A 7090 MACHINE LANGUAGE PROGRAMMING WORKBOOK AND LABORATORY

by

JEROME HOWARD SALTZER

S.B., Massachusetts Institute of Technology

(1961)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 1963

Signature of Author _
Department of Electrical Engineering, August 19, 1963

Certified by _
Thesis Supervisor

Accepted by _
Chairman, Departmental Committee on Graduate Students

A 7090

MACHINE LANGUAGE PROGRAMMING

WORKBOOK AND LABORATORY

by

JEROME HOWARD SALTZER

Submitted to the Department of Electrical Engineering on August 19, 1963 in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

A self-teaching laboratory workbook has been written on the subject of machine language programming for the I.B.M. 7090/7094 computer. This workbook presents the important ideas needed to code the computer in its symbolic machine language, from the viewpoint of an algebraic language programmer who finds that present versions of his language do not allow him sufficient flexibility to describe efficiently some parts of his problem. The workbook is complete, however, in that no previous knowledge of the computer is essential. An important feature of the workbook is that it is intended to be used in conjunction with remote consoles attached to a 7090 or 7094 computer controlled by a time-sharing system. The resulting interaction between the student and the computer makes possible an approach similar to that of a chemistry laboratory.

Thesis Supervisor: Fernando J. Corbató, Associate Professor of Electrical Engineering; Associate Director, M.I.T. Computation Center.

ACKNOWLEDGEMENT

The M.I.T. Computation Center has lent its facilities for the reproduction of this thesis.

Organization

This thesis is divided into two major parts. The first is a description of a self-teaching laboratory workbook for use in learning fundamentals of machine language programming for the I.B.M. 7090/7094 computer. The second part is the workbook itself. While the first part will make many references to the second, the second can stand by itself, and is preceded by its own introduction. To maintain this unilateral independence the first part will contain everything not intended to be a part of the workbook itself.

TABLE OF CONTENTS

PART I

A 7090

MACHINE LANGUAGE PROGRAMMING

WORKBOOK AND LABORATORY

Outline of the Project.

In programming a computer for scientific uses, one can distinguish between four different levels of computer "language." First is the algebraic language of the compilers, such as FORTRAN, ALGOL, or MAD. These algebraic languages are somewhat problem-oriented and bear little resemblance to the actual language of the computer.

Second is the symbolic assembly language commonly used for coding programs which require more flexibility. The symbolic language closely resembles the actual computer machine language, but usually includes many features for the convenience of the programmer, and to aid debugging.

The third level, for lack of a better tern, we may call "BSS form" although BSS is only one example of this level. Programs at this level are in numeric form; no translation as such need be performed on them before they may be loaded into the computer and run. However, sufficient information from the earlier levels has been retained with this form so that certain operations may be done to the program; retained are certain symbols to allow linking, and information to permit the program's relocation.

The fourth and last level is the machine language program, either punched in cards or loaded in the machine itself, which is self-contained, absolute, and to a certain extent, irrevocable.

The process of translation has become a matter of taking a program written at one level and reducing it to the next level; the execution of a program is often taken to include the final translation of level three to level four, which is more accurately described as loading and linking.

It has been well established that a programmer can learn to use a computer for quite sophisticated problems by learning only the highest level, an algebraic language; with really little or no information about what goes on below. Several modern texts teach programming from just this point of view. (Arden, Ledley, Galler*)

Learning level two, however, often brings with it much bigger implications; in particular the person wishing

* See bibliography.

to find out a little about the more flexible language of the computer itself is saddled with all the details from level two, three, and four.

The objective here is to formulate a text which covers only level two; the symbolic machine language, without requiring that the programmer know anything about the ultimate absolute program so produced, or about the intermediate BSS program involved. (If he wishes a higher level of sophistication, he should then progress to the BSS level.

Why is this attack possible now if it was not before? The development of timesharing for a large-scale computer has allowed the user more direct interaction with his program. For many users at level one, this greater interaction ability will bring about a desire to learn fundamentals of the machine language.

