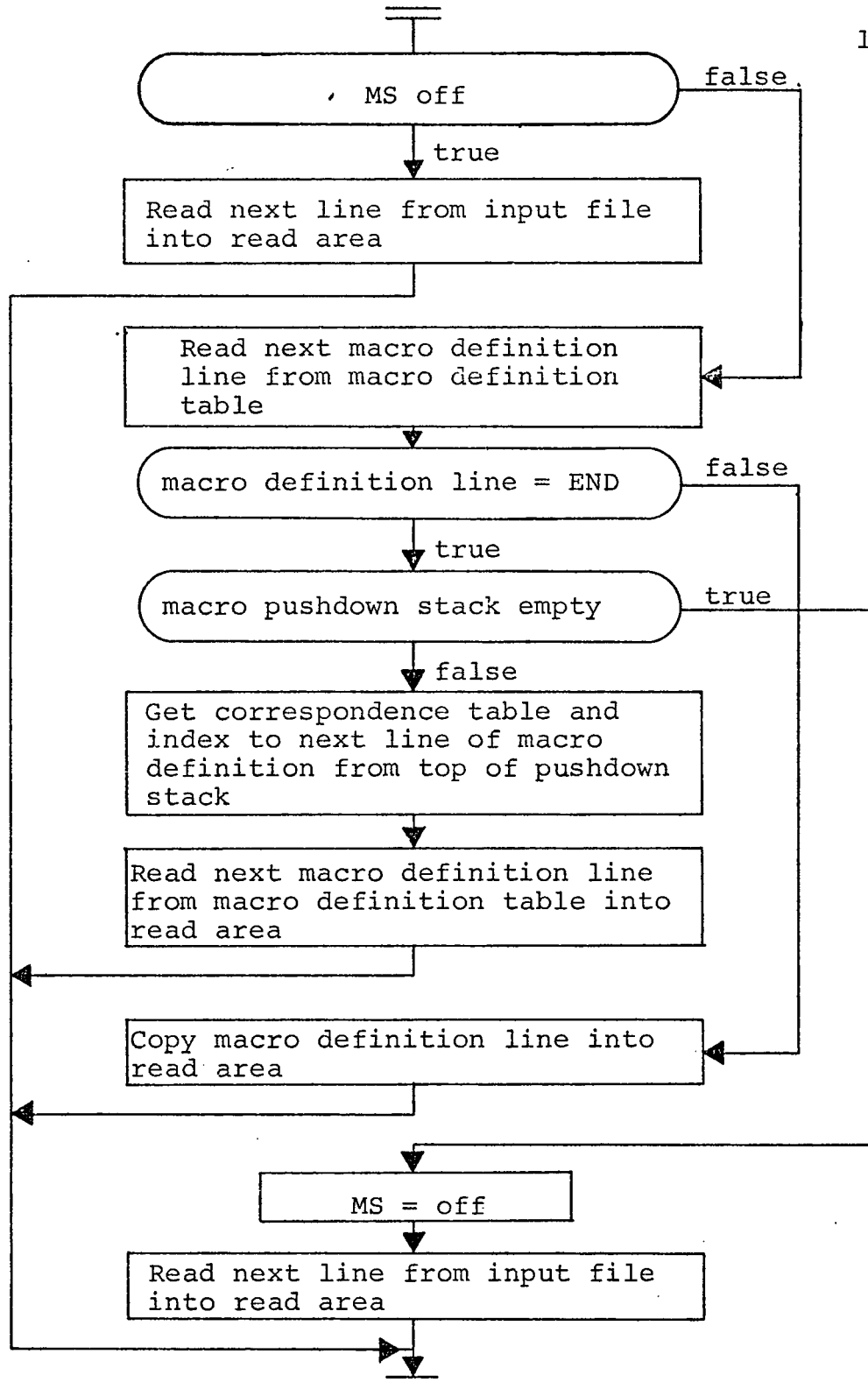
Next in processing the definition is a loop which copies the instructions in the definition into the macro definition table. Before insertion, the source line is scanned for dummy arguments in the label field, operation code field, and operand field. The location of the dummy arguments and their identity (index number) is noted. Two more are set which indicate the positions of the operation code and operand in the source line. The source line is packed (no blanks except in literals) and inserted into the macro definition table. Also inserted with the packed source line are the indexes indicating the identity and location of any dummy arguments, indexes to the location of the operation code, and operand in the source line prior to

packing. The complete macro definition is copied into the macro definition table including the END directive. The copying terminates after the END directive has copied into the macro definition table.

4.13 Macro Expansion

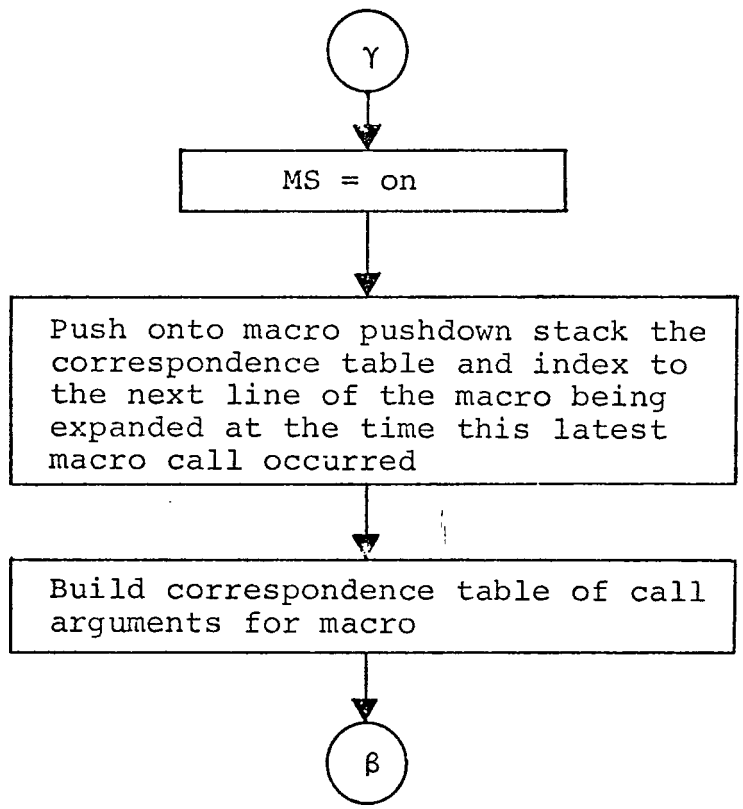
When a macro operation is recognized in a macro call, two tasks have to be performed; the assembler must be made to process each instruction in the definition in the usual way. The call arguments must be substituted for the dummy variables before this processing takes place. The macro processor tricks the assembler and makes it think the lines of the expanded definition are coming from the input file. This is accomplished by moving each line from the macro definition table to the input read area. Figure 4.4 shows the expansion of the read-next-source-line box in Figure 4.1.

Figure 4.5 shows the processing which takes place when a macro call is recognized. Figure 4.5 is a patch to be made between γ and β in Figure 4.1. A switch, MS, is turned on (if not already on) to indicate that the assembler is expanding a macro definition. Since the macro processor allows macro calls to be used in defining other macro, a macro call may occur during a macro expansion. To handle this situation, the current macro expansion is halted, and the correspondence table and index to the next line of the current macro are pushed onto a pushdown stack.



EXPANSION OF "READ NEXT LINE" BOX

Fig. 4.4



PROCESSING A MACRO OPERATION

Fig. 4.5

From the operation table the index to the macro definition of the new macro is obtained. A correspondence table is built. This table contains the call arguments in the operand field of the new macro call. The call arguments are placed in the correspondence table in the sequence in which they occur in the operand field of the macro call. The place of the call arguments in the correspondence table correspond with the identification numbers of the dummy variables they are to be substituted for.

The macro switch, MS, indicates to the assembler that the next source line is to come from the macro definition table rather than being read from the input file. This is seen in Figure 4.4. Each time the read routine is called, the macro switch is interrogated. If the switch is on the next line of the definition is copied from the macro definition table into the read area. As it is being copied, call arguments are substituted for any dummy variables which appear in the line. The proper substitution is made by examining the indexes placed in the macro definition table with the source line. The indexes indicate the identification number of the dummy argument and where they are located (label field, operation code field, or operand field). When a call argument is to be substituted for a dummy argument in the source line, the identification number of the dummy argument is used to obtain the proper call

argument from the correspondence table. The source line is also unpacked and re-assembled to its original form. When the END directive is encountered the definition has been completely copied. The next step is to check the macro pushdown stack for the presence of the correspondence table and index to the next line of a previous macro call. If the macro pushdown stack contains a correspondence table and a macro table index, macro expansion continues with the macro from the top of the macro pushdown stack. When the macro expansion has been completed and the macro pushdown stack has been emptied, the macro switch is turned off. Succeeding lines will come from the input file.

