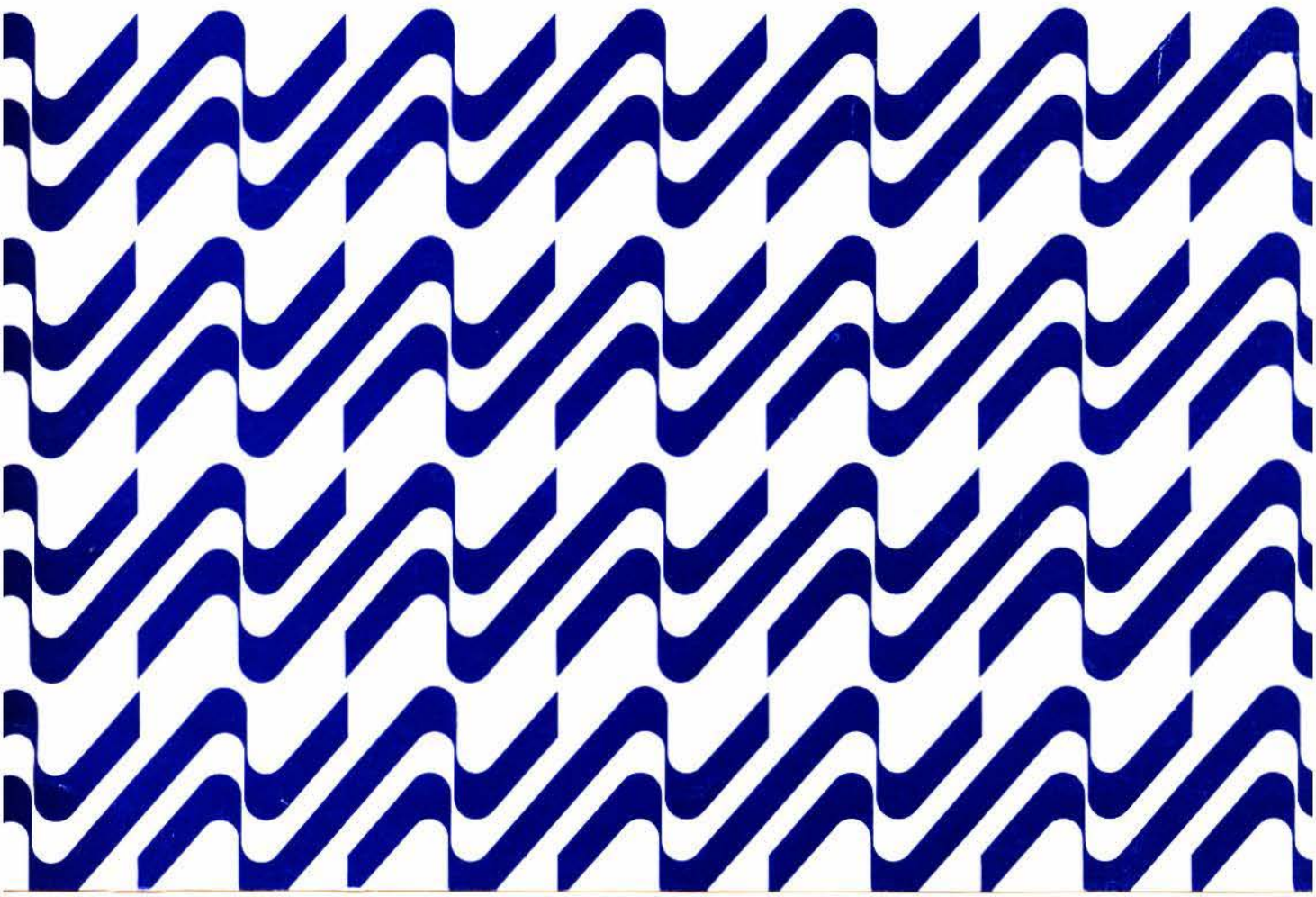
National Semiconductor
Pub. No. 4200094B
Order No. ISP-85/994Y

SC/MP Programming And Assembler Manual



Simple Cost-effective
MicroProcessor

Publication Number 4200094B
Order Number ISP-8S/994Y

SC/MP

PROGRAMMING AND ASSEMBLER MANUAL

February 1976

2900 Semiconductor Drive
Santa Clara, California 95051

PREFACE

The SC/MP Programming and Assembler Manual provides tutorial and reference information for devising user application programs. The manual is written for the benefit of both engineers and programmers for SC/MP programming indoctrination. Information pertaining to the SC/MP microprocessor and microcomputer equipment is not provided in this manual.

The material in this manual is for information purposes only and is subject to change without notice.

It is suggested that the reader thoroughly review the tables of contents, illustrations, and tables to familiarize himself with an overview of the organization of the manual before reading the contents. By so doing, the reader may then be prepared to appreciate the extent-of-coverage; such an appreciation shall likely be useful during the initial reading of this manual.

Copies of this publication and other National Semiconductor publications may be obtained from the sales offices listed on the back cover.

TABLE OF CONTENTS

Chapter			Page
1	GENERAL INFORMATION		1-1
	1.1	INTRODUCTION	1-1
	1.2	SCOPE OF MANUAL	1-1
	1.3	ORGANIZATION OF MANUAL	1-1
2	BASIC CONCEPTS		2-1
	2.1	INTRODUCTION	2-1
	2.2	COMPUTERS, MICROCOMPUTERS, MICROPROCESSORS	2-1
	2.3	BASIC ELEMENTS OF A COMPUTER SYSTEM	2-1
	2.3.1	Hardware	2-1
	2.3.2	Firmware	2-2
	2.3.3	Software	2-2
	2.4	PROGRAMMING LANGUAGES	2-3
	2.5	USING A COMPUTER	2-4
	2.5.1	Problem Definition	2-4
	2.5.2	Program Flowchart	2-4
	2.5.3	Writing a Program	2-5
	2.5.4	Desk-checking Code	2-5
	2.5.5	Assembling a Program	2-5
	2.5.6	Loading	2-7
	2.5.7	Debugging	2-7
3	DEVELOPMENT SYSTEM OVERVIEW		3-1
	3.1	GENERAL	3-1
	3.2	DEVELOPMENT SYSTEM CONFIGURATION	3-1
	3.3	REGISTERS	3-1
	3.3.1	Accumulator (AC)	3-2
	3.3.2	Status Register (SR)	3-2
	3.3.3	Extension Register (E)	3-3
	3.3.4	Program Counter (PC)	3-3
	3.3.5	Pointer Registers (PTR)	3-3
	3.4	MEMORY ADDRESS STRUCTURE	3-3
	3.5	METHODS OF ADDRESSING	3-6
	3.5.1	PC-Relative Addressing	3-6
	3.5.2	Immediate Addressing	3-7
	3.5.3	Indexed Addressing	3-7
	3.5.4	Auto-Indexed Addressing	3-7
	3.6	INPUT/OUTPUT FACILITIES	3-8
	3.6.1	Address Lines	3-8
	3.6.2	Parallel Input/Output	3-8
	3.6.3	Serial Input/Output	3-8
	3.6.4	I/O Status	3-9
4	ASSEMBLY LANGUAGE		4-1
	4.1	CHARACTER SET	4-1
	4.2	ASSEMBLER CODING CONVENTIONS	4-1
	4.2.1	Label Field	4-1
	4.2.2	Operation Field	4-3
	4.2.3	Operand Field	4-3
	4.2.3.1	Self-Defining Terms	4-4
	4.2.3.2	Symbolic Terms	4-4
	4.2.3.3	Expressions	4-6
	4.2.4	Comment Field	4-6

TABLE OF CONTENTS (Continued)

Chapter			Page
4 (Cont'd)	4.2.5	Identification Sequence Field	4-7
	4.2.6	Example Statement	4-7
5	STATEMENTS		5-1
	5.1	COMMENT STATEMENTS	5-1
	5.2	INSTRUCTION STATEMENTS	5-1
	5.2.1	Memory Reference Instructions	5-5
	5.2.2	Memory Increment/Decrement Instructions	5-8
	5.2.3	Immediate Instructions	5-9
	5.2.4	Transfer Instructions	5-12
	5.2.5	Extension Register Instructions	5-13
	5.2.6	Pointer Register Move Instructions	5-16
	5.2.7	Shift, Rotate, Serial Input/Output Instructions	5-17
	5.2.8	Miscellaneous Instructions	5-19
	5.3	PSEUDO INSTRUCTIONS	5-23
	5.4	ASSIGNMENT STATEMENT	5-23
	5.5	DIRECTIVE STATEMENTS	5-24
	5.5.1	.TITLE Directive	5-24
	5.5.2	.END Directive	5-25
	5.5.3	.LIST Directive	5-25
	5.5.4	.SPACE Directive	5-25
	5.5.5	.PAGE Directive	5-26
	5.5.6	.BYTE Directive	5-26
	5.5.7	.DBYTE Directive	5-26
	5.5.8	.ADDR Directive	5-27
	5.5.9	.ASCII Directive	5-27
	5.5.10	.LOCAL Directive	5-27
	5.5.11	Conditional Assembly Directives	5-28
	5.5.12	.FORM Directive	5-29
6	PROGRAMMING TECHNIQUES		6-1
	6.1	INTRODUCTION	6-1
	6.2	STACK PROGRAMMING	6-1
	6.2.1	Stack Operations	6-1
	6.2.2	Repeatable Subroutine Calls	6-3
	6.3	SUBROUTINES	6-3
	6.3.1	Multilevel Subroutines	6-3
	6.3.2	Jump Immediate	6-4
	6.3.3	Conditional Subroutine Jumps	6-4
	6.3.4	Multiple Subroutine Return	6-4
	6.3.5	Transferring Data to Subroutines	6-5
	6.4	LOOP COUNTER	6-5
	6.5	PAGE CONSIDERATIONS	6-6
	6.5.1	Instructions at the Page Boundary	6-6
	6.5.2	Programs Residing Across Page Boundaries	6-6
	6.6	TEXT PROGRAMMING TECHNIQUES	6-6
	6.7	INPUT AND OUTPUT PROGRAMMING TECHNIQUES	6-8
	6.7.1	Programmed Input/Output	6-8
	6.7.2	Interrupt Input/Output	6-9

TABLE OF CONTENTS (Continued)

Chapter			Page
6 (Cont'd)	6.8	USING THE STATUS REGISTER	6-11
	6.8.1	General	6-11
	6.8.2	Arithmetic Operations	6-12
	6.8.2.1	Arithmetic with Unsigned Data Bytes	6-13
	6.8.2.2	Arithmetic with Signed Data Bytes	6-14
	6.8.3	Overflow and Carry/Link	6-14
	6.8.3.1	Add Operation with CY/L initially reset to 0	6-15
	6.8.3.2	Decimal Add Operation with CY/L initially reset to 0	6-15
	6.8.3.3	Complement and Add Operation with CY/L initially set to 1	6-15
7	(FORTRAN) CROSS ASSEMBLER PROGRAM		7-1
	7.1	INTRODUCTION	7-1
	7.2	INPUT AND OUTPUT	7-1
	7.2.1	Source File (Input)	7-1
	7.2.2	Program Listing File (Output)	7-2
	7.2.3	Load Module (Output)	7-2
	7.2.4	Format of LM File	7-2
	7.3	OBTAINING AN OBJECT CARD DECK	7-5
A	APPENDIX — ANSI CHARACTER SET		A-1
B	APPENDIX — OPCODE INDEX OF INSTRUCTIONS		B-1
C	APPENDIX — MNEMONIC INDEX OF INSTRUCTIONS		C-1
D	APPENDIX — INSTRUCTION FORMATS.		D-1
E	APPENDIX — INSTRUCTION EXECUTION TIMES		E-1
F	APPENDIX — DIRECTIVE STATEMENTS — INDEX		F-1
G	APPENDIX — PROGRAMMERS CHECKLIST		G-1
H	APPENDIX — PROGRAM DIAGNOSTIC MESSAGES		H-1
I	APPENDIX — (FORTRAN) CROSS ASSEMBLER (SAS) G.E. TIMESHARING OPERATING PROCEDURE		I-1
J	APPENDIX — (IMP-16) CROSS ASSEMBLER OPERATING PROCEDURE		J-1

LIST OF ILLUSTRATIONS

Figure		Page
2-1	Major Components of a Microcomputer	2-2
2-2	Flowchart for Simple Sort Routine	2-6
2-3	Typical Programming Process	2-7
3-1	SC/MP Registers	3-2
3-2	Memory Organization	3-4
3-3	Interface to Peripheral Device Controller	3-9
3-4	I/O Status	3-9
4-1	Sample Coding Form	4-2
4-2	Relationship of Terms	4-3
6-1	Programmed Input/Output	6-8
6-2	Interrupt Input/Output Initiation	6-10
7-1	LM File and General Formats	7-3
7-2	Title Record Format	7-4
7-3	Data Record Format	7-4
7-4	End Record Format	7-5
H-1	(IMP-16) Cross Assembler Error Detection, Listing Output	H-1
I-1	Preparing User's SAS$$$ Programs (General Electric Timesharing System)	I-4
J-1	Sample Listing of Assembler	J-7

LIST OF TABLES

Table		Page
3-1	Operational Features	3-1
3-2	Addressing Formats	3-6
4-1	Arithmetic and Logical Operators	4-6
5-1	Symbols and Notation	5-2
5-2	SC/MP Instruction Summary	5-3
5-3	Memory Reference Formats	5-5
5-4	Summary of Assembler Directives	5-24
6-1	Status Register Bits	6-11
A-1	ANSI Character Set in Hexadecimal Representation	A-1
A-2	Legend for Nonprintable Characters	A-2
J-1	Operator Selectable Options	J-4

Chapter 1

GENERAL INFORMATION

1.1 INTRODUCTION

SC/MP represents a significant breakthrough in low-cost computer systems. Providing many of the features of higher-priced systems, SC/MP has sufficient hardware to serve most controller and switching applications where processing speed is not a critical factor. With read/write memory, read-only memory, power supply, chassis, and console, SC/MP becomes a stand-alone microcomputer.

SC/MP is a programmable 8-bit parallel processor implemented on a single chip. One 8-bit accumulator, four 16-bit pointer registers (one dedicated as the Program Counter), an 8-bit status register, and an 8-bit extension register are provided. SC/MP can address 65,536 bytes of memory directly.

Architecturally, SC/MP uses a unified bus, whereby the CPU, memory, and peripheral devices are connected to a common data bus. This configuration enables memory-reference instructions also to reference peripheral devices.

The SC/MP assembly language is supported by two cross assemblers: (1) a (FORTRAN) Cross Assembler written in ANSI FORTRAN IV and (2) a (IMP-16) Cross Assembler written in the IMP-16 Assembly Language. Thus, SC/MP assembly language program listings and object modules may be generated on any computer with an ANSI FORTRAN IV compiler and sufficient memory or on an IMP-16 microprocessor.

1.2 SCOPE OF MANUAL

This manual describes assembly language programming for SC/MP. It contains tutorial and reference information needed for writing application programs.

The manual is structured so the user not familiar with computers can learn to generate code with a minimum of effort, and the experienced user is not hindered by the basic information included for the beginning user.

1.3 ORGANIZATION OF MANUAL

The following is a brief description of the contents of chapter 2 through 7.

Chapter 2, Basic Concepts, is a short introduction to microprocessors, assembly language programming, and the steps used to write and assemble a program.

Chapter 3, System Overview, describes the registers available to the user, the types of addressing used in SC/MP, and the input/output facilities of SC/MP.

Chapter 4, Assembly Language, describes the elements, the structure, and the coding conventions of the SC/MP assembly language.

Chapter 5, Statements, is a detailed description of the five statement types processed by SC/MP; the comment, the instruction, the pseudo-instruction, the assignment, and the directive.

Chapter 6, Programming Techniques, shows programming examples; how to write efficient code, how to address subroutines, and how to perform input/output (programmed and interrupt).

Chapter 7, (FORTRAN) Cross Assembler, is a description of the SC/MP Assembler Program and the formats of its input and output files.

Chapter 2

BASIC CONCEPTS

2.1 INTRODUCTION

This chapter discusses basic programming concepts so the user with little or no programming experience may be able to produce efficient assembly language code. The topics include components of a computer system and their functions, the architecture of a computer system, and a step-by-step approach to programming a computer. The user familiar with these basic topics may skip this chapter.

2.2 COMPUTERS, MICROCOMPUTERS, MICROPROCESSORS

The discussion of computers in this manual is limited to digital electronic computers. Keeping that in mind, our definition of a computer is an electronic device capable of executing a sequence of instructions stored in binary format.

Computers come in various sizes from very large to very small. The very large to medium size computers tend to be general-purpose machines, while the small (minicomputers) to very small (microcomputers) tend to be special-purpose machines. The smaller computers are often used as dedicated controllers. Microcomputers in particular serve this function well since they are small, sometimes contained on a single printed-circuit board, their power requirements are low, and they are the lowest-priced computers currently available.

The primary component of a microcomputer is the CPU (Central Processing Unit), normally referred to as the microprocessor. A microprocessor is a general term referring to any Large-Scale-Integration (LSI) function with processing power resembling that of a CPU. Microprocessor loosely covers various types of processors using large-scale integrated circuits; in other words, the prefix "micro" describes the scale of the circuit and not necessarily a microprogrammed architecture.

Because of the increasing concentration of processing power in LSI devices, microprocessors are being used in applications that until now have been the exclusive domain of minicomputers.

2.3 BASIC ELEMENTS OF A COMPUTER SYSTEM

Any computer system may be divided into two or three basic areas: hardware, software, and in some newer systems, firmware.

2.3.1 Hardware

Hardware refers to the physical equipment; the mechanical, magnetic, electrical, and electronic components of a computer. The major components of a microcomputer are shown in figure 2-1, and discussed in the paragraphs following.

The most important component of any computer is the CPU, the part that does the processing. The main elements of the CPU are the Control Unit and the Register, Arithmetic, and Logic Unit. The control unit fetches the instructions stored in memory, decodes, interprets, and implements them. It manages the temporary storage and retrieval of data, and regulates the exchange of information with the outside world through the input and output ports. Finally, it coordinates all the units in a timed logical sequence.

The Register, Arithmetic, and Logic Unit does the actual operations of the CPU under the direction of the control unit.

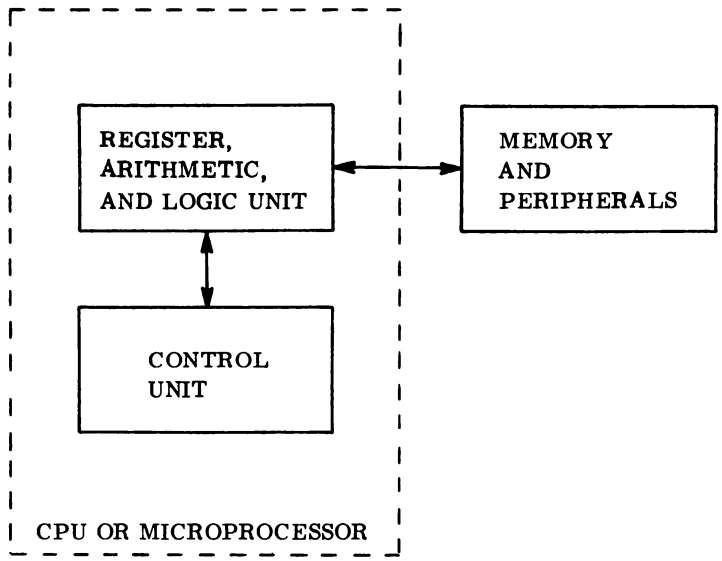


Figure 2-1. Major Components of a Microcomputer

Peripherals may be any input/output or storage device attached to the CPU by an address and data bus. Some examples of peripherals are Teletypes, card readers, line printers, CRTs, magnetic or paper tape units, disc units, and read-only or read/write memory.

2.3.2 Firmware

Firmware is a relatively new feature in computer systems but in microcomputer systems is rapidly becoming a standard feature. Firmware is a term that loosely covers programs resident in read-only memory (ROM) or Programmable Read-Only Memory (PROM). ROMs or PROMs generally contain programs that are a fundamental part of the microcomputer system and are not likely to change, such as loaders or a debug package.

2.3.3 Software

Software, in contrast to firmware, generally refers to those programs that reside in read/write memory (RAM). These programs are often maintained off-line on punched cards or paper tape, or on some kind of magnetic media such as tape or disc.

For a general-purpose computer system to perform a particular task, the software required to execute the task is loaded into the computer read/write memory (RAM). After the first task is completed, another set of programs may be loaded to perform another task. Thus, the software easily modifies the operation and use of the system.

Any software function may be implemented by firmware for a particular computer. In the remainder of this manual, no differentiation will be made between software and firmware.

The following is a list of typical software packages:

- DEBUG PROGRAMS — Debug programs aid the programmer in finding and correcting errors in his programs as they are running on the computer.
- DIAGNOSTIC PROGRAMS — These programs check the various hardware components of a system for proper operation.

- LOADERS — The various software packages and applications (user written) programs must be placed in the proper locations of the system memory. The programs that perform this task are called loaders.
- EDITOR — Editors are programs that aid in preparing source programs by allowing easy manipulation or editing of text material.
- INPUT/OUTPUT HANDLERS — Input/output handlers, sometimes called device drivers, are subroutines that service specific peripheral devices such as teletypewriters or card readers.

  In many systems, the standard input/output handlers are contained in firmware rather than software.
- SIMULATORS — Simulators are programs that simulate the operations of one computer on another computer. Simulators are especially useful if the actual computer is not available (or hasn't been built). If hardware is available, the use of a simulator may be an unnecessary extra step if the software must still be debugged on the hardware. The cost of the computer time to run the simulator effectively is often more than the cost of a development system.

2.4 PROGRAMMING LANGUAGES

Programming is communicating with the computer by a written language. In written English, there are rules about starting and ending sentences and paragraphs, spelling words, and so forth. A programming language has rules of spelling and punctuation also, but these rules are more strictly enforced. If you misspell a few words or incorrectly punctuate in written English probably you will still be understood. A computer will not produce the desired result if its language rules are broken.

There are a number of levels of programming languages. The most basic is machine language.

Each instruction of machine language is uniquely defined by a binary code of ones and zeros. The CPU examines each instruction code and performs the sequence of events to produce the operation defined by that instruction. For example, assume a 0000 0001 code tells the computer to exchange the contents of the accumulator and the extension register. When programming in machine language, the programmer must enter 0000 0001 to perform this instruction. This can be slow and awkward, and errors may be difficult to trace and correct. However, the use of machine language may be a reasonable way to program when the application is simple and must be accomplished on a low budget.

Assemblers were developed to make programming easier. An assembler is a computer program that accepts symbolic codes or "mnemonics" and translates them into binary machine code the computer can execute. Compared to machine codes, the mnemonics used for each instruction are much easier to remember and use, and they make a listing of the program much easier to read. For example, the mnemonic for the 0000 0001 code mentioned above might be XAE, for Exchange Accumulator and Extension Register.

The use of symbolic codes in place of the ones and zeros of machine language is not the only improvement that assemblers can provide. An assembler keeps track of the location of each instruction. This is important because it allows the programmer to use symbolic labels for important locations in the program. These labels allow references to be made to locations in a program without requiring the programmer to keep track of the exact memory locations (which might change if the program is modified).

In addition to allowing the use of mnemonics and labels, assembler listings include comments to document the program, macros that assign a mnemonic to groups of code, listings of labels and their locations, and flagging of errors.

At this point we must stop and define source programs and object programs and how they differ. A source program is a program written by the programmer in any symbolic language. The object program is the list of binary machine instructions (and data) that can be loaded into the computer for execution. The object program is produced from the source program by the assembler.

2.5 USING A COMPUTER

To solve a problem using a computer, the following sequence of operations may be followed from problem definition to loading the final object program.

- Problem Definition
- Program Flowchart
- Writing a Program
- Desk-checking Code
- Assembling a Program
- Loading
- Debugging

2.5.1 Problem Definition

The initial step in programming a computer for a particular application is problem definition. Problem definition requires specification of the following items.

- Outputs required from the program or programs.
- Inputs required for generating the outputs.
- Determination of how the outputs are generated from the inputs (the system transfer function).
- Determination of the acceptable response time (time required for system to react to particular inputs).
- Actions taken as a result of erroneous inputs, alarm conditions, or other interferences.

For a computer program to be well defined, the course of action to be taken must be specified for any possible combination of inputs.

An example problem might be the design of a subroutine that sorts a table of single-byte constants into ascending order in computer memory. The inputs to the subroutine are the addresses of the first and last bytes of the table. The output of the subroutine is the table itself, sorted in ascending order. There are no error conditions or alarm conditions that must be considered. However, sorting generally requires a rather significant amount of CPU execution time, so a method of determining when the table is completely sorted should be included in the sort subroutine so the sort may be terminated.

2.5.2 Program Flowchart

Step two consists of actually designing the program. An important tool used in program design is the program flowchart. The advantages of using a flowchart during program design are as follows:

- A flowchart shows the multidimensional aspects of program flow.
- Excessive branching within a program is shown by a flowchart.
- Function duplications within a program are more easily noticed.
- Program maintenance is made easier by the use of a flowchart.

Standard symbols for drawing program flowcharts have been specified by the American National Standards Institute (ANSI).

As an example, a program flowchart for the sort subroutine is shown in figure 2-2. The method used to sort the table compares two bytes at a time and interchanges the bytes if they are not in ascending order. Bytes one and two are compared, then byte two and three, and so on until the table is exhausted. In order to sort the table properly, multiple passes are required. On any particular pass, a flag is maintained to indicate whether or not an exchange was made. When no exchange occurred on the last pass, the table is completely sorted and the operation can be terminated.

2.5.3 Writing a Program

After a program is defined and flowcharted, it must be coded into assembly language and transferred to a media that is computer readable. Examples of such media are cards and paper tape. The actual program writing procedure involves writing the assembly language on a coding sheet and, then, transferring the program to the selected media.

In writing a program for a microcomputer, two areas of concern arise: memory management and register usage. Memory management relates to the organization of a program in main memory, and register usage relates to the dynamic allocation of hardware registers to various functions within the program.

The following suggestions are offered to aid the user in the process of coding his assembler language program.

- Follow the coding format suggested on the coding form. This results in a program that is easy to read and, therefore, easier to check out and to maintain.
- Neatness in coding results in fewer errors during transcription of the program to computer-readable media.
- As the program is being written, include comments describing the function of each major section, calling sequences for all subroutines, assumptions made, obscure coding techniques employed, and any other information useful for usage or maintenance.
- Subroutines contained in the program should be grouped at the end of the listing for ease of reference.

2.5.4 Desk-checking Code

After a program is written, significant amounts of assembly and checkout time may be saved if the program is desk-checked. Desk-checking a program consists of rereading a program to check it for accuracy. It might even extend to emulating manually computer execution of a portion of the program with pencil and paper.

A list of important points to consider during the desk-checking of a program is given in appendix G, Programmers Checklist.

2.5.5 Assembling a Program

The conversion of a source (assembler language) program into a form that can be loaded into the computer is performed during the assembly process by the assembler.

Two outputs are generated as a result of running a source program through the assembler program: (1) an object program consisting of loadable machine instructions corresponding to the source program statements, and (2) a program listing showing source statements side-by-side with the object code instructions created from the statements. Most programmers work with the program listing once it is available.

As a source program is assembled, it is analyzed for errors in the use of the assembler language. Any detected errors are indicated on the program listing to assist the programmer in debugging.