This more direct interaction means for many users the ability to examine dynamically programs which are not working correctly, and symbolic machine language debugging programs are being presently developed to aid in this dynamic interaction. Having available the supporting systems to allow the user to display and interact with his program at the symbolic level means that it is not necessary for him to learn any of the details of the levels below.

This completely symbolic interaction with a program is rapidly being approached for users of the algebraic languages, and it is the intent of this work to help allow a similar completely symbolic attitude be taken toward machine language programs.

In addition, although with not quite so much emphasis, this work is intended to explore and experiment with the use of the time-shared computer as an aid to teaching this subject; the atmosphere is unique in that the computer will in a sense be used as a laboratory tool for learning about the computer. Rather than exploiting so-called "teaching machine" techniques, the attitude taken will be more of that in a chemistry lab, in which the student simply attempts to obtain practice in the use of a tool by working with it.

For What Use?

Part II of this report, then, is a workbook designed to explain all important principles of machine language programming for the 7090/7094 computer, and intended to go hand in hand with use of the Compatible Time-Sharing System developed for that computer.*

This workbook represents the minimum amount of knowledge about the subject of machine language programming

* Corbató et al., see bibliography.

which would be desired in a beginning course on techniques of computer usage, and as such is intended for use in such a course. The M.I.T. course 6.41, Introduction to Automatic Computation, is just such a beginning course, and it is expected that this course will provide an experimental testing ground in the Fall term, 1963.

The workbook also represents the minimum knowledge prerequisite for an advanced level course in programming techniques, and could be used for that purpose.

The Idea.

One of the important ideas which will be observed to follow through the entire workbook is that the user should be programming a computer entirely in a symbolic language even though it be the machine language. He need not know anything about the explicit binary numbers into which his programs are translated.

The question which arises is, can the user understand what he is doing, can he have a feeling of knowing what is going on, and more importantly, can he use the computer to its maximum ability (the only reason for using machine language, anyway) if he does not know of the actual numbers which represent his symbolic instructions?

We must distinguish carefully between knowledge of the existence of the octal numbers as codes, and detailed knowledge of the codes themselves. For example, any machine language programmer on the 7094 realises that there are standard BCD codes for each of the 48 characters in the character set of the computer hardware. He understands that they require six bits of storage space, and therefore that six of them may be stored in each 7094 word. He may even know that their numerical values are in ascending order, as one goes through the alphabet, and that this feature may be used to alphabetize a list. On the other hand, he does not need to know that the octal code for "Y" is 70; and he probably does not know this. While this is but one example in which assembly programs and symbolic dumps have relieved the programmer of memorizing 48 otherwise arbitrary and uninteresting BCD codes, some other examples show the opposite extreme. For example, floating point numbers are occasionally listed as a piece of necessary knowledge; many textbooks have exercises in conversion in and out of this representation. The programmer should only be burdened with the knowledge that the number A x 2^b can be stored in one word, with the value of A in one place and the value of b in another. The details may be left to dump and assembly programs, and to computer hardware.

Available Texts.

To what extent do available texts already supply us with the materials described? The answer is that to a very great extent, the viewpoint of most texts presently available tends to one of two extreme attitudes, neither of which allows the desired presentation. The first is that the only way to program a computer is in machine language; use of higher level languages and compilers is reduced to the level of a technique at best.[1] Such an approach leads to the covering of involved subject matter such as subroutines, interpreters, arithmetic calculation, and input-output programming with the highly detailed machine language description. Despite the proven abilities of compilers to save much programming effort in many routine jobs, after reading one of the standard machine language programming texts, the student would probably believe that there is only one effective way to tackle a programming problem, the machine language of the computer.

The second approach, characteristic of some recent texts is that the primary way of expressing oneself to a computer is via an algebraic or symbolic language far removed from the actual machine representation.[2] While this viewpoint may well have some merit, current algebraic languages are inadequate to efficiently describe

---

1. Sherman, Leeds & Weinberg--see bibliography.
2. Galler, Arden, Ledley--see bibliography.

and compile for certain problems. Therefore this viewpoint tends to lead to a programming text which hardly mentions the machine language of the computer at all, and in any case leaves the student with the distinct idea that there is no earthly reason for wishing to go to the machine language for any type of problem. One has the feeling that some of these symbolic language oriented texts would get around to describing machine language in its proper context if they had room to do so, although this would require that they commit themselves to a particular machine language for the description and thus cost their present generality.