4.13 Executing the Source Program Across Group Boundaries on the Athena

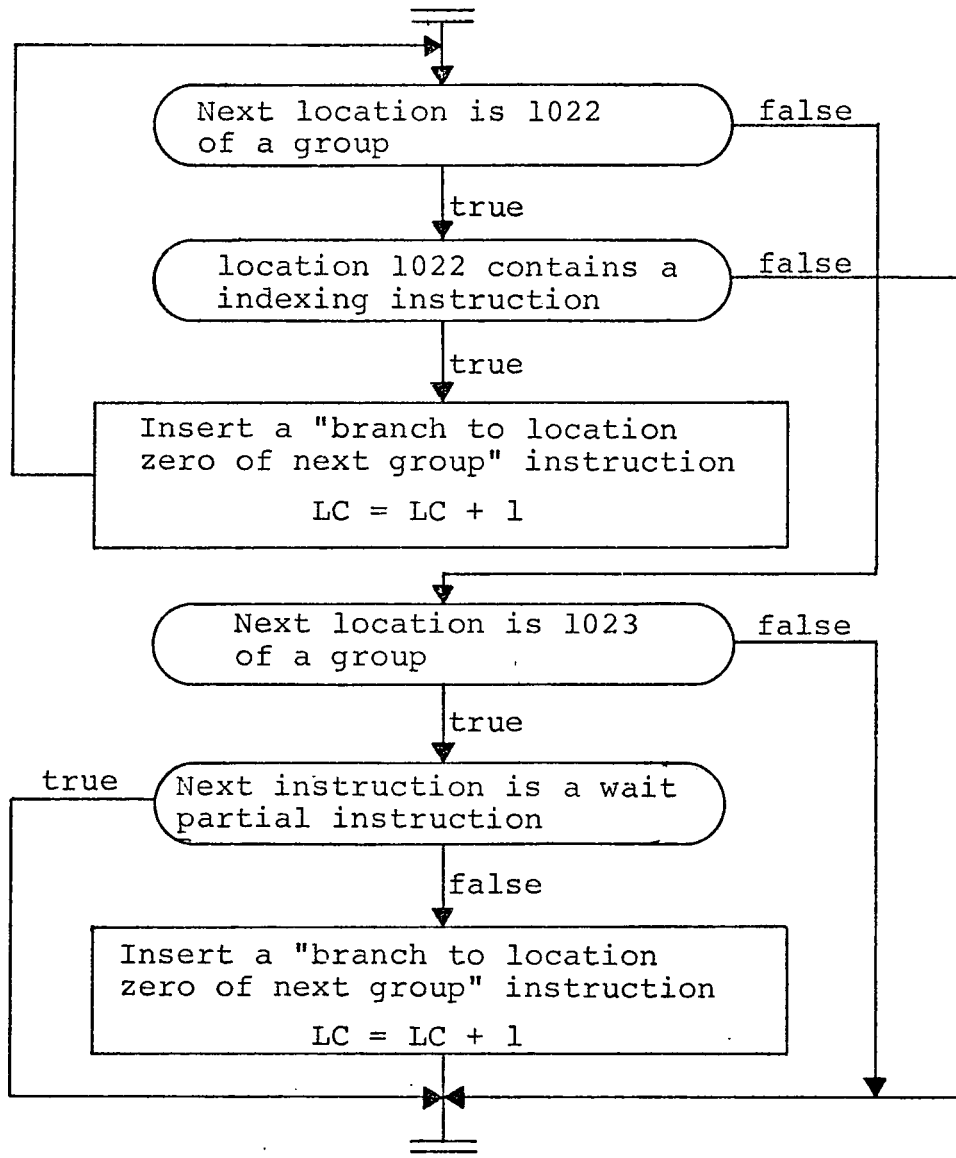
Provisions must be made on the Athena computer to increment the drum address portion of the program address register when source program execution reaches a drum group boundary. When program execution arrives at the last word of say group 3, execution does not continue with the first word (location zero) of zero 4, but instead, goes back to location zero of group 3. To cause execution to continue with location zero of group 4, a branch instruction (opcode 200000_8) to the first word of the next group (in this case location zero of group 4) has to be inserted in the source program at location 1023.

There are three variations to this problem. First, if a branch instruction is inserted at location 1023 of a particular group, the branch instruction cannot be preceded by an indexing instruction in location 1022. The solution is to move the indexing instruction so that it will follow the branch instruction. This is accomplished by inserting another branch instruction at location 1022 and move the indexing instruction to the first location (location zero) of the next group.

A second variation is the presence of a wait partial instruction at location 1023 already. No insertion of a branch instruction is required in this case. The source program is not changed.

A third variation involves storage locations such as data words and reserved locations. No branch instruction is inserted here.

Pass I is modified as shown in Figure 4.6 to test for a drum boundary and make the proper branch instruction insertion if required. This patch in the flow graph of Pass I is inserted immediately preceding the start of the processing of machine instructions (preceding the test for a label on the left side of the flow graph in Figure 4.1).



INSERTING BRANCH INSTRUCTIONS AT GROUP BOUNDARIES

Fig. 4.6

BIBLIOGRAPHY

1. USAF Technical Manual T.O. 21-SM68-2F-6-0, SM 68, Missile Weapon System, Radio Inertial Guidance System Computer (Functional Manual, Computer and Peripheral Equipment), 1963.

2. Baca, B. and B. Sibley. A Machine, published by Ron Baca and Bob Sibley, University of Houston, Houston, Texas, 1969.

3. Graham, R. M., Programming Systems, unpublished book, 1970.

4. Morris, R., "Scatter Storage Techniques," Communications of the ACM, Vol. 11, No. 1, January, 1968, pp. 38-44.

APPENDIX A

PROGRAM DOCUMENTATION FOR THE ATHENA ASSEMBLER

APPENDIX A

PROGRAM DOCUMENTATION FOR THE ATHENA ASSEMBLER

A.1 General Program Description

The Athena assembler program is written for the Univac 1108, Exec. 8, system. The language used throughout the program is FORTRAN V.

The assembler program has two main programs. Program MAIN performs the function of Pass I as described in Chapter IV. Subroutine Pass II performs the functions of Pass II also described in Chapter IV. The functions of Interpass are performed by subroutine INTPAS.

The processing of source lines is performed by a series of functions, and subroutines, each corresponding to one of the syntactical definitions for the Athena assembly language as described in Chapter IV. Functions and subroutines which have a corresponding syntactical definition are the following.

ARG - processes the syntactical definition for a "constant"

CHAR - processes the syntactical definition for a "character string constant"

DATA - processes the syntactical definition for the "data directive"

EQU - processes the syntactical definition for an "equivalance directive"

EXPR - processes the syntactical definition for an "expression"

FLOATI - processes the syntactical definition for a "floating-point constant"

INTEGR - processes the syntactical definition for a "decimal integer constant"

LABL - processes the syntactical definition for a "label"

LITRAL - processes the syntactical definition for a "literal"

MACDEF - processes the syntactical definition for a "macro definition directive" and enters the macro definition into the macro definition table

MACOPC - processes the syntactical definition for a "macro call"

OCTAL - processes the syntactical definition for an "octal constant"

OPRND - processes the syntactical definition for a "machine operand"

RES - processes the syntactical definition for a "reserve directive"

SYMBOL - processes the syntactical definition of a "symbol"

In addition to the above syntactical functions and subroutines, the program has numerous utility routines. They are the following.

ATHENA - converts Univac 1108 hollerith characters to Athena hollerith characters

DEFLK - looks up symbol in the define symbol (program linking) table and returns their value

ENTDEF - enters symbols and their value into the define symbol (program linking) table

ENTROP - enters operation codes and associated information into the operation table

ENTRSY - enters symbols and their value into the symbol table

ERRER - prints out diagnostic error messages

HTONUM - converts the hollerith representation of an integer number to an integer value

ICHECK - scans a character string for a particular character or name

LITLK - looks up literals in the literal table and returns their address

LITPRT - prints out the content of the literal table at the completion of the assembly process and punches an object program out on paper tape for the Athena computer. When triggered by another subroutine SETUP, it prints out the content of the operation table and macro definition table.

LOCATE - parses the source line and isolates the label, operation code and operand

LTABLE - enters literals into the literal table if not already entered

MACLIN - reads the next line of a macro definition from macro definition table, substitutes macro call arguments for dummy arguments, and returns the macro definition line as a return argument

MACLK - reads a macro definition line from the macro definition table

MACTAB - enters a macro definition line into the macro definition table

OPLOOK - looks up an operation code in operation table and returns its values

OUT - assembles a numeric machine instruction and then prints out the location counter, the numeric machine instruction, and the source line

PULLAS - pulls information off the macro pushdown stack

PUSHAS - pushs information onto the macro pushdown stack

READ - reads the next source line from the input file. If the macro switch is on, it reads the next source line from the macro definition table.

READF - reads an assembly language instruction line off the copy file, unpacks the line, and returns the line

SETUP - enters operation codes into the operation table and sets a trigger to cause the content of the

operation table and macro definition table to be printed out

SYMLK - looks up symbols in the symbol table and returns their value

UTOATH - converts binary object code to ASC II code useable by the Athena flexowriter tape reader

WRITEF - takes an instruction line, packs the line into 15 words, and then copies the packed instruction onto the copy file

A.1.1 Usage

A.1.1.1 Input Description

Anything input by the user will be read by the input routine, READ, and an attempt will be made to assemble it. However the program will recognize as legitimate operation codes only the following.

1. Machine operation codes given in Appendix E.
2. Macro operation codes entered by the user.
3. Macro operation codes for floating-point arithmetic stored permanently in the macro definition table.
4. Directives to the assembler such as the DATA, RES, DEF, LOC, EQU, and END directives.

These operation codes are explained in detail in Chapter III and will not be repeated here.

The format of the source line or punched card is free form in Columns 1-72. The label field, operation code field, and operand field are delineated from each other by blank spaces. An * in Column 1 designates the source line as a comment statement. The user indicates the completion of input with a dollar sign "$" in Column "1". The dollar sign indicates that no more source lines will follow.

A.1.1.2 Output Description Process

Output of the assembler program is a series of lines giving the location counter, the numeric machine instruction generated by the source line and the source line. On completion of the printout of the assembled source program, the object program for the Athena is punched out on paper tape. It is necessary for the paper tape punch to be turned on before the start of the punching of the object program, therefore, as an aid to the user, a message is printed out saying "TURN ON PAPER TAPE PUNCH". A few seconds then elapsed before the start of punching of the object program. After completion of the punching, a message is printed saying "TURN OFF PAPER TAPE PUNCH."

A.1.1.3 Executing the Athena Assembler From a Teletype Terminal

Figure A-1 shows the statements required to execute the program on the Univac 1108, Exec. 8 system from a terminal. File ASSEMBLER contains the symbolic, relocatable, and absolute

```
Col. 1

     "RUN       name, i.d., i.d.
     "XQT       ASSEMBLER.MAP1
                Source Program
                      -
                      -
                      -
                      -
                      -
                      -
                      -
                      -
                      -
     $
     "FIN
```

Fig. A-1. Executing the Athena Assembler from a Teletype Terminal

elements. The assembler program is assumed to have been placed on the files at a prior time using the deck setup shown in Figure A-3.

A.1.1.4 Executing Athena Assembler with a Card Deck

In Figure A-2 the deck setup for executing the assembler using a card deck is shown. Executing the assembler using cards is a debugging aid only. The source program is executed on the teletype terminal to obtain a paper tape. The deck setup is for execution on the Univac 1108, Exec. 8 system. The assembler is assumed to have been placed on files at a prior time. The file ASSEMBLER contains the symbolic, relocatable, and absolute elements of the Athena assembler.

A.1.1.5 Deck Setup for Storing Athena Assembler on Files

Figure A-3 shows the deck setup for storing the Athena assembler on file. The Athena assembler is placed on file to allow execution of the Athena assembler from the teletype terminal. The file ASSEMBLER contains the symbolic, relocatable, and absolute elements. The absolute element is called MAP1.