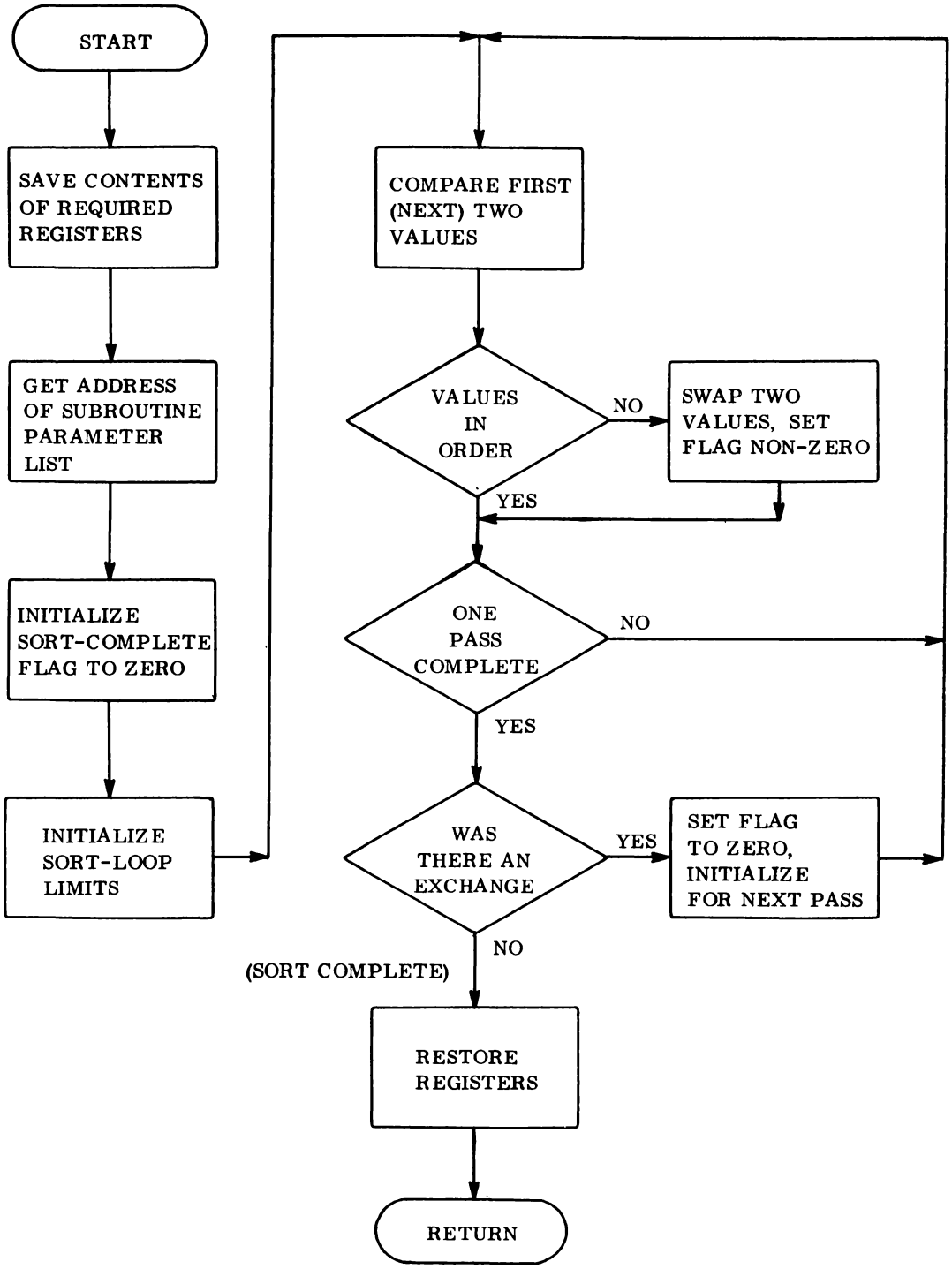
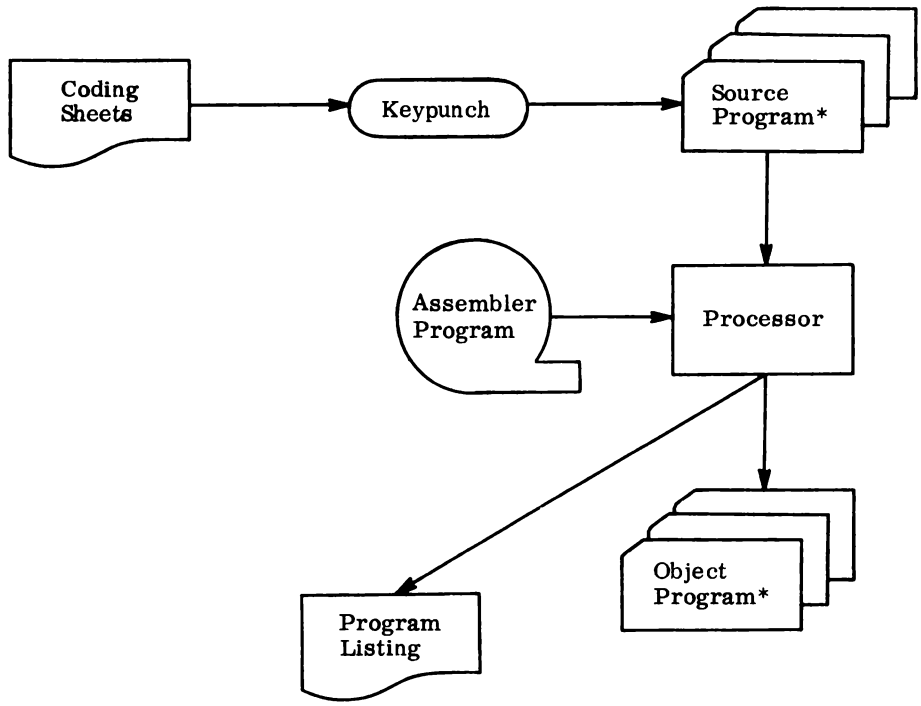


Figure 2-2. Flowchart for Simple Sort Routine

The flowchart in figure 2-3 shows the relationship of the assembler program to the programming process.



* Cards, paper tape, or magnetic tape

Figure 2-3. Typical Programming Process

Some assembler programs, called one-pass assemblers, completely process the symbolic code during one pass. Other assemblers make two passes through the source code. On the first pass, the assembler program determines the number of words of storage required for each statement and assigns a value for the first location in every statement line. It generates the machine language program and assembly listing during pass two.

2.5.6 Loading

The object load module produced by the assembler is loaded into the computer from cards or paper tape, using an absolute or a relocating loader. The absolute loader is used for load modules that have been specified at assembly time to be loaded into specific memory locations. The relocating loader is used for load modules that may be loaded into memory locations specified at load time. For example, the starting address of an absolute load module may be X'100; the program must be loaded starting at that address and no other. The relocatable load module could be loaded at X'100 or X'200, or any location the programmer cared to use.

2.5.7 Debugging

Errors flagged by the assembler, or errors discovered while running the program may be corrected with a debug package. A debug package normally consists of a trace routine for evaluating code (instruction-by-instruction), a routine for dumping portions of memory, an editor for correcting errors in the source code, and a patch routine to temporarily correct the object code.

Chapter 3

DEVELOPMENT SYSTEM OVERVIEW

3.1 GENERAL

This chapter describes the main features of the SC/MP microprocessor system. Only those features the programmer is primarily concerned with are discussed. Detailed information on the system development hardware is contained in the appropriate users manual. Detailed information on the SC/MP device is contained in the SC/MP Data Sheet.

3.2 DEVELOPMENT SYSTEM CONFIGURATION

The SC/MP is an 8-bit parallel processor with 16-bit memory and peripheral device addressing. Functionally, SC/MP has a bidirectional data bus connecting the CPU, memory, and peripheral devices. Peripheral devices are assigned memory addresses, and any standard memory reference instruction can be used for input/output operations. Memory is expandable to 65,536 bytes. Table 3-1 lists the operational features of SC/MP.

Table 3-1. Operational Features

Data Length	8 Bits (Byte)
Instruction Set	46 Instructions
Arithmetic	Parallel, binary, fixed point, twos complement 2-digit BCD addition
Memory	Up to 65,536 bytes
Registers	One 8-bit Accumulator One 8-bit Status Register One 8-bit Extension Register Four 16-bit Pointer Registers (one is the Program Counter)
Addressing Modes	Program Counter Relative Indexed Auto-indexed Immediate
Input/Output and Control	16-bit Address Bus 8-bit Bidirectional Data Bus

3.3 REGISTERS

The seven registers available to the SC/MP assembly language programmer are shown in figure 3-1 and are discussed in the following paragraphs.

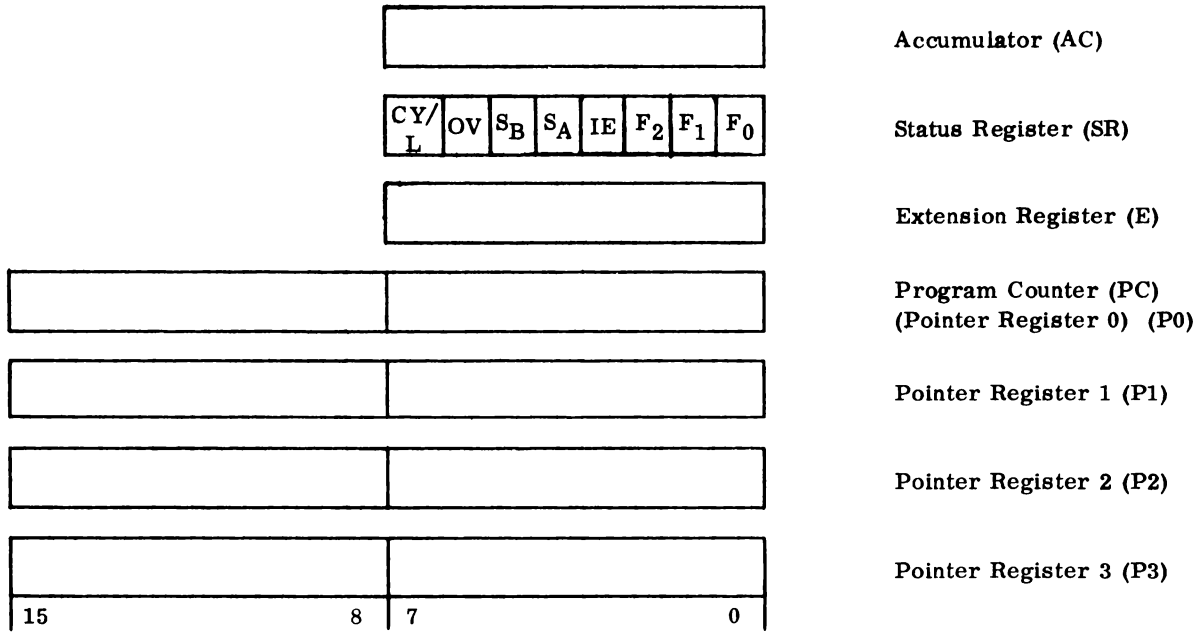


Figure 3-1. SC/MP Registers

3.3.1 Accumulator (AC)

The 8-bit Accumulator (AC) is the primary working register of SC/MP. The accumulator is used in performing arithmetic and logic operations and for storing the results of those operations. Data transfers, shifts, and rotates also use the accumulator. In all, 37 of the 46 SC/MP instructions use the accumulator.

3.3.2 Status Register (SR)

7	6	5	4	3	2	1	0	Bit Positions
CY/L	OV	S_B	S_A	IE	F_2	F_1	F_0	Flags

The Status Register (SR) provides storage for arithmetic, control, and software status flags. The function of each bit in the register is shown below.

Bit	Description
0	User Flag 0 (F0). User assigned for control function or for software status. The output of this bit is available at a pin of the SC/MP device.
1	User Flag 1 (F1). Same as F0.
2	User Flag 2 (F2). Same as F0.
3	Interrupt Enable Flag (IE). The processor recognizes the interrupt input if this flag is set.
4	Sense Bit A (SA). This bit is tied to a package pin and may be used to sense external conditions. This bit is "read-only"; thus, the Copy Accumulator to Status Register (CAS) instruction does not affect this bit. When Interrupt Enable is set, Sense Bit A serves as the interrupt input.

Bit	Description
5	Sense Bit B (SB). Same as SA, except it is not used as an interrupt input.
6	Overflow (OV). This bit is set if an arithmetic overflow occurs during an add (ADD, ADI, or ADE) or a complement-and-add instruction (CAD, CAI, or CAE). Overflow is not affected by the decimal-add instructions (DAD, DAI, or DAE).
7	Carry/Link (CY/L). This bit is set if a carry from the most significant bit occurs during an add, a complement-and-add, or decimal-add instruction. The bit is also included in the Shift Right with Link (SRL) and the Rotate Right with Link (RRL) instructions. CY/L is input as a carry into the bit 0 position of the add, complement-and-add, and decimal-add instructions.

3.3.3 Extension Register (E)

The 8-bit Extension Register (E) is used primarily with the accumulator to perform arithmetic, logic, and data-transfer operations. If the displacement in an indexed or an auto-indexed memory reference instruction equals -128_{10} (X'80), then the contents of E are substituted for the displacement for the given instruction.

Another function of the Extension Register is serial input/output. This feature is explained in detail in the description of the Serial Input/Output Instruction (SIO) in Chapter 5.

3.3.4 Program Counter (PC)

The Program Counter (PC) is the dedicated 16-bit Pointer Register P0. The Program Counter contains the address of the instruction being executed. In the event of an interrupt or a subroutine call, the contents of the Program Counter may be stored on a software stack and retrieved when the processor returns to the main program. The use of a software stack is explained in chapter 6.

The Program Counter is incremented just prior to an instruction fetch. Therefore, the effective address of any transfer of control should be one less than the actual address to be executed (taking into account the modulo 2^{12} address arithmetic as explained in section 3.4).

Arithmetic affecting the Program Counter is performed on the low-order twelve bits; the high-order four bits are unaffected. A further explanation of this may be found in section 3.4.

3.3.5 Pointer Registers (PTR)

There are three 16-bit Pointer Registers (PTR) available for memory and peripheral device addressing, and for use as page pointers, stack pointers, or index registers. Typically the programmer assigns a specific function to each register. The following assignments are used typically in the SC/MP development system software.

P1 — ROM Pointer

P2 — Stack Pointer

P3 — Subroutine Pointer

As mentioned previously, P0 is assigned the function of program counter by the design of the hardware.

3.4 MEMORY ADDRESS STRUCTURE

Memory is organized as a sequence of 8-bit bytes. Each byte is identified by a 16-bit address that represents its sequential position in memory from 0 to X'FFFF ($65,535_{10}$).

In the internal architecture of the computer, memory is divided into 16 pages of 4,096 bytes each, as shown in figure 3-2. Each address consists of a 4-bit page address and a 12-bit page displacement.

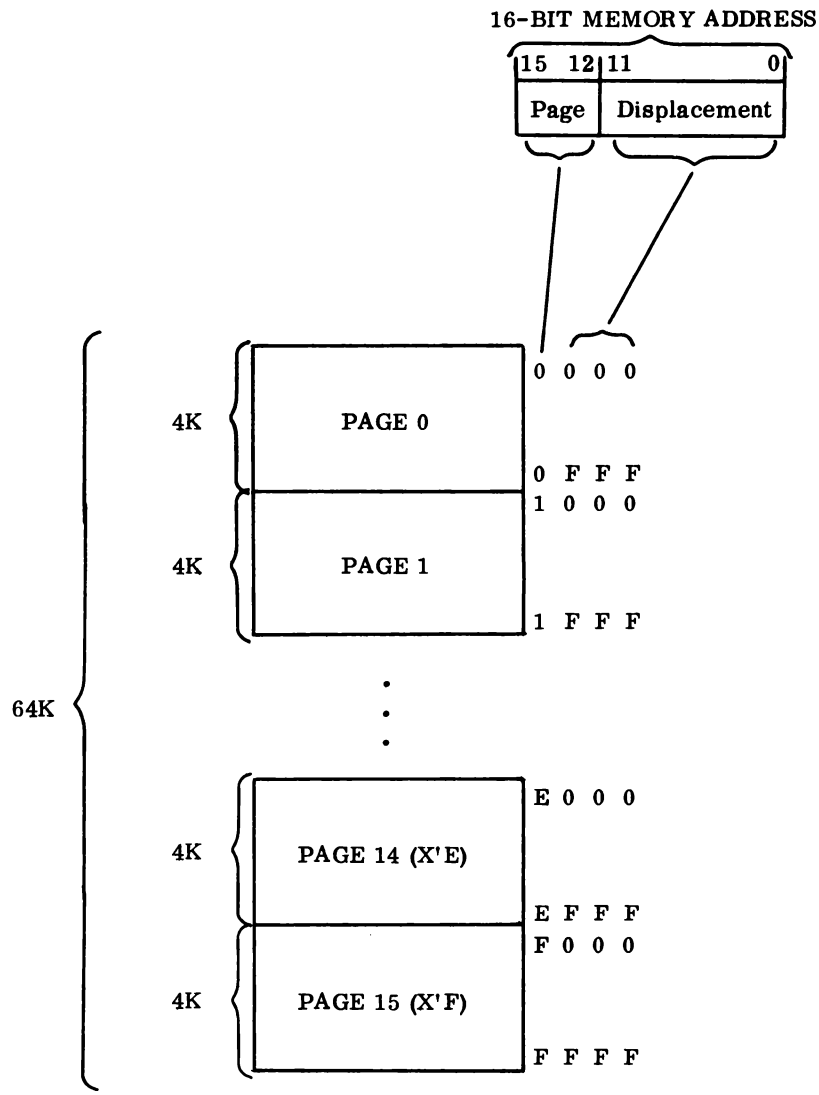


Figure 3-2. Memory Organization

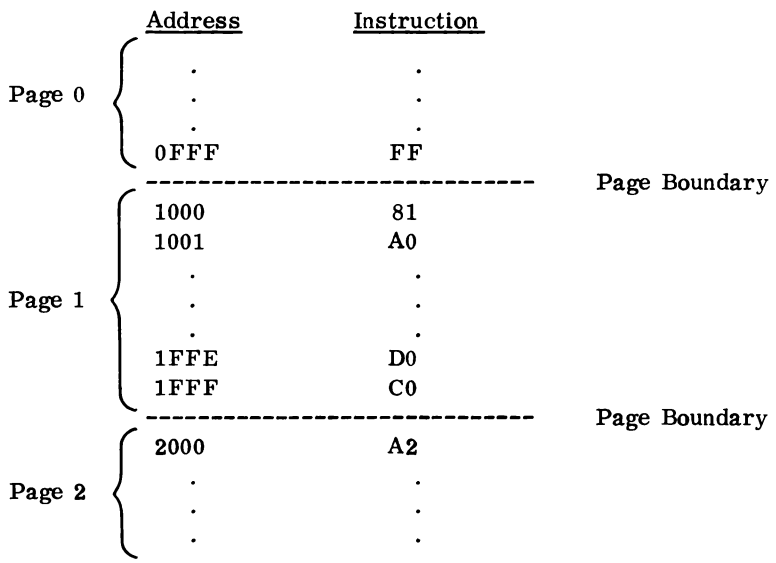
When performing arithmetic to calculate the effective address of an operand, the calculations are performed on the low-order (displacement) portion of the address with no carry into the high-order (page) portion. For example:

	Address Displacement Remains Within Page		Address Displacement Exceeds Page Size	
	Address of Page	Displacement Within Page	Address of Page	Displacement Within Page
Current Address	0	FB4	0	FB4
Displacement From Instruction		05		4D
New Address	0	FB9	0	001

As shown in the previous example, when the address displacement remains within the current page, no carry is generated because the sum of the displacements did not produce a carry. In the example where the displacement exceeds the page size, a carry is normally generated when the two numbers are summed, but it is not carried into the page address field.

When incrementing the address to fetch the next instruction, the same page/displacement arithmetic occurs.

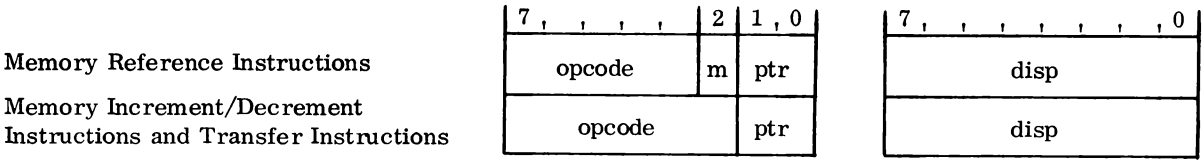
If a two-byte instruction is inadvertently separated by a page boundary, an error occurs. Consider the following sequence of instructions on pages 0, 1, and 2 — with the first digit of the address designating the page and the next three digits, the location within the page.



The instruction intended, when the PC = 1FFF (last word in page 1), is X'C0A2 (LD 20A2). However, instead of fetching the latter half of the instruction from page 2, a wrap-around is made to the first word of page 1; the instruction that will be executed is X'C081 (LD 1081). The SC/MP assembler assumes the user will organize his programs in pages of 4,096 words each to provide protection from the situation described above. If a boundary violation occurs, the assembler issues an alarm message.

3.5 METHODS OF ADDRESSING

During execution, instructions and data defined in a program are stored into and loaded from specific memory locations, the accumulator, or selected registers. Because the CPU, memory (read/write and read-only), and peripherals are on a common data bus, any instruction used to address memory may also be used to address the peripherals. The formats of the instruction groups that reference memory are shown below.



Memory-reference instructions use the PC-relative, indexed, or auto-indexed methods of addressing memory. The memory increment/decrement instructions and the transfer instructions use the PC-relative or indexed methods of addressing.

The various methods of addressing memory and peripherals are shown in table 3-2.

Immediate addressing is an addressing mode specific to the immediate instruction group.

Table 3-2. Addressing Modes

Type of Addressing	Operand Formats		
	m	ptr	disp
PC-relative	0	0	-128 to +127
Indexed	0	1, 2, or 3	-128 to +127
Immediate	1	0	-128 to +127
Auto-indexed	1	1, 2, or 3	-128 to +127

For PC-relative, indexed, and auto-indexed memory-reference instructions, another feature of the addressing architecture is that the contents of the extension register are substituted for the displacement if the instruction displacement equals -128 (X'80).

3.5.1 PC-Relative Addressing

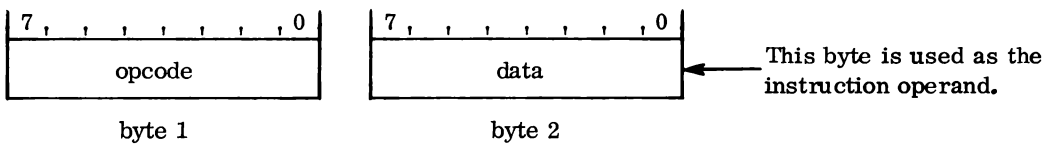
A PC-relative address is formed by adding the displacement value specified in the operand field of the instruction to the current contents of the program counter. The displacement is an 8-bit twos-complement number, so the range of the PC-relative addressing format is -128_{10} to $+127_{10}$ bytes from the current location of the Program Counter. During execution of an instruction, the program counter contains the address of the last byte of the instruction. The following examples show the use of PC-relative addressing.

```
Location    Generated
Counter     Code
0005        C00E        LOOP:   LD      TEMP    ;LOAD THE VALUE IN TEMPORARY STORAGE
                                .
                                .
                                .
000E        90F5                JMP     LOOP    ;REPEAT
                                .
                                .
                                .
0014        04          TEMP:   .BYTE   X'04
```

The assembler assumes PC-relative addressing in the memory-reference and transfer instructions when no pointer-register operand is specified.

3.5.2 Immediate Addressing

Immediate addressing uses the value in the second byte of a double-byte instruction as the operand for the operation to be performed (see below).



For example, compare a Load (LD) instruction to a Load Immediate (LDI) Instruction. The Load Instruction uses the contents of the second byte of the instruction in computing the effective address of the data to be loaded. The Load Immediate Instruction uses the contents of the second byte as the data to be loaded. Because the operand occurs as the second byte of a two-byte instruction, page boundary conditions should be observed as mentioned in 3.4.

3.5.3 Indexed Addressing

Indexed addressing enables the programmer to address any location in memory through the use of the pointer register and the displacement. When indexed addressing is specified in an instruction, the contents of the designated pointer register are added to the displacement to form the effective address. The contents of the pointer register are not modified by indexed addressing. Indexed addressing is used to access tables or subroutines, to transfer control to another page, or to transfer control to a section of the current page that is outside the range of the PC-relative transfer. The rules for page boundaries still apply, so the user is cautioned about crossing page boundaries when using indexed addressing to access tables. Such a reference results in a wrap-around from the end to the beginning of the page, or vice-versa (see 3.4).

3.5.4 Auto-Indexed Addressing

Auto-indexed addressing provides the same capabilities as indexed addressing along with the ability to increment or decrement the designated pointer register by the value of the displacement.

If the displacement is less than zero the pointer register is decremented by the displacement before the contents of the effective address are fetched or stored. If the displacement is equal to or greater than zero, the pointer register is used as the effective address, and the pointer register is incremented by the displacement after the contents of the effective address are fetched or stored.

NOTE

The contents of the pointer register are modified by auto-indexed addressing.

An at sign '@' before the displacement operand designates an auto-indexed operation. Example:

```
Generated
Code
C601        LD      @1(P2)          ;GET A BYTE FROM THE TABLE, AUTO-INDEX
```

3.6 INPUT/OUTPUT FACILITIES

SC/MP uses a single bidirectional input/output bus to interconnect the CPU, memory, and peripheral devices. Peripheral devices are assigned memory addresses, so standard memory-reference instructions can be used for input/output operations.

Peripheral device addressing, data exchange, status reporting, and control signal operations are performed by an external device controller. Because of variations in peripheral devices, depending on the function performed, a standard input/output operation cannot be described here. Similarly, the device controller operations vary widely, depending on the peripheral device being serviced. SC/MP provides the following facilities, which may be used in various applications, providing they match the device controller in use. Refer to figure 3-3.

- 16-bit Address
- 8-bit Parallel Input/Output
- 1-bit Serial Input/Output
- 3 Flag Outputs
- 2 Sense Inputs

3.6.1 Address Lines

The 12-bit address lines contain the displacement portion of the effective memory address generated in response to a memory-reference instruction. The 4-bit page portion of the effective memory address is output on the data bus (see 3.6.4). Because peripheral devices are assigned memory addresses, a decoder in the associated device controller looks for its address or addresses. (Multiple addresses may be assigned to multifunction devices.) Details of interfacing hardware to SC/MP are found in the SC/MP Data Sheet and the SC/MP Users Manual.

3.6.2 Parallel Input/Output

An 8-bit bidirectional data bus transfers data between the peripheral device controller and the processor. This data is associated with the memory reference instruction that addressed the peripheral device. It is the function of the peripheral device controller to place the data on the line when the device is addressed for input; and to transfer data from the lines when the device is addressed for output. The direction of the data transfer depends on the nature of the memory-reference instruction.

3.6.3 Serial Input/Output

Serial Input/Output is provided by using one of the eight data lines, or dedicated flag and sense input, or the extension register as a serial input/output shift register.

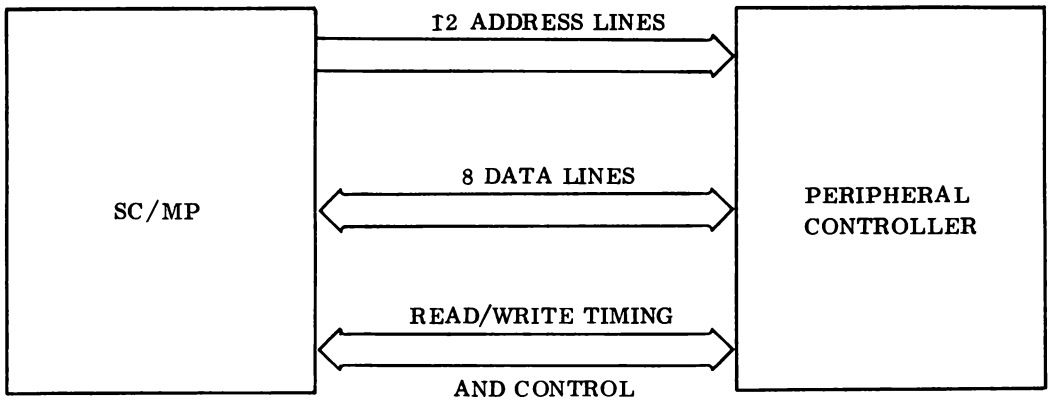


Figure 3-3. Interface to Peripheral Device Controller

In systems that have only one serial input/output device, the serial input and output pins may be tied directly to the input/output device and no address decoding is necessary. The Serial Input/Output Instruction (SIO) is then used for serial input/output. Timing may be provided by program loops using the delay instruction or by an external timing element that is tested by the jump condition inputs. For asynchronous systems a flag may be pulsed each time a new bit is shifted in/out, and a sense condition tested to detect bit received/ready.

Systems that have several serial input/output devices, must be multiplexed, and device selection may be provided by the control flags, or by use of the parallel input/output commands to load an external latch.

The serial data input and output pins may be used as sense input and flag output lines in systems that do not require the serial input/output function.

3.6.4 I/O Status

The I/O status (figure 3-4) is output on the data bus, along with the appropriate timing information on the timing lines, so that the peripheral controller has the additional information available if it is required by a particular system. Two of the I/O status bits are for hardware functions (R-flag and I-flag); the remaining six I/O status bits are generated under software control.