The present work, then, is intended more as a supplement to a description of compiler languages, for the programmer who has used FORTRAN, for example, and suspects that certain sections of the next problem he will tackle will exceed practical bounds in time or computer size, if he attempts to use that language. Being now forced to utilize a more flexible language, he needs a minimum description of the ideas involved in machine language programming, not a complete discussion of the ins and outs of all possible computer applications from a text convinced that all programs whould be written in SOS or FAP.

In addition to covering this minimum amount of information needed for use of machine languages, the present workbook intends to reap fullest possible benefit from the use of a time sharing system to permit ready access to the computer. The idea of an integrated laboratory is again missing from most available texts. (A review of several recent texts on computer programming is found in the appendix.)

Description of the Workbook.

The workbook begins with a brief description of itself, a condensed version of part I of this report. It tells the reader what to expect (and not to expect) from a study of the workbook, and how to use it in connection with the time-shared consoles of the M.I.T. Computation Center.* Detailed information on the method of interaction with the consoles, and as to the status or method of having programs is left out of this section, however, and in fact does not appear at any point in the workbook. The reason for this omission at such a critical point is that the time-sharing system and its consoles and support programs are so highly experimental that no precise description would remain valid for a sufficient length of time to

* See reference one in the bibliography for information about this time-sharing system.

warrant inclusion in the workbook. Also, the particular support programs to go with the workbook are experimental too and subject to change as more information is gained, and as the time-sharing system evolves. Therefore, the student is to obtain a brief memo describing present status and methods of the time-sharing system to use with his study.

A technical introduction to the computer follows, in which the reader is very quickly introduced to instruction sequencing, loops, program modification, conditional branching, and symbolic programming. While this is a big package, remember that the primary emphasis is on the computer user who knows an algebraic language such as FORTRAN or MAD, and has seen many of these ideas before. The rank beginner will indeed find that this collection of ideas is a big one, but nothing is left out from the description so with a little extra study he will be able to follow. As his first interaction with a time-sharing console, the user is told to assemble and run a program which has been previously written and placed into the time-sharing system. This exercise has the value that it can be successfully accomplished without completely understanding what is going on at first, although hopefully the execution of the exercise will bring this understanding.

Each of the following lessons has an organization which goes roughly like this: 5-6 pages of material including illustrations of each of the major points, an example program using each of the points covered, and a problem to be solved by writing a short program segment and debugging it at the computer console. It is expected that the student would study the textual material and examples about 2-4 hours, and spend another hour working out a solution to the problem. He would then check out his program at a computer console. The amount of time for this last requirement is open to question, but experience with the consoles indicates that if he can obtain rapid response to his requests roughly 1/2 to 1 hour should be required per problem, which represents 3 to six minutes of computer time.

Since there are eight lessons, the student can expect to spend about 40 hours learning the material in the workbook.

Before beginning an examination of the lessons and the material they cover, we should first consider the overall picture and the question of how far they go: what is the stopping point for the workbook?

Stopping Point

The most sophisticated instruction (and technique) to be covered is the TSX instruction for subprogram linkage, the construction of an external subroutine, and the use of the calling sequence to transmit parameters; this does not involve a discussion of BSS linkage and transfer vectors. As far as possible, no techniques as such are discussed, this is left for later texts. The main purpose is to discuss the rules of proper construction of simple programs in the symbolic 7090/7094 machine language, in as realistic framework as possible. The user should then be able to apply his knowledge to a course in detailed techniques or to his own specialized problem.

The Lessons.

Perhaps the most controversial thing in the first lesson is that the first item discussed is an index register, and the first two instructions introduced are AXT and TIX. Having discussed these, however, the student has been introduced to a typical use of the machine language, and the great advantage is obtained that future programs, all of which require loops, are much simplified.

Shifting is the other item of interest introduced here, along with the AC and MQ registers. The first program described can thus be done entirely within the live registers, without reference to core storage for pieces of data.