A.1.1.6 Deck Setup for Entering Macro Definition Permanently Into the Athena Assembler Program

Entering macro definitions into the Athena assembler permanently is an aid to users. Storing widely used macro

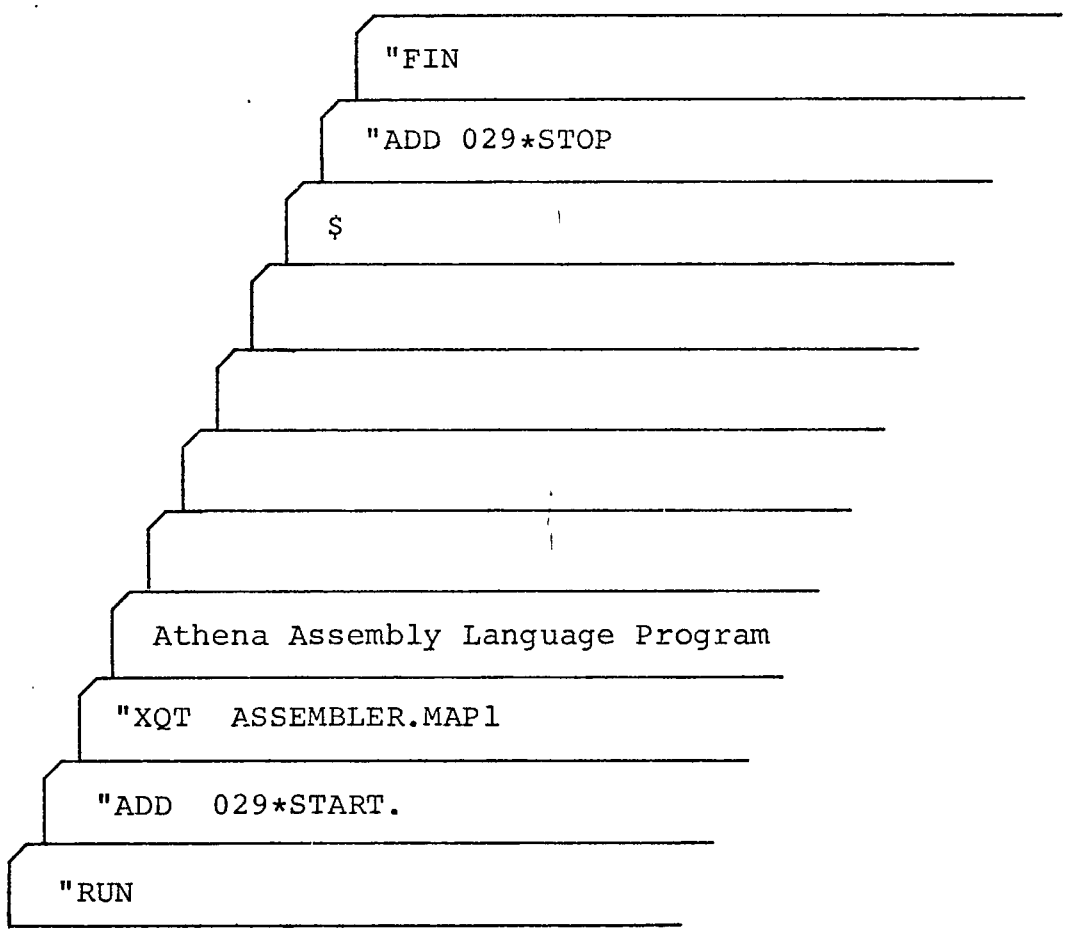


Figure A-2

DECK SETUP FOR EXECUTING THE ATHENA ASSEMBLER PROGRAM WITH CARDS.

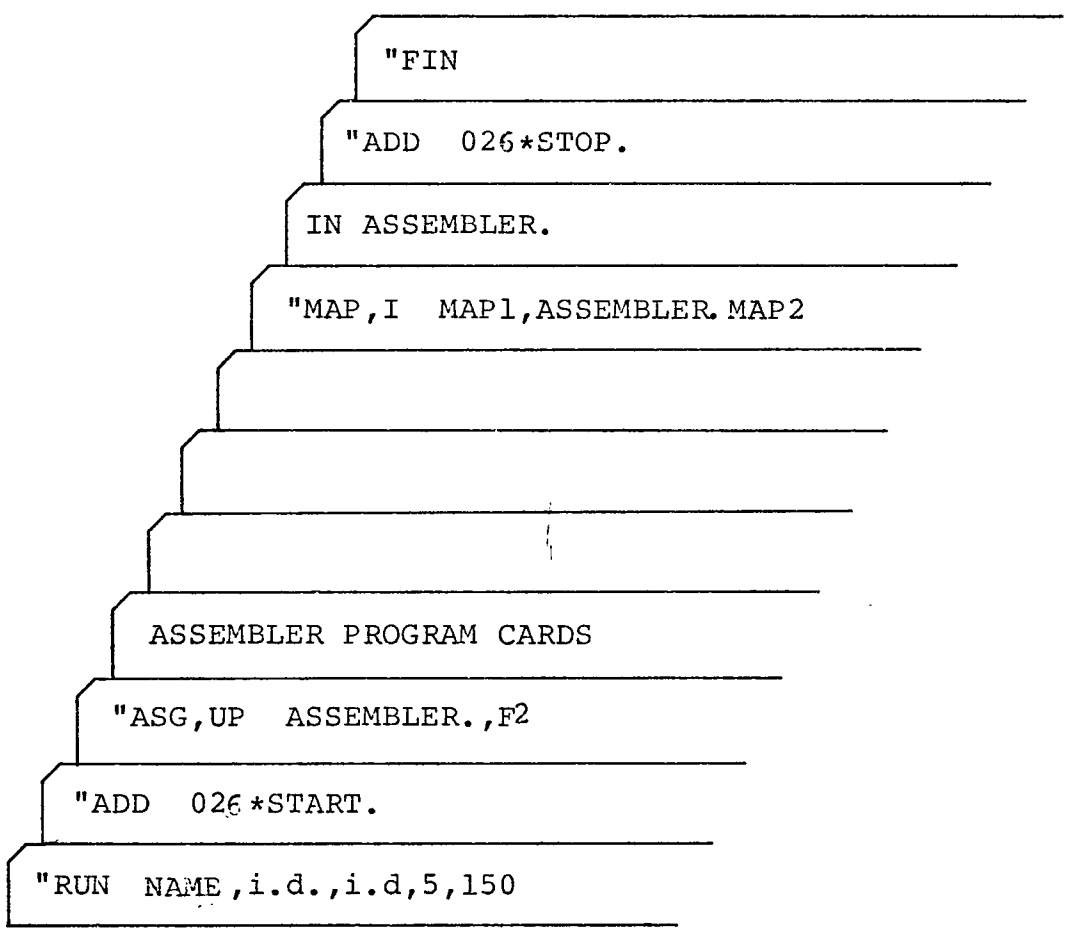


Figure A-3

DECK SETUP OF PLACING ATHENA ASSEMBLER PROGRAM ON FILE.

definitions permanently in the macro definition table and their names and macro definition index in the operation table makes macro definitions easily available to users. Permanent storage eliminates the need to supply every macro definition needed with the source program on each occasion the Athena assembler is used. To facilitate entering macro definitions permanently, the Athena assembler uses a utility subroutine, SETUP, to aid the systems programmer.

To store macro definitions permanently, macro definitions must be entered into the macro definition table and the macro name and macro definition table index entered into the operation table. The contents of the macro definition table and the operation table must then be printed out. Fortran data cards are punched from the printout and entered into the body of the Athena assembler program in subroutines MACTAB and ENTROP. The Athena assembler is then placed on file again using the deck setup in Figure A-3.

Figure A-4 shows the deck setup for entering macro definitions into the macro definition table and operation table and then obtaining a printout of the contents of these tables. The Athena assembler program is assumed to have been placed on file previously using the deck setup in Figure A-3. In Figure A-4 Cards 3, 4, and 5 insert a call to subroutine SETUP in program MAIN. Card 9 is read by subroutine SETUP and causes