7							0
H	D	I	R	A15	A14	A13	A12

A15 to A12 — Four most significant (page) address bits
H — H-flag generated by HALT instruction
D — D-flag generated by DLY instruction
I — I-flag generated by hardware for instruction fetch cycle
R — R-flag generated by hardware for read I/O cycle

Figure 3-4. I/O Status

Chapter 4

ASSEMBLY LANGUAGE

SC/MP assembly language statements have well-defined formats constructed from the elements described in this chapter.

4.1 CHARACTER SET

Statements are written using the following letters, numbers, and special characters:

Letters:	A through Z
Numbers:	0 through 9
Special Characters:	! $ % & ' ( ) * + , - . / : ; = @ ␣
	Note: ␣ means blank

Any of the printable characters listed in appendix A, "ANSI Character Set in Hexadecimal Representation," may be specified with the ASCII Directive Statement. Other nonprintable characters (those consisting of multiple letters) may be generated by using one or more hexadecimal constants in a .BYTE Directive Statement. Directive statements are described in chapter 5.

4.2 ASSEMBLER CODING CONVENTIONS

Assembly language programs are structured around source statements that contain from one to five fields as follows: label (optional), operation (mandatory), operand (usually required), comment (optional), and identification sequence (optional). The fields must be entered in the following order with one or more blanks separating each field:

[label field] operation field operand field [comment field] [identification field]

A sample coding form shown in figure 4-1 has the five fields delineated. However, since the assembler program accepts free-form statements, the programmer is permitted to disregard field boundaries. Use of field boundaries, wherever possible, is highly recommended.

The entry in each of the five fields must meet certain specifications and, in many cases, the programmer must understand how the assembler program executes certain types of instructions in order to code legal statements. The following paragraphs describe the entry requirements for the five fields.

4.2.1 Label Field

The label field is optional and may contain a symbol used to identify the current statement when referenced in other statements. More than one label may appear in the label field, in which case any of the labels may be used to reference the labeled location. A label may appear by itself in a statement, in which case it refers to the next instruction or data word in the source program.

CODING FORM

PROGRAM	PUNCHING INSTRUCTIONS	GRAPHIC									PAGE OF
PROGRAMMER		PUNCH									DATE

STATEMENT

LABEL 1 2 3 4 5 6 7 8	OPERATION 9 10 11 12 13 14 15 16	OPERAND 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	COMMENTS 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	IDENTIFICATION 73 74 75 76 77 78 79 80
1 2 3 4 5 6 7 8	9 10 11 12 13 14 15 16	17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80

Figure 4-1. Sample Coding Form

The following rules apply to labels:

1. A label may contain from 1 to 32 alphanumeric characters and must conclude with a colon (:); for example, TABEND:. Only the first six characters are used by the assembler to uniquely identify a label.
2. The first character must be alphabetic or a dollar sign ($).
3. Blanks cannot appear within the label.
4. For nonlocal labels (that is, ones that do not begin with a $), the first six characters must be unique. For local labels, the first five characters including the $ must be unique (see 5.5.9 .LOCAL Directive).

A space is not necessary between labels or following the label field.

4.2.2 Operation Field

The operation field is mandatory and contains a mnemonic operation code (opcode) defining an assembler or a machine operation.

Operation mnemonics are used in directive and instruction statements. Instruction statements define the machine operations necessary to perform the desired function. Valid operation mnemonics for instruction statements are defined in detail in 5.2. Directive statements control the process of program assembly and may generate data.

4.2.3 Operand Field

The operand field contains entries that identify data to be acted upon by the statement. A space is not required to terminate the field. An operand entry is composed of one or more terms which represent a value. The value may be inherent in the term, in which case the term is self-defining (4.2.3.1) or the value may be assigned by the assembler program during assembly, in which case the term is symbolic (4.2.3.2). An arithmetic combination of terms is reduced to a single value by the assembler program as described in 4.2.3.3. The relationship of terms is shown in figure 4-2. The various types of terms are described in the following paragraphs.

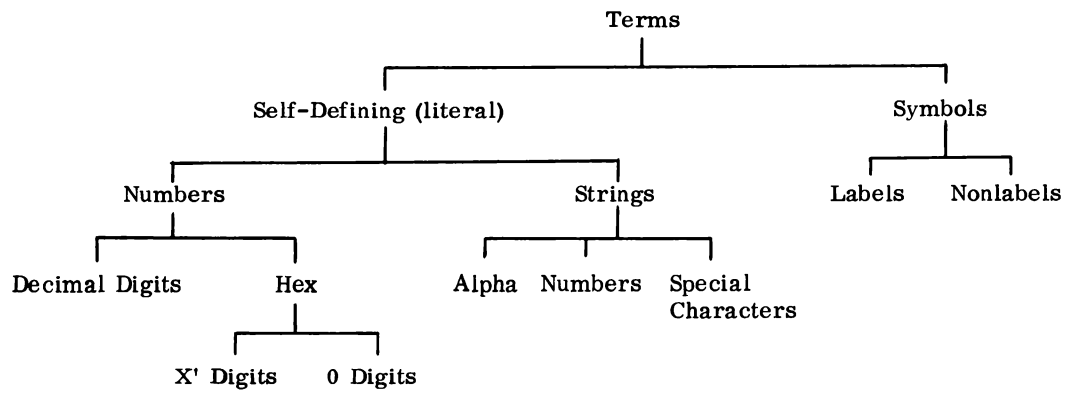


Figure 4-2. Relationship of Terms

4.2.3.1 Self-Defining Terms

A self-defining term has its value inherent in the term. The assembler program does not assign a value to the term, but derives the value from the term.

Self-defining terms are used to specify immediate data, addresses, registers, and input/output information to the assembler program. Three types of self-defining terms are available: decimal, hexadecimal, and character (or string).

A decimal self-defining term is zero, or a positive decimal integer that does not begin with zero. The allowable range of decimal numbers is 0 through 65,535. Examples: 32761, 10, 5, 0.

A hexadecimal self-defining term may be specified in either of two ways. The term may start with X'; or the term may start with a leading zero. The range of hexadecimal numbers is 0_{16} to $FFFF_{16}$. Examples: X'A2, 0A2, X'1234, 01234, 0, X'0, 0ABCD, X'ABCD.

A character self-defining term is defined as a string. A string is a series of characters or a single character enclosed in single quote marks (for example, 'THIS IS A STRING'). All letters, numbers, and special characters (including blanks) may be specified in a string. If a single quote mark is part of the character string, it should immediately be preceded by another single quote mark; for example, 'DON''T DO IT' represents DON'T DO IT. String characters are translated to ASCII code (see appendix A) in memory with each character occupying 8 bits. Refer to the .ASCII directive described in 5.5.8.

```
                        Self-defining term
                            ⌒
LDI                      X'AB
```

A null string ('') will cause the assembler to generate a single blank.

4.2.3.2 Symbolic Terms

Symbols are the most common means of referencing address locations or arbitrary values. Symbols are defined (assigned values) by one of three methods:

1. By appearing in a label field in a statement (see 4.2.1).

   ```
   symbol
     ⌒
   SUB1:    LDI      0          ;CLEAR AC
   ```

   The value assigned to a symbol appearing in the label field is the address of the instruction, data, or storage location named by the symbol.

2. By using an assignment statement to assign a specific value to a symbol (see 5.4).

   ```
   symbol
     ⌒
   P2 = 2                      ;STACK POINTER
   ```

3. By using a .FORM directive statement to assign a value to a symbol (see 5.5.12).

   ```
              symbol
                ⌒
   .FORM      DATA,2,2,4(X'A)
   ```

Symbol construction must meet the following restrictions:

1. A symbol may contain one or more alphanumeric characters, the first of which must be either a letter or a dollar sign ($).

2. Although up to 32 characters may be included, only the first six characters are recognized by the assembler program. Therefore, the programmer must ensure that a long symbol is unique in the first six characters.

   Example:
   ```
   LONGSY
   LONGSYMBOL1   } are identical to the assemblers
   LONGSYMBOL2
   ```

3. If the first character in the symbol is a dollar sign ($), the symbol is defined as a local symbol. The .LOCAL operator allows the programmer to specify that local symbols appearing between two .LOCAL directive statements have a certain meaning only within that region of the program (see 5.5.9 .LOCAL Directive). This enables the programmer to use common mnemonics throughout a program without causing a conflict of names.

   NOTE

   Within a local region, a long local symbol must be unique in the first five characters, including the dollar sign ($).

   Example:
   ```
   $ABCD
   $ABCDEF   } are identical symbols to the assembler.
   ```

4. No special characters or embedded blanks may appear within a symbol.

5. Symbol values cannot exceed a positive value of 65,535 or a negative value of 32,768 for 16-bit data; or 255 and 128, respectively, for 8-bit data.

Several examples of symbols follow:

Legal Symbols	Illegal Symbols
$ABC	
LONGSYMBOL	
$AB2	2AB
$2	#CDE
XYZ	XYZ$
$ABCDEF	
$ABC2EF	

A program assembled on the (FORTRAN) Cross Assembler may contain 900 symbols. A program assembled on the (IMP-16) Cross Assembler may contain about 175 symbols if the 4K version is used or 715 symbols if the 8K version is used.

A symbolic term may represent a memory address and, hence, may have a value ranging between 0 and $65{,}535_{10}$. Since SC/MP is an 8-bit machine, such a value will require two 8-bit bytes for its containment. In order to facilitate working with such values, they have been divided by the assembler into two halves, designated the "high" and the "low" parts of the value. The high part represents the upper half of the value (bits 15-8) and the low part represents the lower half (bits 7-0). These may be referred to in assembly language by using the forms, H(SYMBOL) and L(SYMBOL). For example:

```
Address represented by SYMBOL  = 58615(10) = 0E4F7
H(SYMBOL)                      =   228(10) = 0E4
L(SYMBOL)                      =   247(10) = 0F7
```

The forms, H(SYMBOL) and L(SYMBOL), may be used in any context where an 8-bit value would be appropriate.

In the previous example, note the following:

```
OTHER = SYMBOL+1  =  0E4F8
        H(OTHER)  =  0E4
        L(OTHER)  =  0F8
H(SYMBOL)+1       =  0E5
L(SYMBOL)+1       =  0F8
```

4.2.3.3 Expressions

Operand entries, consisting of either a single term or an arithmetic or logical combination of terms, are called expressions. Expressions are either simple or multiterm. Simple expressions are single terms, such as a symbol or a self-defining term. Multiterm expressions are formed in the same manner as normal arithmetic expressions and are evaluated by the assembler program in a strict left-to-right order without regard to treating a particular operator before any other. Parentheses are not permitted for the purpose of grouping arithmetic and/or logical operations; they have special significance in defining certain assembler functions.

Examples:

```
L(TABLE) + X'10
100 - 1
ENTRY1 + ENTRY2 - 4
```

Expressions are evaluated as 16-bit values.

Table 4-1 lists the arithmetic and logical operators available for forming expressions.

A unary operator operates upon one operand and appears in the format 'op opnd' (for example, -9). A binary operator operates upon two operands and appears in the format '$opnd_1$ op $opnd_2$' (for example, A&B).

Table 4-1. Arithmetic and Logical Operators

Operator	Function	Type
+	Addition	Binary
-	Subtraction	Unary or binary
*	Multiplication	Binary
/	Division	Binary
%	Logical NOT	Unary
&	Logical AND	Binary
!	Logical OR	Binary

4.2.4 Comment Field

Comments are optional descriptive notes printed on the program listing for programmer reference. Comments should be included throughout the program to explain subroutine linkages, assumptions made, formats of inputs processed, and so forth. A comment may follow a statement, or the comment may be entered on a separate statement line(s) since the comment has no affect on the assembled program and is printed only on the listing.

The following conventions apply to comments:

1. A comment must be preceded by a semicolon (;).
2. All valid characters, including blanks, may be used in comments.
3. Comments should not extend beyond column 72, but a comment may be carried over on the following line (preceded by a semicolon).

4.2.5 Identification Sequence Field

The identification sequence field is an optional entry that specifies program identification and/or statement sequence characters. If the field or a portion of the field is used for program identification, the identification is punched in the statement cards and is listed on the program listing. This field is generally not used with paper tape input.

As an aid to keeping source statements in order, the programmer may code a sequence of characters in ascending order in the identification sequence field.

The identification sequence field is fixed in columns 73 through 80 of the source image. Columns 73 through 80 are ignored by the assembler but are printed in the program listing.

4.2.6 Example Statement

An example assembler statement is as follows:

Label	Operation	Operand	Comment
LOOP:	LD	1(P2)	;GET A VALUE

The label, LOOP, is used to refer to the example statement in later (or previous) statements; in effect, to loop back to the statement. The mnemonic operation code, LD, stipulates the type of operation. The operand field specifies a pointer, P2, and a displacement, +1, and the comment field contains a note that may be used by the programmer to identify quickly the action defined by the statement. See chapter 6 for other statement examples.

Chapter 5

STATEMENTS

The SC/MP assembler accepts five types of statements: comment, instruction, pseudo instruction, assignment, and directive.

5.1 COMMENT STATEMENTS

Comment statements are defined by a semicolon (;) in the first character position of the record. They do not generate code, but serve only to document the symbolic output listing of the program. For example:

```
;
;THIS IS A COMMENT STATEMENT
;
```

5.2 INSTRUCTION STATEMENTS

The assembly language instruction set of the SC/MP provides arithmetic, logic, shift, transfer, and other operations between the accumulator and memory or the other registers.

Instruction statements, when assembled, generate the object (machine) code that defines the operations the processor will perform. Depending on the instruction type, one or two bytes of object code are generated for each instruction assembled.

In the following descriptions, any user accessible register or bit that is not explicitly mentioned will not be altered by the instruction.

There are 46 SC/MP instruction statements that comprise the following eight classes:

- Memory Reference
- Memory Increment/Decrement
- Immediate
- Transfer
- Extension Register
- Pointer Register Move
- Shift, Rotate, and Serial Input/Output
- Miscellaneous

Refer to table 5-1 for definitions of the symbols used in the notation for describing the SC/MP instruction set. Upper-case mnemonics refer to units designated by fields of the instruction words; lower-case mnemonics refer to the numerical values of the corresponding fields. For example, ptr in an assembler statement denotes the number of a pointer register, whereas (AC) ← (PTR) denotes the contents of the accumulator are replaced by the contents of a pointer register. In the case where both a lower-case mnemonic and an upper-case mnemonic are composed of the same letters, only the lower-case mnemonic is given in table 5-1. Lower-case notation designates a variable.

The SC/MP instruction set is summarized in table 5-2.

Table 5-1. Symbols and Notation

Symbol and Notation	Meaning
AC	8-bit Accumulator.
CY/L	Carry/Link Flag in the Status Register.
data	Signed, 8-bit immediate data field.
disp	Displacement, represents an operand in a nonmemory reference instruction or an address modifier field in a memory reference instruction. It is a signed twos-complement number.
EA	Effective Address as specified by the instruction.
E	Extension Register; provides for temporary storage, variable displacements and separate serial input/output port.
i	Unspecified bit of a register.
IE	Interrupt Enable Flag.
m	Mode bit, used in memory reference instructions. Blank parameter sets m = 0, @ sets m = 1.
OV	Overflow Flag in the Status Register.
PC	Program Counter (Pointer Register 0); during address formation, PC points to the last byte of the instruction being executed.
ptr	Pointer Register (ptr = 0 through 3). The register specified in byte 1 of the instruction.
$ptr_{n:m}$	Pointer register bits; n:m = 7 through 0 or 15 through 8.
SIN	Serial Input pin.
SOUT	Serial Output pin.
SR	8-bit Status Register.
( )	Means "contents of." For example, (EA) is contents of Effective Address.
[ ]	Means optional field in the assembler instruction format.
~	Ones complement of value to right of ~.
→	Means "replaces."
←	Means "is replaced by."
↔	Means "exchange."
@	When used in the operand field of the instruction, sets the mode bit (m) to 1 for auto-incrementing/auto-decrementing indexing.
10^+	Modulo 10 addition.
∧	AND operation.
∨	Inclusive-OR operation.
⊽	Exclusive-OR operation.
≥	Greater than or equal to.
=	Equals.
≠	Does not equal.

Table 5-2. SC/MP Instruction Summary --- Double-Byte

Description	Op Code	Source Statement	Object Format	Operation	Micro-cycles	Page
Memory Reference Instructions			7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0			
Load	C0	LD	1 1 0 0 0 m ptr | disp	(AC)←(EA)	18	5-5
Store	C8	ST	1 1 0 0 1	(EA)←(AC)	18	5-6
AND	D0	AND	1 1 0 1 0	(AC)←(AC) ∧ (EA)	18	5-6
OR	D8	OR	1 1 0 1 1	(AC)←(AC) ∨ (EA)	18	5-6
Exclusive-OR	E0	XOR	1 1 1 0 0	(AC)←(AC) ▽ (EA)	18	5-6
Decimal Add	E8	DAD	1 1 1 0 1	(AC)←$(AC)_{10}$ + $(EA)_{10}$ + (CY/L); (CY/L)	23	5-7
Add	F0	ADD	1 1 1 1 0	(AC)←(AC) + (EA) + (CY/L); (CY/L), (OV)	19	5-7
Complement and Add	F8	CAD	1 1 1 1 1	(AC)←(AC) + ~ (EA) + (CY/L); (CY/L), (OV)	20	5-8
		disp, disp(ptr), @disp(ptr)				
Memory Increment/Decrement Instructions			7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0			
Increment and Load	A8	ILD	1 0 1 0 1 0 ptr | disp	(AC), (EA)←(EA) + 1	22	5-8
Decrement and Load	B8	DLD	1 0 1 1 1 0	(AC), (EA)←(EA) - 1	22	5-9
		disp, disp(ptr)				
Immediate Instructions			7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0			
Load Immediate	C4	LDI	1 1 0 0 0 1 0 0 | data	(AC)←data	10	5-9
AND Immediate	D4	ANI	1 1 0 1 0 1 0 0	(AC)←(AC) ∧ data	10	5-10
OR Immediate	DC	ORI	1 1 0 1 1 1 0 0	(AC)←(AC) ∨ data	10	5-10
Exclusive OR Immediate	E4	XRI	1 1 1 0 0 1 0 0	(AC)←(AC) ▽ data	10	5-10
Decimal Add Immediate	EC	DAI	1 1 1 0 1 1 0 0	(AC)←$(AC)_{10}$ + $data_{10}$ + (CY/L); (CY/L)	15	5-11
Add Immediate	F4	ADI	1 1 1 1 0 1 0 0	(AC)←(AC) + data + (CY/L); (CY/L), (OV)	11	5-11
Complement and Add Immediate	FC	CAI	1 1 1 1 1 1 0 0	(AC)←(AC) + ~ data + (CY/L); (CY/L), (OV)	12	5-11
		data				
Transfer Instructions			7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0			
Jump	90	JMP	1 0 0 1 0 0 ptr | disp	(PC)←EA	11	5-12
Jump If Positive	94	JP	1 0 0 1 0 1	If (AC) ≥ 0, (PC)← -EA	9, 11	5-13
Jump If Zero	98	JZ	1 0 0 1 1 0	If (AC) = 0, (PC)←EA	9, 11	5-13
Jump If Not Zero	9C	JNZ	1 0 0 1 1 1	If (AC) ≠ 0, (PC)← -EA	9, 11	5-13
		disp, disp(ptr)				
Double-byte Miscellaneous Instructions			7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0			
Delay	8F	DLY disp	1 0 0 0 1 1 1 1 | disp	count AC to -1, delay = 13 + 2 (AC) + 2 disp + 2^9 disp microcycles	13 to 131593	5-22

(Continued on next page)

Table 5-2. SC/MP Instruction Summary --- Single-Byte

Description	Op Code	Source Statement	Object Format	Operation	Micro-cycles	Page
Extension Register Instructions			7 6 5 4 3 2 1 0			
Load AC from Extension	40	LDE	0 1 0 0 0 0 0 0	(AC)←(E)	6	5-14
Exchange AC and Extension	01	XAE	0 0 0 0 0 0 0 1	(AC)←→(E)	7	5-14
AND Extension	50	ANE	0 1 0 1 0 0 0 0	(AC)←(AC) ∧ (E)	6	5-14
OR Extension	58	ORE	0 1 0 1 1 0 0 0	(AC)←(AC) ∨ (E)	6	5-14
Exclusive-OR Extension	60	XRE	0 1 1 0 0 0 0 0	(AC)←(AC) ▽ (E)	6	5-15
Decimal Add Extension	68	DAE	0 1 1 0 1 0 0 0	(AC)←$(AC)_{10}$ + $(E)_{10}$ + (CY/L); (CY/L)	11	5-15
Add Extension	70	ADE	0 1 1 1 0 0 0 0	(AC)←(AC) + (E) + (CY/L); (CY/L), (OV)	7	5-15
Complement and Add Extension	78	CAE	0 1 1 1 1 0 0 0	(AC)←(AC) + ~ (E) + (CY/L); (CY/L), (OV)	8	5-16
Pointer Register Move Instructions			7 6 5 4 3 2 1 0			
Exchange Pointer Low	30	XPAL ptr	0 0 1 1 0 ptr	(AC)←→$(PTR_{7:0})$	8	5-16
Exchange Pointer High	34	XPAH ptr	0 0 1 1 0 1	(AC)←→$(PTR_{15:8})$	8	5-17
Exchange Pointer with PC	3C	XPPC	0 0 1 1 1 1	(PC)←→(PTR)	7	5-17
Shift, Rotate, Serial I/O Instructions			7 6 5 4 3 2 1 0			
Serial Input/Output	19	SIO	0 0 0 1 1 0 0 1	(E_i)→(E_{i-1}), SIN→(E_7), (E_0)→SOUT	5	5-17
Shift Right	1C	SR	0 0 0 1 1 1 0 0	(AC_i)→(AC_{i-1}), 0→(AC_7)	5	5-18
Shift Right with Link	1D	SRL	0 0 0 1 1 1 0 1	(AC_i)→(AC_{i-1}), (CY/L)→(AC_7)	5	5-18
Rotate Right	1E	RR	0 0 0 1 1 1 1 0	(AC_i)→(AC_{i-1}), (AC_0)→(AC_7)	5	5-18
Rotate Right with Link	1F	RRL	0 0 0 1 1 1 1 1	(AC_i)→(AC_{i-1}), (AC_0)→(CY/L)→(AC_7)	5	5-19
Single-byte Miscellaneous Instructions			7 6 5 4 3 2 1 0			
Halt	00	HALT	0 0 0 0 0 0 0 0	Pulse H-flag	8	5-19
Clear Carry/Link	02	CCL	0 0 0 0 0 0 1 0	(CY/L)←0	5	5-20
Set Carry/Link	03	SCL	0 0 0 0 0 0 1 1	(CY/L)←1	5	5-20
Disable Interrupt	04	DINT	0 0 0 0 0 1 0 0	(IE)←0	6	5-21
Enable Interrupt	05	IEN	0 0 0 0 0 1 0 1	(IE)←1	6	5-20
Copy Status to AC	06	CSA	0 0 0 0 0 1 1 0	(AC)←(SR)	5	5-21
Copy AC to Status	07	CAS	0 0 0 0 0 1 1 1	(SR)←(AC)	6	5-21
No Operation	08	NOP	0 0 0 0 1 0 0 0	(PC)←(PC) + 1	5	5-22

5.2.1 Memory Reference Instructions

This group of eight instructions provides logic, arithmetic, and data transfer operations between the accumulator and the effective address. The Memory-Reference Instructions and mnemonics are as follows:

Load . . . . . LD
Store . . . . . ST
AND . . . . . AND
OR . . . . . OR
Exclusive-OR . . . . . XOR
Decimal Add . . . . . DAD
Add. . . . . ADD
Complement and Add . . . . . CAD

The Effective Address (EA) may be PC-relative, indexed, or auto-indexed as shown in table 5-3.

Table 5-3. Memory Reference Formats

Addressing	Operand Formats			
	Object			Source
	m	ptr	disp*	
PC-relative	0	0	-128 to +127	disp
Indexed	0	1, 2, or 3	-128 to +127	disp(ptr)
Auto-indexing	1	1, 2, or 3	-128 to +127	@disp(ptr)

* Note: If disp = -128, then (E) is substituted for disp in calculating EA as well as in performing auto-indexing.

PC-relative addressing is assumed when only a displacement (disp) value is specified (example: LD VALUE). Indexed addressing requires a displacement and one of the four pointer registers (example: ADD 10(P1)). Auto-indexing requires the "at sign," a displacement, and a pointer register (other than PC) (example: ST @-1(3)).

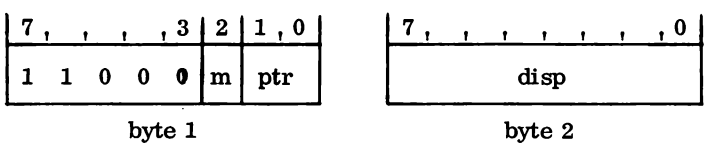
The formats of the source statement and the object code and the description of the operation of each Memory Reference Instruction follow.

LOAD (LD)

SOURCE STATEMENT

Operation	Operands
LD	disp disp(ptr) @disp(ptr)

OBJECT FORMAT



Operation: (AC) ← (EA)

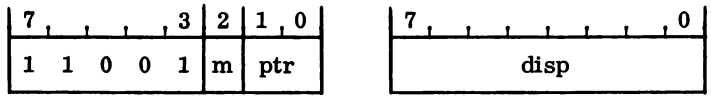
The contents of the Accumulator (AC) are replaced by the contents of the Effective Address (EA). The initial contents of AC are lost; the contents of EA are unaltered.

STORE (ST)

SOURCE STATEMENT

Operation	Operands
ST	disp disp(ptr) @disp(ptr)

OBJECT FORMAT



Operation: (EA) ← (AC)

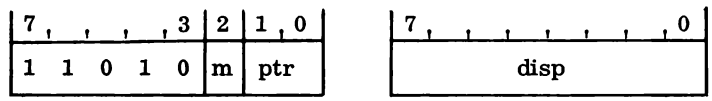
The contents of the Effective Address (EA) are replaced by the contents of the Accumulator (AC). The initial contents of EA are lost; the contents of AC are unaltered.

AND (AND)

SOURCE STATEMENT

Operation	Operands
AND	disp disp(ptr) @disp(ptr)

OBJECT FORMAT



Operation: (AC) ← (AC) $\wedge$ (EA)

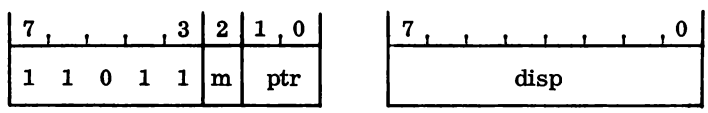
The contents of the Accumulator (AC) are ANDed with the contents of the Effective Address (EA), and the result is stored in AC. The initial contents of AC are lost; the contents of EA are unaltered.

OR (OR)

SOURCE STATEMENT

Operation	Operands
OR	disp disp(ptr) @disp(ptr)

OBJECT FORMAT



Operation: (AC) ← (AC) $\vee$ (EA)

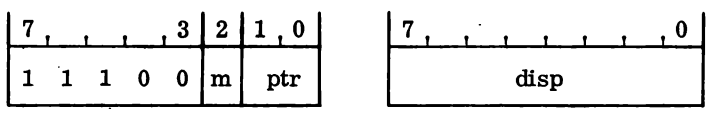
The contents of the Accumulator (AC) are inclusive-ORed with the contents of the Effective Address (EA), and the result is stored in AC. The initial contents of AC are lost; the contents of EA are unaltered.

EXCLUSIVE-OR (XOR)

SOURCE STATEMENT

Operation	Operands
XOR	disp disp(ptr) @disp(ptr)

OBJECT FORMAT



Operation: (AC) ← (AC) $\overline{\vee}$ (EA)

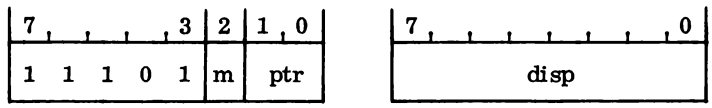
The contents of the Accumulator (AC) are exclusive-ORed with the contents of the Effective Address (EA), and the result is stored in AC. The initial contents of AC are lost; the contents of EA are unaltered.

