THE DESIGN AND IMPLEMENTATION OF A TWO PASS ASSEMBLER

THESIS

Presented to the Graduate Council of
Southwest Texas State University
in Partial Fulfillment of
the Requirements

For the Degree of
MASTER OF SCIENCE

By

Jack Joseph Murphy
San Marcos, Texas

August, 1976

TABLE OF CONTENTS

LIST OF FIGURES

PREFACE

This thesis is written with the student in mind. It was conceived to be a study in the design of one of the most basic systems programs, namely the assembler. My motivation in writing this thesis came after studying assembly language programming and systems programming almost independently of each other. Very rarely is a student given the opportunity to actually code and implement systems software. The material in this thesis provides just such an opportunity. In fact, the novice assembly language programmer should be able to write the assembler defined in the following chapters. The appendices provide projects that will help the student develop his assembler. It is hoped that these chapters and the appendices may be used in a classroom environment to teach assembly language in a way that the student sees and appreciates the assembler that he or she is using by building another assembler. It is not the purpose of this thesis to study everything there is to know about assemblers or assembly languages, however, it is a good first step.

It is very difficult to thank the myraids of people who helped in the research and writing of this thesis. As the reader will discover, each machine has hundreds of idiosyncrasies that are not found in the manuals. When a problem of this nature occurs and the answer is not in a text-

book, the student should do as the author has done--ask a friend. Of the many friends who provided assistance, the writer especially wishes to express his appreciation to the supervising professor, Dr. Grady G. Early, Assistant Professor of Mathematics and Computer Science, for his guidance and assistance in the preparation of the material presented here. Also, he wishes to express his gratitude to Dr. James L. Poirot, Assistant Professor of Mathematics and Computer Science, for his careful reviewing of the material and to thank Dr. Henry N. McEwen, Professor of Mathematics and Computer Science, for serving on the committee and for his cooperation and assistance. Finally, the writer wishes to express his deep gratitude and appreciation to his parents, Mr. and Mrs. John D. Murphy, for their care, encouragement, and continuous support.

J.J.M.

Southwest Texas State University
San Marcos, Texas
August, 1976

CHAPTER I

PRELIMINARY CONCEPTS

INTRODUCTION

The intent of this thesis is to study the purpose, design, and implementation of an assembler. In Chapter I we present some basic concepts which motivate the creation of an assembler and we discuss such basic ideas as the internally stored program concept. Several types of assemblers are in existence; some whose purposes will be considered in this chapter. The material presented in this chapter will show that an assembler is a necessary systems program. After reading this thesis and studying the appendices the reader should be able to develop a simple assembler of his own.

In Chapter II a ficticious but typical machine and associated instruction set is defined. Chapter II will serve as a programming manual for the assembly language which we will call the JACK-1 assembly language, named after our hypothetical machine, the JACK-1 computer. The assembly language contains some "pseudo ops" that are representative of many machines, thus allowing programming flexibility as well as an opportunity to investigate how an assembler handles such instructions.

After the JACK-1 assembly language and the JACK-1 computer have been defined the data base for the assembler will be defined in Chapter III. Table formats and record formats will be defined in addition to a discussion of sorting algorithms, insertion algorithms, and search procedures which are a very important part of any assembler. To show the overall process involved, Chapter III concludes with a discussion of a more sophisticated algorithm which combines advantages of the other algorithms discussed.

In Chapter IV the necessary algorithms and subroutines are developed. To acquaint the programmer with fundamental concepts, a general algorithm is provided. While reading this chapter the reader should refer often to the appendices which present projects that ultimately lead to the completion of an assembler.

Since the machine and language are hypothetical, simulation will be used to test the output for validity. The PDP-11 computer will be used for this purpose since it is available. It is assumed that the reader is familiar with the assembly language of the PDP-11, although this is not required.

INTERNALLY STORED PROGRAM

The internally stored program was introduced by mathematician Jon Von Neumann in the late 1940's. Prior to this time all programming was done by means of hardwired circuits.

Programs written after the introduction of the Von Neumann machine were coded into the machine's primary memory as binary 1's and 0's. Each word of memory contained an instruction or part of an instruction which was the binary representation of a machine executable instruction. The programmer had to know the binary code of all instructions as well as the absolute memory address of any operands. After the programmer had devised an algorithm and flowcharted the necessary steps, he then coded his program in machine language; that is, each instruction was coded into a machine language instruction interpretable by the control unit of the Central Processing Unit (CPU). Both instructions and data had to be coded in this manner. When he had finished coding the program, his pad of paper was an image of primary memory as it would appear in the machine immediately prior to execution of the program. He then performed the following sequence of steps to load and execute the program:

(1) Toggle into the Switch Register (SR) the initial program load address (IPLA), push the load address button to deposit the contents of the Switch Register into the Address Register (AR).

(2) Toggle the next instruction into the Switch Register.

(3) Depress the Deposit Button (DEP) to deposit the contents of the Switch Register in primary memory at the address specified in the Address Register.

(4) Push an increment button to add 1 to the Address

Register so the next instruction will be loaded at the proper address.

(5) If not last instruction go to (2).

(6) Toggle into the SR the initial program load address (address of first instruction to execute). Push a load address button to deposit this address into the Address Register (AR).

(7) Push the start button to begin execution of the program.

This procedure was difficult and tedious. If the program did not produce the expected results, the machine language programmer had to "debug" his program--a very difficult task which was sure to produce a headache if nothing else. Despite its difficulties, the concept of an internally stored program was one of the most significant advances in computer technology. Surely it was better than sitting with a handful of wires and a giant circuit board trying to decide which wire to plug in which hole. The internally stored program was here to stay. New programming techniques such as address modification were made possible for the first time.

ADDRESS MODIFICATION

The internally stored program concept permits instructions as well as data to be stored simultaneously in memory.

In fact it is difficult to distinguish data from instructions; visual inspection alone of a specific address in memory is not sufficient to determine if that word is data or instruction. Actually a particular word could be interpreted as either or both depending on the context of a given program. The following MACRO-11 assembly language program, assumed to be loaded at memory location 10_8, illustrates this capability. The technique is referred to as address modification. It is not recommended as a general programming technique, but can occasionally be useful.

ABSOLUTE ADDRESS	CONTENTS		INSTRUCTION
10	062767	A:	ADD #10,72
12	000010		
14	000054		
16	062767		ADD #2,14
20	000002		
22	177770		
24	000771		BR A
26			

Notice that the first instruction requires three consecutive memory locations for storage. The first word of the instruction signals the controller that this is an add instruction and therefore requires two operands. The low order byte indicates that the first operand is in the next consecutive word, and the address of the second operand is to be obtained by adding the contents of the third word of

the instruction and the contents of the program counter. Upon execution of this instruction, 10 will be added to the contents of memory location 72_8. The second instruction has a similar form, but consider the consequences upon execution: the number 2 will be added to the contents of memory location 14_8. Notice that 14_8 was previously part of an instruction and is now the operand. After the branch is executed the next instruction to be executed is the first ADD instruction. But this instruction has been changed! It is now equivalent to:

```
A:      ADD #10,74      062767
                        000010
                        000056.
```

This program, then, adds 10_8 to each memory location starting with 72_8. Hence that which is data, and that which is instruction can be determined only in the context of a specific program. This type of programming was not possible before the internally stored program was invented, because programs were previously hardwired and hence unalterable.

RELOCATION

The problem of relocation is very important, especially if a program is to be used as a subroutine by many different main programs, which is frequently the case. To understand this problem consider the third word of the first add

instruction. The number 10_8 is to be added to memory location 72_8, not 54_8, because the PDP-11 relative addressing mode is used. Fifty-four is the address of the operand relative to the current program counter (PC). To get the absolute address, 16_8 (the current content of the PC) must be added to 54_8 giving 72_8, the desired absolute memory address. Relative addressing mode is a must if we are to allow symbolic addressing (a topic to be discussed later); therefore, it should be understood. In terms of programming, relative addressing means the third word (of the ADD instruction in our example) must be changed if the program is to be stored anywhere else in memory and that the machine language version must be loaded beginning at address 10. Hence the machine language programmer must be concerned with the absolute memory addresses the program will occupy. If he must load the program starting at address 20 then he must change the third word of the first instruction and second instruction accordingly. But notice the symbolically coded program. The code remains the same whether it is loaded at address 0 or at address 024610. The assembler, by means of an assembly location counter (ALC) can keep track of relative addresses and record all address dependent constants (constants who value depends on where the program is loaded) that need to be altered before execution. This is extremely important if the routine is to be used as a subroutine for several different programs and must reside in memory wherever there is space available.

SYMBOLIC ADDRESSING

The machine language programmer must compute the address of each operand. He frequently does not know the address until the program is almost completely coded. He must then go back and fill in all these addresses. Programming in assembly language circumvents this problem by allowing the programmer to assign symbolic names to memory locations. In the example given earlier, there was a symbolic label on the first instruction. Presumably this location will be referenced at some other point in the program. In fact, the third instruction commands the processor to transfer control to the instruction which is identified by the symbol A. A is thus a user defined symbol that corresponds to the memory address where the first ADD instruction will be located. Now consider the alternative machine code for the branch instruction. The offset in words must be computed and stored in the low order byte of the instruction. The transfer address is 7 words away and in the negative direction so -7 must be stored in the low order byte of the BR instruction. This is tedious work but to complicate matters suppose it is decided that an additional instruction should immediately precede the branch command. In this case the offset is no longer correct and must be recomputed. However, with a symbolic label no change in the existing instructions needs to be made. The assembler does the work. It maintains a table of user defined symbols and their

values so that instructions that reference these symbols can be translated. Thus symbolic addressing makes programming much simpler for the user.