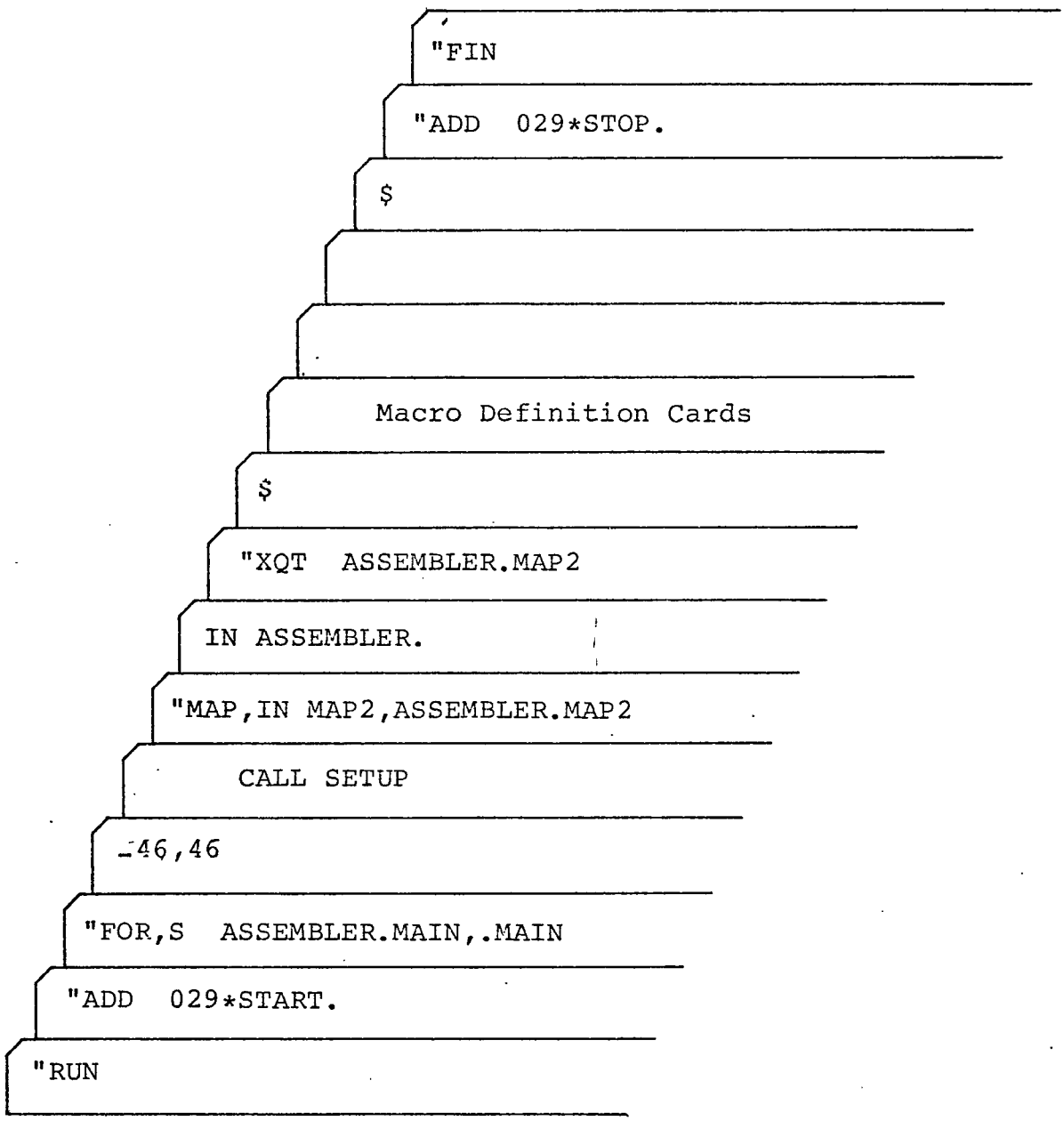


Figure A-4

DECK SETUP USED TO AID IN CHANGING THE OPERATION TABLE AND ENTERING MACRO DEFINITIONS PERMANENTLY INTO THE MACRO DEFINITION TABLE.

a return in subroutine SETUP to program MAIN after SETUP has set a trigger to cause subroutine LITPRT to print out the content of the macro definition table and operation table after the macro definitions have been entered. The macro definitions begin with Card 10.

A.1.1.7 Changing the Operation Codes

Changing the operation codes requires that new operation codes first be entered into the operation table and then the printing out of the content of the operation table. Fortran data cards are punched from the printout and entered into the body of the Athena assembler program in subroutine ENTROP. Subroutine SETUP aids the systems programmer by entering new operation codes into the operation table and then sets a trigger to cause subroutine LITPRT to print out the content of the operation table. To make use of subroutine SETUP, a call to subroutine SETUP must be inserted into program MAIN. Comment cards in program MAIN indicate where the call on subroutine SETUP is inserted.

Figure A-4 shows the deck setup for entering new operation codes into the operation table and obtaining a printout of the content of the operation table. To enter new operation codes into the operation table, four data are punched for each new operation code and inserted following Card 8 in Figure A-4. The first data card has the operation code name

starting in Column 1. Any number of characters may be used in the name. The second data card has a number in Column 1 identifying the "type" of operation code involved. The "type" number is "6" for all machine instructions. The third card gives the numeric value of the operation code. The numeric value of the operation code is a six digit octal number starting in Column "1" and is read with a "Ø6" format. The fourth card gives the mask to be used in masking off the address (or value) before it is added to the least significant bits of the numeric operation code to form a complete numeric machine instruction. The mask is a 12 digit octal number beginning in Column "1" and is read with a "Ø12" format. The lower six digits of the mask are then used in masking the address. The upper six digits of the mask are always "777777" octal. An example of a mask for numeric operation code "060000" is "777777777400". Numeric operation code "060000" cause the content of the magnetic core storage location specified in its lower "8" bits to be loaded into the accumulator. The mask has its lower eight bits set to "0". The data cards for macro definitions starting with Card 10 in Figure A-4 are not required. The last three cards on the deck setup, starting with the dollar sign card, are required.

A.1.2 Execution Characteristics

A.1.2.1 Program Restrictions

The Athena assembler program is machine dependent. Bit level manipulation of data makes the program dependent on the particular computer used in executing the program. The Athena assembler executes on the Univac 1108 or computers similar to the Univac 1108. Features of the Univac 1108 which the program takes into consideration are the computer word length (36 bits per word), the byte length (six bits per byte), the computer's internal floating-pointing number representation, the computers representation for numbers (one's complement on the Univac 1108), and a special bit level manipulation function, the field function (FLD). All subroutines and functions that are machine dependent have notations in their listings identifying them as machine dependent and indicating the particular statements which are machine dependent.

The Athena assembler program assembles 13,106 lines of assembly language instructions. No more than 666 labels, two characters long or 210 labels, fifteen characters long, may appear in the label field of the assembly language instructions. The capacity of the symbol table where the labels are stored depends on the length of the labels. No more than 1000 literals with values less than $2^{18}-1$ may appear in the operand field. No more than 800 literals with values greater than $2^{18}-1$ but less than $2^{36}-1$ may appear in the

operand field. The operation table has capacity for 500 operation codes and macro names two characters in length or 444 operation codes and macro names three characters in length. The longer the operation codes and macro names, the fewer that can be placed in the operation table.

The number of lines of macro definition which can be stored in the macro definition table depends on the number of characters used in the macro definition line. Assuming that each macro definition line contains a total of 11 characters in the label, operation code, and operand, 500 lines of macro definition can be stored in the macro definition table.

There may be a maximum of 100 symbols, two characters in length, appearing in the operand field of DEF directives. If the symbols are 15 characters long, only 21 symbols may appear in the operand field of DEF directives. The number of symbols the define symbol (program linking) table will hold depends on the length of the symbols used.

A.1.2.2 Storage Requirements

Code (or instructions) for the Athena assembler occupy 11628 words of memory. Data occupies 33932 words of memory. Total storage required is 45560 words.

A.1.2.3 Run Time

Execution time averaged 40 millisecond per assembly language instruction in test runs with the Athena assembler program on the Univac 1108 computer.

A.2 Program MAIN

The MAIN program performs the functions of Pass I. The functions of Pass I are described in Chapter IV and will not be repeated here.

A.3 Subroutine Pass II

Subroutine Pass II performs the functions of Pass II as described in Chapter IV. The functions of Pass II are described very adequately in Chapter IV and will not be repeated here.

The calling sequence is

```
CALL     PASS II
```

A.4 Function ARG

Function ARG is used to examine the operand field for a decimal integer, octal constant, floating-point number, or a character string constant.

The calling sequence is:

```
ARG (FIELD, FLDCNT, ARGVAL, CNT, TYPE, PTR)
```

The function ARG examines the character string in the vector FIELD, starting with the character being pointed at by PTR, for a decimal integer, octal constant, floating-point number or a character string constant. If a constant is found, ARG is set to 1, and the value of the constant is returned in the vector ARGVAL. CNT is set equal to the number

of words of ARGVAL used to store the value of the constant. PTR is left pointing to the next character in FIELD following the constant. If a character string constant is found, TYPE is set to 2, otherwise it is set to 1. FLDCNT is the count on the number of characters in vector FIELD.

A.5 Subroutine ATHENA

Subroutine ATHENA converts Univac 1108 octal representations for characters to their equivalent representation for the Athena.

The calling sequence is:

CALL ATHENA (SYMBL, ATHSYM, CNT)

The subroutine ATHENA takes a Univac 1108 octal representation for a character (for example, the Univac 1108 representation for "A" is "06_8") passed to it in SYMBL and converts it to the equivalent octal representation for the Athena computer. The equivalent Athena octal representation is returned in ATHSYM. As an example, subroutine ATHENA will take the Univac 1108 representation for an "A", (06_8), and convert it to the Athena computer representation for "A" (61_8). Subroutine ATHENA keeps track of the case (upper case or lower case) of the characters and prefixes characters as required (i.e., places a 74_8 before a string of upper case characters and places a 72_8 before a string of lower case characters). CNT is set equal to the number of bytes being

returned in ATHSYM. CNT will always be 1 or 2.

A.6 Function CHAR

Function CHAR processes a character string constant.

The calling sequence is:

CHAR (FIELD, FLDCNT, STRING, SCNT, PTR)

Function CHAR scans the content of the vector FIELD for a character string constant starting with the character pointed at by PTR. If a character string constant is found, function CHAR is set to 1 and the value of the character string constant is stored in vector STRING as a return argument. The count of words of STRING needed to store the value of the character string constant is returned in SCNT. PTR is returned, pointing to the first character beyond the character string constant. FLDCNT is the count on the number of characters contained in vector FIELD.

If function CHAR does not find a character string constant, function CHAR is set to zero and PTR is reset to its original value.

A.7 Subroutine DATA

Subroutine DATA processes source lines containing a "DATA" directive in its operation code field.

The calling sequence is:

CALL DATA (LINE, LABEL, OPCODE, OPERAND, LABCNT, OPCCNT,
OPRCNT)

Subroutine DATA processes the "DATA" directive in accordance with the value of METHOD. If METHOD = 1, the "DATA" directive is processed as needed in Pass I of the assembly process. If METHOD = 2, the "DATA" directive is processed in accordance with the needs of Pass II. A detailed description of the processing of the "DATA" directive on Pass I and Pass II is given in Chapter IV.

Passed to subroutine DATA is the source line in vector LINE, the contents label field in vector LABEL, the contents of the operation code field in OPCODE, and the contents of the operand field in vector OPRAND. The count on the number of characters in LABEL, OPCODE and OPRAND are stored in LABCNT, OPCCNT, and OPRCNT, respectively.

A.8 Subroutine DEFLK

Subroutine DEFLK looks up symbols contained in the define symbol table. Symbols which appear in the operand field of "DEF" directives are contained in the define symbol table. With a call to subroutine DEFLK, the value of any symbol contained in the define symbol table may be obtained.

The calling sequence is:

CALL DEFLK (OPRAND, OPRCNT, FOUND, VALUE)

Subroutine DEFLK looks up the symbol contained in vector OPRAND in the define symbol table and returns its value in VALUE. FOUND is set to 1 if the symbol was found in the table.

FOUND is set to 0 if the symbol was not found. To change the size of the define symbol table, change the dimension of vector TABLE in subroutine DEFLK and subroutine ENTDEF. Subroutine ENTDEF enter symbols into the define symbol table.

A.9 Subroutine ENTDEF

Subroutine ENTDEF enters symbols found in the operand field of "DEF" directives into the define symbol table. The value of the symbol is also entered into the define symbol table with the symbol.