DECIMAL ADD (DAD)

SOURCE STATEMENT

Operation	Operands
DAD	disp disp(ptr) @disp(ptr)

OBJECT FORMAT

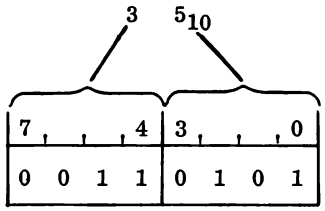


byte 1 byte 2

Operation: $(AC) \leftarrow (AC)_{10} + (EA)_{10} + (CY/L)$; (CY/L)

The contents of the Accumulator (AC) and the contents of the Effective Address (EA) are treated as 2-digit binary-coded-decimal numbers greater than or equal to zero, and less than or equal to ninety-nine ($0 \leqslant n \leqslant 99$). The contents of AC and EA and the Carry (CY/L) are added, and the 2-digit binary-coded-decimal sum is stored in AC. The initial contents of AC are lost; the contents of EA are unaltered. The Carry Flag in the Status Register is set if a carry occurs from the most significant decimal digit; otherwise, it is cleared. The Overflow Flag is not affected.

Example of a binary-coded-decimal number:



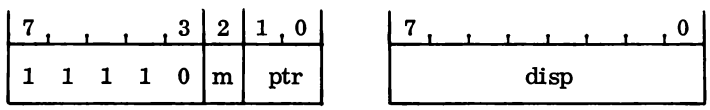
(AC) or (EA)

ADD (ADD)

SOURCE STATEMENT

Operation	Operands
ADD	disp disp(ptr) @disp(ptr)

OBJECT FORMAT



byte 1 byte 2

Operation: $(AC) \leftarrow (AC) + (EA) + (CY/L)$; (CY/L), (OV)

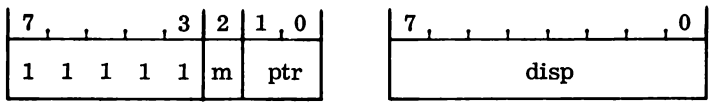
The contents of the Accumulator (AC) and the contents of the Effective Address (EA) are treated as 8-bit binary twos-complement numbers. The contents of the Accumulator (AC), the Effective Address (EA), and the Carry (CY/L) are added algebraically, and the sum is stored in AC. The Carry Flag in the Status Register is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow (OV) Flag in the Status Register is set if an overflow occurs (that is, if the sign of the results differs from the sign of both operands); otherwise, the overflow flag is cleared.

COMPLEMENT AND ADD (CAD)

SOURCE STATEMENT

Operation	Operands
CAD	disp disp(ptr) @disp(ptr)

OBJECT FORMAT



byte 1 byte 2

Operation: (AC) ← (AC) + ~ (EA) + (CY/L); (CY/L), (OV)

The contents of the Accumulator (AC) and the contents of the Effective Address (EA) are treated as 8-bit binary numbers. The contents of the Accumulator (AC), the ones complement of the contents of the Effective Address (EA), and the Carry (CY/L) are added algebraically, and the sum is stored in AC. The initial contents of AC are lost; the contents of EA are unaltered. The Carry Flag (CY/L) in the Status Register is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag (OV) in the Status Register is set if the sign of the result is the same as the sign of (EA) and opposite the sign of (AC); otherwise, it is cleared.

NOTE

If the CY/L Flag is cleared initially, the logical (ones) complement of (EA) is added to the Accumulator. If the CY/L Flag is set, the twos complement of (EA) is added.

5.2.2 Memory Increment/Decrement Instructions

The two double-byte instructions in this group may be used to maintain a software counter in memory. The Memory Increment/Decrement Instructions are as follows:

Increment and Load . . . . . ILD
Decrement and Load . . . . . DLD

The formats of the source statement and object code and the description of each Memory Increment/Decrement Instruction follow.

NOTE

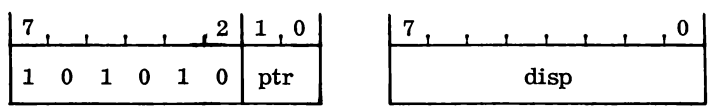
At the hardware level, these instructions access the memory in a read-alter-write mode. The processor retains control of the input/output bus between the data read and write operations.

INCREMENT AND LOAD (ILD)

SOURCE STATEMENT

Operation	Operands
ILD	disp disp(ptr)

OBJECT FORMAT



byte 1 byte 2

Operation: (AC), (EA) ← (EA) + 1

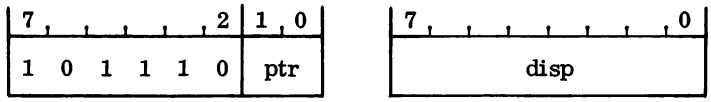
The contents of the Effective Address (EA) are incremented by 1, and the result is stored in the Accumulator (AC) and, also, in EA. The initial contents of AC and EA are lost. The Carry and Overflow Flags are not affected.

DECREMENT AND LOAD (DLD)

SOURCE STATEMENT

Operation	Operands
DLD	disp disp(ptr)

OBJECT FORMAT



byte 1 byte 1

Operation: (AC), (EA) ← (EA) - 1

The contents of the Effective Address (EA) are decremented by 1, and the result is stored in the Accumulator (AC) and, also, in EA. The initial contents of AC and EA are lost. The Carry and Overflow Flags are not affected.

5.2.3 Immediate Instructions

The immediate instructions perform most of the same operations as the memory-reference instructions. The data used in the operations comes from the byte immediately after the opcode byte; that is, the data byte is the displacement. The Immediate Instructions are as follows:

Load Immediate . . . . . .LDI
AND Immediate . . . . . .ANI
OR Immediate . . . . . .ORI
Exclusive-OR Immediate . . . . . .XRI
Decimal Add Immediate . . . . . .DAI
Add Immediate . . . . . .ADI
Complement and Add Immediate . . . .CAI

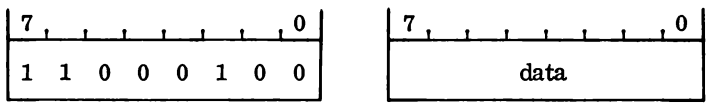
The formats of the source statement and the object code and the description of each Immediate Instruction follow.

LOAD IMMEDIATE (LDI)

SOURCE STATEMENT

Operation	Operand
LDI	data

OBJECT FORMAT



byte 1 byte 2

Operation: (AC) ← data

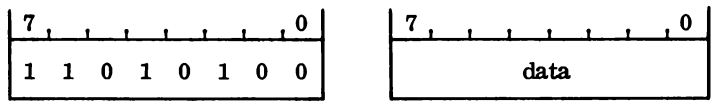
The contents of the Accumulator (AC) are replaced by the data byte. The initial contents of AC are lost; the data byte is unaltered.

AND IMMEDIATE (ANI)

SOURCE STATEMENT

Operation	Operand
ANI	data

OBJECT FORMAT



Operation: (AC) ← (AC) ∧ data

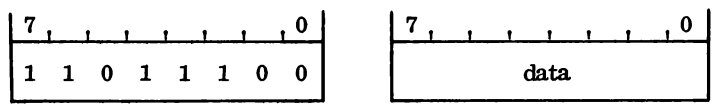
The contents of the Accumulator (AC) are ANDed with the data byte, and the result is stored in AC. The initial contents of AC are lost; the data byte is unaltered.

OR IMMEDIATE (ORI)

SOURCE STATEMENT

Operation	Operand
ORI	data

OBJECT FORMAT



Operation: (AC) ← (AC) ∨ data

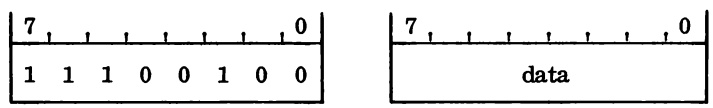
The contents of the Accumulator (AC) are inclusive-ORed with the data byte, and the result is stored in AC. The initial contents of AC are lost; the data byte is unaltered.

EXCLUSIVE OR IMMEDIATE (XRI)

SOURCE STATEMENT

Operation	Operand
XRI	data

OBJECT FORMAT



Operation: (AC) ← (AC) ⊽ data

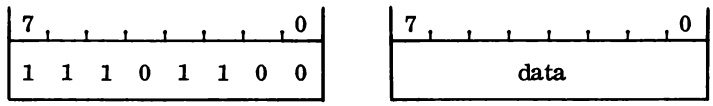
The contents of the Accumulator (AC) are exclusive-ORed with the data byte, and the result is stored in AC. The initial contents of AC are lost; the data byte is unaltered.

DECIMAL ADD IMMEDIATE (DAI)

SOURCE STATEMENT

Operation	Operand
DAI	data

OBJECT FORMAT



Operation: $(AC) \leftarrow (AC)_{10} + data_{10} + (CY/L)$; (CY/L)

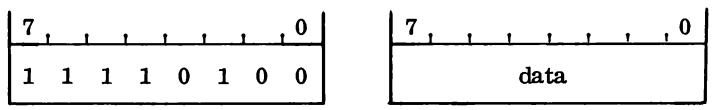
The data byte and the contents of the Accumulator are treated as 2-digit binary-coded-decimal numbers. The contents of the Accumulator (AC), the data byte, and the Carry (CY/L) are added, and the 2-digit binary-coded-decimal sum is stored in AC. The initial contents of AC are lost; the data byte is unaltered. The Carry Flag in the Status Register is set if a carry from the most significant decimal digit occurs; otherwise, it is cleared. The Overflow Flag is not affected.

ADD IMMEDIATE (ADI)

SOURCE STATEMENT

Operation	Operand
ADI	data

OBJECT FORMAT



Operation: $(AC) \leftarrow (AC) + data + (CY/L)$; (CY/L), (OV)

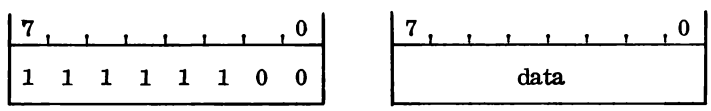
Data and the contents of the Accumulator (AC) are treated as 8-bit twos-complement numbers. The contents of the Accumulator (AC), the data byte, and the Carry (CY/L) are added algebraically, and the sum is stored in AC. The initial contents of AC are lost; the data byte is unaltered. The Carry Flag in the Status Register is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag (OV) in the Status Register is set if the sign of the result differs from the sign of both operands; otherwise, it is cleared.

COMPLEMENT AND ADD IMMEDIATE (CAI)

SOURCE STATEMENT

Operation	Operand
CAI	data

OBJECT FORMAT



Operation: $(AC) \leftarrow (AC) + \sim data + (CY/L)$; (CY/L), (OV)

The data byte and the contents of the Accumulator (AC) are treated as 8-bit numbers. The contents of the Accumulator (AC), the ones complement of the data byte, and the Carry (CY/L) are added algebraically and the result is stored in AC. The initial contents of AC are lost; the data byte is unaltered. The Carry Flag in the Status Register is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag (OV) in the Status Register is set if the sign of the result is the same as the sign of the data byte and opposite the sign of (AC); otherwise, it is cleared.

NOTE

If the CY/L Flag is set initially, this operation is equivalent to subtracting the data byte from the Accumulator.

5.2.4 Transfer Instructions

The four double-byte instructions in this group are used for conditional and unconditional jumps within a routine, and jumps to subroutines. The Transfer Instructions are as follows:

Jump . . . . . JMP
Jump if Positive . . . . . JP
Jump if Zero . . . . . JZ
Jump if Not Zero . . . . . JNZ

The effective address of a jump that is PC-relative is the PC plus the displacement (disp). The range of a PC-relative jump is -126 to +129 bytes from the jump instruction. As with the memory-reference instructions, the effective address does not affect the 4 most significant address bits; thus, wrap-around can occur at 4K page boundaries. When 'ptr' specifies a pointer register other than 0, the 4 most significant bits of the PC are replaced by the 4 most significant bits of the specified pointer.

When assembling Transfer Instructions, the assembler reduces the specified displacement by one (1), so if the jump is taken the next instruction executed is located at the label specified in the jump instruction; that is, the effective address specified by the user.

Example:

```
Location  Object
Counter   Code
 1000      9012      START:   JMP     LABEL    ;JUMP TO LABEL
                              .
                              .
                              .
 1014                LABEL:

PC (during execution)  1001
disp                    12
EA                    1013
```

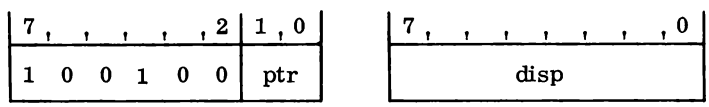
The formats of the source statement and the object code and the description of the operation of each Transfer Instruction follow.

JUMP (JMP)

SOURCE STATEMENT

Operation	Operands
JMP	disp disp(ptr)

OBJECT FORMAT



byte 1 byte 2

Operation: (PC) ← EA

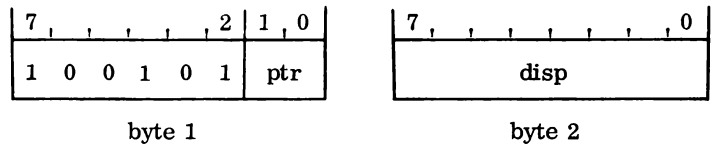
The Effective Address (EA) replaces the contents of the Program Counter (PC). The next instruction is fetched from the location designated by the new contents of PC + 1.

JUMP IF POSITIVE (JP)

SOURCE STATEMENT

Operation	Operands
JP	disp disp(ptr)

OBJECT FORMAT



Operation: If (AC) ≥ 0, (PC) $\leftarrow$ EA

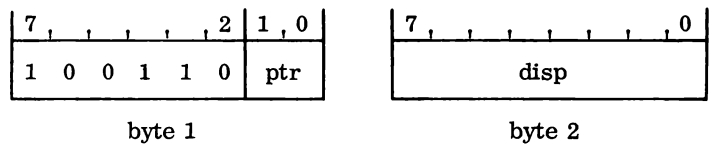
If the contents of the Accumulator (AC) are positive or zero, the Effective Address (EA) replaces the contents of the Program Counter (PC). The next instruction is fetched from the location designated by the new contents of PC + 1.

JUMP IF ZERO (JZ)

SOURCE STATEMENT

Operation	Operands
JZ	disp disp(ptr)

OBJECT FORMAT



Operation: If (AC) = 0, (PC) $\leftarrow$ EA

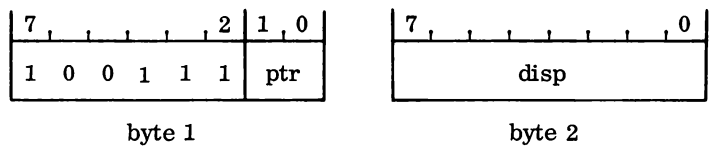
If the contents of the Accumulator (AC) are zero, the Effective Address (EA) replaces the contents of the Program Counter (PC). The next instruction is fetched from the location designated by the new contents of the PC + 1.

JUMP IF NOT ZERO (JNZ)

SOURCE STATEMENT

Operation	Operands
JNZ	disp disp(ptr)

OBJECT FORMAT



Operation: If (AC) $\neq 0$, (PC) $\leftarrow$ EA

If the contents of the Accumulator (AC) are not zero, the Effective Address (EA) replaces the contents of the Program Counter (PC). The next instruction is fetched from the location designated by the new contents of the PC + 1.

5.2.5 Extension Register Instructions

This group of eight single-byte instructions is used for arithmetic and logic operations between the Extension Register (E) and the Accumulator (AC). The Extension Register Instructions are as follows:

Load AC from Extension . . . . . LDE
Exchange AC and Extension . . . . . XAE
AND Extension. . . . . ANE
OR Extension . . . . . ORE
Exclusive-OR Extension . . . . . XRE
Decimal Add Extension . . . . . DAE
Add Extension . . . . . ADE
Complement and Add Extension . . . . CAE

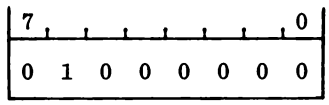
The formats of the source statement and the object code and the description of the operation of each Extension Register Instruction follow.

LOAD FROM EXTENSION (LDE)

SOURCE STATEMENT

Operation
LDE

OBJECT FORMAT



Operation: (AC) ← (E)

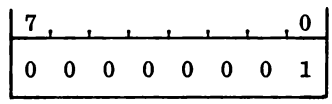
The contents of the Accumulator (AC) are replaced by the contents of the Extension Register (E). The initial contents of AC are lost; the contents of E are unaltered.

EXCHANGE AC AND EXTENSION (XAE)

SOURCE STATEMENT

Operation
XAE

OBJECT FORMAT



Operation: (AC) ←→ (E)

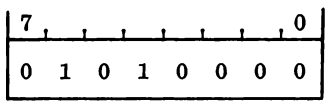
The contents of the Accumulator (AC) are exchanged with the contents of the Extension Register (E).

AND EXTENSION (ANE)

SOURCE STATEMENT

Operation
ANE

OBJECT FORMAT



Operation: (AC) ← (AC) $\wedge$ (E)

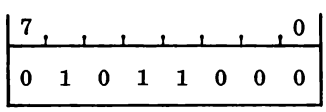
The contents of the Accumulator (AC) are ANDed with the contents of the Extension Register (E), and the result is stored in AC. The initial contents of AC are lost; the contents of E are unaltered.

OR EXTENSION (ORE)

SOURCE STATEMENT

Operation
ORE

OBJECT FORMAT



Operation: (AC) ← (AC) $\vee$ (E)

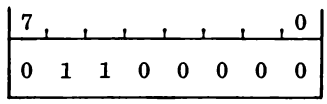
The contents of the Accumulator (AC) are inclusive-ORed with the contents of the Extension Register (E), and the result is stored in AC. The initial contents of AC are lost; the contents of E are unaltered.

EXCLUSIVE-OR EXTENSION (XRE)

SOURCE STATEMENT

Operation
XRE

OBJECT FORMAT



Operation: $(AC) \leftarrow (AC) \bigtriangledown (E)$

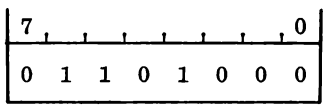
The contents of the Accumulator (AC) are exclusive-ORed with the contents of the Extension Register (E), and the result is stored in AC. The initial contents of AC are lost; the contents of E are unaltered.

DECIMAL ADD EXTENSION (DAE)

SOURCE STATEMENT

Operation
DAE

OBJECT FORMAT



Operation: $(AC) \leftarrow (AC)_{10} + (E)_{10} + (CY/L); (CY/L)$

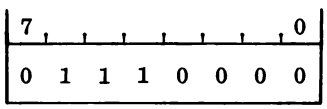
The contents of the Accumulator (AC) and the Extension Register (E) are treated as 2-digit binary-coded-decimal numbers, greater than or equal to zero. The contents of the Accumulator (AC), Extension Register (E), and the Carry (CY/L) are added, and the sum is stored in AC. The initial contents of AC are lost; the contents of E are unaltered. The Carry Flag in the Status Register is set if a carry from the most significant decimal digit occurs; otherwise, it is cleared. The Overflow Flag is not affected.

ADD EXTENSION (ADE)

SOURCE STATEMENT

Operation
ADE

OBJECT FORMAT



Operation: $(AC) \leftarrow (AC) + (E) + (CY/L); (CY/L), (OV)$

The contents of the Accumulator (AC) and the Extension Register (E) are treated as 8-bit binary, twos-complement numbers. The contents of the Accumulator (AC), Extension Register (E), and the Carry (CY/L) are added algebraically, and the sum is stored in AC. The initial contents of AC are lost; the contents of E are unaltered. The Carry Flag (CY/L) in the Status Register is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag (OV) in the Status Register is set if the sign of the result differs from the sign of both operands; otherwise, it is cleared.

COMPLEMENT AND ADD EXTENSION (CAE)

SOURCE STATEMENT

Operation
CAE

OBJECT FORMAT

7							0
0	1	1	1	1	0	0	0

Operation: (AC) ← (AC) + ~ (E) + (CY/L); (CY/L), (OV)

The contents of the Accumulator (AC) and Extension Register (E) are treated as 8-bit binary numbers. The contents of the Accumulator (AC), the ones complement of the contents of the Extension Register (E), and the Carry (CY/L) are added algebraically, and the result is stored in AC. The initial contents of AC are lost; the contents of E are unaltered. The Carry Flag (CY/L) in the Status Register is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag (OV) in the Status Register is set if the sign of the result is the same as the sign of (E) and opposite the sign of (AC); otherwise, it is cleared.

NOTE

If the CY/L Flag is set initially, this operation is equivalent to subtracting the contents of E from the contents of AC.

5.2.6 Pointer Register Move Instructions

The three single-byte instructions in this group are used for transfers between the Pointer Registers and the Accumulator or the Program Counter. The Pointer Register Move Instructions are as follows:

Exchange Pointer Low . . . . . XPAL
Exchange Pointer High . . . . . XPAH
Exchange Pointer with Program Counter . . . . . XPPC

The formats of the source statement and the object code and the description of each Pointer Register Move Instruction follow.

EXCHANGE POINTER LOW (XPAL)

SOURCE STATEMENT

Operation	Operand
XPAL	ptr

OBJECT FORMAT

7						2	1	0
0	0	1	1	0	0		ptr	

Operation: (AC) ←→ ($PTR_{7:0}$)

The contents of the Accumulator (AC) are exchanged with the low-order byte (bits 7 through 0) of the designated Pointer Register (PTR).

EXCHANGE POINTER HIGH (XPAH)

SOURCE STATEMENT

Operation	Operand
XPAH	ptr

OBJECT FORMAT

7	6	5	4	3	2	1	0
0	0	1	1	0	1	ptr	

Operation: $(AC) \longleftrightarrow (PTR_{15:8})$

The contents of the Accumulator (AC) are exchanged with the high-order byte (bits 15 through 8) of the designated Pointer Register (PTR).

EXCHANGE POINTER WITH PC (XPPC)

SOURCE STATEMENT

Operation	Operand
XPPC	ptr

OBJECT FORMAT

7	6	5	4	3	2	1	0
0	0	1	1	1	1	ptr	

Operation: $(PC) \longleftrightarrow (PTR)$

The contents of the Program Counter (PC) are exchanged with the designated Pointer Register (PTR).

5.2.7 Shift, Rotate, Serial Input/Output Instructions

The five single-byte instructions in this group shift or rotate the Accumulator or perform serial input/output operations using the Extension Register. The Shift, Rotate, and Serial Input/Output Instructions are as follows:

Serial Input/Output	SIO
Shift Right	SR
Shift Right with Link	SRL
Rotate Right	RR
Rotate Right with Link	RRL

The formats of the source statement and the object code and the description of the operation of each Shift, Rotate, or Serial Input/Output Instruction follow.

SERIAL INPUT/OUTPUT (SIO)

SOURCE STATEMENT

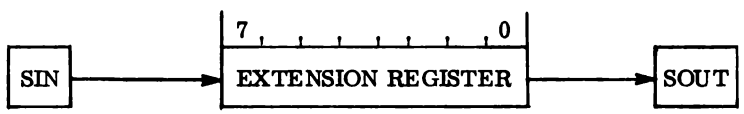
Operation
SIO

OBJECT FORMAT

7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1

Operation: $(E_i) \rightarrow (E_{i-1})$, $SIN \rightarrow (E_7)$, $(E_0) \rightarrow SOUT$

The contents of the Extension Register (E) are shifted right one bit. The initial content of bit 0 is shifted to the data output pin SOUT. The signal on the data input pin SIN is shifted into bit 7.

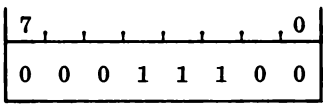


SHIFT RIGHT (SR)

SOURCE STATEMENT

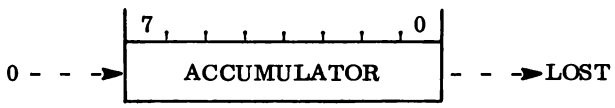
Operation
SR

OBJECT FORMAT



Operation: $(AC_i) \rightarrow (AC_{i-1})$, $0 \rightarrow (AC_7)$

The contents of the Accumulator (AC) are shifted right one bit. The initial content of bit 0 is lost. Zero is shifted into bit 7.

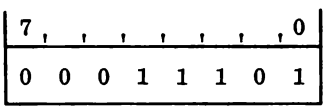


SHIFT RIGHT WITH LINK (SRL)

SOURCE STATEMENT

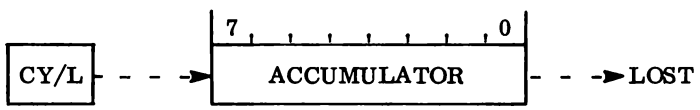
Operation
SRL

OBJECT FORMAT



Operation: $(AC_i) \rightarrow (AC_{i-1})$, $(CY/L) \rightarrow (AC_7)$

The contents of the Accumulator are shifted right one bit. The initial content of bit 0 is lost. The Link (CY/L) Flag from the Status Register is shifted into bit 7. The Link Flag is not altered.

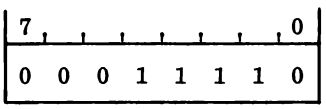


ROTATE RIGHT (RR)

SOURCE STATEMENT

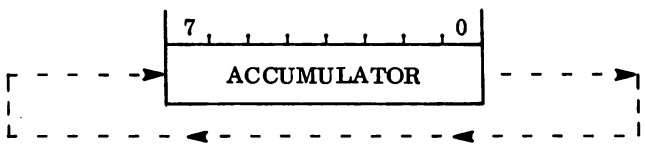
Operation
RR

OBJECT FORMAT



Operation: $(AC_i) \rightarrow (AC_{i-1})$, $(AC_0) \rightarrow (AC_7)$

The contents of the Accumulator (AC) are rotated right one bit. The initial content of bit 0 is shifted into bit 7.



ROTATE RIGHT WITH LINK (RRL)

SOURCE STATEMENT

Operation

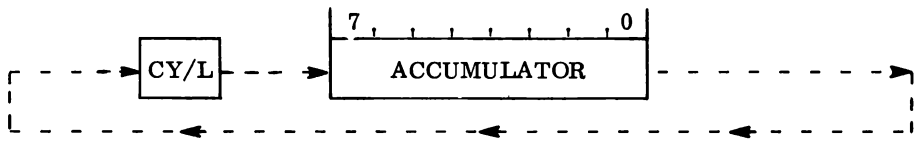
RRL

OBJECT FORMAT

7							0
0	0	0	1	1	1	1	1

Operation: $(AC_i) \rightarrow (AC_{i-1})$, $(AC_0) \rightarrow (CY/L) \rightarrow (AC_7)$

The contents of the Accumulator (AC) are rotated right one bit. The initial content of bit 0 is shifted into the Link Flag (CY/L) of the Status Register, and the initial content of the Link Flag is shifted into bit 7 of AC.



5.2.8 Miscellaneous Instructions

There are nine instructions in this group, eight single-byte, and one double-byte. The Miscellaneous Instructions are as follows:

Halt	HALT
Clear Carry/Link Bit	CCL
Set Carry/Link Bit	SCL
Disable Interrupt	DINT
Enable Interrupt	IEN
Copy Status to Accumulator	CSA
Copy Accumulator to Status	CAS
No Operation	NOP
Delay	DLY

HALT (HALT)

SOURCE STATEMENT

Operation

HALT

OBJECT FORMAT

7							0
0	0	0	0	0	0	0	0

Operation: Pulse H-flag at I/O status time.

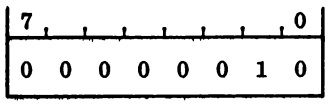
In a particular application system, this instruction may be used for functions other than HALT. For detailed information on the hardware operation of the halt instruction, see the SC/MP Users Manual.

CLEAR CARRY/LINK (CCL)

SOURCE STATEMENT

Operation
CCL

OBJECT FORMAT



Operation: (CY/L) ← 0

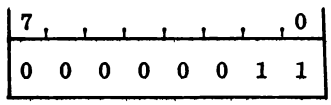
The Carry/Link (CY/L) Flag in the Status Register (SR) is cleared. The remaining bits in SR are not affected.

SET CARRY/LINK (SCL)

SOURCE STATEMENT

Operation
SCL

OBJECT FORMAT



Operation: (CY/L) ← 1

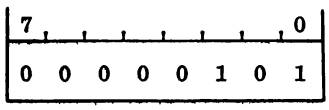
The Carry/Link Flag in the Status Register (SR) is set. The remaining bits in SR are not affected.

ENABLE INTERRUPT (IEN)

SOURCE STATEMENT

Operation
IEN

OBJECT FORMAT



Operation: (IE) ← 1

The Interrupt Enable (IE) Flag in the Status Register (SR) is set; the remaining bits in SR are not affected. The processor interrupt system is enabled. Interrupts will be processed as received after the next instruction is fetched and executed.