THE ASSEMBLER

Coding a program in binary or octal digits, although much simpler than the old hardwired method, is still somewhat tedious and difficult. If the hardware of a machine can interpret a large number of instructions, then the programmer must learn the binary code for each of the commands. To further complicate matters these binary codes need have no logical connection with the operations they perform. As has already been shown, coding programs in this manner can be difficult, especially when the program does not function properly and needs to be debugged.

If each machine instruction had associated with it a three or four letter mneumonic which suggested the operation performed by that instruction, then reading, writing, and coding programs could be simplified. The association of a meaningful mneumonic to each machine instruction must be a bijective (one to one) relation. Remember, though, that the computer only understands machine language instructions, not manmade mneumonics that identify specific operations. Now, however, a program could be written in a symbolic language and translated into machine instructions because of the one to one correspondence between mneumonic instruction

and machine instruction. We still have the problem of translation. This translation could be done by hand, but the bijective nature of the mapping suggests that the process could be automated. A program could be written to perform the translation--a process called "assembling". This program should also be able to handle problems like symbolic addressing and relocation. The program to perform the translation is called an assembler. The purpose then of an assembler is to accept as input a file whose records contain mneumonic instructions and create as an output file a relocatable object program written in the binary code of the machine. This is a somewhat oversimplified statement of the assembler's purpose but is an adequate statement of the overall problem.

Some important remarks should be made about an assembler. First, it is a program like any other program, not a magic box; therefore, it must be designed and coded as such. But in what language is this "assembler" program written? No assembler exists to translate an assembly language program into an executable form which can then accept an assembly language program and translate it into a machine language program. We might therefore assume the assembler is written in machine language. Although the first assemblers were written this way, it is doubtful that any assemblers are written this way today. When a new machine is designed, it is usually simulated on another system. Hence, the assembler for one machine can be written on a different

machine which simulates the assembly language of the new machine.

Second, the assembly language program which is input to the assembler is not executed by the machine or by the assembler. The assembler's only function is to create a file which is the machine language version of the input file. A separate systems program called a linker is used to link all subroutines used and prepare them for execution. A loader then loads the final program and transfers control to the user program. Hence, the assembler must be provided at least two pieces of information: (1) the name and location of the input file, and (2) the name of the output file and where it is to be saved for later execution. The input for the assembler is called the source program; the output file is called an object program or relocatable file.

Third, the assembler could conceivably load the program without the aid of the linker or loader, but there are some very important objections to this. The assembler is generally very large compared to a loader; hence, there may not be space available in memory for both the assembler and the program to be executed. Furthermore, every time the program needs to be executed it would first have to be translated, an unnecessary procedure if the program is debugged. It is also a time consuming operation.

Fourth, the assembler, if written well, can detect some programming errors. Syntax errors can be detected, but it is impossible for the assembler to detect logic errors.

Fifth, unlike other programming languages such as FORTRAN and COBOL, an assembly language is not machine independent. This means the assembly language is unique to the machine for which it was designed. This is unfortunate but any attempt to create a standard assembly language would be foolhardy since each machine has different capabilities and hardware configurations.

CHAPTER II

MACHINE AND ASSEMBLY LANGUAGE

We now define the hardware and instruction set for our hypothetical computer. During the design stage of any piece of hardware the manufacturer must decide on the characteristics that make the machine marketable. These hardware characteristics are primarily dictated by user needs and satisfaction. Some hardware considerations are: speed, timesharing capability, efficient I/O processing and expandable memory units. These constraints, although necessary to the function the machine is to perform, may sometimes result in machine and assembly languages that are very complex. Since our purpose is to explore the basic components of an assembler, however, the machine which we will define shall contain the basic components in computer design, without an extensive instruction set which would serve only to increase the complexity of the assembler without a resulting increase in the benefits gained.

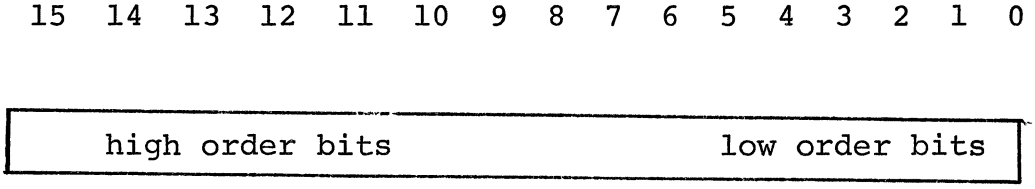
This fictitious machine, dubbed JACK-1, is a small but typical machine. It is very similar in design to the PDP-8 manufactured by Digital Equipment Corporation. The PDP-8 was chosen as a pattern for the JACK-1 for several reasons. First, it is a small simple machine, but it is very powerful

in the functions it is capable of performing. Second, the hardware configuration is simple yet typical of many minicomputers. Also, the instruction set is limited to eight basic machine instructions.

MEMORY

The primary means of storing data on the JACK-1 is the memory unit. The machine is a stored program computer; consequently the memory unit must often contain machine instructions as well as any data in the form of ASCII characters or two's complement binary numbers. The basic unit of data storage is the word which on the JACK-1 consists of 16 binary digits. There are 4K (4096) addressable words of primary storage each of which can be accessed as readily as any other. With 4096_{10} words of primary storage it is easy to see that 12 bits are required to address any word. Some larger machines can uniquely address halfwords called bytes thus doubling the number of uniquely addressable locations. However, it should also be apparent that an additional bit is required each time the number of addressable locations is doubled. Since only 12 bits are required for an address then 4 bits are free to contain an operation code. On the JACK-1 each instruction requires exactly one word of storage thus simplifying the assembly process. Numerical data on the JACK-1 is stored in two's complement form allowing integer values in the range -32,768 to 32,767. This range is

quite good for most applications on minicomputers. The following diagram illustrates how bits will be numbered in the remainder of this thesis.



THE ACCUMULATOR (A)

All computers have special purpose registers, some of which are accessible to the programmer and some that are not. All registers on the JACK-1 are high speed storage devices that are not a part of primary storage. General purpose registers are the type available for programmer use. There is only one general purpose register in the JACK-1. This is called the accumulator or the A register. It is given this name because its primary purpose by a user is to accumulate sums and differences. It is also used for various logical operations such as testing and branching. The A register may be thought of as a scratch pad. A typical memory reference instruction requires two pieces of data, but there is only room in the instruction for one address; the accumulator is assumed to be the other operand. In general the operand that is altered is called the "destination" operand. The unaltered operand is called the "source" operand. The accumulator is the primary means of performing

arithmetic in the computer. An extension of the A register called the link bit is used to test the state of a previous operation in the accumulator. Special instructions are available to operate on the link bit. For example the link bit is complemented if a carry out of the high order bit was necessary in an add instruction.

THE INSTRUCTION REGISTER (IR)

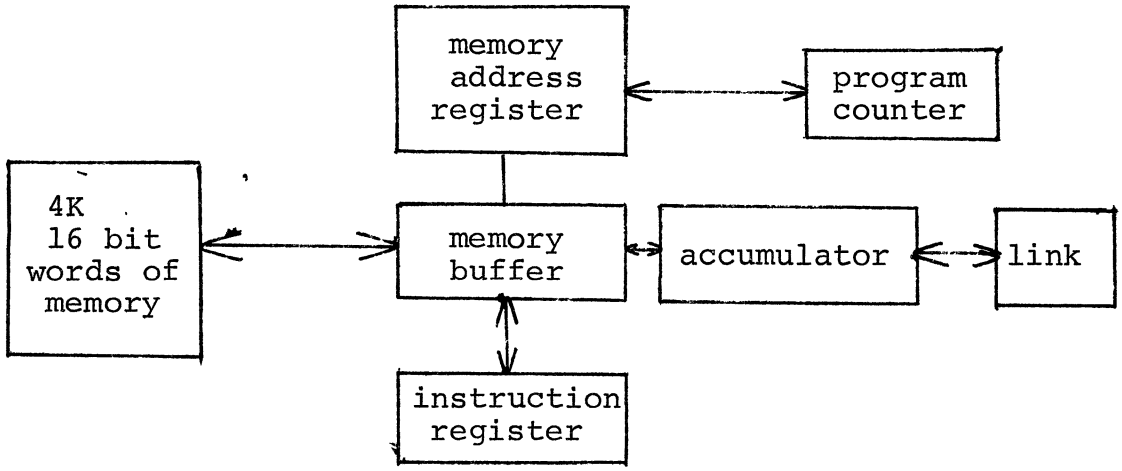
The instruction register is a register not directly accessible to the programmer. It contains a copy of the current instruction being executed. It can be considered a hardware memory device. Before an instruction can be executed it must first be fetched from primary memory and sent to the instruction register to be decoded. The JACK-1 is constantly in either of two states. In the fetch state the control section is fetching an instruction of data from primary storage. During the execution state of the computer the function specified in the current instruction is being executed.

THE PROGRAM COUNTER (PC)

The program counter is a special 12-bit register that contains the location of the next instruction to be executed. It is initialized by a loader program to the first instruction to be executed. Since each instruction is only one word long and instructions are stored consecutively in memory,

the PC is incremented by one after each instruction fetch. When a branch is executed, the control unit resets the PC to the effective address of the desired branch, thus pointing to the instruction specified by the branch. The PC is the only means by which the programmer may alter the normal flow of control in the program.