The calling sequence is:

CALL ENTDEF (OPRAND, OPRCNT)

Subroutine ENTDEF obtains the value of symbol contained in vector OPRAND from the symbol table and enters OPRAND and its value into the define symbol table. OPRCNT is the count on the characters contained in vector OPRAND. To change the size of the define symbol table, change the dimension of vector TABLE in subroutine ENTDEF and subroutine DEFLK. Subroutine DEFLK obtains values for symbols from the define symbol table.

A.10 Subroutine ENTROP

Subroutine ENTROP enters operation code names, macro names and directive names into the operation table. Other information pertinent to these names is also entered.

The calling sequence is:

CALL ENTROP (KEY, KEYCNT, TYPE, MACIND, OPCNUM, MASK)

Subroutine ENTROP enters the operation code name, macro name, or directive name passed to it in vector KEY into the operation table. The count of characters KEYCNT, contained in KEY and the type code, TYPE, are entered with KEY into the operation table. The type code, TYPE, identifies the name in KEY as a machine operation code, name, macro name, or directive name. The following type codes are used: TYPE = 1 for a EQU directive, TYPE = 2 for a DATA directive, TYPE = 3 for a RES directive, TYPE = 4 for a MACRO directive, TYPE = 5 for a macro name, TYPE = 6 for a machine operation code, TYPE = 7 for an END directive, TYPE = 8 for a DEF directive, and TYPE = 9 for a LOC directive. If the name is a macro name, an index, MACIND, to the location of the macro in the macro definition table is entered into the operation table. If the name is a machine operation code, the numeric value of the machine operation code, OPCNUM, is entered into the operation table along with a mask, MASK, for the numeric operation code.

Subroutine ENTROP computes a hash address, KPLACE, from KEY which it uses as an index to the scatter table, TABLE. A pointer (the current value of PTR) is left in TABLE (KPLACE) which points to the place in the operation table storage area, FSL (PTR), where the subroutine calling arguments, KEY, KEYCNT,

TYPE, MACIND, OPCNUM and MASK are stored. A detailed description of the operation table is given in Chapter IV.

All operation code names and directive names are stored in the operation table permanently. A few macro names are also stored permanently, but most macro names are entered on a temporary basic as each programmer defines them in his particular program. Permanent entries into the operation table are made by inserting data cards with the operation code name, directive name or macro name and associated information into subroutine ENTROP. Special provisions have been made for assisting the systems programmer in making permanent entries into the operation table. A detailed discussion of these provisions is presented in subroutine SETUP.

The size of the operation table may be changed by changing the dimensions of vector FSL in subroutine ENTROP and subroutine OPLOOK. Subroutine OPLOOK looks up operation codes, macro names, and directive names in the operation table.

A.11 Subroutine ENTRSY

Subroutine ENTRSY enters symbols into the symbol table. The value of the symbol is entered also.

The calling sequence is:

CALL ENTRSY (KEY, KEYCNT, VALUE)

Subroutine ENTRSY enters the symbol in vector KEY into the symbol table. The symbol character

count, KEYCNT, the value of the symbol, VALUE, and the identification number of the program the symbol is being utilized in, are entered into the symbol table. The program identification number, PRGNUM, is passed to subroutine ENTRSY through a common block from the MAIN program. The program number, PRGNUM, allows different assembly language programs to be assembled together while using the same symbol table for all of the programs. PRGNUM is different for each assembly language program. A more comprehensive discussion of the program identification number is given in Chapter IV in the discussion on the END directive.

Since a symbol may not be entered into the symbol table more than once, a flag is set when the symbol is entered the first time. Any attempts to enter the symbol again will result in an error message being printed out declaring the symbol to be multiply defined.

Subroutine ENTRSY computes a hash address, KPLACE, from KEY which is used as an index to the scatter table, TABLE. A pointer is left in TABLE (KPLACE) which points to the place in the symbol table storage area, FSL (PTR), where the KEY, KEYCNT and VALUE are stored. A detailed description of the operation of the symbol table is presented in Chapter IV in the discussion and data tables.

The size of the symbol table may be changed by changing the dimension of vector FSL in subroutine ENTRSY and subroutine

SYMLK. Subroutine SYMLK looks up symbols in the symbol table and returns their value.

A.12 Subroutine EQU

Subroutine EQU processes the "EQU" directive.

The calling sequence is:

CALL EQU (LABEL, OPRAND, LABCNT, OPRCNT)

Subroutine EQU processes the label field and operand field of an "EQU" directive. The content of the label field is passed to subroutine EQU in vector LABEL. The label in LABEL is verified to be a legal symbol and then the operand field, stored in vector OPRAND, is evaluated for an expression, octal constant, decimal integer, or a location counter symbol (a dollar sign). LABEL and the value of the operand field are entered into the symbol table.

A.13 Subroutine ERRER

Subroutine ERRER prints out error messages and their number.

The calling sequence is:

CALL ERRER (ARG)

Subroutine ERRER writes out an error message and an error message number, ARG. When ERRER is called during Pass I, the error message is written onto File 3 with the rest of the source program. During Pass II, the error message is written out on the printer.

A.14 Function EXPR

Function EXPR examines the operand field for an expression of the form:

octal constant ± decimal integer

symbol ± decimal integer

decimal integer ± decimal integer

dollar sign ± decimal integer

± integer

± octal constant

The calling sequence is:

EXPR (OPRAND, OPRCNT, VALUE, VALCNT, PTR)

Function EXPR scans the operand field in array OPRAND for an expression. Function EXPR starts scanning OPRAND at the character pointed at by pointer, PTR. If an expression is found, function EXPR is set to 1 and the value of the expression is returned in array VALUE. VALCNT gives the count on the number of words of array VALUE needed to hold the value of the expression. The pointer, PTR, is returned pointing to the first character following expression.

If no expression is found, the function EXPR is set to zero and the pointer PTR is reset to its original value.

OPRCNT is the count on the number of characters contained in array OPRAND.

A.15 Function FLOATI

Function FLOATI scans the operand field for a floating-point number. If a floating-point number is found, it is processed and a value is returned.

The calling sequence is:

FLOATI (FIELD, FLDCNT, FNUM, FCNT, PTR)

Function FLOATI scans vector FIELD, starting with the character pointed at by PTR, for a floating-point number. Function FLOATI searches for a floating-point number with the following format.

F' a maximum of 6 digits and a decimal point'

or F' a maximum of 6 digits and a decimal point E ± 2 digits'

Function FLOATI converts the real number found in the floating-point declaration to a floating-point number consisting of a fraction and an exponent. The fraction is returned in FNUM(1) and the exponent is returned in FNUM(2). The fraction is stored in the least significant 18 bits of FNUM(1), left justified to bit 18. If the real number is positive, the 18th bit will be zero and the 17th bit equal to 1. If the real number is negative, the 18th bit is 1 and the 17th bit zero. The exponent is stored right justified in FNUM(2) with the decimal point to the right of the least significant bit. The exponent is an integer number with no offset added to its value. The exponent is negative for real numbers less than one. If

the real number is zero, FNUM(1), the fraction, and FNUM(2), the exponent are set to zero.

If a floating-point number is found, the function FLOATI is set to 1. The pointer, PTR, is returned pointing to first character to the right the floating-point declaration. If no floating-point number is found, function FLOATI is set to zero, and the pointer, PTR, reset to its original value.

A.16 Subroutine HTONUM

Subroutine HTONUM converts a decimal number expressed as a string of hollerith characters to its equivalent binary value.

The calling sequence is:

CALL HTONUM (IV, KOUNT, NUM)

Subroutine HTONUM converts the hollerith decimal number passed to it in vector IV, to its equivalent binary value. KOUNT contains a count on the number of hollerith characters being passed to subroutine HTONUM in Vector IV. A single hollerith number in each word of Vector IV is assumed. The most significant digit of the decimal number is assumed to be stored in IV(1) and IV(2) contains the next most significant digit, etc. The binary value of the hollerith decimal number is returned in NUM.

A.17 Function ICHECK

Function ICHECK examines a string of characters for a particular character or string of characters.

The calling sequence is:

ICHECK(VALUE, SYMBOL, COUNT, PTR)

Function ICHECK test to see if next 'COUNT' characters in the vector VALUE, and starting with the character pointed to by PTR (i.e., VALUE(PTR) through VALUE(PTR + COUNT - 1), match the characters found in SYMBOL. If a match is found, ICHECK returns a value of "1" and sets PTR = PTR + COUNT, if not ICHECK returns a value of "0" and PTR remains where it was.

A.18 Function INTEGR

Function INTEGR examines a string of characters for a decimal integer. A decimal integer is defined to be a decimal number consisting of six digits or less.

The calling sequence is:

INTEGR (FIELD, FLDCNT, INTGR, INTCNT, PTR)

Function INTEGR tests the character string in vector FIELD, starting with the character point to by PTR, for a decimal integer. Scanning continues until the first non-digit is encountered or until "7" digits (one more than the maximum digits allowed in a decimal number) have been found. If the decimal number contains too many digits, it is truncated and an error message is printed out.

If a decimal integer is found, INTEGR is set to "1" and the decimal integer is stored in vector INTGR for return. INTCNT is set equal to the digit count of the decimal integer. PTR is set to PTR = PTR + INTCNT.

FLDCNT is the character count in FIELD.

A.19 Subroutine INTPAS

Subroutine INTPAS performs the duties of described for Interpass in Chapter IV. Subroutine INTPAS assigns drum storage location to each literal in the literal table. Literals are assigned locations beginning with a displacement equal to the current program length. The program length is in fact the value of the location counter at the end of Pass I. It is the number of words of machine code generated by instructions, data directives and reserve directives in the source program.

The calling sequence is:

CALL INTPAS

A.20 Function LABL

Function LABL checks for a legal symbol in the label field of a source line.

The calling sequence is:

LABL (LABEL, LABCNT)

Function LABL checks the content of the label field which is passed to it in vector LABEL for a legal symbol. A legal

symbol is defined to consist of one to fifteen letters or digits, at least one of which is a letter. If a legal symbol is found, LABL is set to "1", if no symbol is found, LABL is set to "0", if an illegal symbol is found, LABL is set to "-1".

LABCNT is the character count in LABEL.

A.21 Subroutine LITLK

Subroutine LITLK looks up literals in the literal table and returns their drum location.

The calling sequence is:

CALL LITLK (KEY;KEYCNT, FOUND, VALUE)

Subroutine LITLK probes the literal table for the literal placed in vector KEY. If KEY is found, its drum storage location is returned in VALUE and FOUND is set to logical TRUE. KEYCNT is the count of the number of words of vector KEY required for the literal.