DISABLE INTERRUPT (DINT)

SOURCE STATEMENT

Operation
DINT

OBJECT FORMAT

7							0
0	0	0	0	0	1	0	0

Operation: (IE) ← 0

The Interrupt Enable (IE) Flag in the Status Register (SR) is cleared; the other bits in SR are not affected. The processor interrupt system is disabled. Interrupts which occur while the system is disabled will not be processed.

COPY STATUS TO AC (CSA)

SOURCE STATEMENT

Operation
CSA

OBJECT FORMAT

7							0
0	0	0	0	0	1	1	0

Operation: (AC) ← (SR)

The contents of the Accumulator (AC) are replaced by the contents of the Status Register (SR). The initial contents of AC are lost; the contents of SR are not altered.

COPY AC TO STATUS (CAS)

SOURCE STATEMENT

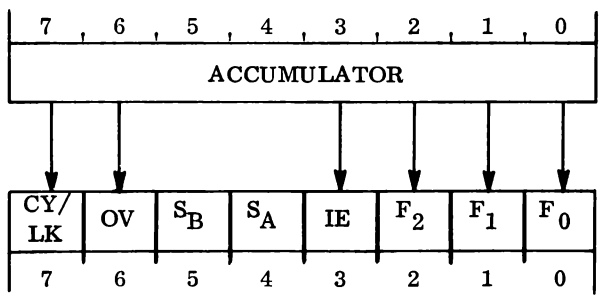
Operation
CAS

OBJECT FORMAT

7							0
0	0	0	0	0	1	1	1

Operation: (SR) ← (AC)

The contents of the Accumulator (AC) replace the contents of the Status Register (SR). SR bits 4 and 5 are external sense bits and are not affected by this instruction. The initial contents of SR (except for bits 4 and 5) are lost; the contents of AC are not altered.



Status Register

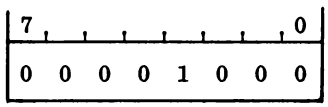
If IE is changed from 0 to 1 by this instruction, the interrupt system will be enabled after the next instruction is fetched and executed.

NO OPERATION (NOP)

SOURCE STATEMENT

Operation
NOP

OBJECT FORMAT



Operation: (PC) ← (PC) + 1

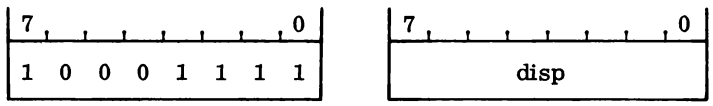
The Program Counter (PC) is incremented by 1. The NOP instruction takes the minimum 5-microcycle execution time. Undefined opcodes encountered are considered to be one-byte or two-byte NOPs and may take 5 to 10 microcycles to execute depending on the code.

DELAY (DLY)

SOURCE STATEMENT

Operation	Operand
DLY	disp

OBJECT FORMAT



byte 1 byte 2

Operation: DELAY = 13 + 2(AC) + 2 disp + 2^9 disp

This instruction delays processing a variable length of time. The contents of the Accumulator (AC) and the Displacement (disp) are considered unsigned binary numbers (maximum value of each is 255). The number computed from the given equation is the execution time in microcycles. The following table gives some typical execution times. Range of delay is from 13 to 131593 microcycles.

		AC									
		0	25	50	75	100	125	150	175	200	225
	0	13	63	113	163	213	263	313	363	413	463
	1	527	577	627	677	727	777	827	877	927	977
	2	1041	1091	1141	1191	1241	1291	1341	1391	1441	1491
	3	1555	1605	1655	1705	1755	1805	1855	1905	1955	2005
	4	2069	2119	2169	2219	2269	2319	2369	2419	2469	2519
disp	5	2583	2633	2683	2733	2783	2833	2883	2933	2983	3033
	6	3097	3147	3197	3247	3297	3347	3397	3447	3497	3547
	7	3611	3661	3711	3761	3811	3861	3911	3961	4011	4061
	8	4125	4175	4225	4275	4325	4375	4425	4475	4525	4575
	9	4639	4689	4739	4789	4839	4889	4939	4989	5039	5089
	10	5153	5203	5253	5303	5353	5403	5453	5503	5553	5603

To determine AC and disp for a specific number of microcycles (m) use the following equations:

disp = truncate ((m-13)/514)
AC = ((m-13) - 514(disp))/2

Using these equations, the delay time will be either exact or one microcycle less than the required number of microcycles.

5.3 PSEUDO INSTRUCTIONS

A Pseudo Instruction is a statement that appears to the assembler the same as any other assembly-language statement but results in the generation of several machine-language instructions.

Only one Pseudo Instruction statement is implemented in SC/MP assembly language; it is the Jump to Subroutine statement described below.

JUMP TO SUBROUTINE (JS)

	Operation	Operands
Format:	JS	ptr, expression
Memory:	7 bytes	
Generated Code:	LDI	H(expression-1)*
	XPAH	ptr
	LDI	L(expression-1)*
	XPAL	ptr
	XPPC	ptr

* This is not legal notation for the assembler

When a Jump to Subroutine is invoked, the code generated results in the setting of the specified Pointer Register (PTR) to the value, expression-1. In calculating this value, the memory page structure of SC/MP is taken into account (see 3.4). The contents of the PTR is then exchanged with the contents of the PC. As a result, the next instruction to be executed will be that at the location addressed by expression. The Pointer Register (PTR) will contain the address of the XPPC instruction, allowing a subroutine return to the instruction following.

The 4K page boundary should be kept in mind when assigning expression (3.4).

5.4 ASSIGNMENT STATEMENT

[label] symbol = expression [;comments]

The 'Assignment Statement' assigns the value of the expression on the right of the equals sign to the symbol on the left of the equals sign. The Statement may be preceded by a series of labels.

Example:

```
RETURN = 0D          ;SYMBOL HAS CARRIAGE RET CODE
```

The Assignment Statement may set the location counter or refer to the current value of the location counter in an expression. The period '.' is a special symbol used to specify the location counter. The location-counter symbol may appear on either side of the equals sign. If it appears on the left, it is assigned the value on the right side of the equals sign. The programmer may refer to the current setting of the location counter by referencing the '.' in the expression to the right of the equals sign. Assignment statements using the location-counter symbol are coded as free-form statements. For example:

```
         .=20                ;SET LOCATION COUNTER TO 20
                             ;LOCATION CTR IN ABSOLUTE MODE
TABLE:   .=.+10              ;RESERVE 10 LOCATIONS FOR TABLE
```

If the '.' appears on the left, the expression on the right must be defined during the first pass so subsequent label assignments may be made.

If the symbol on the left is not '.', then the expression on the right need not have a value during the first pass, but must have a value during the second pass. This permits only one level of forward referencing. An example of more than one level of forward referencing follows:

```
FST:    A=B+2     This expression undefined during pass 2.
SND:    B=C-1     This expression undefined during pass 1.
THD:    C=25      This expression absolute.
```

5.5 DIRECTIVE STATEMENTS

The Directive Statements control the assembly process and may generate data in the object program. The directive operator may be preceded by one or more labels, and may be followed by a comment. It occupies the operator field and is followed by either no operand or as many operands as required by the particular operator.

Assembler directive operators and their functions are summarized in table 5-4. Note that all directive operators begin with a period for easy visual differentiation from the instruction operator mnemonics in the output listing. Each directive operator is described in more detail in the following paragraphs.

Table 5-4. Summary of Assembler Directives

Directive Name	Function
.TITLE	Identification of program.
.END	Physical end of source program.
.LIST	Listing output control.
.SPACE	Space n lines in output listing.
.PAGE	Output listing to top-of-form.
.BYTE	8-bit (single-byte) data generation.
.DBYTE	16-bit (double-byte) data generation.
.ASCII	Data generation for character strings.
.LOCAL	Establish a new local symbol region.
.IF	Conditional assembly directive.
.ELSE	
.ENDIF	
.FORM	Field Specification.
.ADDR	Address constant generation.

5.5.1 .TITLE Directive

```
[label]     .TITLE     symbol [,string]     [;comments]
```

The .TITLE directive identifies the load module in which it appears with a symbolic name and an optional definitive title. If a .TITLE directive does not appear in the program, the load module is given the name MAINPR. If more than one .TITLE directive is used, the last one encountered specifies the symbolic name.

The symbolic name and the string must meet the symbol and string construction restrictions discussed in chapter 4.

Example:

```
.TITLE  TLLKP,'TABLE LOOKUP - 06/15/75'
```

5.5.2 .END Directive

[label] .END [address] [;comments]

The .END directive signifies the physical end of the source program. The optional address in the operand field may be either a symbol or a constant and indicates an execution address to the loader. In other words, it causes a branch to the address of the first executable instruction (entry point in contrast to load point) after the load is complete.

Examples:

1. No branch required

```
.END
```

2. Jump to the entry point at X'00A9

```
.END X'00A9
```

3. Jump to the entry point labeled START

```
.END START
```

5.5.3 .LIST Directive

[label] .LIST immediate [;comments]

The .LIST directive suppresses or reinstates the assembly program listing. Normally, the assembler is initialized in list mode; that is, a listing is produced as a result of an assembly operation. If a .LIST directive is encountered with a negative or zero operand, the output listing is suppressed. Otherwise, the directive reinstates the listing.

Examples:

1. Suppress the listing

```
.LIST 0
```

2. Reinstate the listing

```
.LIST 1
```

5.5.4 .SPACE Directive

[label] .SPACE immediate [;comments]

The .SPACE directive skips forward a specified number of lines on the output listing.

Examples:

1. Skip 20 (decimal) lines

```
.SPACE 20
```

2. Skip 20 (hexadecimal) lines

```
.SPACE 020
   or
.SPACE X'20
```

5.5.5 .PAGE Directive

[label] .PAGE [string] [;comments]

The .PAGE directive spaces forward to the top of the next page on the output listing. The optional string is printed as a page title on each page until a .PAGE directive containing a new string is encountered. No action is taken (except for a new page title) if the .PAGE directive is issued immediately after an assembler generated top-of-page request.

Example:

```
.PAGE   'TTY I/O ROUTINES'
```

5.5.6 .BYTE Directive

[label] .BYTE expression[,expression...] [;comments]

The .BYTE directive stores sequentially in memory one 8-bit byte for each given expression. If the directive has a label, it refers to the address of the first expression. The value of each expression must be in the range, -128 through +255.

Examples:

1. Single expression without a label

```
.BYTE   X'FF
```

2. Multiple expressions with a label

```
TBL:  .BYTE  MPR+10,X'FF,X'00
```

NOTE

TBL is assigned the location counter value of the byte containing the expression MPR+10.

5.5.7 .DBYTE Directive

[label] .DBYTE expression[,expression...] [;comments]

The .DBYTE directive stores 16-bit data in two consecutive 8-bit memory locations. Each expression of a .DBYTE directive is evaluated, and its value is placed in the next available pair of memory locations. The value of each expression must be in the range, -32768 through +65535.

The .DBYTE directive generates 16-bit address constants for use with memory-reference or memory-increment/decrement instructions (5.2.1 and 5.2.2). If the directive has a label, it refers to the memory address of the first byte generated by the directive.

Examples:

1. Without a label

```
.DBYTE   X'77FF
```

2. With a label

```
LABL:   .DBYTE   X'77FF
```

NOTE

LABL is assigned the value of the location of X'77.

5.5.8 .ADDR Directive

```
[label]      .ADDR      expression[,expression...]      [;comments]
```

The .ADDR directive generates 16-bit address constants to be used by transfer instructions. Each expression in the directive is evaluated, is decremented by 1, and then is placed in the next available pair of memory bytes. The decrementing takes into account the modulo-4096 address arithmetic used in SC/MP.

The effect of this directive is that if a pointer register is loaded with the resulting constant and exchanged with the program counter, the next instruction to be executed will be the one addressed by the value, expression.

Example:

```
AD1:    .ADDR     OUTPUT
          .
          .
          .
        LD        AD1
        XPAH      P3
        LD        AD1+1
        XPAL      P3
        XPPC      P3
```

The subroutine OUTPUT is executed.

5.5.9 .ASCII Directive

```
[label]      .ASCII      string[,string...]      [;comments]
```

The .ASCII directive stores data in successive memory locations by translating the characters in the string into their 7-bit ASCII equivalent code. Each string must be enclosed in single quote marks ('). Each character occupies one byte in memory. The .ASCII directive is used primarily to generate messages for output on teletypewriter or printer.

Example:

```
        .ASCII    'INPUT OF VALUE OF X'
```

5.5.10 .LOCAL Directive

```
[label]      .LOCAL      [;comments]
```

The .LOCAL directive establishes a new program section for local symbols (symbols beginning with a dollar sign ($)). Designated symbols between two .LOCAL directive statements have the value assigned to them only within that particular section of the program. Note that a .LOCAL directive is assumed at the beginning and the end of a program; thus, one .LOCAL directive within a program divides the program into two sections.

If the first character of a symbol is a dollar sign ($), the assembler attaches a unique character from the ANSI character set to the end of the symbol. Initially, this character is an exclamation point '!' (X'21). Each time a .LOCAL directive is encountered, the value of the added character is advanced by one with the letter "Z" (X'5A) as the last legal value. Therefore, up to 58 .LOCAL directives can appear in one assembly.

Example:

```
        .LOCAL
```

5.5.11 Conditional Assembly Directives

```
[label]     .IF         expression1[,expression2]   [;comments]
            .ELSE                                   [;comments]
            .ENDIF                                  [;comments]
```

The conditional assembly directives selectively assemble portions of a source program based on the value of the initial expression in the .IF directive statement.

All source statements between a .IF directive and its associated .ENDIF are defined as a .IF-.ENDIF block. These blocks can be nested to a depth of ten. The .ELSE directive can be included optionally in a .IF-.ENDIF block. The .ELSE directive segments the block into two parts. The first part of the source statement block is assembled if the .IF expression is greater than zero; otherwise, the second part is assembled. When the .ELSE directive is not included in a block, the block is assembled only if the .IF expression is greater than zero. If expression2 >0, the source code not assembled is listed anyway; if expression2 ≤ 0, the source code is listed only if it is assembled. The initial condition for this feature is not to list. If the condition is changed by including expression2, the new condition remains until modified. If an error is detected in either expression1 or expression2, the assembler assumes a TRUE value (greater than zero).

Examples:

1. Two-part conditional assembly

```
        .IF     COMPR   ┐
          .             │
          .             │  Assembled if COMPR greater than zero.
          .             │
        .ELSE           ┤
          .             │
          .             │  Assembled if COMPR less than or equal to zero.
          .             │
        .ENDIF          ┘
```

Chapter 6

PROGRAMMING TECHNIQUES

6.1 INTRODUCTION

This chapter discusses the programming techniques used to produce efficient SC/MP object code. Examples of coding are included to illustrate the method by which the techniques are implemented.

6.2 STACK PROGRAMMING

A convenient way of temporarily saving status and return addresses from subroutines and interrupt service routines is to maintain a stack in read/write memory. An advantage of a software stack compared to a hardware stack is that the hardware stack is limited in size to a fixed number of storage locations; any additional data pushed onto a stack cause an overflow and loss of data at the bottom of the stack. A software stack, on the other hand, can be made virtually any length, so overflow cannot occur. Another advantage of using software stacks is that more than one stack can be maintained.

The system software uses the following pointer register assignments:

Pointer Register	Function
P1	ROM Pointer and miscellaneous
P2	Stack Pointer
P3	Subroutine Pointer

Storing and retrieving data from the stack is accomplished by the following methods:

1. Store one byte of data or address

```
ST      @-1(P2)      ;PUSH A BYTE ONTO THE STACK
```

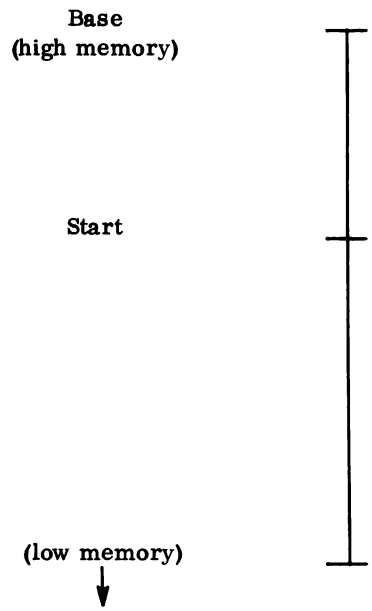
2. Retrieve one byte of data or address

```
LD      @1(P2)       ;PULL A BYTE OFF THE STACK
```

It should be noted that the auto-indexing feature is used to move the stack pointer address up or down the stack. The stack pointer (P2) always points to the last value pushed onto the stack.

6.2.1 Stack Operations

Using the stack conventions previously stated creates a stack that begins in high memory and extends downward. The most effective method of using this stack consists of fixing the base location of the stack, allocating any permanent locations required by the program, and then allowing the dynamic portion of the stack to expand and contract below that. For example see following page.



Permanent Area — contains global data for one or more programs. Fixed structure.

Dynamic Area — used by a particular program or subroutine at a particular time. The structure of this area depends upon the state of the system.

The permanent area of the stack may always be accessed by providing two words (labeled STKPT) for saving the stack pointer and then using the following code.

```
BASE    =    .                    ;PERM. AREA OF STACK
        .
        .
        .
        .=.-2
STKPT   =    .-BASE
        .
        .
        .
LDI     L(BASE)
XPAL    P2
XAE                               ;SAVE LO-HALF OF PTR
LDI     H(BASE)
XPAH    P2
ST      STKPT(P2)
LDE
ST      STKPT + 1 (P2)
```

Loading a Pointer from a word-pair pointed to by the same pointer.

```
LD      0(P2)                     ;LOAD UPPER POINTER ADDRESS
XAE                               ;SAVE
LD      +1(P2)                    ;LOAD LOWER POINTER ADDRESS
XPAL    P2                        ;TRANSFER TO LOWER P2
LDE                               ;RESTORE UPPER ADDRESS
XPAH    P2                        ;TRANSFER TO UPPER P2
```

6.2.2 Repeatable Subroutine Calls

If P3 is being used as a subroutine pointer, the subroutine may be called repeatably without reloading P3 as long as P3 is not disturbed. The subroutine must be set up as follows:

```
SIN:    .
        .
        .

        XPPC    P3          ;SUBROUTINE RETURN
        JMP     SIN         ;FOR REENTRY
```

All other things being equal, subroutines should be coded as repeatable.

6.3 SUBROUTINES

Because of the problems involved when a program crosses a page boundary, it is suggested that the programmer code his programs in modules smaller than a page. Organizing code into small subroutines is a more efficient way of coding, since it is easier to verify several small subroutines than one large program.

There are two methods used to implement subroutines, depending on whether the subroutine is single level or nested. Nesting is a condition where a subroutine contains calls to other subroutines.

6.3.1 Multilevel Subroutines

To implement multilevel (nested) subroutines, a stack must be created in memory. As an example, pointer P2 could be defined as the stack pointer. The address loaded into P2 would point to the top of the stack. This address would be a location in read/write memory.

If nesting is not required, subroutines can be called using the pointer registers to save the return address.

The following examples assume that SUBR is the label on the first instruction of the subroutine.

```
SUBROUTINE JUMP
SUBR1   =       SUBR-1
        LDI     L(SUBR1)        ;LOAD LOWER SUBROUTINE ADDRESS
        XPAL    P3              ;TRANSFER LOWER TO P3L
        LDI     H(SUBR1)        ;LOAD UPPER SUBROUTINE ADDRESS
        XPAH    P3              ;TRANSFER UPPER TO P3H
        XPPC    P3              ;EXCHANGE PC AND P3
SUBROUTINE RETURN
        XPPC    P3              ;RETURN FROM SUBROUTINE EXCHANGE
```

If multilevel subroutines are used, the current contents of the pointer register should be saved on the top of the stack and restored upon return from the subroutine.

```
        LDI     L(SUBR1)
        XPAL    P3
        ST      @-1(P2)
        LDI     H(SUBR1)
        XPAH    P3
        ST      @-1(P2)
        XPPC    P3
```

6.3.2 Jump Immediate

A jump immediate can be implemented directly using the subroutine jump as shown in example 6.3.1, but not executing a subroutine return. This facility allows a jump to any address in memory.

6.3.3 Conditional Subroutine Jumps

Conditional subroutine jumps can be implemented using a condition jump test to bypass the subroutine call. For example:

```
            JP      NOJSR           ;NO SUBROUTINE JUMP IF AC POSITIVE
             .        .
                      SUBROUTINE CALL AS IN OTHER EXAMPLES
             .        .
             .        .
NOJSR:
             .
             .
```

6.3.4 Multiple Subroutine Return

The same programming technique described in 6.3.1 can be used to establish more than one return address after a subroutine has been executed. This technique can be used to test a flag condition and branch conditionally to one of two or more locations, depending upon the condition of the flag. For example, a subtract routine might require three returns: one for a positive result, one for a negative result, and one for a zero result.

The example below affects a conditional call from a subroutine on the current page.

```
            LDI     LOWER           ;LOAD LOWER RETURN ADDRESS
            XPAL    P3
            ST      @-1(P2)         ;PUSH ONTO STACK
            LDI     UPPER           ;LOAD UPPER RETURN ADDRESS
            XPAH    P3
            ST      @-1(P2)         ;PUSH ONTO STACK
            XPPC    P3              ;JUMP TO SUBROUTINE
            JMP     ERR1            ;ERROR RETURN
            ST      RESULT          ;TEST OK, CONTINUE
             .
             .
             .
TEST1:       .
             .
             .
             .
            LD      @1(P2)          ;LOAD UPPER RETURN ADDRESS
            XPAH    P3              ;TRANSFER TO P3 HIGH
            LD      @1(P2)          ;LOAD LOWER RETURN ADDRESS
            XPAL    P3              ;TRANSFER TO P3 LOW
            JP      RETURN          ;ERROR IF AC POSITIVE OR ZERO
            LD      @2(P3)          ;INCREMENT RETURN ADDRESS
RETURN:     XPPC    P3              ;EXCHANGE P3 AND PC
```

6.3.5 Transferring Data to Subroutines

Frequently parameters must be passed to a subroutine when it is called; this is accomplished by listing the parameters in the bytes following the subroutine call and by incrementing the return address to the next executable instruction. Below is an example of the coding for this data transfer technique:

```
        JS      P3, MATH        ;JUMP TO SUBROUTINE
        .BYTE   X'01            ;2 BYTES PASSED TO
        .BYTE   X'02            ;SUBROUTINE
          .
          .
          .
MATH:   LD      @1(P3)          ;ADJUST PTR TO 1ST PARAMETER
        LD      @1(P3)          ;FETCH PARAMETER
        ST      PARM1
        LD      (P3)            ;FETCH PARAMETER 2
        ST      PARM2
```

At this point, the return address is in P3. The programmer may elect to leave it in P3, save it on the stack or store it locally until it is needed to return from the subroutine.

It also may be convenient to store all of the subroutine input parameters on the stack before calling the subroutine and then to have the subroutine place any output parameters on the stack before executing a return.

6.4 LOOP COUNTER

When executing a routine in which a group of instructions is repeated a given number of times, it may be convenient to use a memory location as a counter register. The address of the memory location used as a counter would be stored in one of the pointer registers.

An advantage of the use of a memory location as a counter rather than an internal register (such as E) that is the Increment and Load (ILD) and Decrement and Load (DLD) Instructions associated with memory-location increments and decrements do not affect the value of the carry bit in the status register. This is particularly important in serial arithmetic operations, where the carry bit must be saved for the next step.

It should be noted that both the ILD and DLD instructions destroy the contents of the accumulator; so the contents of the accumulator should be saved temporarily if they are needed in additional calculations.

The following exemplifies the implementation of a memory counter for a program containing a loop that is to be executed eight times.

```
        LDI     H(CNTR)         ;LOAD HIGH ORDER ADDRESS OF COUNTER
        XPAH    P1              ;P1 AS COUNTER POINTER
        LDI     L(CNTR)         ;LOAD LOW ORDER ADDRESS OF COUNTER
        XPAL    P1              ;P1 AS COUNTER POINTER
        LDI     8               ;LOAD NUMBER OF TIMES TO LOOP
        ST      (P1)            ;STORE COUNTER VALUE IN MEMORY
          .
          .
LOOP:   first instruction of loop
          .
          .
       *XPAL    P3              ;SAVE AC IN P3L
        DLD     0(P1)           ;DECREMENT COUNTER
        JZ      NEXT            ;IF COUNTER = 0, END OF LOOP
       *XPAL    P3              ;RECOVER AC FROM P3L
        JMP     LOOP            ;REPEAT LOOP
NEXT:  *XPAL    P3              ;RECOVER AC FROM P3L
```

* These instructions are required only if the value of the accumulator must be saved for next loop.

In a similar manner, the counter and temporary storage for the AC can be saved on the stack, thereby eliminating the overhead of initializing P1 (in the preceding example). The Extension Register may also be used as temporary storage for saving the Accumulator.

6.5 PAGE CONSIDERATIONS

PC-relative memory-reference instructions can only reference memory within the current page (4096 bytes); this requires the programmer to take certain precautions and use the techniques described in this section to avoid problems at page boundaries.

6.5.1 Instructions at the Page Boundary

The program counter does not automatically increment across page boundaries, but effects a "wrap-around" in the same page. Therefore, a two-byte instruction might occupy the last two bytes of a page or the last and first byte of the same page, but not the last byte of one page and the first byte of the next page. The assembler flags the condition of a two-byte instruction whose first byte is at the page boundary so the programmer can modify the source code.

6.5.2 Programs Residing Across Page Boundaries

Since PC-relative memory references are limited to the page occupied by the instruction, the simplest way of writing a program is to organize it as short subroutines, each of which resides within one page of memory. It is usually advisable not to fill a page with one subroutine, since corrections that require additional program steps could not easily be incorporated.

Using indexed addressing, it is relatively simple to write programs that occupy more than one page of memory. The first instruction on each page loads a pointer register with the alternate page address; indexed addressing is used to reference the alternate page, and PC-relative addressing is used to reference the current page. The techniques used to load a subroutine address apply here.

6.6 TEXT PROGRAMMING TECHNIQUES

When programs require extensive dialog, textual display, or printout, attention should be given to the technique that programs the textual printout, since it is likely to be subject to modification. One technique that readily lends itself to the SC/MP is the Literal Pool. Using this technique, all text is stripped of duplication; the resulting text then is stored in one area of memory. For example, consider the following five messages:

1. ENTER 5 COEFFICIENTS
2. COEFFICIENT OUT OF ALLOWED RANGE. RE-ENTER
3. ANSWER =
4. ANOTHER?
5. NO VALID ANSWER. RE-ENTER 5 COEFFICIENTS

Text for the five messages may be stored in a literal pool as shown below.

```
L1:     .ASCII      '.RE-'
L2:     .ASCII      'ENTER '
L3:     .ASCII      '5'
L4:     .ASCII      ' COEFFICIENT'
L5:     .ASCII      'S'
L6:     .ASCII      ' OUT OF ALLOWED RANGE'
L7:     .ASCII      'ANOTHER? '
L8:     .ASCII      'ANSWER'
L9:     .ASCII      '= '
L10:    .ASCII      'NO VALID '
L11:    .=.+1
```

Messages are created by indexing the literal pool using a two-byte repeating sequence. Byte 1 holds the displacement from the base of the literal pool to the first required character. Byte 2 holds the value of the number of characters to be printed. If the first byte of the byte pair holds the value X'FF, it signifies the end of the message; otherwise, another segment of the message is sought in the next byte pair. Each of the five messages described above could be created by the index sequence shown below.

```
I1:     .BYTE       L2-L1           ;ENTER 5 COEFFICIENTS
        .BYTE       L6-L2
        .BYTE       X'FF
I2:     .BYTE       L4-L1           ;COEFFICIENT
        .BYTE       L5-L4
        .BYTE       L6-L1           ;OUT OF ALLOWED RANGE
        .BYTE       L7-L6
        .BYTE       L1-L1           ;.RE-ENTER
        .BYTE       L3-L1
        .BYTE       X'FF
I3:     .BYTE       L8-L1           ;ANSWER =
        .BYTE       L10-L8
        .BYTE       X'FF
I4:     .BYTE       L7-L1           ;ANOTHER?
        .BYTE       L8-L7
        .BYTE       X'FF
I5:     .BYTE       L10-L1          ;NO VALID ANSWER
        .BYTE       L11-L10
        .BYTE       L8-L1
        .BYTE       L9-L8
        .BYTE       L1-L1           ;.REENTER 5 COEFFICIENTS
        .BYTE       L6-L1
        .BYTE       X'FF
```

A subroutine generates the printed messages. To write a message, the procedure is to call the subroutine and to specify the index that identifies the message to be printed. For example, to print "Answer =" the call would be as shown below.

```
LABEL   JMP         WRIT            ;WRITE "ANSWER ="
        .DBYTE      I3
         .                          ;RETURN POINT
         .
         .
```

Subroutine WRIT calculates the section of the literal pool to be printed. The first character is at the address: Li plus the contents of I3. The number of characters to be printed is derived from the contents of I3 + 1. The next byte contains X'FF, so printing is finished; otherwise, printing would continue with the next pair of index bytes specifying the next string of characters.