There are many other registers internal to the CPU that serve such special purposes as the interpretation of instructions, control of I/O devices and buffer registers. However, excluding the registers mentioned above, the rest are transparent to the user. The following diagram illustrates the conceptual relationship between hardware components.



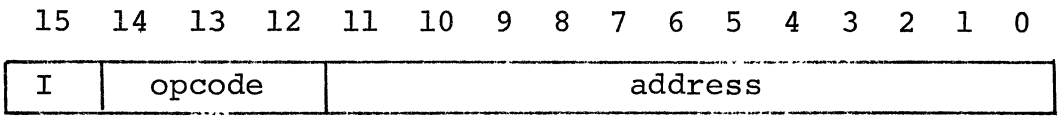
INSTRUCTION SET

There are three broad categories of instructions available on the JACK-1. (1) Memory Reference Instructions allow the user to operate on data stored in the memory unit.

(2) Micro Instructions allow bit manipulation, testing, and branching. (3) I/O Instructions enable the programmer to perform simple data transfers to and from peripheral devices.

MEMORY REFERENCE INSTRUCTIONS

There are six memory reference instructions having the following format.



I = 1: address is pointer to effective address

I = 0: address is absolute memory address

The three bit opcode field contains an octal number in the range 0 through 5 designating which of the six memory reference instructions is to be executed. The 12 bit address portion of the instruction contains the absolute memory address of the operand or the address of the address if the I bit is set to 1. In the following description of each instruction, the mneumonic indicated is the code recognized by the assembler corresponding to the machine translation indicated. Also nnnn is the absolute memory address of the operand or a pointer to the location containing the absolute address depending on whether I = 1 or I = 0.

(1) AND IOnnnn A ← A ∧ I(nnnn)

The AND instruction causes a bit by bit boolean "and" of the accumulator and the contents of the effective address. The result is left in the accumulator. Only the A register is altered.

(2) TAD I1nnnn A ← A + I(nnnn)

The TAD instruction performs a two's complement add of the accumulator and the contents of the effective address. The result is left in the A register. If the addition results in a carry out of the high-order bit the link bit is complemented.

(3) ISZ I2nnnn I(nnnn) ← I(nnnn) + 1, if I(nnnn) = 0 PC ← PC + 1

The ISZ instruction is an increment and skip if result is zero. The contents of the effective address is incremented and the result compared to zero. If the new contents of the effective address is zero, the PC is incremented thus skipping the next instruction.

(4) DCA I3nnnn I(nnnn) ← A, A ← 0

The DCA instruction deposits the contents of the A register in the effective address and then sets the A register to all zeroes. The old contents of the effective address are lost as a result of this operation.

(5) JMS I4nnnn I(nnnn) ← PC, PC ← #I(nnnn) +1

The JMS instruction allows the user to jump to a

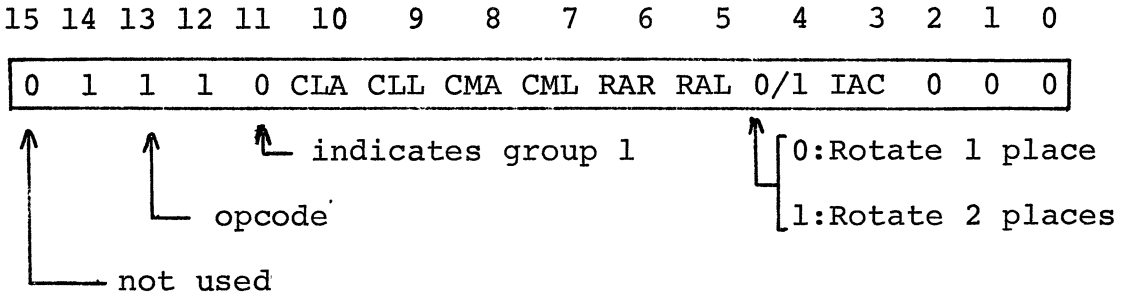
subroutine and save the return address in the first word of the subroutine. The PC is saved in the effective address and then control is transferred to the instruction immediately following the effective address. To return from a subroutine an indirect jump may be made to the word containing the return address.

(6) JMP I5nnnn PC ← I(nnnn)

The JMP instruction transfers control to the effective address by setting the PC to that effective address. Unlike the JMS instruction no return address is saved.

MICRO INSTRUCTIONS

Micro Instructions, which do not require operands, allow the user to manipulate and/or test the data that is stored in the accumulator and link bit. They may also be used to branch on certain conditions. The opcode for all micro instructions is 7_8 in bits 12-14, however the micro instructions may be subdivided into two groups. Group 1 instructions alter the contents of the A register and the link bit. The following diagram illustrates the use of group 1 instructions.



It is seen from this diagram that the entire word is used to specify which operation is to be performed. This is possible since no operand is required for these instructions. Bits 5-10 and bit 3 specify the operation indicated below.

CLA Clear the accumulator.

CLL Clear the link bit.

CMA Compliment the accumulator.

CML Compliment the link bit.

RAR Rotate the accumulator and link to the right one position. The least significant bit (bit 0) replaces the link bit. The link bit is shifted into the high order bit of the A register.

RTR Rotate the accumulator and link to the right two positions. This is equivalent to executing RAR twice.

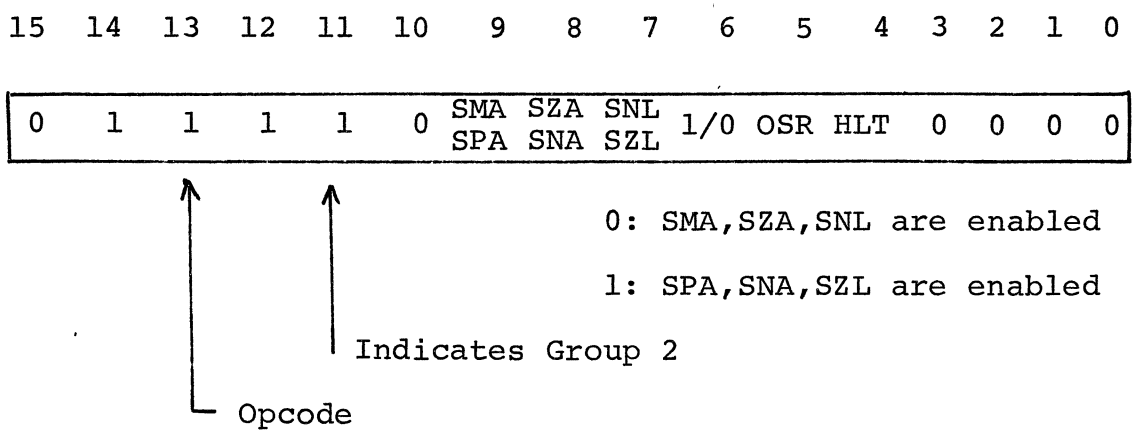
RAL Rotate left one position.

RTL Rotate left two positions.

IAC Increment the accumulator.

NOP If bit 0-11 are all set to zero no operation is performed.

Group 2 micro instructions are used to perform branches depending on certain conditions. These instructions test the state of the accumulator and link bit to skip the next instruction depending on the result of the test. Group 2 micro instructions have the following format.



Each bit that is set specified the following action to be performed.

SMA Skip next instruction if accumulator is negative.

SPA Skip next instruction if accumulator is positive.

SZA Skip next instruction if accumulator is zero.

SNA Skip next instruction if accumulator is not zero.

SNL Skip on non-zero link bit.

SZL Skip on zero link bit.

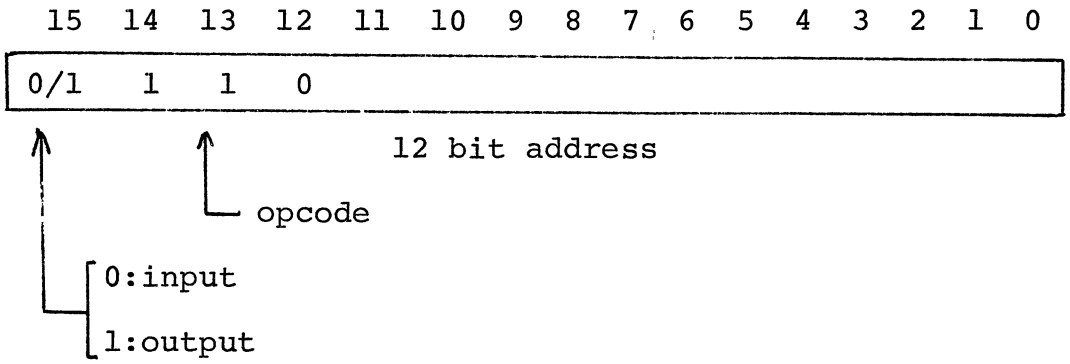
SKP Unconditional skip. This instruction is specified if all of bits 6-9 are clear.

OSR Inclusive "or" of switch register with A register. The result is left in the A register.

INPUT/OUTPUT INSTRUCTIONS

There are two basic instructions that allow data transfers from peripheral devices. In both I/O instructions control is not transferred back to the user until the data transfer is completed, therefore no waiting loops are

necessary. The diagram below illustrates the format for an I/O instruction. Notice that indirect addressing is not allowed on I/O instructions.



GET 16nnnn

This instruction places a card image, 80_{10} ASCII characters, in 40 consecutive memory locations beginning at the address specified in the operand field. Two characters are placed in each word. The low order half of the word is filled first, the next character is placed in the high order half of the word. For example the word "computer" would appear in core as:

O	C
P	M
T	U
R	E

.