Subroutine LITLK probes the literal table by first computing a hash address, KPLACE, from KEY which is used as an index to the scatter table, TABLE. A pointer, KPROBE, stored in TABLE (KPLACE) points to the place in the literal table storage area FSL (KPROBE + 1) where KEY and its drum location are stored. If KEY is not stored starting at FSL (KPROBE + 1), then the content of FSL (KPROBE) is checked for a pointer to the next literal of this chain. If FSL (KPROBE) = 0, then the literal is not in the table and the search is terminated. If

FSL (KPROBE) contains a pointer, KPROBE is set to KPROBE = FSL(KPROBE) and the next literal on the chain is checked for a match with KEY. If a KEY is not found in the chain, then the literal KEY is not in the table and FOUND is set equal to logical FALSE. A description of the operation of the literal table is given in Chapter IV in the discussion on data tables.

The size of the literal table may be changed by changing the dimension of vector FSL in subroutine LITLK, subroutine LTABLE, subroutine INTPAS, and subroutine LITPRT.

A.22 Subroutine LITPRT

Subroutine LITPRT has three functions. It prints out the content of the literal table at the end of Pass II. It reads the content of the object code copy file and punches a paper tape for the Athena. If commanded to by subroutine SETUP, it prints out the content of the operation table and macro definition table.

The calling sequence is:

CALL LITPRT

A.23 Function LITRAL

Function LITRAL examines the operand field for a literal.

The calling sequence is:

LITRAL (FIELD, FLDCNT, LITVAL, VALCNT, PTR)

Function LITRAL scans the content of vector FIELD, starting with the character pointed to by PTR, for a literal. A literal is defined to consist of an equal sign in front of a constant (= constant) or a constant enclosed in quote marks and preceded by a "L" (L 'constant'). The literal is converted to its binary form and returned in vector LITVAL. VALCNT is the word count of the literal value. PTR is returned pointing to the next character to the right of the literal in FIELD.

A.24 Subroutine LOCATE

Subroutine LOCATE parses the source line for the label operation code and operand.

The calling sequence is:

CALL LOCATE (CARD, LABEL, LABCNT, OPCODE, OPCNT, OPRAND, OPRCNT)

Subroutine LOCATE scans vector CARD for the label, operation code, and operand. The label and its character count are returned in LABEL and LABCNT. The operation code and its character count are returned in OPCODE and OPCCNT. The operand and its character count are returned in OPRAND and OPRCNT.

A.25 Subroutine LTABLE

Subroutine LTABLE enters literals into the literal table.

The calling sequence is

CALL LTABLE (KEY, KEYCNT)

Subroutine LTABLE enters the literal contained in vector KEY and the word count on KEY, KEYCNT, into the literal table if the literal table does not already contain a duplicate.

Subroutine LTABLE uses a hash technique to enter KEY into the literal table. A discussion of literal table operation is given in Chapter IV in the discussion on data tables.

A.26 Subroutine MACDEF

Subroutine MACDEF processes macro definitions and stores them into the macro definition table.

The calling sequence is:

```
CALL MACDEF (CARD, LABEL, LABCNT, OPCODE, OPCNT, OPRAND,
             OPRCNT)
```

Subroutine MACDEF enters the macro name passed to it in vector LABEL into the operation table. Stored with LABEL is a flag identifying the content of LABEL as a macro name and the index to the definition of the macro in the macro definition table. The operand field passed to subroutine MACDEF in OPRAND, is scanned for dummy arguments. Dummy arguments are stored. The source line is stored in the macro definition table. The succeeding lines of the macro definition are read. Each line is parsed to isolate the content of the label field, operation code field and operand field. Each field is in turn checked for a match with each dummy argument. If a dummy argument is found in the label field, operation code field or

operand field, its identity and the field it is located in is noted. The macro definition line is then stored in the macro definition table along with the identity and location of any dummy argument. Processing of the macro definition lines is terminated when the "END" directive is identified on the macro definition.

Calling arguments for subroutine MACDEF not identified yet are vector OPCODE, LABCNT, OPCCNT, and OPRCNT. Vector OPCODE passes the content of the opcode field to subroutine MACDEF. LABCNT, OPCCNT and OPRCNT are respectively the count of characters in vectors LABEL, OPCODE and OPRAND.

A.27 Function MACLIN

Function MACLIN reads the next line out of the macro definition table, substitutes macro calling arguments for dummy arguments, and reassembles the packed label, operation code, and operand into an assembler instruction line.

The calling sequence is:

MACLIN (CODE)

Function MACLIN reads the next line of macro definition code from the macro definition table. If the line contains dummy arguments, the appropriate call argument is substituted for it from the correspondence table. The packed macro definition line is reassembled to its original form and placed in vector CODE as a return argument. Function MACLIN is set to "1" and a return is made.

If the next line read from the macro definition table was an "END" directive, and the macro pushdown stack is empty, function MACLIN is set to "0" and a return is made.

If the next line read from the macro definition table was an "END" directive and the macro pushdown stack is not empty, the correspondence table and macro definition table index is pulled off the pushdown stack. The macro definition table index from the pushdown stack is used to read the next line from the macro definition table. Using the correspondence table just pulled off the pushdown stack, call arguments are substituted for dummy arguments in the macro definition line. The packed macro definition line is reassembled to its original form and placed in vector CODE as a return argument. Function MACLIN is set to "1".

A.28 Subroutine MACLK

Subroutine MACLK reads a line of macro definition from the macro definition table.

The calling sequence is:

CALL MACLK (MACIND, LINE)

Subroutine MACLK reads a line of macro definition from the macro definition table starting at the index passed to the subroutine in MACIND. The macro definition line is returned in vector LINE.

The size of the macro definition table may be changed by changing the dimension on vector MDEF in subroutine MACLK and

subroutine MACTAB. Subroutine MACTAB enters macro definition lines into the macro definition table.

A.29 Subroutine MACOPC

Subroutine MACOPC processes a call on a macro.

The calling sequence is:

```
CALL MACOPC (LABEL, LABCNT, OPCODE, OPCCNT, OPRAND,
             OPRCNT, MACIX)
```

Subroutine MACOPC scans the label field and operand field passed to it in vector LABEL, and vector OPRAND for calling arguments and builds a correspondence table. The index to the macro in the macro definition table is passed in MACIX. If a macro was being expanded at the time this macro call occurred, the previous macro's correspondence table and index to the next line of that macro in the macro definition table are both pushed onto the macro pushdown stack.

Other calling arguments of subroutine MACOPC are OPCODE, OPCCNT, LABCNT, and OPRCNT. Vector OPCODE contains the content of the operation code field. LABCNT, OPCCNT, and OPRCNT are the character count in vectors LABEL, OPCODE, and OPRAND.

A.30 Subroutine MACTAB

Subroutine MACTAB stores lines of macro definition in the macro definition table.

The calling sequence is:

CALL MACTAB (CARD, LABEL, LABCNT, OPCODE, OPCCNT, OPRAND, OPRCNT)

Subroutine MACTAB stores in the macro definition table the content of the label field, operation code field, and operand field passed to MACTAB in vectors LABEL, OPCODE, and OPRAND. LABCNT, OPCCNT and OPRCNT are the count on the number of characters in vectors LABEL, OPCODE, and OPRAND, respectively. Stored in the macro definition table with the macro definition line are indexes identifying any dummy arguments and their location in the line. Indexes indicating the original positions of the operation code and operand in the macro definition line are stored to allow the macro definition to be re-assembled to its original form. Also stored in the macro definition table with the macro definition line are the count on the number of characters in LABEL, OPCODE, and OPERAND.

To change the size of the macro definition table, the dimension of vector MDEF is changed in subroutine MACTAB and subroutine MACLK.

A.31 Function OCTAL

Function OCTAL scans the operand field for an octal constant. The octal constant is converted to its binary form and returned.

The calling sequence is:

OCTAL (FIELD, FLDCNT, OCTNUM, OCNT, PTR)

Function OCTAL scans vector FIELD, starting with the character pointed to by pointer, PTR, for an octal constant. An octal constant is defined to consist of an octal number inside two quote marks and preceded by a "Ø" (i.e., Ø' 12 octal digits or less'). The octal number inside the quote marks is converted from its hollerith representation to its binary representation and stored in vector OCTNUM as a return argument. ØCNT is the number of words of OCTNUM required to hold the octal constant value. PTR is returned pointing to the first character to the right of the octal constant. Function OCTAL is set to "1".

If no octal constant is found, function OCTAL is set to "0" and the point is reset to its original value.

A.32 Subroutine OPLOOK

Subroutine OPLOOK probes the operation table for operation codes.

The calling sequence is:

CALL OPLOOK (KEY, KEYCNT, FOUND, TYPE, MACIND, OPCNUM, MASK)

Subroutine OPLOOK probes the operation table for the operation code passed to it in vector KEY. If KEY is found, FOUND is set to logical TRUE, otherwise FOUND is set to

logical FALSE. The type number of the operation code is returned in TYPE. If the type of operation code is a macro call, the index to the definition of the macro in the macro definition table is returned in MACIND. If the operation code type is a machine operation code, its numeric value is returned in OPCNUM. The numeric operation code mask is returned in MASK.

Subroutine OPLOOK uses hashing techniques to probe the operation table. A discussion of the operation of the operation table is presented in Chapter IV in the discussion on data tables.

To change the size of the operation table, vector FSL in subroutine OPLOOK and subroutine ENTROP must be changed.

A.33 Function OPRND

Function OPRND scans the operand field of a machine instruction for a machine operand. A machine operand is defined to be an expression, an octal constant, a symbol, a decimal integer or a location counter symbol (a dollar sign). A constant or an expression involving only constants is converted to its equivalent binary and returned. When a symbol appears in the operand field alone, the symbol's address (or value if the symbol was defined in an EQU directive) is obtained from the symbol table and returned as the symbol's value.

If the symbol appears in an expression, the value obtained from the symbol table is added or subtracted from the decimal integer appearing in the expression with the symbol, and returned as the expression's value. If a dollar sign is found, the current location counter value is returned.