6.7 INPUT AND OUTPUT PROGRAMMING TECHNIQUES

The programming of data transfers between read/write memory and peripheral devices is generally classified as input/output programming. Depending on the significance of the input/output operations in the overall program, different approaches to input/output program implementation are recommended; these approaches are described in the following sections.

6.7.1 Programmed Input/Output

A programmed input/output operation is initiated and completed under the control of the initiating program. In figure 6-1, the program being executed starts the input/output operation; then the program waits for the operation to be completed before continuing.

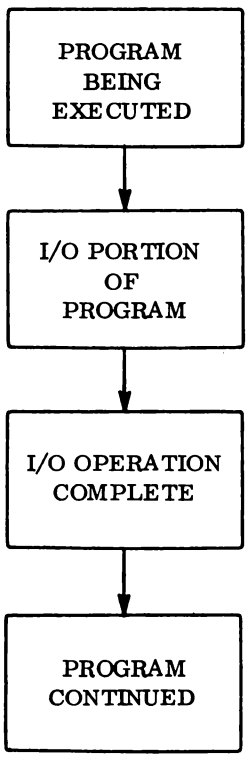


Figure 6-1. Programmed Input/Output

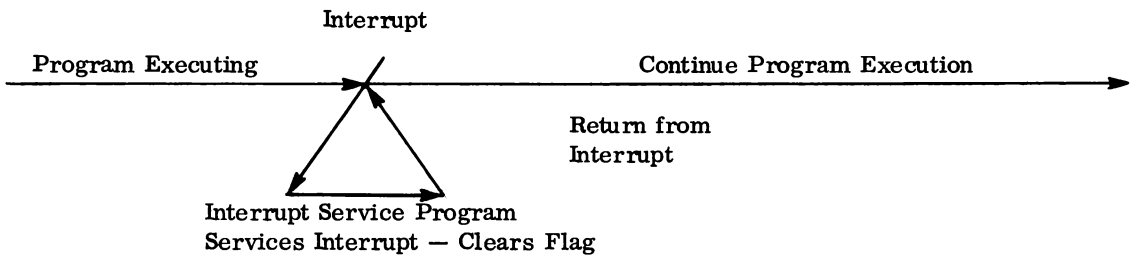
SC/MP allows any memory-reference instruction to execute a programmed input/output operation. Peripheral device controllers are assigned specific memory addresses, which when referenced by a memory-reference instruction, execute the input/output operation. It is necessary that the memory addresses assigned to the peripheral device be unique to the device, that is, no other device uses the same assigned memory addresses, nor is there memory with the same addresses. Also, the device controller must contain the necessary logic to decode its assigned addresses, then gate data on and off the data bus. By convention, memory addresses X'7FFF and below are reserved for read/write or read-only memory; memory addresses X'8000 to X'FFFF are reserved for peripheral device controllers. The user, however, can change this convention, as described in the SC/MP Users Manual.

The actual program steps required to enable programmed input/output will depend on the design of the device controller.

6.7.2 Interrupt Input/Output

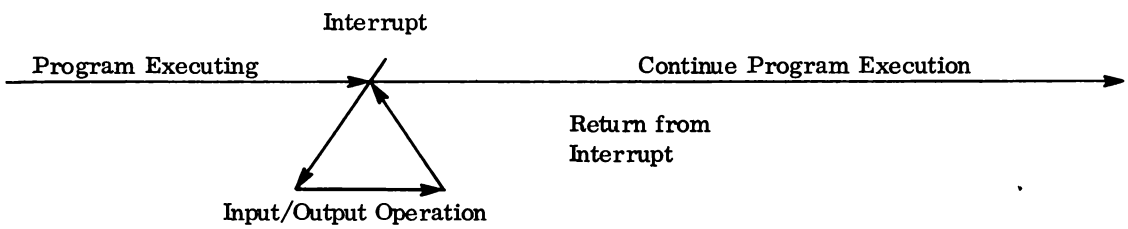
In certain cases, an input/output operation initiated by a program requires a significant length of time (many milliseconds) for execution, during which the program might perform other tasks. In other cases, the frequency of input/output service that requires a certain input/output device might be such that it would be convenient for the program to ignore the device unless it specifically requires service. Each of these situations may be handled by taking advantage of the SC/MP interrupt system and employing interrupt input/output for devices which have interrupt capability.

In figure 6-2, the program might initiate the input/output operation as part of its normal sequence of operation and set a flag indicating that such action was taken. An input/output device that has interrupt capability will, upon completion of an input/output operation, transmit an interrupt to SC/MP to indicate completion of the operation. The executing program is then interrupted for the time required to service the interrupt.



The flag may be employed by the original program to determine whether or not the input/output operation has been completed and whether or not the input/output device is still busy.

When the input/output device has data available for executing program, it might also transmit an interrupt to the CPU, causing interruption of the executing program for the duration of the executing program being executed.



Interrupt input/output requires a definite and specific sequence of events, irrespective of what peripheral device is to be serviced; the sequence is as follows:

1. In order for an interrupt to be accepted by SC/MP, the Interrupt Enable (IE) flag (bit 3) in the status register must be set and the interrupt system armed. The interrupt system is armed when the IE flag is changed from 0 to 1 (by an IEN or CAS instruction) and the next instruction is fetched and executed. If the interrupt is disabled (IE cleared), interrupt signals from peripheral devices are ignored, and no interrupt input/output operation starts. The instruction DINT disables interrupts.
2. Once an interrupt has been received and accepted by SC/MP, the following steps occur automatically under control of SC/MP.
   a. The instruction currently being executed is completed.
   b. Interrupts are disabled (IE cleared). Therefore, no further interrupts are accepted by SC/MP until interrupts are re-enabled.
   c. The contents of the PC are exchanged with the contents of P3.
   d. The instruction located at the location specified by the new (PC) + 1 is executed.

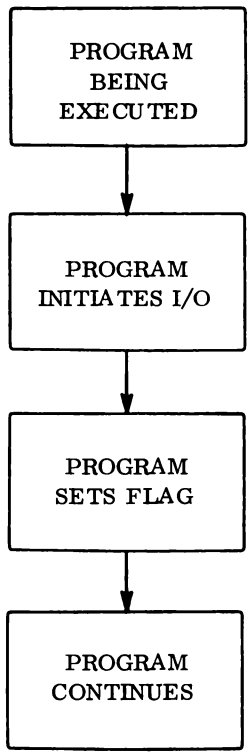


Figure 6-2. Interrupt Input/Output Initiation

3. The instructions at entry to the interrupt service routine must perform a number of housekeeping tasks before the required input/output operation can proceed. Tasks, in order of normal execution are as follows:

   a. Save the contents of registers that will be used by the interrupt routine so they can be restored just before returning from the interrupt. Register contents are typically saved on a stack, usually the same stack being maintained for subroutines.
   b. Determine the source of the interrupt. The way this is done depends on the design of the peripheral device controllers, but, usually, controllers are designed to respond to an interrupt acknowledge signal by transmitting a data byte (or word) that identifies the source of the interrupt. This acknowledge signal could be a particular address to which all interrupting devices respond.
   c. Once the interrupt has been identified, jump to the routine that services the identified device.

4. Execute input/output service routine of the selected device.

5. Restore the appropriate register contents that were saved in step 3a.

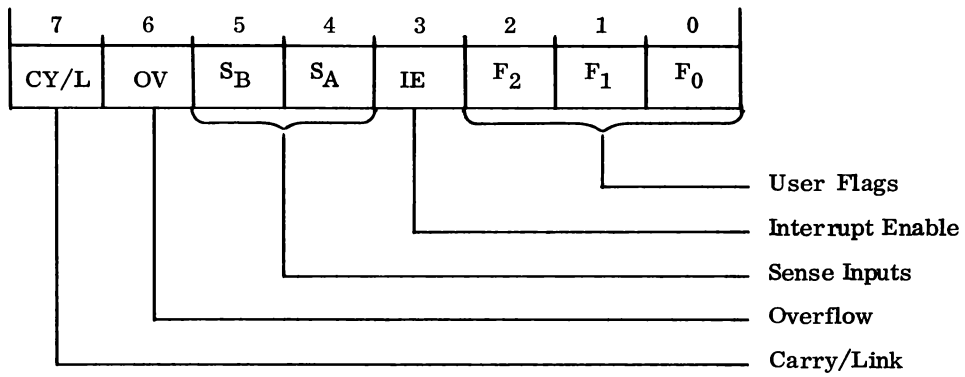
6. Return from the interrupt by enabling the interrupt, then exchanging the return address in P3 (step 2c) with the program counter, so the execution continues at the program instruction following the interrupt. The following example shows the technique to insure the contents of P3 upon return from the interrupt.

```
RETURN:   IEN                    ;ENABLE INTERRUPTS
          XPPC    P3             ;INTERRUPT SYSTEM IS ARMED AFTER THIS
                                 ;INSTRUCTION IS FETCHED.
INTSVC:   ...                    ;INTERRUPT SERVICE STARTS HERE
          <Service Routine>
          JMP     RETURN
```

6.8 USING THE STATUS REGISTER

6.8.1 General

The status register is an 8-bit register, where two bits automatically reflect the result of accumulator operations; one bit is the interrupt enable, and the five remaining bits are sense bits or user flags. The Sense-A input also serves as the interrupt request input.



The status register bits are modified as indicated in table 6-1.

Table 6-1. Status Register Bits

Flag	Automatic Operation	Special Instruction	CAS Instruction
CY/L	Set/reset by arithmetic operations and rotate with link	SCL sets CCL resets	Loads from AC_7
OV	Set/reset by arithmetic operations	——	Loads from AC_6
S_B	Reflects input line	——	Not affected, read only
S_A	Reflects input line	——	Not affected, read only
IE	Reset by interrupt	IEN sets DINT resets	Loads from AC_3
F_2	——	——	Loads from AC_2
F_1	——	——	Loads from AC_1
F_0	——	——	Loads from AC_0

All bits of the Status Register except the sense inputs can be set or cleared by the CAS Instruction.

The CY/L and OV bits are automatically set or cleared according to the result of accumulator operations, as described below. CY/L can also be set by the SCL Instruction or cleared by the CCL Instruction. The IE bit is automatically cleared when an interrupt is accepted. It can be set by the IEN Instruction and cleared by the DINT Instruction. The two sense bits reflect the state applied to the external sense pins. The 3 general-purpose flag bits are set or cleared only as directed by the CAS Instruction.

The Status Register is a passive depository of status information, and apart from the operations on CY/L, OV, and IE described, all status operations (for example, testing of status or setting/clearing the overflow or flag bits) take place in the accumulator. Thus, the normal sequence of status register operations is as follows:

1. Transfer status to accumulator
2. Test/set/clear status
3. Return accumulator contents to status register if update is required.

The contents of the Status Register may be tested by copying SR to AC, masking with a logical instruction, and testing with a conditional jump as shown in the test for overflow example below:

```
CSA                     ;COPY STATUS TO AC
ANI     X'40            ;CLEAR ALL BITS EXCEPT 6 (OV)
JNZ     OVFL            ;IF OVERFLOW, JUMP
```

Individual bits may be set or cleared using the following methods:

```
CSA                     ;COPY STATUS TO AC
ANI     X'FE            ;CLEAR F0.
ORI     2               ;SET F1.
CAS                     ;COPY AC TO STATUS
```

6.8.2 Arithmetic Operations

Arithmetic operations may be signed or unsigned. Consider first one byte. Unsigned, its numbering range is as follows:

from 0_{10} (X'00) = 0 0 0 0 0 0 0 0
to 255_{10} (X'FF) = 1 1 1 1 1 1 1 1

Signed, the high order bit is 0 for + (plus), 1 for - (minus), and the numbering range is as follows:

$+127_{10}$ = 0 1 1 1 1 1 1 1
⋮
0_{10} = 0 0 0 0 0 0 0 0
⋮
$-\ 1_{10}$ = 1 1 1 1 1 1 1 1
⋮
-128_{10} = 1 0 0 0 0 0 0 0

6.8.2.1 Arithmetic with Unsigned Data Bytes

Unsigned arithmetic uses the carry bit, but not the overflow bit. For example:

```
+ X'2E  =  0 0 1 0 1 1 1 0
  X'52  =  0 1 0 1 0 0 1 0
= X'80     1 0 0 0 0 0 0 0     CY/L = 0, no carry. Note that OV would be
                               set, but the program is not concerned.
+ X'AE  =  1 0 1 0 1 1 1 0
  X'52  =  0 1 0 1 0 0 1 0
= X'100    0 0 0 0 0 0 0 0     CY/L = 1. Note that OV would be cleared.
```

Consider the following subtraction using twos-complement arithmetic:

X'AE - X'52 = X'AE + ~ X'52 + X'01 = X'AE + X'AD + X'01 = X'5C

```
+ X'AE  =  1 0 1 0 1 1 1 0
+ X'AD  =  1 0 1 0 1 1 0 1
  X'01  =  0 0 0 0 0 0 0 1
= X'5C  =  0 1 0 1 1 1 0 0     CY/L = 1 answer positive. Note that OV = 1
```

The code for implementing the above operation is as follows:

```
LDI     0AE
SCL
CAI     052
ST      ANS
```

X'52 - X'AE = X'52 + ~ X'AE + X'01 = X'52 + X'51 + X'01 = X'A4 = -X'5C

```
          X'52  =  0 1 0 1 0 0 1 0
          X'51  =  0 1 0 1 0 0 0 1
          X'01  =  0 0 0 0 0 0 0 1
          X'A4  =  1 0 1 0 0 1 0 0     CY/L = 0 answer negative so take twos
ones complement =  0 1 0 1 1 0 1 1     complement. Note OV = 1.
twos complement =  0 1 0 1 1 1 0 0
                =  X'5C
```

The code for taking the twos complement is as follows:

```
SCL
XAE
LDI     0
CAE
```

Rules for addition and subtraction using unsigned data bytes are as follows:

1. Ignore the OV bit.
2. When adding, if CY/L is set, add 1 to the next high-order digit. CY/L is automatically an input to the add operation.
3. When subtracting, if CY/L is set, the answer is positive. If CY/L is cleared, the answer is negative and is present in its twos-complement form.

In multibyte arithmetic, the preceding three rules apply to the leftmost (terminal or high-order) byte. Between lower-order bytes, the CY/L bit is always treated as a carry into the low-order bit of the next byte:

```
+ X'13E7 = 0 0 0 1 0 0 1 1   1 1 1 0 0 1 1 1
  X'24C2 = 0 0 1 0 0 1 0 0   1 1 0 0 0 0 1 0
= X'38A9 = 0 0 1 1 1 0 0 0   1 0 1 0 1 0 0 1
                    ↖   ↗
                  CY/L = 1
                  Carry into next byte
```

6.8.2.2 Arithmetic with Signed Data Bytes

Since in signed arithmetic the high-order bit represents the sign, carries out of bit 7 different from carries into bit 7 represent overflow.

When performing signed arithmetic, the rules are essentially the same as for unsigned arithmetic with the following exceptions.

In a single-precision (one-byte) operation, or when operating upon the most significant byte in a multibyte operation, the overflow (OV) flag is set when an incorrect sign bit is generated as a result of the operation.

In performing multibyte arithmetic, the low-order byte should be processed first and any carries generated should be added into the next higher byte. This can be done automatically if the CY/L bit is not modified between the two operations.

6.8.3 Overflow and Carry/Link

The overflow bit is set whenever an add or complement-and-add operation causes a different carry bit into bit 7 from that out of bit 7; otherwise, it is not reset:

```
                7 6 5 4 3 2 1 0          Bit Number
  90        +   0 1 0 1 1 0 1 0    =      90
+ 117           0 1 1 1 0 1 0 1    =      117
  207       =   1 1 0 0 1 1 1 1    =      -49 signed twos complement
                ↖
                Carry out of bit 6, but not out of 7 bit.
                OV set to 1.

                7 6 5 4 3 2 1 0          Bit Number
            +   0 1 0 1 1 0 1 0    =      90
                0 0 1 0 0 1 0 0    =      36
            =   0 1 1 1 1 1 1 0    =      126
                ↓
                No carry out of bit 6, nor out of bit 7.
                OV set to 0.
```

The overflow bit is useful with signed arithmetic operations to indicate the generation of a result with an incorrect sign.

In addition to being used in rotate operations when specified, the carry/link bit is set whenever an add, decimal add, or complement-and-add operation causes a carry out of bit 7; otherwise, it is reset.

6.8.3.1 Add Operation with CY/L initially reset to 0

```
                  7 6 5 4 3 2 1 0          Bit Number
  X'B4          + 1 0 1 1 0 1 0 0    =       -76
+ X'D6            1 1 0 1 0 1 1 0    =       -42
                = 1 0 0 0 1 0 1 0    =      -118   signed twos complement
```

[1] ← Carry out of bit 7; CY/L is set to 1.
(Note that in this case OV would be reset.
Ones are carried into and out of bit 7.)

```
                  7 6 5 4 3 2 1 0          Bit Number
  X'34          + 0 0 1 1 0 1 0 0    =        52
+ X'56            0 1 0 1 0 1 1 0    =        86
                = 1 0 0 0 1 0 1 0    =      -118   signed twos complement or 138 unsigned 8-bit binary
```

No carry out of bit 7; CY/L is set to 0.
(Note that in this case OV would be set.)

6.8.3.2 Decimal Add Operation with CY/L initially reset to 0

(Note positive integers assumed)

```
                  7 6 5 4 3 2 1 0          Bit Number
  X'52          + 0 1 0 1 0 0 1 0    =        52
+ X'86            1 0 0 0 0 1 1 0    =        86
                = 0 0 1 1 1 0 0 0             38
```

Carry out of high-order digit; CY/L is set to 1.

6.8.3.3 Complement and Add Operation with CY/L initially set to 1

```
                                7 6 5 4 3 2 1 0    Bit Number
X'34                            0 0 1 1 0 1 0 0     52
X'56  0 1 0 1 0 1 1 0  ──→      1 0 1 0 1 0 0 1    -86
                                1 1 0 1 1 1 1 0    -34
```

No carry out of bit 7; CY/L is set to 0.
(Note that in this case OV would be reset to 0.)

Chapter 7

(FORTRAN) CROSS ASSEMBLER PROGRAM

7.1 INTRODUCTION

The (FORTRAN) Cross Assembler Program assembles a source program on a host computer for subsequent execution by an SC/MP Microprocessor. The assembler may be used on different host processors since it is written in FORTRAN IV (USA Standard Language Subset). It requires the following minimum peripheral hardware complement: processor input unit, scratch unit, list output unit, and binary output unit. The scratch unit is a mass storage device on which the source file must be written in order that the assembler be able to rewind it for multipass processing.

The Assembler accepts free-format source statements and, in two passes, produces a load module (object program) and a program listing.

Salient features of (FORTRAN) Cross Assembler Program are:

- Absolute load module generation.
- Conditional assembly facilities.
- Local symbols.
- Wide variety of assembly time operators (+, -, *, /, AND, OR, NOT).
- Diagnostic messages that include error position in source line.

Appendix I describes the use of the (FORTRAN) Cross Assembler and the related programs installed on the General Electric nation-wide timesharing system.

7.2 INPUT AND OUTPUT

The input and output files (data sets) required by the assembler are listed below:

Fortran File Name (DDNAME)	Function	File Format	Logical Record Length
FT05F001	Source File (Input)	Sequential	80 bytes
FT06F001	Listing File (Output)	Sequential	121 bytes
FT09F001	Load Module (Output)	Binary	36 words

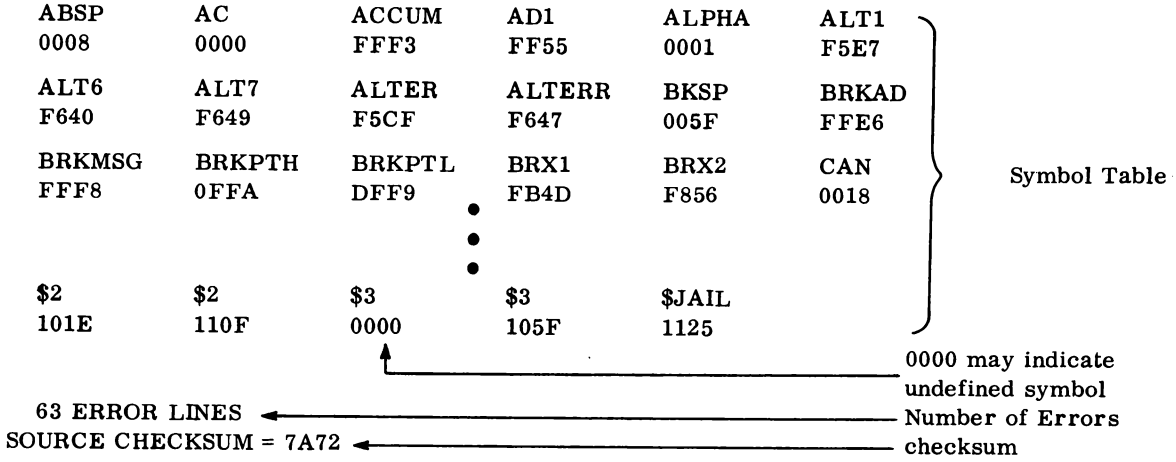
7.2.1 Source File (Input)

The Source File may be input via punched cards, paper tape, or from the keyboard of a computer terminal.

7.2.2 Program Listing File (Output)

The Program Listing contains ANSI-standard carriage control characters.

At the end of the Program Listing, a symbol table is produced, a message is printed noting the number of errors discovered by the assembler program, and the source and object checksums are printed; see below.



7.2.3 Load Module (Output)

The Load Module (LM) contains the object code produced from the source statements and loading information. The LM file is written as an unformatted file.

7.2.4 Format of LM File

The LM file is composed of a series of records, each containing 36 words. The representation of these records depends on the storage medium. There are three types of LM records:

Title Record (one per LM)

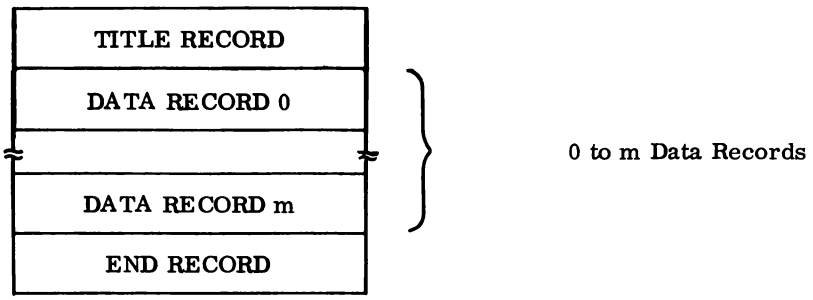
Data Record (variable number per LM)

End Record (one per LM)

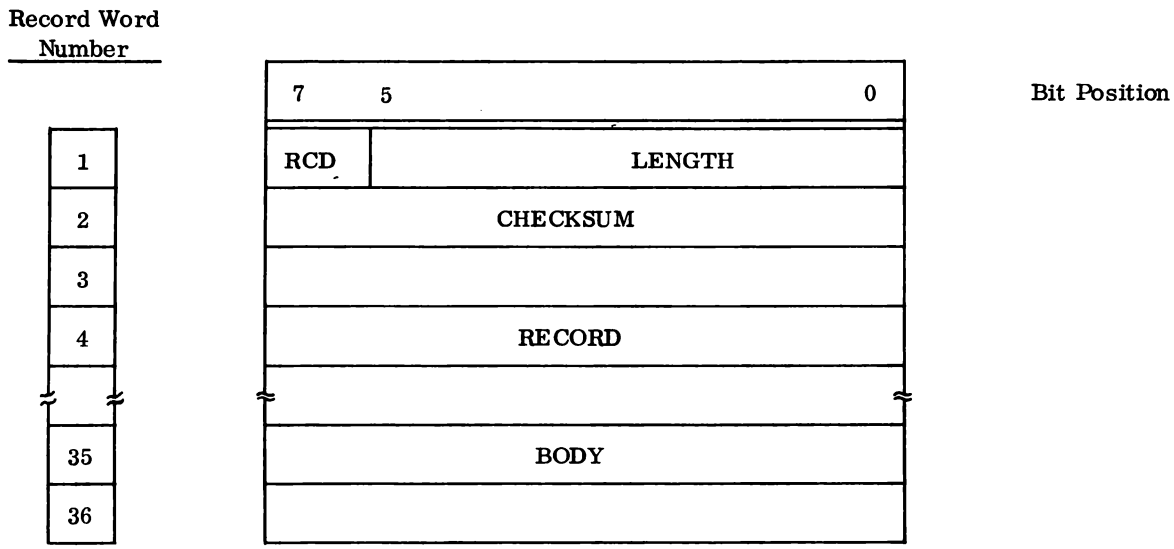
The records are produced in the sequence illustrated in figure 7-1A. Independent of the record type, the first two words (figure 7-1B) in each record always have the same interpretation. The first word specifies the record type and the length of the record body. The second word contains a checksum for error detection.

The Title Record identifies the load module by name and, optionally, by a descriptive character string. These two items are supplied by the last .TITLE Directive Statement in the source program. If this directive is not included, a default name (MAINPR) is used. If the default name is assigned, the qualifying character string is empty. Figure 7-2 illustrates the format of the Title Record.

The Data Records contain the actual data and instruction bytes to be loaded into memory. Each data record contains the load address for the initial data byte of the record. Each time a discontinuity (empty area or change-of-page) occurs in a program, the current record is terminated and outputted, and a new record is initiated. Figure 7-3 illustrates the format of the Data Record.



View A. LM File Format



Notes

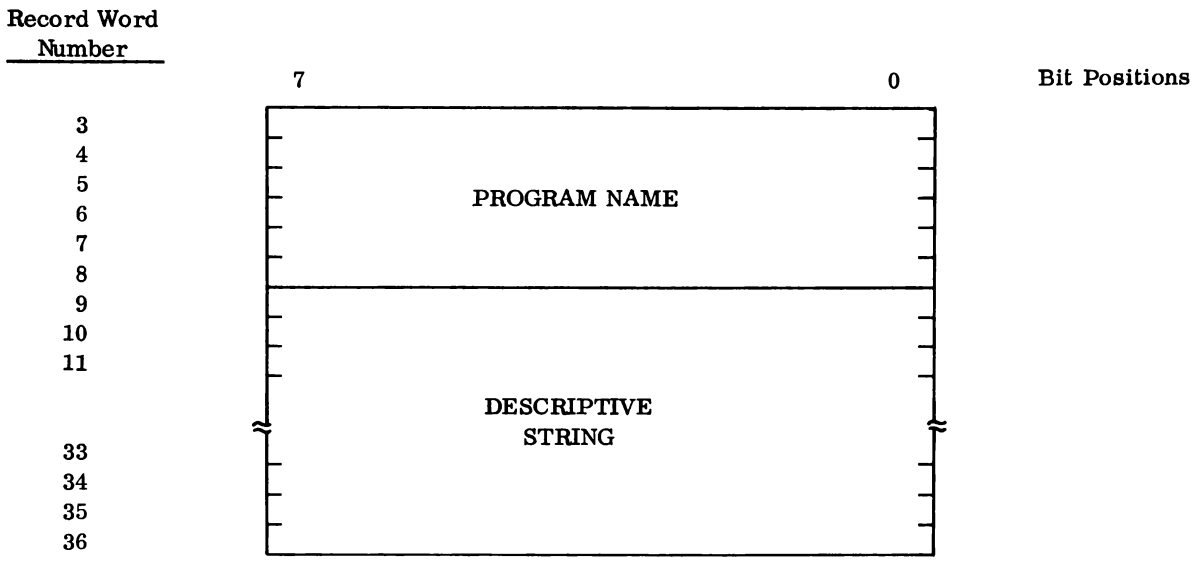
1. RCD specifies the type of record

RCD	RECORD TYPE
0	Title
1	Data
3	End

2. The CHECKSUM is formed by taking the arithmetic sum of all the words in the record body.

View B. General Record Format

Figure 7-1. LM File and General Formats



NOTE

1. The program name and descriptive string are made up of 7-bit ASCII characters.
2. If there are more than 28 characters in the descriptive string, only the first 28 characters are used.