PUT 06nnnn

This instruction causes 132_{10} ASCII characters beginning at the address specified to be printed on the line printer. Formatting of character strings is the same as the GET instruction. This means 66 words are assumed to

contain 132 characters. Carriage return and line feed characters are supplied, therefore the programmer need not insert these characters.

DEFINITION OF JACK-1 ASSEMBLY LANGUAGE

A source program is composed of a sequence of source lines each of which contains a single assembly language statement. All source programs used as input to the assembler must be disk resident contiguous files in which each record contains exactly 80 characters followed by a carriage return and a line feed character. It is suggested that the file utility program (PIP on the PDP-11) be used with the appropriate switches to create a source program satisfying the above constraints. The user creates the output file by specifying its name in the output field of a command string. For example:

```
$RUN PAL8
#OUTFIL.OBJ<INFILE.PAL
```

A command string must appear immediately following $RUN PAL8 and the command string must contain exactly one input file and one output file. No switches are permitted.

STATEMENT FORMAT

Each record contains a single 80 character assembly language statement. Each statement may contain up to four

fields which are identified by special characters and the order of appearance on the card. The general format for a JACK-1 assembly language source statement is:

<label>, <operation> <operand> /<comment>.

LABEL FIELD

A label (optional) is a user defined symbol which is assigned a value at assembly time by the assembly location counter. The label and its value are entered into the user symbol table created by the assembler during PASS1. JACK-1 does not distinguish absolute from relocatable user defined symbols. The label is a symbolic means of referencing a particular location in the text of a source program. If a label appears it must appear first on the card and must be terminated by a comma. Only one label per source card is permitted. Labels are composed of letters A-Z and the digits 0-9. However, a label must begin with an alphabetic character. Labels may contain more than 6 characters, however each label must be unique in the first 6 characters.

OPERATION FIELD

Every source line must contain an operation field. This field is the only field that may not be omitted. Legal operations that may appear in the operation field include: (1) any of the instructions defined in the preceding pages

of this chapter, or (2) assembler directives, called pseudo ops, defined later in this chapter. Pseudo ops are instructions that direct the assembler to perform certain functions during assembly. Pseudo ops are not executed by the machine, they merely aid in the translation process. In all cases the mneumonic that appears in this field must be followed by at least 1 blank. Only one operation may be specified per source line.

OPERAND FIELD

The operand field may or may not be present depending on which operation is specified in the operation field. Only memory reference instructions and some pseudo ops may have operands. Valid operands include symbolic labels defined in the label field of some instruction or octal or decimal numbers on some pseudo ops. On memory reference instructions an operand may be declared "indirect" by inserting the character "I" between the operation code and the operand. The assembler will recognize this and set bit 15 accordingly. The character "I" must be preceded and followed by at least one blank character. The effect of declaring an operand indirect is the topic of a later section.

Example: DCA I NUMB

This declares NUMB to be an indirect operand of DCA. The following example illustrates how I may be used as an

indirect indicator as well as an operand. The second I is assumed to be a symbolic name defined in the label field of some source line, while the first I indicates this operand is declared indirect.

Example: DCA I I

COMMENTS

Comments, if they appear, must be the last field on the card. The character "/" indicates the beginning of a comment. Comments are for the benefit of user documentation, therefore any characters in the comment field are ignored by the assembler.

Example:

```
LABEL,   TAD I NUMB   / THIS IS AN EXAMPLE ITS VALID
```

ASSEMBLER DIRECTIVES

Assembler directives are "pseudo-operations" which are processed by the assembler. They aid in the translation of the program, but have no machine language equivalent. Pseudo ops are used to allocate storage and define the contents of that storage. Also, since source programs are of varying length, we must have a pseudo instruction which indicates the end of the source deck. Such an instruction has no machine language equivalent, however it does contain

important information necessary for the assembler. The purpose of each pseudo op is given below.

OCTL The octal pseudo op contains one operand which consists of a string of ASCII digits representing the octal integers from 0 to 77777. The purpose of the OCTL pseudo op is to define one word of storage containing the value indicated in the operand.

DCML The DCML pseudo op fulfills the same purpose as the OCTL but allows the user to express the operand as a decimal integer in the range 0 to 32767. No decimal point is permitted since the instruction already assumes decimal.

CHAR This pseudo op permits the user to define storage to contain the 26 characters, the blank, and the 10 decimal digits. The operand consists of two characters to be stored in the assigned word preceded immediately by a single quote.

END The END pseudo instruction contains one operand. The END pseudo op designates the physical end of the program source deck. It must therefore always be the last card in a source deck. The operand must be a user symbol identifying the first instruction to be executed when the program is loaded and ready to run. This location is called the transfer address and need not be physically first in the program.

PROGRAMMING EXAMPLES

Even though the instruction set for the JACK-1 is fairly small, many programming techniques are possible. Some of these techniques are illustrated below.

Example 1: Move X to Y

```
    CLA          / CLEAR THE A REGISTER
    TAD X        / ADD X
    DCA Y        / MOVE X TO Y
```

Example 2: Subtraction. Subtraction is done by using the two's complement and adding. To subtract X from Y and leave the result in Z.

```
X,  DCML 23
Y,  DCML 25
Z,  DCML  0
    CLA          / CLEAR THE A REGISTER
    TAD X        / PUT X IN THE A REGISTER
    CMA          / COMPLEMENT A
    IAC          / NOW -X IS IN A REGISTER
    TAD Y        / A REGISTER NOW HAS X - Y
    DCA Z        / STORE IT IN Z
```

Example 3: Comparisons. To compare two numbers X and Y subtract and compare to zero. This program compares X and Y and branches to GREATER if $X \geq Y$ or LESS if $X < Y$.

```
CLA             / CLEAR THE A REGISTER
TAD Y           / MOVE Y TO A REGISTER
CMA
IAC             / -Y IS IN A REGISTER
TAD X           / X - Y IS IN A REGISTER
SMA             / SKIP NEXT INSTRUCTION IF X - Y < 0
JMP GREATER     / X - Y ≥ 0   X ≥ Y
JMP LESS        / X - Y < 0   x < Y
```

USE OF INDIRECT ADDRESSING

If an operand is declared indirect by inserting I between the mneumonic and the operand then the specified operand is assumed to be a pointer to the effective operand. An indirect operand (pointer address) identifies the location which contains the effective address. When the instruction is executed the processor will detect this condition and interpret the value in the address field of the instruction to be a pointer to a location which is assumed to contain the effective address. Such an addressing mode is particularly useful especially in the use of subroutines and array processing. The following example illustrates how indirect addressing may be used to return from a subroutine and to simplify handling arguments. The example puts A - B in C.

Example 4:

```
    JMS ABS
A,  DCML 12
```

```
B,      DCML 13
C,      DCML 0

ABS,    NOP             / POINTER TO ARGUMENT AND RETURN
        CLA
        TAD I ABS       / PUT A IN A REGISTER
        DCA TEMP        / SAVE A
        ISZ ABS         / POINT AT NEXT ARGUMENT
        TAD I ABS       / PUT B IN A REGISTER
        CLA
        IAC             / A REGISTER CONTAINS -B
        TAD TEMP        / NOW IT CONTAINS A-B
        ISZ ABS         / POINT TO C
        SMA
        JMP RETURN      / ONLY IF A - B ≥ 0
        CMA             / OTHERWISE
        IAC             / FIND ABSOLUTE VALUE
RETURN, DCA I ABS       / PUT RESULT IN C
        ISZ ABS         / POINT TO NEXT INSTRUCTION
        JMP I ABS       / RETURN
```

OUTPUT OF JACK-1

JACK-1 provides the programmer with the following:

(1) A list of the user symbol table.

(2) Program length and transfer address.

(3) A listing of the program including any error messages relating to syntax or semantics.

(4) An object module.

The object module is a disk resident contiguous file identified by the name specified in the output specification of the command string. The first word of the object module contains the transfer address. The second word contains the program length. The remainder of the object module contains the machine text code for the source program. Since all addressing is absolute the module may be loaded directly and executed. Execution and loading are the topic of Chapter IV.

CHAPTER III

DATA BASES AND ALGORITHMS

Several data bases or tables are required for assembly and several algorithms capable of processing these data bases are also required. Such algorithms may be considered small but important subroutines to a larger algorithm--the assembler itself. Chapter IV presents a detailed study of the assembler algorithm; however, it is beneficial at this point to present an overview of the assembly process.

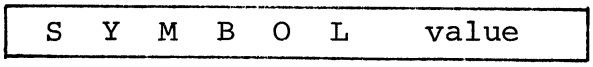
Our assembler processes each record or statement twice and consequently is called a two pass assembler. To facilitate this task the first phase of the assembler called PASS1 must save a card image of each statement so it may be processed again by PASS2. This is easy to implement since we require the source file to be a disk resident random access file. All that is required is to reset a pointer at the end of PASS1 to indicate that the next input will be the first record in the file. Although processing during each pass is sequential, the contiguous nature of the file allows rapid reread capability. The primary function of PASS1 is to collect all alphanumeric symbols found in the label field, assign an addressing value, and put them in a user symbol table, the UST. These values must correspond to the

relative address (equivalent to absolute address in this problem) of the instruction or data identified by the symbol.

During PASS2 the actual translation of the assembly language code is performed. This translation must be delayed until PASS2 because of forward referencing of symbols which would otherwise have no value assigned to them. The machine code which is the most important output from the assembler is stored in another contiguous disk file called a load module. Once loaded into primary memory the load module is ready for execution. In the problem presented here the function of loading and execution is done by a simulator program, a topic to be treated in the second half of Chapter IV.