The calling sequence is:

OPRND (OPRAND, OPRCNT, OPRVAL, VALCNT, RELABS, PTR)

Function OPRND scans vector OPRAND, starting at the character pointed at by pointer PTR, for an expression an octal constant, a symbol, a decimal integer, or a dollar sign. The operand field is evaluated and its value is returned in vector OPRVAL. VALCNT is the count on the number of words of OPRVAL used to store the operand field value. OPRCNT is the count of characters in vector OPRAND. PTR is returned pointing to the first character to the right of the expression, octal constant, symbol, decimal integer or dollar sign.

Argument RELABS is set to "2" if the value returned for the operand field is an address. RELABS is set to "1" for a constant value. RELABS is provided to make the assembler code usable as input to a relocatable loader. RELABS is not used as the program is presently set up.

A.34 Subroutine OUT

Subroutine OUT assembles a numeric machine instruction and prints it out along with the source line.

The calling sequence is:

CALL OUT (NUMOPC, MASK, RELABS, OPRVAL, CNT, LINE)

Subroutine OUT takes the numeric operation code in NUMOPC, and the value of the operand field contained in OPRVAL and combines them into a single numeric machine instruction. In combining NUMOPC and OPRVAL, the upper bits of OPRVAL are masked off using MASK to eliminate any bits from overlapping into the numeric operation code. The masked off OPRVAL and NUMOPC are "anded" together to form the complete numeric machine instruction. The source line number, location counter value, numeric machine instruction and source line, LINE, are printed out. The location counter is increment one.

If the numeric operation code is zero, as with a data directive, the content of vector OPRVAL is printed out along with the location counter. The source line, LINE, is printed out with OPRVAL(1). CNT specifies the number of words of OPRVAL to be printed out.

The numeric machine instruction is also copied into the copy file for punching the paper tape.

RELABS specifies if the content of OPRVAL is a relative address or an absolute value. RELABS is provided to make the assembler code usable as input to a relocatable loader. RELABS is not used as the program is presently set up. Absolute loading is used. RELABS is equal to "1" if OPRVAL is an

absolute value, and "2" if OPRVAL is a relative address.

A.35 Subroutine PULLAS

Subroutine PULLAS pulls information off the macro pushdown stack.

The calling sequence is:

CALL PULLAS (MACIND, CNT, ARGSIZ, ARG, ARGERR)

Subroutine PULLAS pulls off top of the macro pushdown stack the index, MACIND, to the next line of the macro in the macro definition table, the count, CNT, on the number of calling arguments in correspondence table, a vector ARGSIZ, containing the count of the characters in each calling argument store in the correspondence table ARG. ARGERR is a flag stored with the correspondence table, which is set to "1" when the number of calling arguments in the macro call is less than the number of dummy arguments in the macro definition.

A.36 Subroutine PUSHAS

Subroutine PUSHAS pushes information onto the macro pushdown stack.

The calling sequence is:

CALL PUSHAS (MACIND, CNT, ARGSIZ, ARG, ARGERR)

The calling arguments of subroutine PUSHAS are discussed in subroutine PULLAS.

A.37 Subroutine READ

Subroutine READ reads the next source line from the input file, or if the macro switch is set to "1", the next source line is read from the macro definition table.

The calling sequence is:

CALL READ (CARD)

Subroutine READ returns the next source line from the input file or macro definition table in vector CARD.

A.38 Subroutine READF

Subroutine READF reads a packed assembly language instruction line off the copy file, unpacks it and returns the unpacked line in vector CARD.

The calling sequence is:

CALL READF (CARD)

A.39 Subroutine RES

Subroutine RES processes the "RES" directive.

The calling sequence is:

CALL RES (LINE, LABEL, OPRAND, LABCNT, OPRCNT, METHOD)

Subroutine RES processes the "RES" directive in accordance with the requirements of Pass I if METHOD = 1, and in accordance with the requirements of Pass II if METHOD = 2. A description of the processing of the "RES" directive is presented in Chapter IV.

Vector LINE contains the source line. LABCNT and OPRCNT give the character count of the label field and operand field stored in vector LABEL and vector OPRAND.

A.40 Subroutine SETUP

Subroutine SETUP enters operation codes and directive names with associated information into the operation table. A trigger is set to cause the operation table and macro definition table to be printed out by subroutine LITPRT at the completion of the assembly process.

The calling sequence is:

CALL SETUP

A description of the use of subroutine SETUP in making updates to the operation table is given in the front of this appendix.

A.41 Function SYMBOL

Function SYMBOL scans the label field or operand field for a symbol. A symbol is defined to consist of no more than 15 letters or digits, at least one of which is a letter.

The calling sequence is:

SYMBOL (FIELD, FLDCNT, SYMBL, SYMCNT, PTR)

Function SYMBOL scans the content of vector FIELD starting with the character pointed at by pointer PTR for a symbol. The symbol is returned in vector SYMBL and SYMCNT

gives the character count of the symbol. If a symbol is found, function SYMBOL is set to "1" and PTR is returned pointing to the first character to the right of the symbol in vector FIELD.

If no symbol is found, function SYMBOL is set to "0", and PTR is reset to its original value.

FLDCNT is the count on characters in vector FIELD.

A.42 Subroutine SYMLK

Subroutine SYMLK probes the symbol table for a symbol and returns its value.

The calling sequence is:

CALL SYMLK (KEY, KEYCNT, FOUND, VALUE)

Subroutine SYMLK probes the symbol table for the symbol passed to it in vector KEY. If KEY is found, FOUND is set to logical TRUE, otherwise FOUND is set to logical FALSE. The value of the symbol is returned in VALUE. KEYCNT is the symbol character count in KEY.

Subroutine SYMLK uses hashing techniques to probe the symbol table. A discussion of the operation of the symbol table is presented in Chapter IV in the discussion on data table.

A.43 Subroutine UTOATH

Subroutine UTOATH takes a 18 bit binary numeric machine instruction and converts it to ASCII code usable by the Athena

tape reader.

The calling sequence is:

CALL UTOATH (INST, IVEC)

Subroutine UTOATH takes the 18 bit numeric machine code and converts it three bits at a time, starting with the three most significant bits, to ASCII code readable by the Athena tape reader. The Athena tape reader recognizes as octal numbers "0" through "7", the ASCII characters @, A, B, C, D, E, F, G.

A.44 Subroutine WRITEF

Subroutine WRITEF packs a 82 word assembly language source line into 15 words and stores it on the copy file. The instruction line is packed to save file space.

The calling sequence is:

CALL WRITEF (CARD)

Subroutine WRITEF takes the 82 characters stored in 82 words of vector CARD and packs them into 15 word (six characters to a word). The packed instruction line is then copied onto the copy file.

APPENDIX B

ERROR MESSAGES

APPENDIX B

ERROR MESSAGES

Error Number	Error Message
1.	Opcode not recognized
2.	Illegal label; syntax error. Label contains a character other than a letter or digit or only digits.
3.	Operand value is too large. Truncation possible.
4.	Illegal decimal integer
5.	The DATA directive below reserved more than 100 words
6.	Operand missing
7.	Label appears more than once in a label field
8.	Improper operand
9.	Label missing
10.	Non-octal number (an 8 or 9) appears in octal number declaration.
11.	Octal number greater than 12 digits long.
12.	Right quote mark missing.
13.	Truncation of an integer number has taken place. Integer has more than 6 digits.
14.	Symbol greater than 15 characters in length.

Error Number	Error Message
15.	Bad literal
16.	Location counter set to a value greater than 8192.
17.	Exponent of floating number missing.
18.	Exponent of floating number greater than 38.
19.	Decimal point missing in floating-point number.
20.	Character string declaration contains no characters between quote marks.
21.	Character string contains more than 24 characters.
22.	Coefficient of floating-point number missing.
23.	Symbol in operand field has not appeared in the label field of a statement.
24.	Capacity of macro definition storage table has been exceeded. Too many macros have been defined or else an "END" statement has left off a macro definition.
25.	An integer number must follow the "+" or "-" sign in the expression appearing in the operand field.
26.	Error in macro definition; possibly an END statement missing.
27.	The symbol appearing in the operand field of the DEF directive below cannot be entered into the define symbol (program linking) table. The define symbol table is filled to capacity already.

Error Number	Error Message
28.	Need to provide more words for data in the operand field of the "DATA" directive below.
29.	Only one word of storage required for literal appearing in the operand field of the "DATA" directive below.
30.	DATA directive is not properly suffixed with an integer number.
31.	Illegal argument in operand field. The operand field of a LOC directive must contain an integer or octal constant.
32.	Dummy arguments count in macro definition is greater than count of calling arguments.
33.	Dummy arguments count in macro definition does not equal the calling arguments count.
34.	Argument list too long; greater than 20 in length.
35.	A character not recognized in a character string declaration.
36.	Insufficient space allowed for insertion of calling arguments into macro definition line. Truncation resulted at Column 72.
37.	Truncation of contents of address field of numeric machine instruction has occurred.
38.	Symbol in operand field of DEF directive below has not been defined. The symbol must appear in the label field of a statement to be defined.

Error Number	Error Message
39.	Dummy argument too long.
40.	Macro calls are nested more than 20 deep.