Figure 7-2. Title Record Format

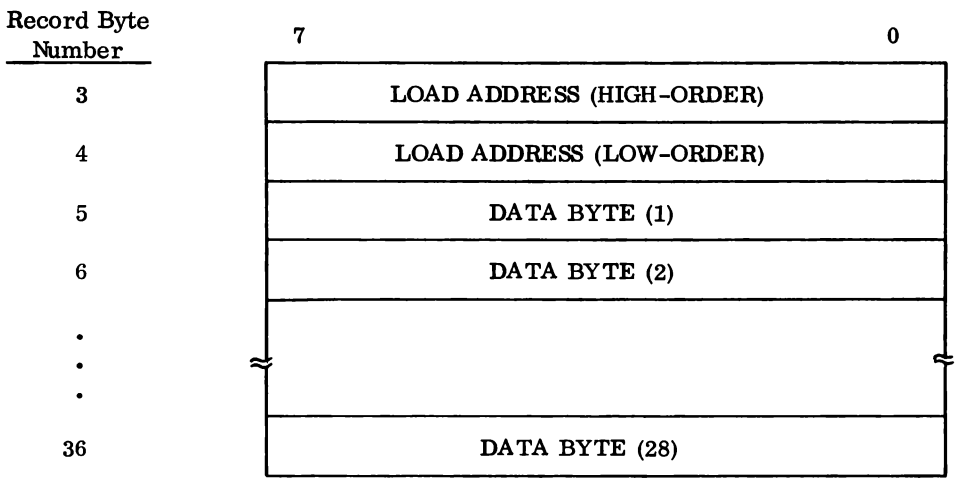


Figure 7-3. Data Record Format

The End Record marks the end of the LM file and specifies an entry address for the load module. The format of the End Record is illustrated in figure 7-4.

The source checksum represents the sum (modulo-2^{16}) of all the characters, taken one at a time, in the program source file. This sum is printed on the program listing following the symbol table printout.

The object checksum represents the modulo-2^{16} sum of all the individual record checksums of the LM. This sum is also printed on the program listing following the symbol table.

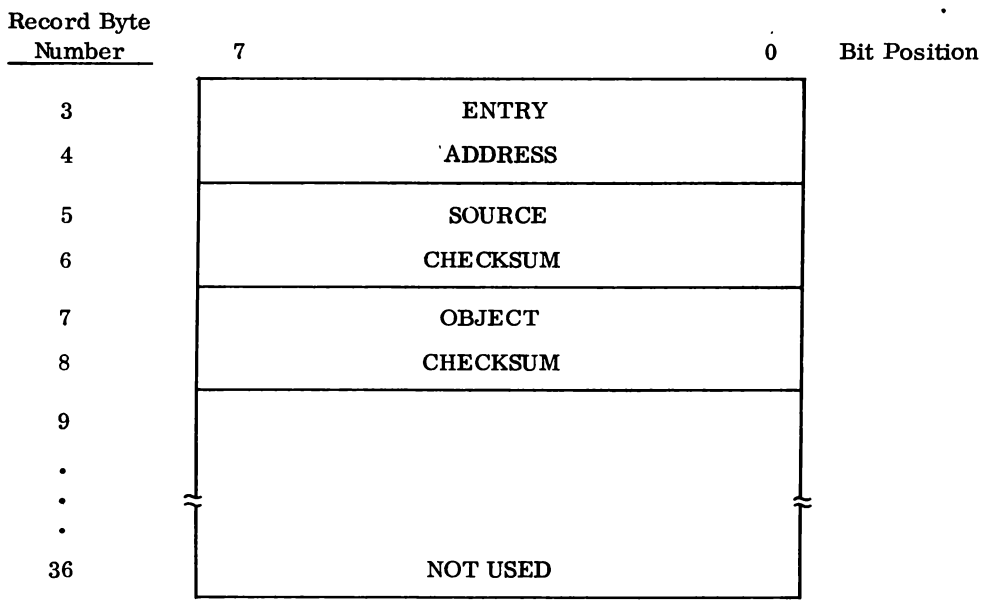


Figure 7-4. End Record Format

7.3 OBTAINING AN OBJECT CARD DECK

To obtain a load module in card format, the user must execute the FORTRAN program PRLM8. This program reads the assembler load module output file on FORTRAN file FT09F001 (unit 9), and outputs card images to FORTRAN unit 7, which is generally assigned to the card punch.

In a load module (LM) card deck, the first card contains !RLM in columns 1 through 4, and the following cards each contain an LM record. Each word of the record is represented by two hexadecimal characters.

Appendix A

ANSI CHARACTER SET

Table A-1 contains the 7-bit hexadecimal code for each character in the ANSI character set. The printable characters in this set may be set up as program data by use of the .ASCII directive. The remaining characters may be set up in hexadecimal constants with a .WORD directive. Table A-2 contains the legend for nonprintable characters.

Table A-1. ANSI Character Set in Hexadecimal Representation

Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number
NUL	00	SP	20	@	40	`	60
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	'	27	G	47	g	67
BS	08	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	-	2D	M	4D	m	6D
SO	0E	.	2E	N	4E	n	6E
SI	0F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DC1	11	1	31	Q	51	q	71
DC2	12	2	32	R	52	r	72
DC3	13	3	33	S	53	s	73
DC4	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B		7B
FS	1C	<	3C	\	5C		7C
GS	1D	=	3D	]	5D	ALT	7D
RS	1E	>	3E	↑	5E	ESC	7E
US	1F	?	3F	←	5F	DEL, RUBOUT	7F

Table A-2. Legend for Nonprintable Characters

Character	Definition
NUL	NULL
SOH	START OF READING; ALSO START OF MESSAGE
STX	START OF TEXT; ALSO EOA, END OF ADDRESS
ETX	END OF TEXT; ALSO EOM, END OF MESSAGE
EOT	END OF TRANSMISSION (END)
ENQ	ENQUIRY (ENQRY); ALSO WRU
ACK	ACKNOWLEDGE. ALSO RU
BEL	RINGS THE BELL
BS	BACKSPACE
HT	HORIZONTAL TAB
LF	LINE FEED OR LINE SPACE (NEW LINE): ADVANCES PAPER TO NEXT LINE BEGINNING OF LINE
VT	VERTICAL TAB (VTAB)
FF	FORM FEED TO TOP OF NEXT PAGE (PAGE)
CR	CARRIAGE RETURN TO

Character	Definition
SO	SHIFT OUT
SI	SHIFT IN
DLE	DATA LINK ESCAPE
DC1	DEVICE CONTROL 1
DC2	DEVICE CONTROL 2
DC3	DEVICE CONTROL 3
DC4	DEVICE CONTROL 4
NAK	NEGATIVE ACKNOWLEDGE
SYN	SYNCHRONOUS IDLE (SYNC)
ETB	END OF TRANSMISSION BLOCK
CAN	CANCEL (CANCL)
EM	END OF MEDIUM
SUB	SUBSTITUTE
ESC	ESCAPE. PREFIX
FS	FILE SEPARATOR
GS	GROUP SEPARATOR
RS	RECORD SEPARATOR
US	UNIT SEPARATOR
SP	SPACE

Appendix B

OPCODE INDEX OF INSTRUCTIONS

Opcode	Mnemonic	Operation	μcycles	Page
00	HALT	Pulse H-flag	8	5-19
01	XAE	Exchange AC and Extension	7	5-14
02	CCL	Clear Carry/Link	5	5-20
03	SCL	Set Carry/Link	5	5-20
04	DINT	Disable Interrupts	6	5-21
05	IEN	Enable Interrupts	6	5-20
06	CSA	Copy Status to AC	5	5-21
07	CAS	Copy AC to Status	6	5-21
08	NOP	No Operation	5	5-22
19	SIO	Serial Input/Output	5	5-17
1C	SR	Shift Right	5	5-18
1D	SRL	Shift Right with CY/L	5	5-18
1E	RR	Rotate Right	5	5-18
1F	RRL	Rotate Right with CY/L	5	5-19
30	XPAL	Exchange Pointer Low	8	5-16
34	XPAH	Exchange Pointer High	8	5-17
3C	XPPC	Exchange Pointer with PC	7	5-17
40	LDE	Load from Extension	6	5-14
50	ANE	AND Extension	6	5-14
58	ORE	OR Extension	6	5-14
60	XRE	Exclusive-OR Extension	6	5-15
68	DAE	Decimal Add Extension	11	5-15
70	ADE	Add Extension	7	5-15
78	CAE	Complement and Add Extension	8	5-16
8F	DLY	Delay	13-131593	5-22
90	JMP	Jump	11	5-12
94	JP	Jump If Positive	9, 11	5-13
98	JZ	Jump If Zero	9, 11	5-13
9C	JNZ	Jump If Not Zero	9, 11	5-13
A8	ILD	Increment and Load	22	5-8
B8	DLD	Decrement and Load	22	5-9
C0	LD	Load	18	5-5
C4	LDI	Load Immediate	10	5-9
C8	ST	Store	18	5-6
D0	AND	AND	18	5-6
D4	ANI	AND Immediate	10	5-10
D8	OR	OR	18	5-6
DC	ORI	OR Immediate	10	5-10
E0	XOR	Exclusive-OR	18	5-6
E4	XRI	Exclusive-OR Immediate	10	5-10
E8	DAD	Decimal Add	23	5-7
EC	DAI	Decimal Add Immediate	15	5-11
F0	ADD	Add	19	5-7
F4	ADI	Add Immediate	11	5-11
F8	CAD	Complement and Add	20	5-8
FC	CAI	Complement and Add Immediate	12	5-11

Appendix C

MNEMONIC INDEX OF INSTRUCTIONS

Mnemonic	Opcode	Description	μcycles	Page
ADD	F0	Add	19	5-7
ADE	70	Add Extension	7	5-15
ADI	F4	Add Immediate	11	5-11
AND	D0	AND	18	5-6
ANE	50	AND Extension	6	5-14
ANI	D4	AND Immediate	10	5-10
CAD	F8	Complement and Add	20	5-8
CAE	78	Complement and Add Extension	8	5-16
CAI	FC	Complement and Add Immediate	12	5-11
CAS	07	Copy AC to Status	6	5-21
CCL	02	Clear Carry/Link	5	5-20
CSA	06	Copy Status to AC	5	5-21
DAD	E8	Decimal Add	23	5-7
DAE	68	Decimal Add Extension	11	5-15
DAI	EC	Decimal Add Immediate	15	5-11
DINT	04	Disable Interrupts	6	5-21
DLD	B8	Decrement and Load	22	5-9
DLY	8F	Delay	13-131593	5-22
HALT	00	Pulse H-flag	8	5-19
IEN	05	Enable Interrupts	6	5-20
ILD	A8	Increment and Load	22	5-8
JMP	90	Jump	11	5-12
JNZ	9C	Jump If Not Zero	9, 11	5-13
JP	94	Jump If Positive	9, 11	5-13
JZ	98	Jump If Zero	9, 11	5-13
LD	C0	Load	18	5-5
LDE	40	Load from Extension	6	5-14
LDI	C4	Load Immediate	10	5-9
NOP	08	No Operation	5	5-22
OR	D8	OR	18	5-6
ORE	58	OR Extension	6	5-14
ORI	DC	OR Immediate	10	5-10
RR	1E	Rotate Right	5	5-18
RRL	1F	Rotate Right with Link	5	5-19
SCL	03	Set Carry/Link	5	5-20
SIO	19	Serial Input/Output	5	5-17
SR	1C	Shift Right	5	5-18
SRL	1D	Shift Right with Link	5	5-18
ST	C8	Store	18	5-6
XAE	01	Exchange AC and Extension	7	5-14
XOR	E0	Exclusive-OR	18	5-6
XPAH	34	Exchange Pointer High	8	5-17
XPAL	30	Exchange Pointer Low	8	5-16
XPPC	3C	Exchange Pointer with PC	7	5-17
XRE	60	Exclusive-OR Extension	6	5-15
XRI	E4	Exclusive-OR Immediate	10	5-10

Single-byte

Opcode	Mnemonic	Byte 1 7	6	5	4	3	2	1	0
00	HALT	0	0	0	0	0	0	0	0
01	XAE	0	0	0	0	0	0	0	1
02	CCL	0	0	0	0	0	0	1	0
03	SCL	0	0	0	0	0	0	1	1
04	DINT	0	0	0	0	0	1	0	0
05	IEN	0	0	0	0	0	1	0	1
06	CSA	0	0	0	0	0	1	1	0
07	CAS	0	0	0	0	0	1	1	1
08	NOP	0	0	0	0	1	0	0	0
19	SIO	0	0	0	1	1	0	0	1
1C	SR	0	0	0	1	1	1	0	0
1D	SRL	0	0	0	1	1	1	0	1
1E	RR	0	0	0	1	1	1	1	0
1F	RRL	0	0	0	1	1	1	1	1
30	XPAL	0	0	1	1	0	0	ptr	
34	XPAH	0	0	1	1	0	1	ptr	
3C	XPPC	0	0	1	1	1	1	ptr	
40	LDE	0	1	0	0	0	0	0	0
50	ANE	0	1	0	1	0	0	0	0
58	ORE	0	1	0	1	1	0	0	0
60	XRE	0	1	1	0	0	0	0	0
68	DAE	0	1	1	0	1	0	0	0
70	ADE	0	1	1	1	0	0	0	0
78	CAE	0	1	1	1	1	0	0	0

Double-byte

Opcode	Mnemonic	Byte 1 7	6	5	4	3	2	1	0	Byte 2 7	6	5	4	3	2	1	0
8F	DLY	1	0	0	0	1	1	1	1	disp							
90	JMP	1	0	0	1	0	0	ptr									
94	JP	1	0	0	1	0	1										
98	JZ	1	0	0	1	1	0										
9C	JNZ	1	0	0	1	1	1										
A8	ILD	1	0	1	0	1	0										
B8	DLD	1	0	1	1	1	0										
C0	LD	1	1	0	0	0	m	ptr									
C4	LDI	1	1	0	0	0	1	0	0								
C8	ST	1	1	0	0	1	m	ptr									
D0	AND	1	1	0	1	0	m	ptr									
D4	ANI	1	1	0	1	0	1	0	0								
D8	OR	1	1	0	1	1	m	ptr									
DC	ORI	1	1	0	1	1	1	0	0								
E0	XOR	1	1	1	0	0	m	ptr									
E4	XRI	1	1	1	0	0	1	0	0								
E8	DAD	1	1	1	0	1	m	ptr									
EC	DAI	1	1	1	0	1	1	0	0								
F0	ADD	1	1	1	1	0	m	ptr									
F4	ADI	1	1	1	1	0	1	0	0								
F8	CAD	1	1	1	1	1	m	ptr									
FC	CAI	1	1	1	1	1	1	0	0	disp							

Appendix E

INSTRUCTION EXECUTION TIMES

Instruction	Read Cycles	Write Cycles	Total Microcycles
ADD	3	0	19
ADE	1	0	7
ADI	2	0	11
AND	3	0	18
ANE	1	0	6
ANI	2	0	10
CAD	3	0	20
CAE	1	0	8
CAI	2	0	12
CAS	1	0	6
CCL	1	0	5
CSA	1	0	5
DAD	3	0	23
DAE	1	0	11
DAI	2	0	15
DINT	1	0	6
DLD	3	1	22
DLY	2	0	13-131593
HALT	2	0	8
IEN	1	0	6
ILD	3	1	22
JMP	2	0	11
JNZ	2	0	9, 11 for Jump

Instruction	Read Cycles	Write Cycles	Total Microcycles
JP	2	0	9, 11 for Jump
JZ	2	0	9, 11 for Jump
LD	3	0	18
LDE	1	0	6
LDI	2	0	10
NOP	1	0	5
OR	3	0	18
ORE	1	0	6
ORI	2	0	10
RR	1	0	5
RRL	1	0	5
SCL	1	0	5
SIO	1	0	5
SR	1	0	5
SRL	1	0	5
ST	2	1	18
XAE	1	0	7
XOR	3	0	18
XPAH	1	0	8
XPAL	1	0	8
XPPC	1	0	7
XRE	1	0	6
XRI	2	0	10

If slow memory is being used, the appropriate delay should be added for each read or write cycle.

Appendix F

DIRECTIVE STATEMENTS — INDEX

Statement	Operator Mnemonic	Operand Field
Address Directive	.ADDR	expression [,expression...]
ASCII Directive	.ASCII	string [, string,...string]
Byte Directive	.BYTE	expression [,expression,...expression]
Double-Byte Directive	.DBYTE	expression [,expression,...expression]
End Directive	.END	[address]
Form Directive	.FORM	symbol,exp [(exp)] , [exp [(exp)]]
Conditional Directives	.IF	$expression_1$ [, $expression_2$]
	.ELSE	not used
	.ENDIF	not used
List Directive	.LIST	immediate
Local Directive	.LOCAL	not used
Page Directive	.PAGE	[string]
Space Directive	.SPACE	immediate
Title Directive	.TITLE	symbol [, string]

Appendix G

PROGRAMMERS CHECKLIST

The following list of items is suggested for desk-checking a program before assembly.

1. Is the source program terminated by an . END Directive?
2. Is each label in the program terminated by a colon (:)?
3. Is each comment in the program preceded by a semi-colon (;)?
4. Is each string constant in the program set off on both ends by a prime (')?
5. Are all hexadecimal constants preceded by either X' or 0 (zero)?
6. For each .IF Directive in the program, is there a corresponding . ENDIF?
7. Are any symbols defined by two-level forward references? This is illegal.
8. Do transfer address operands consider memory page structure and PC pre-incrementation (before instruction fetch)?
9. Are the jumps relative to the current location specified in bytes?

Appendix H

PROGRAM DIAGNOSTIC MESSAGES

H.1 INTRODUCTION

When a source program error is encountered by either the (FORTRAN) Cross Assembler or the (IMP-16) Cross Assembler, an appropriate error message, together with a pointer, is printed in the output (object listing). The pointer is a "?" character.

H.2 (IMP-16) CROSS ASSEMBLER ERROR MESSAGES

The (IMP-16) Cross Assembler only detects the first eight errors found in each statement. The error is diagnosed and marked in the listing, in the following line, by an error message (described below) and a "?" character under the probable error field. An example of a cross assembler error detection is shown in figure H-1.

```
  158 1115   9C00              JNZ            BB-BASE1(P3)
ERROR ADDRESS                                          ?
```

Figure H-1. (IMP-16) Cross Assembler Error Detection, Listing Output

The following are the error messages:

1.	ERROR MISSING ARG.	This error indicates more arguments are required.
2.	ERROR VALUE	This error indicates value out of range or exceeds field size.
3.	ERROR ADDRESS	This error indicates address out of range.
4.	ERROR USAGE	This error indicates a number of possibilities including: a. A .IF nesting error b. Symbol not previously defined which would affect location counter c. Illegal expression, for example, two operators in sequence
5.	ERROR SYNTAX	Indicates an illegal character or improper statement construction.
6.	ERROR EXCESS ARG.	Indicates an existence of unprocessed arguments.
7.	ERROR TBL OVERFLOW	Indicates the following: a. If nesting level exceeds 10 b. Number of local regions exceeds 64 c. Symbol table overflow
8.	ERROR UNDEFINED	Used to indicate either an undefined symbol or undefined instruction/directive.

9. ERROR DUP. DEF. — Duplicate definition of the symbol.

10. ERROR POINTER — Indicates the following:

    a. Pointer Register should have been specified but was not
    b. Pointer Register 0 (PC) not allowed

H.3 (FORTRAN) CROSS ASSEMBLER ERROR MESSAGES

Each error is diagnosed and marked with a "?" character in the following line of the output listing. The "?" is placed under the probable error field. The error is also marked on the listing with an asterisk (*) in column 1.

1. ATTEMPT TO REDEFINE VALUE OF SYMBOL

   Symbol, assigned a value in assignment statement, is already defined or a symbol changed value from pass 1 to pass 2.

2. CONDITIONAL ASSEMBLY ERROR

   The conditional assembly directives do not balance. They must appear in sets of either .IF-.ENDIF or .IF-.ELSE-.ENDIF.

3. EXPRESSION VALUE EXCEEDS BOUNDS

   The value of an expression is too large for field (for example, immediate operand > 255).

4. ILLEGAL DIRECTIVE NAME

   The directive flagged is not one recognized by the assembler.

5. ILLEGAL EXPRESSION

   During evaluation of an expression, an illegal operator/operand combination was discovered.

6. ILLEGAL POINTER FIELD

   The given value of the Pointer Register must be 0, 1, 2, or 3.

7. ILLEGAL FORM SYMBOL

   The specified symbol is not recognizable as a legal operator or a defined .FORM symbol, or there was an error in the corresponding .FORM declaration.

8. ILLEGAL SYNTAX

   Instruction has incorrect structure (for example, operator not followed immediately by operand in an expression, .ASCII directive not followed by a 'string', illegal character in an expression).

9. INTEGER EXCEEDS LIMITS

   A decimal integer with a value less than -32,768 or greater than 65,535 or a hexadecimal value of more than four characters has been encountered.

10. LOCATION COUNTER OUTSIDE OF RANGE

    The value of location counter exceeds $65,535_{10}$ ($FFFF_{16}$).

11. MULTIPLE DEFINITION

    A symbol that appears in a .FORM statement or as a label is already defined.

12. OUT OF STORAGE

    The maximum number of .FORM statements has been exceded.

13. SYMBOL XXXXXX UNDEFINED DUE TO SYMBOL TABLE OVERFLOW (Pass 1 Message)

    The maximum number of symbols has been exceded.

14. TOO MANY LOCAL DIRECTIVES

    The maximum number of local directives has been exceded.

15. TOO MANY OPERANDS

    More operands appear in a .FORM call than appear in the corresponding declaration. Too many operands in instruction.

16. UNABLE TO GENERATE ADDRESS

    The memory reference instruction violates addressing limitations.

17. UNDEFINED SYMBOL

    The symbol flagged is not defined in this program.

18. END OF MEMORY PAGE

    The end of 4096-byte memory page has been reached.

H.4 OTHER ERROR CONDITIONS

If the assembly process is aborted by the operating system and a message is printed that indicates that an end-of-file condition was detected on the input file, the cause is probably omission of the .END directive at the end of the source program.

Appendix I

(FORTRAN) CROSS ASSEMBLER (SAS) G.E. TIMESHARING OPERATING PROCEDURE

The (FORTRAN) Cross Assembler (SAS) is installed and available to users of the General Electric national timesharing service under the program name, SAS$$$. In this section, instructions are given for (1) preparing and editing source code using any of the standard General Electric editors, (2) assembling the source code into SC/MP object code, and (3) converting the object code into media suitable for loading.

REFERENCES

1. Timesharing System Manual, General Electric Company, Palo Alto, California, Publication Number 711223, Mark II
2. Command System, General Electric Company, Palo Alto, California, Publication Number 3501.011, Mark III Foreground Reference Manual
3. Editing Commands, General Electric Company, Palo Alto, California, Publication Number 3400.01F, Mark III Foreground Reference Manual
4. RMS Remote Media Services, General Electric Company, Palo Alto, California, Publication Number 3710.04B, Mark III Foreground Reference Manual

Users need not be programmers. However, familiarity with the system is required. The General Electric Timesharing System Manual (Reference 1) provides information concerning operation and should be used to supplement the operating information contained in this appendix. The Command System Manual (Reference 2) describes the operating commands needed to operate the timesharing system, and the Editing Commands Manual (Reference 3) describes the commands needed to edit the SC/MP source code. Reference 4 contains information about the Remote Media Service provided by General Electric. This service, referred to in the text, may be used to obtain card output.

I.1 TELETYPE CONFIGURATION RESTRICTIONS

If your Teletype is equipped with the Automatic Answerback facility, this facility must be disabled or faulty object paper tapes may result.

I.2 HOW TO OBTAIN SAS$$$

SAS$$$ is available from the General Electric Timesharing Service. Contact the local representative of the General Electric Timesharing Computer Service in your area and ask for validation on the NAQ54 catalog. A local General Electric representative is listed in most telephone books under General Electric Company, Timesharing Computer Service. If you are unable to locate a General Electric representative in your area, call the Palo Alto, California, office of General Electric, Timesharing Computer Service.

I.3 HOW TO ACCESS THE COMPUTER

In all of the examples following, user input is underlined to distinguish it from the computer output. Pressing the carriage return key is represented by (CR) . The following procedure is used to access the General Electric Timesharing computer.

1. Turn on the terminal. If the terminal can be set to LOCAL or LINE, set it to LINE.
2. Telephone the local General Electric Timesharing Computer.

3. When the ringing stops and a high-pitched whistle begins, place the telephone handset on the acoustic coupler. Type the letter H or HELLO; then, press the carriage return key. The terminal will reply with the following:

   U#=

4. Type your user number, a comma, and your password; then, press the carriage return key. The computer will accept your user number and password and will type the following:

   ID:

   Or, if the computer does not recognize your user number, and/or the password is incorrect, the computer will type:

   VALIDATION FAULT, RETYPE IT --

   Retype your user number, a comma, your password, and the carriage return.

5. When the computer types ID:, type your project number or project identifier; then, press carriage return. The computer will reply with:

   SYSTEM-

6. Type in FIV (for FORTRAN IV) and carriage return. The computer will reply with:

   NEW OR OLD

7. If you are setting up a new data file, type NEW; if you want to work on a previously saved file, type OLD. Follow either entry with carriage return.

8. The computer will reply with:

   ENTER FILE NAME --

   If you are setting up a new file, assign it a name (up to 8 characters); if you are recalling an old file, type in the name of the file, followed with a carriage return. The computer will signal that the requested file is set up with the message:

   READY

You now can set up the new file or perform an analysis on your old file. Each line of the file should have a line number followed by at least one space. The contents of a line can be changed simply by retyping the same line number followed by the new text desired as long as the file still is being actively processed (prior to the SAVE Command). The computer automatically replaces the old line with the new one bearing the same line number. When a file is completed, it should be stored permanently in the computer. To store a new file, type SAVE. To replace an old file, type REPLACE. The computer saves the file and types out READY. Editing cannot be performed on a file that is printed out as the result of a LIST Command. Details of the data file are discussed in the following paragraphs.

I.4 HOW TO SIGN OFF

To terminate the timesharing operation, one of the following operations may be performed:

1. Remove handset from the acoustic coupler and hang up.

2. Type in GOODBYE or BYE, followed with a carriage return. The computer will reply with the total number of CRUs used, the terminal connect hours, and the number of input/output kilo characters transmitted.

To terminate a project, and immediately reestablish access to the computer for a new project, terminate the old project by typing in H or HELLO, followed by a carriage return. The computer will terminate the old project and will set up the new project with the message:

U#=

Repeat the sign-on procedures described in the paragraph titled, HOW TO ACCESS THE COMPUTER, from step 4, onward.

I.5 OVERVIEW OF MAJOR PROGRAMMING ACTIVITIES

Figure I-1 provides a flowchart overview of the use of the General Electric Timesharing Computer System for preparing user SAS programs. The procedures for the major activities in the flowchart are described in the following paragraphs. The major activities after logging into the General Electric System follow:

1. Create program.
2. Edit program.
3. Call assembler (SAS$$$).
4. Enter name of source program.
5. Answer questions regarding format of output desired.
6. If required — obtain printout of assembled code.
7. If required — correct errors in assembled code.
8. If required, execute PRLM8$ to format object output for tape or cards, or PROM8$ to punch PROM programming tape.
9. If required — use RMS to obtain output.

I.6 CREATING AND EDITING USER'S PROGRAM

In order to assemble a SC/MP program, the user first must enter his source program into the General Electric Timesharing System, using its file creation capabilities. To do this, the user must execute the commands shown below and, then, type in his source programs. In the example below, as in all examples in this manual, the user input is underlined and comments are typed in lower case.

```
U# NAQ54000,PSWD   (CR)
ID: IDENT   (CR)
SYSTEM-FIV   (CR)
OLD OR NEW-NEW   (CR)
FILE NAME-SOURCE   (CR)

READY
- - - - - - - - - - - - - - - - - - - - - - - - -   Type in your program at this point.
SAVE SOURCE   (CR)                                 Save the file with the name SOURCE for future reference.
```

Once you have created and saved your program, you may wish to make changes to it. To do this, first call up your program by typing in OLD and the name of your file. Then, to delete a line, merely type in the line number and hit a carriage return; to add or change a line, type in the line number and the new information. When all your changes are made, be sure to type REP to save the corrected version on your file. The following example illustrates these techniques (see page I-5).