THE USER SYMBOL TABLE (UST)

The UST contains a copy of all symbols (labels) and their values. The following diagram describes the format of each entry in the symbol table. Since each symbol must



be unique in the first six characters, the symbol field of the record must be six bytes long (on the PDP-11). Characters in the symbol field are left justified and padded with blanks. Each record in the UST requires four words, three for the symbol and one for the value. In the example program in the preceding chapter there are four user defined

symbols: A, B, C, and ABS whose relative addresses are 1, 2, 3, and 5 respectively. The symbol table for this program might appear as is shown here. Notice that the symbol table

A	1
B	2
C	3
A B S	5

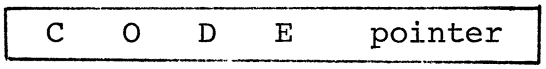
illustrated above is not ordered. For small programs such as the one illustrated it is probably unwise to spend time sorting this table; however, for large programs some order must be imposed on the table before it is used by PASS2. To improve speed of processing it becomes necessary to keep the UST core resident at all times. Consequently, this table must be of some fixed size. Space can be allocated as symbols are found, however, this is difficult to manage. The obvious alternative is to allocate some fixed number of consecutive memory locations capable of containing some sufficiently large number of records. For this application 4096_{10} words is more than adequate since a possible 1024_{10} symbols can be contained in this space. This allocation is accomplished by a single command as shown here.

```
UST:      .BLKW 4096.
```

THE MACHINE OP TABLE (MOT)

The assembler must maintain a permanent list of

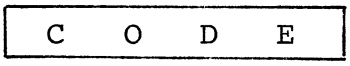
allowable operation mneumonics. Such a list shall be divided into two distinct tables: the Machine Op Table (MOT) and the Pseudo Op Table (POT). The latter of which contains only valid pseudo ops. The MOT is unaltered during execution of the assembler and is only referenced during PASS2. The construction of the MOT must include in each entry the ASCII code identifying a valid instruction; however, to successfully perform the translation to machine language we might also include the machine translation as part of the entry. This is possible but an alternative might be to include instead the address of an action routine to perform the translation. This is desirable since some instructions (MRI's) contain an address field which should be built into the instruction at the same time. This could be accomplished by appropriate action routines. The following diagram illustrates the format of each entry in the MOT.



As with the UST, the code field contains the ASCII code left justified. Each entry in the MOT occupies three words of memory, and there are a fixed number of valid opcodes. Therefore a permanently core resident block of 96 consecutive words is required to contain all 32 valid instructions as defined in Chapter II. The appendices show the table in its entirety. Notice that the table is preordered for faster searching.

THE PSEUDO OP TABLE (POT)

The pseudo op table is referenced during both passes of the assembler, however, it is primarily used during PASS2 to allocate and define the contents of storage locations. Since each pseudo op requires special processing, we define the POT so that each entry has the format shown here.



Since pseudo ops have no direct machine translation it is not possible to include such a code in the entry. Similarly in the algorithm for this assembler a single subroutine is called when a pseudo op is encountered, consequently each entry in the POT need only contain the mneumonic itself and the action taken will be determined by the subroutine. The entire POT is shown here.

```
POT:    .ASCII /DCML/
        .ASCII /END /
        .ASCII /OCTL/
        .ASCII /CHAR/
```

CHARACTER CODE TABLE (CODES)

One additional table whose function is partially incidental to the system we are using is a table of character codes. Each entry in the table contains the ASCII code and the card reader code for each legal character in the assembly

language. This table's only function is to facilitate I/O. Each entry in the table has the following format. Each field

card code	ASCII code

in the record occupies one word. The complete table is provided in the appendices.

THE LOAD MODULE

The primary purpose of the assembler is to create a load module as described in earlier sections. This module is actually a machine executable program which is stored in a contiguous disk file identified by the output specification of the command string for the assembler. Prior to execution of a load module the Loader/Simulator must be provided the following information:

(1) Name of object module.

(2) Program length.

(3) Transfer address (Address of first instruction to execute).

(4) Text of the program.

To accomplish this the following format is adopted for the object program:

length
transfer
text
text
text
.
.

The name of the object module is provided by means of the command string supplied to the Loader/Simulator. The rules for the object program are:

(1) It must occupy contiguous blocks on the disk.

(2) The first word of the first block must contain the number of words occupied by the text of the program.

(3) The second word is the relative address of the first instruction to execute as identified in the END pseudo op.

ALGORITHMS

In this section we present algorithms which aid in the processing and maintaining of the data bases described in the last few sections. A modular approach to programming is presented here so that the reader may program and understand each module and eventually combine them into an assembler.

We start by presenting an algorithm for the semantic and syntactic interpretation of each record. A means of stripping off the various parts of each statement (label, opcode, operand) is desirable so that the tables of our data base may be searched to find a match and perform the appropriate action. If a fixed format were adopted for each record, this task might be somewhat simpler; however, for appropriate error checking the entire card should be

scanned. The free format algorithm presented here is no more complex than a fixed format algorithm with appropriate error checking. This algorithm scans each statement from left to right picking off one token at a time and determining its type. The algorithm requires the following constraints:

(1) The card image is stored in ASCII form in primary memory at an identifiable address.

(2) The card image occupies exactly 80 bytes.

(3) A 6 byte area is reserved in the calling program to store the label.

(4) A 6 byte area is reserved in the calling program to store the operand.

(5) A 4 byte area is reserved in the calling program to store the opcode.

(6) A status word is reserved in the calling program to indicate certain information about the card image.

Each bit in the card status word has some preassigned meaning. The algorithm presented here makes no attempt to account for all possible errors. It should serve as a basic outline to be expanded in actual application. The calling sequence to this algorithm is shown here.

```
JSR  R0,CARDSCAN

.WORD INPUT        ; Address of 80 byte record.
.WORD LABEL        ; For rule (3) above.
.WORD OPCODE       ; For rule (5) above.
.WORD OPERAND      ; For rule (4) above.
.WORD STATUS       ; For rule (6) above.
```

ALGORITHM C

(C1) R1←(RO), R2←address of token, blank out token, I←0.

(C2) I←I+1, if I>80 (blank card) go to ERROR.

(C3) if (R1) = R1←R1+1, go to C2.
if (R1) = /(only a comment) go to ERROR
if (R1) = ,(no label preceding comma) go to ERROR
if (R1) = number go to ERROR

(C4) (R2)+←(R1)+, I←I+1 if I>80 go to C14

(C5) if (R1) = , go to C6
if (R1) = go to C14
if (R1) = / go to ERROR

(C6) (this token is label) R4←2(RO)
R2←address of token
R3←6

(C7) R3←R3-1, if R3<0 go to C9.

(C8) (R4)+←(R2)+ go to C7.

(C9) blank out token, R2 address of token.

(C10) R1←R1+1, I←I+1, if I>80 (label without operation)
then go to ERROR.

(C11) if (R1) = go to C10
if (R1) = / or , or number go to ERROR.

(C12) (R2)+←(R1)+, I←I+1, if I>80 go to C14.

(C13) if (R1) = go to C14
if (R1) = / or , or <number> go to ERROR.

(C14) (this token is opcode) R4←4(RO)
R2←address of token
R3←4.

(C15) R3←R3-1, if R3<0 go to C17.

(C16) (R4)+←(R2)+, go to C15.

(C17) blank out token, R2←address of token.

(C18) R1←R1+1, I←I+1, if I>80 go to C30.

(C19) if (R1) = go to C18,
if (R1) = / go to C30,
if (R1) = , go to ERROR.

(C20) (R2)+←(R1)+, I←I+1, if I>80 go to C27.

(C21) if (R1) = go to C22,
if (R1) = / go to C27,
if (R1) = , go to ERROR,
go to C20.

(C22) (this token is operand, see if there is another operand, if so this one should be I to indicate indirect)
R1←R1+1, I←I+1, if I>80 go to C27.

(C23) if (R1) = go to C22,
if (R1) = / go to C27,
if (R1) = , go to ERROR.

(C24) See if first token is I if so set indirect bit in status word and blank out token otherwise go to ERROR.

(C25) (R2)+←(R1)+, I←I+1, if I>80 go to C27.

(C26) if (R0) = go to C27,
otherwise go to C25.

(C27) (this token is real operand) R4←6(R0),
R2←address of token
R3←6.

(C28) R3←R3-1, if R3<0 go to C30.

(C29) (R4)+←(R2)+, go to C28.

(C30) Set all appropriate bits in status word and return.

SEARCHING ALGORITHMS

The CODES table is ordered only by the card reader codes. It may, however, be necessary to convert a card code to an ASCII code in which case a linear search through the card code portion of the table would provide the appropriate conversion. It is necessary to have a linear search subroutine which might also be used to search the symbol table to determine if a particular symbol is in the table.

Constraints on the algorithm presented here are:

(1) The table must occupy contiguous locations in primary memory at an identifiable address.

(2) The record length, key length, and the maximum number of records in the table must be known.

(3) The address of the key to search for must be known.

The calling sequence to this routine is as follows:

```
JSR RO, LINEAR
.WORD TABLE          ; address of first key to check.
.WORD KEY            ; address of key to search for.
.WORD m              ; # of records in table.
.WORD n              ; # of bytes of each record.
.WORD p              ; # of bytes of each key.
.WORD 0              ; upon return will contain address
                       of key or 0 if not found.
```

ALGORITHM L

(L1) R1←(R0), R2←2(R0), R3←4(R0), R4←6(R0), R5←10(R0).

(L2) R3←R3-1, if R3<0 (searched whole table) then 12(R0)←0, go to L7.

(L3) R5←R5-1, if R5<0 (found it) go to L6.