APPENDIX C

ATHENA CHARACTER CODES

APPENDIX C

ATHENA CHARACTER CODES

Athena Flexowriter Symbol	Athena Internal Code (Octal)
A	61
B	62
C	63
D	64
E	65
F	66
G	67
H	70
I	71
J	41
K	42
L	43
M	44
N	45
Ø	46
P	47
Q	50
R	51
S	22
T	23

Athena Flexowriter Symbol	Athena Internal Code (Octal)
U	24
V	25
W	26
X	27
Y	30
Z	31
0	00
1	01
2	02
3	03
4	04
5	05
6	06
7	07
8	10
9	11
',-	40
;,&	60
=,_	53
,	33
.	73
:,/	21
space	14

Athena Flexowriter Symbol	Athena Internal Code (Octal)
Stop Code	13
Lower Case	72
Upper Case	74
Carriage Return	20
Tab	36
FC ON	56
ON1-ON2	54
OFF	37

APPENDIX D

LISTING OF FLOATING-POINT MACROS

D.1 Floating Add Macro

```
FA   MACRO  LAB,L1,L2,L3,L4,L5,L6,L7,A1,A2,B1,B2,C1,C2
LAB  LA     A2
     SA     TEMP2
     SB     B2
     SA     TEMP1
     BLZ    L1
     B      L2
L1   LCA    TEMP1
     SA     TEMP1
     LA     B2
     SA     TEMP2
     CX     0
     CA     0
     LA     B1
     SA     TEMP3
     LA     A1
     B      L3
L2   LA     A1
     SA     TEMP3
     CX     0
     CA     0
     RS     23
     LA     B1
L3   LIR    TEMP1
     INDEX
     RS     0
     AD     TEMP3
     LIRI   -23
L4   INC
     LS     1
     OJ     L5
     BIZ    L6
     B      L4
L5   RSL    7
     SA     C1
     CX     -16
     CA     1
     SIR    TEMP1
     SB     TEMP1
     AD     TEMP2
     SA     C2
     B      L7
L6   CX     0
     CA     0
     SA     C1
     SA     C2
L7   EQU    $
     END
```

D.2 Floating Multiply Macro

```
FM    MACRO    LAB,L1,L2,L3,L4,A1,A2,B1,B2,C1,C2
LAB   LA   A1
      MP   B1
      LIRI -47
L1    INC
      LS   1
      OJ   L2
      BIZ  L3
      B    L1
L2    RSL  7
      SA   C1
      CS   -35
      CA   1
      SIR  TEMP1
      SB   TEMP1
      AD   A2
      AD   B2
      SA   C2
      B    L4
L3    CX   0
      CA   0
      SA   C1
      SA   C2
L4    EQU  $
      END
```

D.3 Floating Divide Macro

```
FD    MACRO   LAB,L1,L2,L3,L4,A1,A2,B1,B2,C1,C2
LAB   LA      81
      RS      6
      SA      TEMP1
      CX      0
      CA      0
      RS      23
      LA      A1
      RS      18
      DV      TEMP1
      LS      31
      OJ      $+1
      SA      TEMP1
      LIRI    -24
L1    INC
      LS      1
      OJ      L2
      BIZ     L3
      B       L1
L2    RSL     7
      SA      C1
      CX      -18
      CA      1
      SIR     TEMP1
      SB      TEMP1
      AD      A2
      SB      B2
      SA      C2
      B       L4
L3    CX      0
      CA      0
      SA      C1
      SA      C2
L4    EQU     $
      END
```

APPENDIX E

INSTRUCTION SET FOR THE ATHENA COMPUTER

LITERAL CODE	OCTAL CODE	PROGRAM CONTROL REGISTER	INSTRUCTION DESCRIPTION
AC	15-	001 101 XXX XXX XXX XXX	Transmit rightmost 12 bits of instruction word to the X register at bit positions X_{12-23}. Add quantity of X to the accumulator.
AD	064	000 110 100 dCC CCC CCC	Add to contents of accumulator the quantity in magnetic core storage and place results in accumulator. Check for overflow.
B	20-	010 00D DDD DDD DDD DDD	Unconditional Branch. Same as UJ.
BIZ	30-	011 00D DDD DDD DDD DDD	Branch (Jump) if contents of index register is Zero. Same as JIZ.
BL	22-	010 01D DDD DDD DDD DDD	Branch if less than. Same as BLZ and SJ.
BLZ	22-	010 01D DDD DDD DDD DDD	Branch if less than zero. The same as BL and SJ.
BNE	24-	010 10D DDD DDD DDD DDD	Branch if not equal. Identical to BNEZ and ZJ.
BNEZ	24-	010 10D DDD DDD DDD DDD	Branch if not equal to zero. Branches if the quantity in the accumulator is not zero. Identical to BNE and ZJ.
CA	14-	001 100 XXX XXX XXX XXX	Transmit the rightmost 12 bits of instruction word to the X register at bit positions X_{12-23}. Clear the accumulator and add the quantity in X to the accumulator.
CJ	36-,37-	011 11D 000 000 000 000	Coefficient Jump. Jump to a drum group sector in drum groups zero, one, or two for an octal code of "36" or drum groups four, five, and six for an octal code of "37": the specific sector being determined by the TARGET pushbutton selected.

LITERAL CODE	OCTAL CODE	PROGRAM CONTROL REGISTER	INSTRUCTION DESCRIPTION
CX	12-	001 010 XXX XXX XXX XXX	Transmit the rightmost 12 bits of instruction word to the X register at bit positions X_{00-11}.
DA	417	100 001 111 000 000 PPP	Same as DD.
		OPTIONS:	
		PPP = 010	Same as TYPED.
		PPP = 011	Same as TYPEOL.
DD	017	000 001 111 000 000 PPP	Transmit data from a designated source to a designated display destination.
		OPTIONS:	
		PPP = 000	Same as TYPEOS.
		PPP = 001	Same as TYPEA.
		PPP = 010	Same as PRTD.
		PPP = 011	Same as PRTO.
		PPP = 100	Spaces digital printer.
DV	112	001 001 010 dCC CCC CCC	Divide the contents of AQ by a quantity in magnetic core storage and place quotient in Q and the absolute value of the remainder in the accumulator.
INC	024002	000 010 100 000 000 010	Increment the index register (IR).
INDEX	024001	000 010 100 000 000 001	Index the following instruction.
INDEXI	024003	000 010 100 000 000 011	Index the following instruction, then increment the IR.

LITERAL CODE	OCTAL CODE	PROGRAM CONTROL REGISTER	INSTRUCTION DESCRIPTION
JIZ	30-	011 00D DDD DDD DDD DDD	Same as BIZ.
LA	060	000 110 000 dCC CCC CCC	Load Accumulator. Clears the accumulator and adds the quantity in magnetic core storage.
LCA	062	000 110 010 dCC CCC CCC	Load Complement into Accumulator. Clears the accumulator and subtracts the quantity in magnetic core storage.
LIR	026	000 010 110 dCC CCC CCC	Load Index Register from core storage.
LIRI	52-	101 01X XXX XXX XXX XXX	Load Index Register Immediate.
LIRX	426	100 010 110 ddd ddd ddd	Load Index Register from Exchange (X) Register.
LS	104	001 000 100 000 0XX XXX	Left Shift the quantity AQ by K places ($0 \leq K \leq 31$). Check for overflow during shifting.
LV	026	000 010 110 dCC CCC CCC	Same as LIR.
MP	110	001 001 000 dCC CCC CCC	Multiply the contents in the accumulator by the quantity in magnetic core storage and place the result in AQ.
MS	32-	011 01S SSS	Manual Stop selective in maintenance condition or in standby condition for pre-guidance operations only.
OC	114	001 001 100 dCC CCC CCC	If an overflow has occurred since the last OC, OJ, or WC instruction, store this information.
OJ	26-	010 11D DDD DDD DDD DDD	Overflow Jump.
PRTD	017002	000 001 111 000 000 010	Prints 32 least significant bits at AQ in decimal form on digital printer (sign and 7 digits).

LITERAL CODE	OCTAL CODE	PROGRAM CONTROL REGISTER	INSTRUCTION DESCRIPTION
PRTO	017003	000 001 111 000 000 011	Prints contents of A1 in octal form on digital printer (8 digits).
RL	66-	110 11D DDD DDD DDD DDD	Transmit twelve bits of drum word to lower twelve bits of A1.
RS	106	001 000 110 000 0XX XXX	Right Shift the quantity AQ by K places ($0 \leq K \leq 31_0$). Algebraic sign in A_{23} is retained and transferred to A_{22}.
RSL	506	101 000 110 000 0XX XXX	Right Shift the quantity AQ by K places ($0 \leq K \leq 31_0$). Complement of algebraic sign is retained (zeros shifted in from the left if negative, ones if positive)
RU	70-	111 00D DDD DDD DDD DDD	Transmit twelve bits of drum word to upper twelve bits of A1.
RW	76-	111 11D DDD DDD DDD DDD	Transmit drum word to lower 18 bits of A1.
SA	004	000 000 100 dCC CCC CCC	Store the contents of the accumulator in magnetic core storage.
SB	066	000 110 110 dCC CCC CCC	Subtract from the contents of the accumulator the quantity in magnetic core storage and place the results in the accumulator. Check for overflow.
SI	006	000 000 110 dCC CCC CCC	Store the input data in magnetic core storage.
SIR	404	100 000 100 dCC CCC CCC	Store contents of index register in core storage.
SJ	22-	010 01D DDD DDD DDD DDD	Sign Jump. Same as BLZ and BL.

LITERAL CODE	OCTAL CODE	PROGRAM CONTROL REGISTER	INSTRUCTION DESCRIPTION
TD	024	000 010 100 000 000 0II	Transmit the lower twelve bits of the PCR to the Discrete Register.
		OPTIONS:	
		II = 00	No effect.
		II = 01	Same as INDEX.
		II = 10	Same as INC.
		II = 11	Same as INDEXI.
TJ	30-	011 00D DDD DDD DDD DDD	Test Jump; selective in maintenance condition only.
TP	060	000 110 000 dCC CCC CCC	Same as LA.
TQ	102	001 000 010 000 0XX XXX	Left Shift the quantity in AQ in K places ($0 \leq K \leq 37_8$). After shifting place algebraic sign from A_{24} to A_{23}.
TN	062	000 110 010 dCC CCC CCC	Same as LCA.
TYPEA	017001	000 001 111 000 000 001	Types 18 least significant bits of A1 in alphanumeric form on flexowriter (3 characters).
TYPED	417002	100 001 111 000 000 010	Types on the flexowriter the contents of the accumulator (bits 8-0) and Q register (bits 23-0) in BCD form (sign and 7 digits).
TYPEOL	417003	100 001 111 000 000 011	Types on flexowriter the content of the accumulator in octal form (8 digits).
TYPEOS	017000	000 001 111 000 000 000	Types 18 least significant bits of A1 in octal form on flexowriter (6 digits).

LITERAL CODE	OCTAL CODE	PROGRAM CONTROL REGISTER			INSTRUCTION DESCRIPTION
UJ	20-	010 00D DDD	DDD DDD DDD		Unconditional Jump. Same as B.
WC	000	000 000 000			Wait for computation cycle sync.
WL	60-	110 00D DDD	DDD DDD DDD		Transmit lower twelve bits of A1 to drum storage.
WP	34-	011 10D DDD	DDD DDD DDD		Wait for Partial Cycle Sync.
WU	64-	110 10D DDD	DDD DDD DDD		Transmit upper twelve bits of A1 to drum storage.
WW	62-	110 01D DDD	DDD DDD DDD		Transmit lower 18 bits of A1 to drum storage.
ZJ	24-	010 10D DDD	DDD DDD DDD		Non-Zero Jump. Same as BNEZ and BNÈ.

CODE:

d	Don't care
C	Core address
P	Print mode
I	Index control
X	Constant
D	Drum address
S	Identification