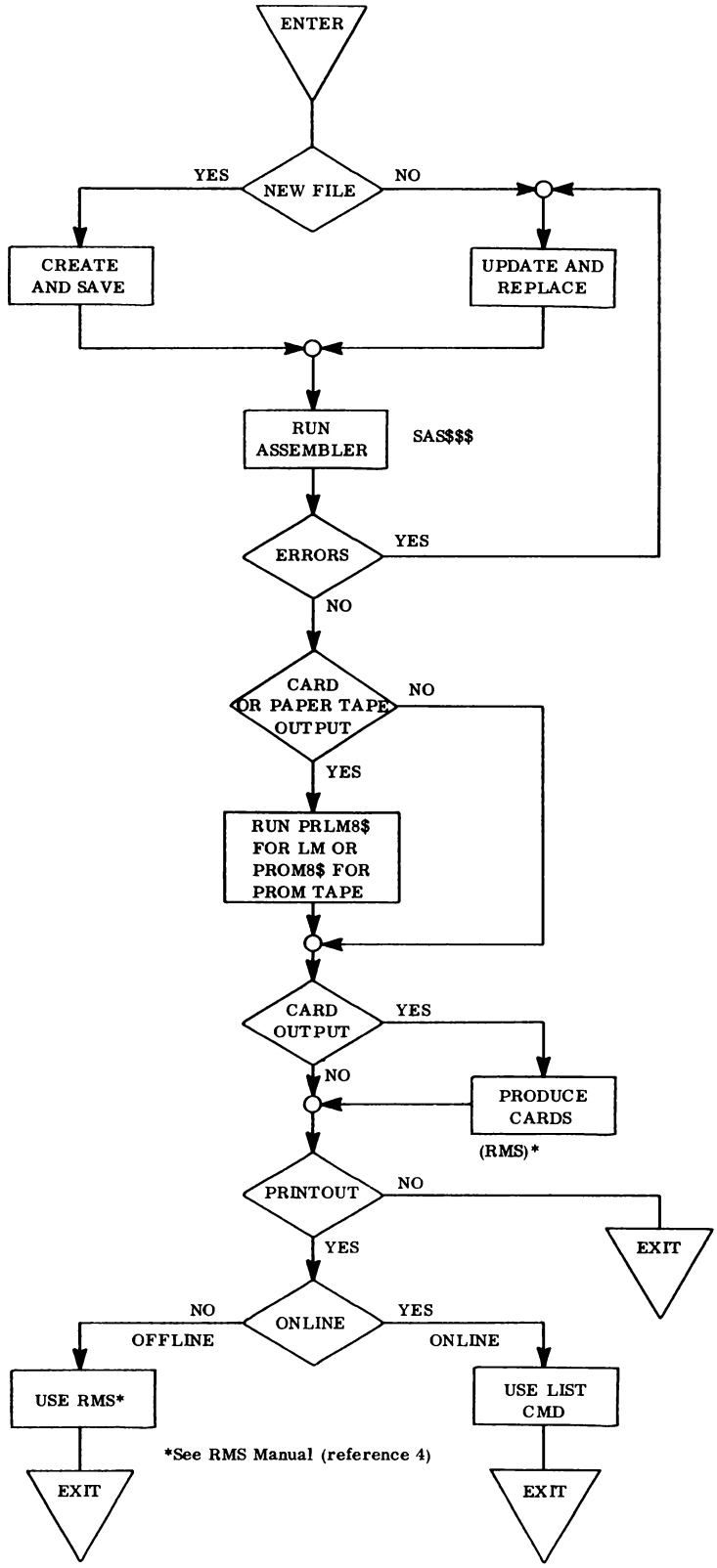


Figure I-1. Preparing User's SAS$$$ Programs (General Electric Timesharing System)

```
READY
OLD  CR
ENTER FILE NAME-SOURCE  CR
READY
15        ST        2(P2)        ;CLEAR MEMORY LOCATION
20        JMP       NEXT         ;GET ANOTHER VALUE
30  INCR: XAE                    ;SAVE REMAINDER

READY
REP SOURCE  CR                   Replace file SOURCE with this corrected version.
READY
```

More sophisticated editing techniques are available and are described in the General Electric Manual, EDITING COMMANDS, 3400.01F.

I.7 RUNNING THE ASSEMBLER

After a source file is created, the SC/MP Assembler may be executed by typing:

RUN SAS$$$ CR

SAS$$$ will prompt the user with various questions to determine:

1. The name of the source file
2. The name of the listing file
3. The name of the object file

After the program has assembled, the listing file may be listed either on-line on the TTY, or off-line on the computer center printer. To list on the TTY, type:

OLD listname CR

LIST CR

The list file can then be purged by typing:

PURGE list name CR

NOTE

If the break key is pressed during execution of SAS$$$, the run will terminate immediately and all scratch files created will be purged automatically.

I.8 OBTAINING OBJECT CARDS OR PAPER TAPE

The output LM of the SC/MP assembler is stored on disk in binary format as described in section 7.2.3 of this manual. To convert the LM to a loadable form on cards or paper tape the program PRLM8$ must be executed.

PRLM8$ is executed by typing:

RUN PRLM8$ CR

Upon initialization, PRLM8$ requests:

OBJECT AND OUTPUT FILENAMES, CARD/PPT

If the user desires card output, he should respond with the name of his assembled object file, the name of the file which will contain his card image output, and the designator, CARD, in the following format:

objectfile, outputfile, CARD CR

If the program is able to create the requested output file, the message

CREATED FILE outputfile

will be printed. Otherwise, the message will be

UNABLE TO CREATE FILE outputfile STATUS=errornr

Where errornr may be looked up in the GE Timesharing manual, "FORTRAN IV SYSTEM ROUTINES", in the section describing the CREATE command.

The system will signal completion of program execution by printing READY, at which time the user may make an RMS request to obtain his cards.

If he desires tape output, he may enter only the name of his assembled object file, followed by two commas and the designator, PPT, as follows:

objectfile,,PPT (CR)

The program will reply with

TURN ON PAPER TAPE PUNCH

and will punch the requested load module at the user's terminal.

I.9 OBTAINING PROM PROGRAMMING TAPE.

The user may also execute PROM8$ to obtain, via his terminal, a BPNF-formatted PROM programming tape to program either MM5203 PROMs (256 x 8) or MM5204 PROMs (512 x 8). This tape may be submitted directly to Customer Service, National Semiconductor Corporation, to have either of the above PROMs programmed.

PROM8$ assumes the user's program has been preallocated to the correct locations in SC/MP memory at assembly time and is organized such that the code and data for any 512-word block is contiguous within the load module.

PROM8$ may be executed following an assembly by typing:

RUN PROM8$ (CR)

Upon initialization, PROM8$ requests:

OBJECT FILE NAME

The user should enter the object file name of the PROM-programming program, followed by a carriage return.

The program then types:

FOR 5203 PROM, ENTER '2'; 5204 PROM, ENTER '4'

The appropriate response is either 2 or 4 followed by a carriage return.

After the requested load module is read into memory, PROM8$ notifies the user to:

TURN ON PAPER TAPE PUNCH

PROM8$ then begins immediate output of the requested tape.

The tape produced by PROM8$ is in standard "BPNF" format as described in the National Semiconductor Corporation MOS Integrated Circuits Manual, April 1974. The tape will contain the following:

1. Up to 12 null characters (X'00)
2. 32 rubout characters (X'FF)
3. Words 0 through 255 or 511 of page (i)
4. 32 rubout characters (X'FF)

This sequence is repeated for each 256 or 512 words.

Appendix J

(IMP-16) CROSS ASSEMBLER OPERATING PROCEDURE

J.1 INTRODUCTION

The (IMP-16) Cross Assembler is a 3-pass assembler designed to run on an IMP-16P Microprocessor. The assembler is available in 4K and 8K versions. The two versions are similar but have different minimum memory requirements and input/output facilities. The 4K assembler requires a minimum of 4K words of computer memory. It uses paper tape or the keyboard for the source input and object module output, and the Teletype for control input/output and for program listing. The map option for the 4K assembler produces an unsorted map. The 8K assembler requires a minimum of 8K words of computer memory. In addition to the options available on the 4K assembler, the 8K assembler provides input and output facilities for a high-speed printer and a card reader. The map option for the 8K assembler produces a sorted map. The .FORM directive is included only in the 8K version.

The (IMP-16) Cross Assembler accepts free-format source statements from either the keyboard, a paper tape reader, or a card reader (8K version) and produces an unlinked Load Module (LM) on paper tape and an object listing on the Teletype or the high-speed printer (8K version).

J.2 LOADING THE ASSEMBLER

This section gives only the steps required to load the (IMP-16) Cross Assembler. The assembler is loaded into memory using the Absolute Card Reader Loader (ABSCR), the Absolute Paper Tape Loader (ABSPT) or the Disk Bootstrap Loader (DBOOT).

J.2.1 Absolute Card Reader Loader (ABSCR)

ABSCR is the stand-alone loader program that reads the assembler from cards into memory. In the IMP-16P, ABSCR is resident in ROM.

The procedure for loading the assembler from the IMP-16P Card Reader into memory follows:

1. Place assembler (SCASM and SCASP) into card reader followed by !GO card and ready card reader.
2. Press INIT.
3. Set Mode Switch to PC.
4. Set Data Switches to X'7F00.
5. Press LOAD DATA.
6. Set Mode Switch to PROG DATA.
7. Press RUN.

J.2.2 Absolute Paper Tape Loader (ABSPT)

ABSPT is the stand-alone loader program that reads the assembler from paper tape into memory. ABSPT is resident in read-only memory (ROM) in the IMP-16. The procedure for loading the assembler from paper tape follows:

1. Press INIT.
2. Place assembler tape in Paper Tape Reader.

3. Press LOAD PROG.
4. Turn on Paper Tape Reader.
5. After assembler is loaded, press RUN.

ABSPT checks only for a checksum error and halts if one is discovered. To try to load again, position the paper tape at the beginning of the record in error, press RUN, and turn on the Paper Tape Reader.

J.2.3 Disk Bootstrap Loader (DBOOT)

DBOOT is a IMP-16P Floppy Disk Bootstrap Loader program resident on the Master Diskette. The procedure for loading the assembler from disk follows:

1. Press INIT.
2. Set Mode Switch to AC0.
3. Set Data Switches to contain the assembler disk address (initial sector number).
4. Press LOAD DATA.
5. Set Mode Switch to PC.
6. Set Data Switches to X'C000.
7. Press LOAD DATA.
8. Set Mode Switch to PROG DATA.
9. Press RUN.

J.3 INITIALIZING THE ASSEMBLER

Two entry points are provided for each version of the (IMP-16) Cross Assembler (in case recovery is required).

	4K	8K
START	2B0	89E
NEW ASSEMBLY	2E3	8DA

Assembly program initialization is provided when the respective program is started at the START entry point. The 4K version can accommodate approximately 175 symbols in its symbol table, while the 8K version can accommodate approximately 715 symbols in its symbol table.

When the assembler program is loaded, push RUN. The assembler program starts and the following message is typed out:

```
NSC SC/MP ASSEMBLER
MEMORY =
```

This message is a request for the user to specify available memory for the assembler's symbol table. Three forms of reply are accepted:

1. Default, by just pressing the carriage return key (CR) . The default condition is address 0 to address 4095_{10} for the 4K assembler, or address 0 to address 8031_{10} for the 8K assembler.
2. A continuous range of usable memory may be specified by entering the memory configuration, in the form

   a:b (CR)

   where "a" represents the lower limit of the specified memory range and "b" represents the upper limit of the memory range. Both "a" and "b" are in 1024-word units, such that a memory range specified as 0:8 represents a continuous memory region from address 0 to

address 8191_{10} (8192 words); and a memory range specified as 2:12 represents a continuous memory region from address 2048_{10} to address $12,288_{10}$.

3. A memory range consisting of two disjointed regions is specified by entering the memory configuration in the form:

   a:b, c:d (CR)

   where "a" and "b" are the same as described above, specifying the first region; "c" and "d" represents the second region. This ability to specify two regions is provided so that the user may reserve the contents of a memory range by specifying the memory below and above the range while excluding the reserved memory. A reply of 0:4, 60:64 indicates a memory configuration of 0 to 4095_{10} and $61,440_{10}$ to $65,535_{10}$.

The assembler uses this memory configuration information so that all memory that is not occupied by the assembler itself will be used for the symbol table. These numbers may vary depending upon the latest assembler release and the length of the user's symbols. Three words of symbol table are required for each symbol which contains four or less characters. Four words of symbol table are required for symbols containing more than four characters.

J.4 SELECTING OPTIONS FOR PROGRAM ASSEMBLY

At the beginning of each assembly, the assembler initializes all of the default modes for the input, listing, and output device selection. If an input, listing, or output device is not specified through a control option command, the assembler uses the default mode for the facility in question. The default mode for all options (if no option is specified) is as follows:

- Keyboard Input
- Full Listing Output on Teletype Printer
- No Object Module
- Symbol Map is output

Following the initialization of the default modes, the assembler prompts with a message to allow the operator to specify optional devices or control options. The prompt message is as follows:

.ASM

The first command (DE) is used for system communication purposes. The remaining commands (Table J-1) allow the user to select among the available assembler options (assuming the various resources are available). Control options may be specified in any order and should be separated by commas. If an erroneous control message is typed in, the assembler reinitializes and reprompts. If the programmer inadvertently requests conflicting options within the same category (for instance, requesting both keyboard and card reader inputs), the assembler accepts only the last entry. This facility may be used to change a requested option before the carriage return key is pressed.

The assembler also permits the user to specify assembly control in his source program. This is done by using the .ASM directive with the control options. If the .ASM directive appears in the source program, it is usually the first record. However, it may appear anywhere in his program: for example, to change the source device. Other than to change the source device, it is not recommended that the .ASM directive appear anywhere in the program except at the first record. The following is a typical example of the .ASM directive:

.ASM OM, NC, DT0300

The (IMP-16) Cross Assembler normally requires three passes over the source program; however, if the object listing is to be suppressed or written to the high-speed printer, or if the load module is to be suppressed or written to the diskette, only two passes are required. Pass two generates the program listing and pass three generates the load module.

J.4.1 Disc Editor (DE)

This command causes immediate transfer to the DOS Disc Editor and takes priority over all other assembler commands.

J.4.2 Disc Input (DI)

This command reads and assembles a source file from disc. The source file must have been previously written to disc using EDIT16 or the DT command (see J.4.3). The command is entered in the form "DIn" — where 'n' is the sector on which the source file begins. The sector number must have a leading zero if a hexadecimal number is desired; otherwise, it will be assumed to be decimal. If the sector number is not specified with the command, SCASM will assume that the source file begins at sector 0200_{16}.

Table J-1. Operator Selectable Options

Category	Command	Description	4K	8K	Ref Section
INPUT DEVICE	KB	Keyboard Input	X	X	J.8
	PT	Paper Tape Input	X	X	J.7
	CR	Card Reader Input		X	J.6
	DI	Diskette Input		X	J.4.2
INTERMEDIATE STORAGE	DT	Diskette Temporary File		X	J.4.3
OBJECT MODULE	OM	Load Module to Paper Tape	X	X	7.2.4, J.11
	DO	Load Module to Diskette		X	J.4.4
LISTING	NL	No Listing	X	X	J.10
	EL	Error Listing	X	X	J.10
	NC	No Comments	X	X	J.10
MAP	NM	No Map	X	X	J.10
LISTING DEVICE	PR	High Speed Printer		X	J.10
SYSTEM CONTROL	DE	Execute Disc Editor*		X	J.4.1

* Used alone, with no other options

J.4.3 Disc Temporary (DT)

This command causes a source file that is being read from cards or paper tape for the first pass of SCASM also to be written to disc. For the remaining passes, SCASM reads from the disc, thereby eliminating the re-reading of cards or paper tape. This command also establishes a source file that can be used by EDIT16.

The command is entered in the form "DTn" — where "n" is the sector where the source file is to begin. The sector number must have a leading zero if a hexadecimal number is desired; otherwise, it will be assumed to be decimal. If the sector number is not specified with the command, sector 0200_{16} is assumed to be the starting sector.

address 8191_{10} (8192 words); and a memory range specified as 2:12 represents a continuous memory region from address 2048_{10} to address $12,288_{10}$.

3. A memory range consisting of two disjointed regions is specified by entering the memory configuration in the form:

   a:b,c:d (CR)

   where "a" and "b" are the same as described above, specifying the first region; "c" and "d" represents the second region. This ability to specify two regions is provided so that the user may reserve the contents of a memory range by specifying the memory below and above the range while excluding the reserved memory. A reply of 0:4, 60:64 indicates a memory configuration of 0 to 4095_{10} and $61,440_{10}$ to $65,535_{10}$.

The assembler uses this memory configuration information so that all memory that is not occupied by the assembler itself will be used for the symbol table. These numbers may vary depending upon the latest assembler release and the length of the user's symbols. Three words of symbol table are required for each symbol which contains four or less characters. Four words of symbol table are required for symbols containing more than four characters.

J.4 SELECTING OPTIONS FOR PROGRAM ASSEMBLY

At the beginning of each assembly, the assembler initializes all of the default modes for the input, listing, and output device selection. If an input, listing, or output device is not specified through a control option command, the assembler uses the default mode for the facility in question. The default mode for all options (if no option is specified) is as follows:

- Keyboard Input
- Full Listing Output on Teletype Printer
- No Object Module
- Symbol Map is output

Following the initialization of the default modes, the assembler prompts with a message to allow the operator to specify optional devices or control options. The prompt message is as follows:

.ASM

The first command (DE) is used for system communication purposes. The remaining commands (Table J-1) allow the user to select among the available assembler options (assuming the various resources are available). Control options may be specified in any order and should be separated by commas. If an erroneous control message is typed in, the assembler reinitializes and reprompts. If the programmer inadvertently requests conflicting options within the same category (for instance, requesting both keyboard and card reader inputs), the assembler accepts only the last entry. This facility may be used to change a requested option before the carriage return key is pressed.

The assembler also permits the user to specify assembly control in his source program. This is done by using the .ASM directive with the control options. If the .ASM directive appears in the source program, it is usually the first record. However, it may appear anywhere in his program: for example, to change the source device. Other than to change the source device, it is not recommended that the .ASM directive appear anywhere in the program except at the first record. The following is a typical example of the .ASM directive:

.ASM OM,NC,DT0300

The (IMP-16) Cross Assembler normally requires three passes over the source program; however, if the object listing is to be suppressed or written to the high-speed printer, or if the load module is to be suppressed or written to the diskette, only two passes are required. Pass two generates the program listing and pass three generates the load module.

J.4.1 Disc Editor (DE)

This command causes immediate transfer to the DOS Disc Editor and takes priority over all other assembler commands.

J.4.2 Disc Input (DI)

This command reads and assembles a source file from disc. The source file must have been previously written to disc using EDIT16 or the DT command (see J.4.3). The command is entered in the form "DIn" — where 'n' is the sector on which the source file begins. The sector number must have a leading zero if a hexadecimal number is desired; otherwise, it will be assumed to be decimal. If the sector number is not specified with the command, SCASM will assume that the source file begins at sector 0200_{16}.

Table J-1. Operator Selectable Options

Category	Command	Description	4K	8K	Ref Section
INPUT DEVICE	KB	Keyboard Input	X	X	J.8
	PT	Paper Tape Input	X	X	J.7
	CR	Card Reader Input		X	J.6
	DI	Diskette Input		X	J.4.2
INTERMEDIATE STORAGE	DT	Diskette Temporary File		X	J.4.3
OBJECT MODULE	OM	Load Module to Paper Tape	X	X	7.2.4, J.11
	DO	Load Module to Diskette		X	J.4.4
LISTING	NL	No Listing	X	X	J.10
	EL	Error Listing	X	X	J.10
	NC	No Comments	X	X	J.10
MAP	NM	No Map	X	X	J.10
LISTING DEVICE	PR	High Speed Printer		X	J.10
SYSTEM CONTROL	DE	Execute Disc Editor*		X	J.4.1

* Used alone, with no other options

J.4.3 Disc Temporary (DT)

This command causes a source file that is being read from cards or paper tape for the first pass of SCASM also to be written to disc. For the remaining passes, SCASM reads from the disc, thereby eliminating the re-reading of cards or paper tape. This command also establishes a source file that can be used by EDIT16.

The command is entered in the form "DTn" — where "n" is the sector where the source file is to begin. The sector number must have a leading zero if a hexadecimal number is desired; otherwise, it will be assumed to be decimal. If the sector number is not specified with the command, sector 0200_{16} is assumed to be the starting sector.

J.4.4 Disc Object (DO)

This command allows the user to specify that an object module is to be produced (Pass 3 of SCASM) and written to disc instead of being punched on paper tape (see J. 11).

The command is entered in the form "DOn" — where "n" is the beginning sector to which the object module is to be written. The sector number must have a leading zero if a hexadecimal number is desired; otherwise, it will be assumed to be decimal. If the sector number is not specified with the command, the object module is written to the sectors immediately following the source file. If the source file is not on disc, and the sector number is not specified with the command, then the object file is written to disc beginning at sector 0200_{16}.

J.5 ASSEMBLING A PROGRAM

After the user enters the response to the .ASM directive, the source program is read from the input device (either card reader, paper tape reader, diskette, or keyboard). The assembler makes two or three passes over the source program. At the end of each pass, unless diskette is being used, the source must be reloaded for the next pass. Because subsequent passes begin automatically, the user must exercise care to ensure the correct input is read. If more than one program is being assembled using the card reader or the paper tape reader as the source input, it might be wise to turn off the reader when the end of pass message is being printed, reload the source, and then turn the reader back on. Obviously, if the user has only one assembly or if he is entering the source from the keyboard or diskette, such precautions are unnecessary.

At the beginning of the third pass, if required, the assembly halts for the operator to turn on the paper tape punch. After the operator turns on the paper tape punch, he strikes any key to continue. After assembling a program, the assembler reinitializes and prompts for the next assembly.

For the user entering the source program from the Teletype keyboard, horizontal tab facilities are provided to automatically position the Teletype carriage to the start of the operation field, the operand field, and the comment field. The operation is initiated by pressing the horizontal tab (HT) key on the Teletype keyboard. The start of these fields are located at columns 9, 17, and 33, respectively. This facility may be used to improve the clarity and order of the source listing.

J.6 CARD READER INPUT

A source program from the card reader contains one statement per card. Columns 1 to 72 contain the statement and columns 73-80 may contain identification information which is ignored by the assembler. A semicolon (;) causes immediate termination of the source statement scan. Otherwise, the source statement is terminated after column 72.

J.7 PAPER TAPE INPUT

A source program from paper tape contains one statement per record. Carriage return characters must terminate each record, and the record may not contain more than 72 characters.

J.8 KEYBOARD INPUT

A source program entered from the keyboard should be formatted one statement per record. Carriage return characters must terminate each record, and the record may not contain more than 72 characters.

The assembler prompts for each statement from the keyboard with a statement number followed by an asterisk (*).

J.9 KEYBOARD/PAPER TAPE SPECIAL EDITING CHARACTERS

Assembler input from the keyboard or paper tape allows the following editing characters:

NULL	(00)	Ignored
RUBOUT	(FF)	Ignored
LINE FEED	(0A)	Ignored
←	(5F)	Delete previous character (backspace)
ALT	(7D)	Delete source line
CR	(0D)	Terminates each source line

The above editing characters are processed as such, even if they appear with a character string.

J.10 LISTING

Figure J-1 illustrates a typical listing and map from the (IMP-16) Cross Assembler. Only the first 53 columns of each source statement are listed, although the entire source statement is processed.

If an Error Listing (EL) is requested, only those source statements in which an error is detected are listed along with the appropriate error messages (see F.2). Under this option no symbol map is output.

If the No Listing (NL) option is specified, no output listing is generated.

If the No Comment (NC) mode of listing output has been specified, one of the two following conditions will hold:

1. If the comment begins in column 1 of the card, the card is counted but is otherwise completely ignored by the resident assembler and no other actions are taken.
2. If the comment begins in other than column 1, the line is numbered and listed up to, but not including, the comment field.

J.11 LOAD MODULE (LM)

Each object module record is punched on paper tape in the following format:

8 null frames
Start of Text Character (02)
Object Module Record (see (7.2.4), Load Module (Output))
Carriage Return (0D)
Line Feed (0A)

The first record is preceded by 8 additional null frames and the last record is followed by 64 null frames.

NOTE

The term object module is used to describe the output of the assembler program (excluding the listing). The term load module is used when the object module is being loaded for execution.

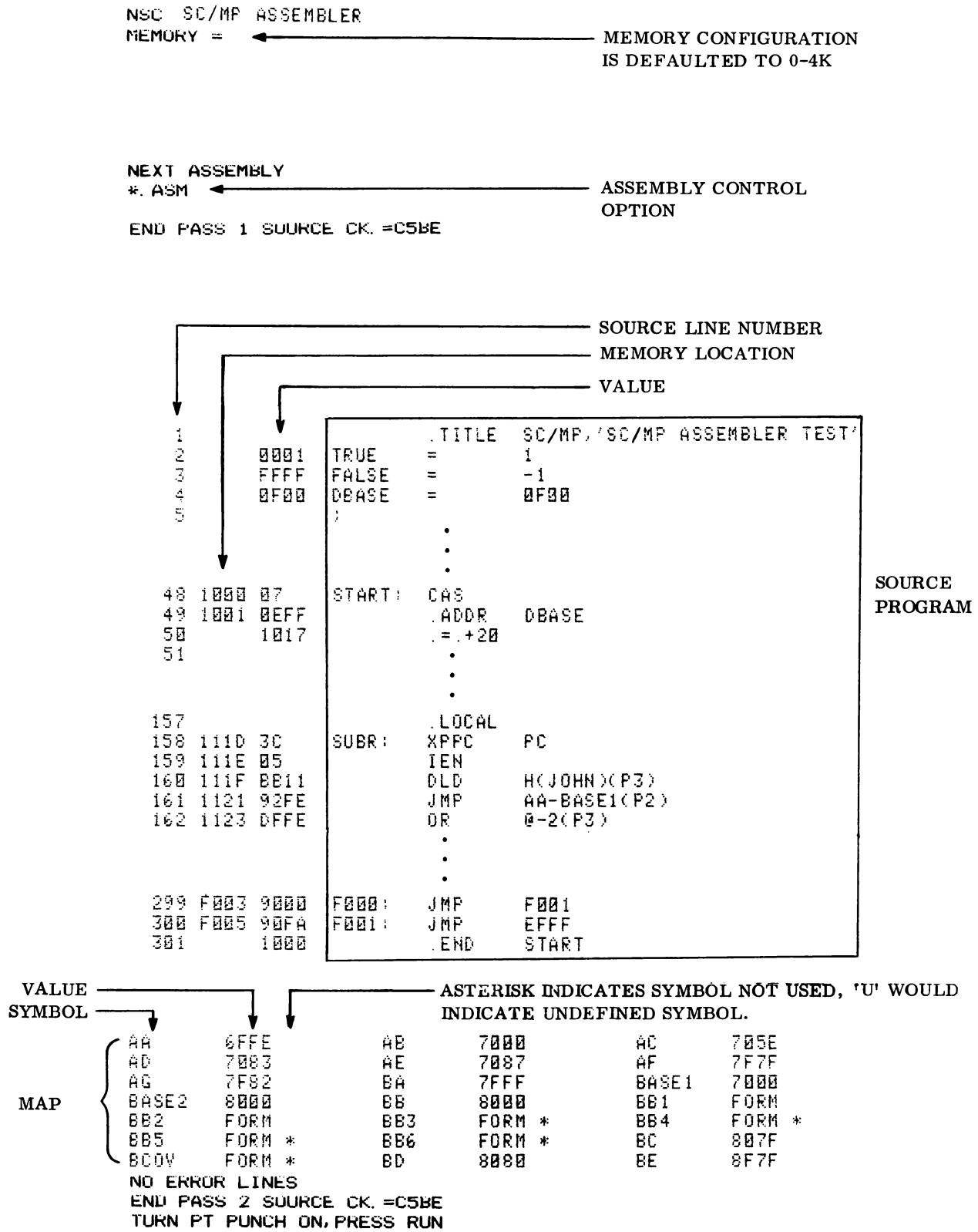


Figure J-1. Sample Listing of Assembler