(L4) if (R1)+ = (R2)+ go to L3.

(L5) (set pointers to next record--next key)
R2←R2-(10(R0)-R5),
R1←R1-(10(R0)-R5)+R4,
R5←10(R0),
go to L2 (get next record).

(L6) 12(R0)←R1-(10(R0)+1)

(L7) R0←R0+14 and return.

Binary searches are notably faster than linear searches particularly if the table to be searched contains a large

number of records. Such a table is the symbol table which may contain up to 1024 entries. If a binary search algorithm is used, then the table searched must satisfy four constraints.

(1) For simplicity the maximum number of records in the table must be a power of two.

(2) Each record in the table must be the same length.

(3) The table must be in order of some key.

(4) All keys must be of the same length.

A search of this nature is easily adaptable to searching the MOT since all the constraints are already satisfied. If this routine is to be used to search the symbol table the user must first order the records and then pad the unused portion of the table with positive infinity to assure proper operation of the algorithm. In a sense a binary search narrows down the table size by chopping it in half at each probe similar to guessing the value of a random number between 0 and 1000 as quickly as possible by first guessing 500 then either 250 or 750 depending on whether the first guess was too high or too low. Calling the binary search subroutine should be as follows:

```
JSR RO, SEARCH
.WORD KEY          ; address of key to search for.
.WORD TABLE        ; address of first key.
.WORD n            ; number of bytes of each record.
.WORD m            ; number of bytes of each key.
.WORD t            ; maximum # of records (power of
                     two).
.WORD 0            ; will contain record # if pre-
                     sent, 0 otherwise.
```

ALGORITHM B

(B1) R5←(R0), R2←2(R0), M←4(R0), N←6(R0), R3←10(R0), R1←R5, R3←R3/2, R5←R3.

(B2) R5←R5/2,
W←R3*M+R2,
N←6(R0).

(B3) (check this key) N←N-1, if N<0 go to B8,
If @W>(R1) R1←R1+1 then go to B4,
If @W<(R1) R1←R1+1 go to B5
otherwise W←W+1, repeat this step.

(B4) (too high in table) R3←R3-R5 and go to B6.

(B5) (too low in the table) R3←R3+R5.

(B6) (reinitialize R1 to first byte of key to search for) R1←R1-(6(R0)-N), if R5≠ 0 go to B2.

(B7) (might be first element) See if desired key is first entry in table, if not then 12(R0)←0 and go to B9.

(B8) (found) 12(R0)←R3+1

(B9) RETURN.

TABLE INSERTION ALGORITHM

During the first pass of the assembler, while an algorithm is stripping off labels and keeping track of the assembly location counter it is necessary to enter this information in the UST. The first method for inserting a label and value into the UST is quite simple. The idea involved is to keep a count of the number of symbols in the table and insert the record in the next available location. Such an algorithm is presented below and is subject to the following constraints.

(1) The table must be of fixed length.

(2) Each record must be of fixed length.

(3) The user must keep a count of the current number of records in the table.

(4) No error checking is done to insure that we do not overflow the table or to determine if the item is not already in the table.

The calling sequence to this algorithm is:

```
JSR RO, ENTER
.WORD TABLE      ; address of table.
.WORD ITEM       ; address of item to enter.
.WORD n          ; # of bytes for each record.
.WORD m          ; current # in the table.
```

ALGORITHM E

(E1) R1←(R0), R2←2(R0), R3←4(R0), R4←6(R0).

(E2) (get offset from top of table) R5←R3*R4, R1←R1+R5.

(E3) (move it) R3←R3-1, if R3<0 go to E5.

(E4) (R1)+←(R2)+ go to E3.

(E5) R0←R0+10 and RETURN.

If this algorithm is to be used to process the symbol table then algorithm S should be initiated to assure that the item is not already in the table. Also note that the Auto-increment mode in step E4 is a byte increment as with all other uses of Auto-increment mode in this chapter.

SORTING ALGORITHMS

In order to efficiently use the symbol table during PASS2 of the assembler as well as to provide a neat listing of the symbol table to the programmer it becomes necessary

to sort the table. The first algorithm proposed is given the name straight insertion sort by Knuth (1975) and is easy to implement especially if algorithm E is used to build the symbol table. Even with intricate devices that are used on the more popular bubble sort the insertion sort is much faster because the number of times a record needs to be moved is much smaller with the insertion sort. The constraints for this algorithm are the same as those for algorithm E; in addition, however, the calling program must set up a work space the size of one record. The calling sequence to this algorithm is as follows:

```
JSR RO, SORT
.WORD TABLE          ; address of table to sort.
.WORD n              ; # of bytes for each key.
.WORD m              ; # of bytes for each record.
.WORD p              ; # of records to sort.
.WORD WORK           ; address of buffer area.
```

ALGORITHM S

(S0) TOP←(RO), KSIZ←2(RO), RSIZ←4(RO), R3←10(RO), R1←0.

(S1) R1←R1+1, if R1≥6(RO) terminate,
otherwise R2←TOP+R1*RSIZ.

(S2.0) (move next item to save area) I←R1-1.

(S2.1) RSIZ←RSIZ-1, if RSIZ<0 then R3←10(RO)
and RSIZ←4(RO) and go to S3.0.

(S2.2) (R3)+←(R2)+, go to S2.1.

(S3.0) (compare saved item with last item in sorted sublist)
R2←TOP+I*RSIZ.

(S3.1) KSIZ←KSIZ-1, if KSIZ<0 then KSIZ←2(RO) and go to S5.0.

(S3.2) (still greater than) if (R3)+≥(R2)+ then go to S3.1.

(S4.0) (Otherwise move one item up)

R4←I*RSIZ+TOP,
R5←(I+1)*RSIZ+TOP.

(S4.1) RSIZ←RSIZ-1. if RSIZ<0 then RSIZ←4(RO) and go to S4.3.

(S4.2) (move some more) (R5)+←(R4)+, go to S4.1.

(S4.3) I←I-1, if I≥0 (more items left to compare this key with) then KSIZ←2(RO) and go to S3.0.

(S5.0) (found where it goes, put it there)
R5←(I+1)*RSIZ+TOP, R3←10(RO).

(S5.1) RSIZ←RSIZ-1, if RSIZ<0 then RSIZ←4(RO) and KSIZ←2(RO) and go to S1.

(S5.2) (put it where it belongs) (R5)+←(R3)+.

Algorithm S above is somewhat complicated, however, it is quite detailed and designed to be implemented immediately on a PDP-11. It is a simple matter to translate this algorithm to MACRO-11 code. Users of this algorithm and of all the others presented in this study should be cautioned that the temptation to alter the value of the variables and registers to the right of the "←" symbol can lead to serious difficulty. Temporary locations should be set up to contain these intermediate results.

Some interesting observations can be made at this point. Suppose it is necessary to search the symbol table during PASS1 (and this is necessary if the assembler is to detect errors resulting from multiple definition of symbols). If algorithm E (ENTER) is used to enter the symbols then algorithm L (LINEAR) must be used to search the table during PASS1 since algorithm S (SORT) is only implemented once to sort the entire table at the end of PASS1. This linear

search is slow since every byte of each key must be examined, especially when the table starts to get full. A possible alternative is to sort the entire table each time a new entry is made, but this is very inefficient since it would then be the case that all items would be already sorted except the last item in the list. This would result in steps S1 through S5.1 of algorithm S to be executed $\sum_{i=1}^{n} i$ times where n is the total number of symbols in the program. The previous considerations suggest that some improved algorithms exist having the following advantages:

(1) It should eliminate the need for a linear search by keeping the table ordered throughout PASS1.

(2) It should enter one item in the table and immediately sort the table thus eliminating algorithm E and S.

(3) It should contain steps similar to S1 through S5.1, but execute them n times instead of $\sum_{i=1}^{n} i$ times thus increasing the speed.

(4) Core requirements should not exceed the sum of the requirements for algorithm E and S.

The calling sequence to this super algorithm is as follows:

```
JSR RO, INSERT
.WORD TABLE          ; address of table.
.WORD ITEM           ; address of item to enter.
.WORD n              ; # of bytes for each record.
.WORD m              ; # of items already in table.
```

```
.WORD p                 ; # of bytes for each key.
.WORD WORK              ; Address of work area.
```

Algorithm S and E define the constraints that apply to algorithm I presented here.

ALGORITHM I

(I0) R1←(R0), R2←2(R0), R3←4(R0), R4←6(R0).

(I1) R5←R3*R4, R1←R1+R5.

(I2) R3←R3-1, if R3<0 go to I4.

(I3) (here we enter the item at the end of the list the previous n-1 records are already sorted) (R1)+←(R2)+, go to I2.

(I4) (initialize for the sort from back to front for the item just entered) TOP←(R0), KSIZ←10(R0), RSIZ←4(R0), R3←12(R0), R1←6(R0).

(I5) (one more item now in table) R1←R1+1, R2←TOP+R1*RSIZ, I←R1-1.

(I6) RSIZ←RSIZ-1, if RSIZ<0 then R3←12(R0) and RSIZ←4(R0) and go to I8.

(I7) (R3)+←(R2)+, go to I6.

(I8) R2←TOP+I*RSIZ.

(I9) KSIZ←KSIZ-1, if KSIZ<0 then go to I15.

(I10) (compare new item to next item of previously sorted list) if (R3)+$\geq$(R2)+ then go to I9.

(I11) (otherwise move this item to next highest position to make room) R4←I*RSIZ+TOP, R5←(I+1)*RSIZ+TOP.

(I12) RSIZ←RSIZ-1, if RSIZ<0 then RSIZ←4(R0) and go to I14.

(I13) (put it there) (R5)+←(R4)+, go to I12.

(I14) I←I-1, if I$\geq$0 then KSIZ←10(R0) and go to I8.

(I15) (this is where it belongs) R5←(I+1)*RSIZ+TOP, R3←12(R0).

(I16) RSIZ←RSIZ-1, if RSIZ<0 RETURN.

(I17) (R5)+←(R3)+, go to I16.

CHAPTER IV

THE ASSEMBLER AND SIMULATOR

There are two primary functions of an assembler. The most obvious function is to translate into machine language the coded instruction mneumonics. Assembler directives aid in this process by providing storage allocation to the assembler. The second function of the assembler is to provide certain important information to the loader program (see Chapter II). We shall discuss these functions separately.

To perform these functions we recall that the assembler is divided into two distinct passes. With the data bases well defined and some important subroutines already developed it becomes a straightforward problem to build PASS1 and PASS2 of the assembler.

PASS1

The primary functions of PASS1 are to create the UST and to initialize tables and monitor buffers. Most importantly PASS1 must determine what file is to be assembled. Secondly it must initialize all tables and work areas. For example + ∞ must be moved to each byte of the UST (See algorithm B in the previous chapter). Finally, PASS1 begins

the process of building the UST.

Determining the Input file, finding it, and opening it is not at all as easy as it might seem. To accomplish these tasks in the simplest manner possible, we make use of some pre-written subroutines. Most monitors contain a library of system subroutines available to user programs by use of macro calls which enable users to easily accomplish file manipulation tasks. In writing an assembler the programmer should review this library to see what programs are available and how they may be used to solve various problems. It is suggested that the reader refer to the Disk Operating System Monitor Programmers Handbook to obtain information on the following subroutines:

.CSI1 .CSI2	command string interpreter
.INIT	to initialize a file block
.OPENI	opens a file for input
.RECRD	for I/O of records
.WAIT	to delay further execution until I/O is completed
.CLOSE	to close a file
.RLSE	to release a file
.ALLOC	to allocate a file
.OPENU	to open a file for output

To determine the input file name we first read a card containing a command string identifying the input and output

files. Next the command string must be checked for proper syntax. (Note: the # symbol is not part of a command string.) We then initialize the file and open the file for input since the input specification is known. Methods of identifying and accessing a file vary from system to system and hence the outline of the necessary steps presented above is intentionally non-specific. For a more detailed discussion, the reader should refer to the handbook of the current monitor being used.

The task of initializing all tables and work areas is easily accomplished, hence no discussion is presented here.

Assuming some method exists for accessing a disk file the assembler is now ready to process that file. The task is simply to find all labels and record the relative address associated with the labels and use this information to build the UST. To accomplish this the following data bases are maintained during PASS1:

ALC (assembly location counter) A pseudo program counter to keep track of relative addresses.

POT The assembler must be able to allocate storage as well as determine the end of a file. This is done by searching the pseudo op table.

CODES Input and output processes may require code translation.

UST This table is created during PASS1.

Algorithm PI below is a simplified outline of PASS1.

(PI0) Initialize record count and assembly location counter.
RCRDCNT ← -1, ALC ← 0.

(PI1) Read next record.
RCRDCNT ← RCRDCNT + 1.
INPUT RECORD OF INPUT FILE IDENTIFIED BY RCRDCNT.

(PI2) Parse the current record.
CALL CARCSCAN.

(PI3) Search POT to see if this instruction is "END" pseudo op.
CALL LINEAR, IF YES GO TO PI10.

(PI4) LENGTH ← 1.

(PI5) Check Cardstatus word to see if a label exists if not GO TO PI9.

(PI6) VALUE ← ALC.

(PI7) See if label is already in UST.
CALL SEARCH, If found GO TO PI9.

(PI8) Make new entry in UST.
CALL ENTER.

(PI9) ALC ← ALC + LENGTH, GO TO PI1.

(PI10) Clean up, print symbol table and errors.
CALL PASS2.

At the beginning of this chapter the two primary functions of an assembler were stated. We have not yet finished the first function. However, at the conclusion of PASS1 there is enough information to perform the second function. The discussion of PASS2 will finish the translation process, but it is convenient at this time to make some remarks concerning the second function.

We may begin PASS2 with an initial routine to make the first two entries in the load module (see Chapter III). The output file must first be allocated. A command string

interpreter may be used to obtain the name to be given to the load module. The length of the program text is known from PASS1, hence an allocation routine may be used to create a file containing the appropriate amount of storage and named appropriately. On most systems the smallest amount of disk storage that may be allocated is one block; therefore, a simple arithmetic computation will give the correct number of blocks to allocate. The program length might be slightly smaller than the amount allocated, hence we will output the program length to the first word of the load module since it is known at this time. Similarly the END pseudo op in conjunction with the UST provides the transfer address so we may output this to the second word of the load module. After these tasks have been performed all the information necessary to the loader has been provided.

PASS2

In PASS2 we actually output the machine language text of the program to the load module immediately following the preface information. For micro instructions and other instructions or data not requiring an operand address this could have been done during PASS1 provided that the file was already allocated at the beginning of PASS1 instead of the end of PASS1. However, it is conceptually and logically simpler to do all translation during PASS2. In order to accomplish this task the following data bases are maintained:

UST This table is used to get the value of an operand so that the address portion of an instruction can be calculated.

POT This table is used to aid in allocation and definition of work areas and to recognize END pseudo ops.

MOT This table is used to build the operation portion of an instruction.

CODES Used for I/O translation.

INPUT PASS2 must reread the input file.

OUTPUT This file is the created load module.

Algorithm PII below is a general outline for PASS2.

(PII0) Initialize record count.
RCRDCNT ← -1.

(PII1) Read next record.
RCRDCNT ← RCRDCNT + 1,
INPUT RECORD OF INPUT FILE IDENTIFIED BY RCRDCNT.

(PII2) Parse the current card.
CALL CARDSCAN.

(PII3) Search the POT to see if this instruction is any pseudo op.
CALL LINEAR, if no GO TO PII5.

(PII4) Branch to a subroutine to process all pseudo ops.
CALL PSEUDO. This subroutine branches to the appropriate step.

(PII5) Search MOT for opcode.
CALL SEARCH.

(PII6) Get address of subroutine to build this instruction.
(N = item # in table, TOP = address of top of MOT)
R1 ← N*6-2+TOP,
R2 ← (R1), branch to appropriate subroutine.
JSR R0, (R2)+

(PII7) Output machine instruction to next free location in load module. (PUNCH contains machine instruction) OUTPUT PUNCH TO OUTPUT FILE IDENTIFIED BY RCRDCNT+2.

(PII8) GO TO PII2.

(PII9) FINISH.

In many computer systems the monitor does not allow the user to have more than one file open at any given time. Since PASS2 must read the input file and write the output file the assembler must open a file, perform the I/O operation, and close the file each time an I/O operation needs to be performed. This is simple to do yet costly in terms of execution time. An alternative which is much faster but harder to implement is to read and write one block at a time. Upon completion of PASS2 all files should be closed and all tables reinitialized.

THE SIMULATOR

Although an assembled program is more meaningful to a machine than the coded one it still must be loaded in an executable form and finally executed before it becomes useful as a tool of data processing. Since our assembler is for a JACK-1 computer it must be executed on a JACK-1. The programs that are output from the assembler (although created by a PDP-11 program) cannot be executed by a PDP-11. However we can put the load module in the memory of a JACK-1 and execute it there. In view of the preceding remarks we see that an assembler could be written on any available

machine to assemble a program designed for some specific machine. In any case, it is still preferable to have the assembler available on the computer system for which it was designed. This could be accomplished in either of the following ways. First, write a translator to translate the assembler we have written in the previous chapters from PDP-11 code to JACK-1 code. Second, rewrite the assembler of the previous chapters in JACK-1 code and assemble it using the assembler written in PDP-11 code. The first method suggested is a more difficult problem but has the advantage that once the translator exists any PDP-11 program can be altered to run on a JACK-1.

Neither of the above alternatives are possible in the problem of this paper since no JACK-1 computer exists. We do, however, desire some means of executing the programs written in JACK-1. For example, suppose that JACK-1 had some elegant instruction that was desirable for a particular application and, furthermore, suppose that such an instruction does not exist on any other machine. Then the programmer would like to write his program in JACK-1 code using all the elegant features of the JACK-1. Since the JACK-1 assembler already exists, then the obvious solution is to simulate execution of the load module by means of a simulator program. This simulator program will perform the exact same functions of the JACK-1 hardware illustrated in Chapter II. The diagram in Chapter II will aid in understanding algorithm SI below which outlines the steps executed by the simulator.

(SI1) Load the file.

(SI2) Initialize instruction counter with initial program load address.
IC ← IPLA + offset.

(SI3) Set the memory address register.
MAR ← IC

(SI4) Put instruction in instruction register.
MBR ← (MAR), IR ← MBR

(SI5) If micro instruction GO TO SI8.

(SI6) Determine address of operand.

(SI7) Execute instruction by simulation.

(SI8) IC ← IC + 1, GO TO SI3.

CHAPTER V

CONCLUSIONS

In summary, we have described an entire computer system. The major thrust of the investigation was the design and implementation of an assembler for the system we have described. An assembler is one of the most fundamental systems programs since it allows users to interface directly with the hardware.

Chapter I presented some basic concepts of computers that relate the intent of this study. Through these concepts and through man's necessity to communicate effectively with a machine, we provided justification for further study in the field of assemblers as necessary systems programs.

Chapter II began by defining a hypothetical machine which was conceived to be a suitable model for educational study. After the basic elements of the hardware were presented, instruction formats, mneumonics and functions were defined. With the machine and instruction set well defined the next step was to describe in detail the form that the assembly language must assume before translation to machine language by the assembler. Chapter II concluded with a few programming examples.

In Chapter III a very brief outline of the function of

an assembler was presented so that an understanding of the material to follow would be possible. All the various tables necessary for the translation process were treated. Their purpose, format, content and restrictions were discussed. In order to effectively translate an assembly language program into machine language several algorithms (procedures) for processing the tables had to be developed. These algorithms, which would be subroutines in the assembler, were developed and defined in the second half of Chapter III.

Chapter IV described in more detail the functions of an assembler and how these functions might be accomplished. A description of a two pass algorithm was presented including data bases, functions of each pass, and algorithms to perform each of these functions. These algorithms and data bases, which constituted the assembler, made use of the subroutine of the previous chapter. Chapter IV concluded with implementation procedures and a general design of a simulator. What follows in this chapter is sample output of the assembler and simulator developed in this thesis.

```
                                    USER DEFINED SYMBOL TABLE
                                    ADDR      000005
                                    BEGIN     000000
                                    DATA      000002
                                    MASK      000001
000000          BEGIN,    DCML 0
007777          MASK,     OCTL 07777
000005          DATA,     DCML 5
000002                    DCML 2
000003                    DCML 3
010002          ADDR,     TAD DATA          / DUMMY INSTRUCTION WHICH HAS OPERAND DATA
072000                    CLA
071000                    CLL
010005                    TAD ADDR          / TO GET INSTRUCTION CONTAINING ADDRESS OF
000001                    AND MASK          / DATA IN AC  AND STRIP OFF THE ADDRESS
030000                    DCA BEGIN
                          END ADDR
```

FIGURE 1.

SAMPLE OUTPUT FROM ASSEMBLER

```
072000:
00: 004060 070000 003462 000200 003606 072054 072054 001722: .H@02G@@FG,4,4RC
20: 001402 104002 177776 001046 000000 000000 000006 072112: BCBH:?%B@@@@F@J4
40: 072070 005706 002066 000352 001724 000000 000352 002060: 84Fk6D*@TC@@*@0D
60: 000006 072070 005706 072116 177777 072150 072150 000000: F@84FkN4??(4(4@@

072100:
00: 000000 000000 077200 000004 000000 000000 000000 000000: @@@@@:>D@@@@@@@@@@
20: 000012 000002 007777 000005 000002 000003 010002 072000: J@B@?0E@B@C@BP@4
40: 071000 010005 000001 030000 000000 000000 000000 000000 @ZEPA@@0@@@@@@@@@
60: 000000 000000 000000 000000 000000 000000 000000 000000 @@@@@@@@@@@@@@@@

072200:
00: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@
20: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@
40: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@
60: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@

072300:
00: 000000 000000 000000 000000 000000 000000 000000 000000 @@@@@@@@@@@@@@@@@@
20: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@
40: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@
60: 000000 000000 000000 000000 000000 000000 000000 000000 @@@@@@@@@@@@@@@@@@

072400:
00: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@
20: 000000 000000 000000 000000 000000 000000 000000 000000 @@@@@@@@@@@@@@@@@@
40: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@
60: 000000 000000 000000 000000 000000 000000 000000 000000: @@@@@@@@@@@@@@@@@@
```

FIGURE 2.

SAMPLE OUTPUT FROM SIMULATOR

APPENDIX A

DATA BASES

DATA BASES

(1) Machine Op Table

```
 .ASCII /AND /          .ASCII /RAR /
.WORD AND               .WORD RAR
.ASCII /CLA /           .ASCII /RTL /
.WORD CLA               .WORD RTL
.ASCII /CLL /           .ASCII /RTR /
.WORD CLL               .WORD RTR
.ASCII /CMA /           .ASCII /SKP /
.WORD CMA               .WORD SKP
.ASCII /CML /           .ASCII /SMA /
.WORD CML               .WORD SMA
.ASCII /DCA /           .ASCII /SNA /
.WORD DCA               .WORD SNA
 .ASCII /GET /          .ASCII /SNL /
 .WORD GET              .WORD SNL
.ASCII /HLT /           .ASCII /SPA /
.WORD HLT               .WORD SPA
.ASCII /IAC /           .ASCII /SZA /
.WORD IAC               .WORD SZA
.ASCII /ISZ /           .ASCII /SZL /
.WORD ISZ               .WORD SZL
.ASCII /JMP /           .ASCII /TAD /
.WORD JMP               .WORD TAD
.ASCII /JMS /            .WORD 55533
.WORD JMS                .WORD 55533
.ASCII /NOP /            .WORD 55533
.WORD NOP                .WORD 55533
.ASCII /OSR /            .WORD 55533
.WORD OSR                .WORD 55533
 .ASCII /PUT /           .WORD 55533
 .WORD PUT               .WORD 55533
.ASCII /RAL /            .WORD 55533
.WORD RAL
```

(2) Codes Table

```
                        CARD READER          ASCII      CHARACTER
CODES:      .WORD           54      ,         50      ;    (
            .WORD           000     ,         40      ;
            .WORD           1       ,         61      ;    1
            .WORD           2       ,         62      ;    2
            .WORD           3       ,         63      ;    3
            .WORD           4       ,         64      ;    4
            .WORD           5       ,         65      ;    5
            .WORD           6       ,         66      ;    5
            .WORD           7       ,         67      ;    7
            .WORD           10      ,         70      ;    8
            .WORD           12      ,         137     ;    -
            .WORD           13      ,         75      ;    -
            .WORD           14      ,         100     ;
            .WORD           15      ,         136     ;
            .WORD           16      ,         42      ;
            .WORD           17      ,         134     ;
            .WORD           20      ,         71      ;    9
            .WORD           40      ,         60      ;    0
            .WORD           41      ,         57      ;    /
            .WORD           42      ,         123     ;    S
            .WORD           43      ,         124     ;    T
            .WORD           44      ,         125     ;    U
            .WORD           45      ,         126     ;    V
            .WORD           46      ,         127     ;    W
            .WORD           47      ,         130     ;    Y
            .WORD           50      ,         131     ;    Y
            .WORD           52      ,         73      ;
            .WORD           53      ,         54      ;    ,
            .WORD           55      ,         42      ;    @
            .WORD           56      ,         43      ;    #
            .WORD           60      ,         132     ;    Z
            .WORD           100     ,         55      ;    -
            .WORD           101     ,         112     ;    J
            .WORD           102     ,         113     ;    K
            .WORD           103     ,         114     ;    L
            .WORD           104     ,         115     ;    M
            .WORD           105     ,         116     ;    N
            .WORD           106     ,         117     ;    O
            .WORD           107     ,         120     ;    P
            .WORD           110     ,         121     ;    Q
            .WORD           112     ,         72      ;    :
            .WORD           113     ,         44      ;    $
            .WORD           114     ,         52      ;    *
            .WORD           115     ,         133     ;    [
```

(2) Codes Table (Continued)

```
        .WORD       116     ,       76      ;   >
        .WORD       117     ,       46      ;   +
        .WORD       120     ,       122     ;   R
        .WORD       200     ,       53      ;   +
        .WORD       201     ,       101     ;   A
        .WORD       202     ,       102     ;   B
        .WORD       203     ,       103     ;   C
        .WORD       204     ,       104     ;   D
        .WORD       205     ,       105     ;   E
        .WORD       206     ,       106     ;   F
        .WORD       207     ,       107     ;   G
        .WORD       210     ,       110     ;   H
        .WORD       212     ,       77      ;   @
        .WORD       213     ,       56      ;   .
        .WORD       214     ,       51      ;   )
        .WORD       215     ,       135     ;   ]
        .WORD       216     ,       74      ;   <
        .WORD       217     ,       41      ;
        .WORD       220     ,       111     ;   I
```

APPENDIX B

ASSIGNMENTS

ASSIGNMENTS

Assignment 1:

Examine and record the contents of memory locations 72-82. Refering to the program in Chapter I: (1) Provide a means of getting out of the loop after iterations. (2) Toggle in the program so that the instruction labeled A: will begin at address 10. (3) Execute the program and re-examine location 72-82 for proper results. What would happen if the program were not loaded at the proper address? Examine the program area. How does it differ from what you toggled in?

Assignment 2:

Read and study the monitor routines discussed in Chapter IV. Write a program to read a command string in the proper form off a card then use the proper monitor routine to:

1. Check the command string for proper syntax.
2. Fill in the missing entries in the FILBLK.
3. Initialize and open the file for input.
4. Read the first record from the file by use of the

.RECRD routine.

5. Print this record on the line printer.

Assignment 3:

Write MACRO subroutines corresponding to each of the algorithms presented in Chapter III. Be sure to call the subroutines in the way specified by writing an appropriate main-line to test the subroutine.

Assignment 4:

Complete the steps necessary to develop your own assembler.

BIBLIOGRAPHY

Disk Operating System Monitor Programmers Handbook, Maynard Massachusetts:Digital Equipment Corporation, 1972.

Donovan, John J. Systems Programming, New York:McGraw-Hill Book Company, 1972.

Eckhouse, Richard H., Jr., Minicomputer Systems: Organization and Programming, Englewood Cliffs, N.J.:Prentice-Hall, Inc., 1975.

Knuth, Donald E., The Art of Computer Programming, Vol. 3, Reading Massachusetts:Addison-Wesley Publishing Company, 1975.