In lesson two the BCD code is discussed with the use of a pseudo-operation to generate BCD codes. Compare instructions are introduced, allowing a program to be written which counts commas. A lot of ground is covered in the area of notation, and questions of which bit in the AC is affected by which instruction are mentioned, so that the student knows that there is an issue.

Arithmetic instructions and number representation are left until exercise three, in which a parity checking program is described. Putting arithmetic and number representation after BCD representation brings home the point that numbers are just another interpretation of bit patterns within the computer, and that they are only a fundamental interpretation in the sense that special machine instructions are provided to enable this interpretation.

In lesson four, program modification is emphasized, and the student is given a program which cannot really be done effectively without program modification: A string pointer list program. The motivation for these important ideas is carefully established.

Lesson five introduces index registers in the role of effective address counters, and it is shown how a program can be much simplified with the use of effective addresses.

In order to make sure that the student can in fact use his knowledge, input and output are discussed in lesson six. However, this is not input-output in the usual sense, as no write selects or channel loading instructions are mentioned. Instead, the reader is encouraged to use subroutines provided by the same system programming staff which provides his assembly programs; he is made to realize that he is not interested in the detailed control of an input-output device which tends to come with direct machine language programming. (Also, even an experienced programmer has some hesitation in attempting to communicate directly with one of the teletype units presently attached to the 7094.)

Having used a subroutine for input-output in lesson six, lesson seven tells the student how to write his own general purpose subroutines. In this connection, really only the operation of the TSX instruction is described; no mention is made of transfer vectors or loading and linking procedures. This brings the student to the specific stopping point mentioned before.

Lesson eight, then, is not really a lesson but an epilogue, in which the student learns where he stands with respect to the massed knowledge of computer techniques. References are given in which he may find more detailed information about various aspects of computer usage, and he is given a send-off in the form of a set of several moderately difficult problems, one of which he is to pick out and solve and check out at the console.

As a help for a beginner, a brief reference description of FAP is included in an appendix, along with a description of eight commonly used pseudo-operations. The student is left to explore in the 7090 or 7094 reference manual.

The Subset Language.

The workbook and its reference appendix do not attempt to present all available 7090 instructions or FAP pseudo-operations; a carefully selected subset of these is used instead. Since this is the case, some discussion is in order regarding the method of selection of the particular subset.

First, with reference to the FAP pseudo-operations, the important consideration is simplicity of usage and expression. One or two examples of each of five important classes of pseudo-operations is described; these pseudo-operations just about provide the minimum repertoire needed for convenient programming, plus a few included primarily for pedagogical purposes. For example, the REM (arbitrary remark) pseudo-operation is not strictly needed in any program, as one can place comments to the right of every instruction. On the other hand, this pseudo-operation is representative of a large class of list control pseudo-operations, which the programmer should realize exist. (The negative impact of the console type-writer on the existence of assembly listings is not yet fully appreciated, and it could be argued that list control pseudo-operations could very well be completely left out.) Three data-generators are provided, for octal, decimal, and BCD formats, primarily to indicate that there exist a variety of means for getting information into a program without going through lengthy conversion procedures.

The SYN pseudo-operation is included to make clear the difference between naming a storage location and naming its contents, a mistake a beginner is particularly prone to make, especially after working with an algebraic language where the distinction is not of such importance.

The remaining pseudo-operations, BSS, END, and ENTRY, are included because they are needed in nearly all programs. The list stops here, though, as one can program for a long time without needing any more in the way of convenience from his assembly program.

In choosing instructions from the order list of the 7090/7094, emphasis was again placed on finding representatives of various classes of instructions. As an indication of the relative complexity of the instruction set provided, it is roughly the same as the instruction set of the much simplified I.B.M. 7040 computer (with index registers.)

Compatibility and Universality.

One of the objectives of this study was to produce a description of machine language programming which is to some extent as "machine-independent" as possible. While any great strides in this direction are very difficult, it may be noted that the specific ideas of a minimum subset language, completely symbolic format, and subroutine calls for input and output permit this text to be considered adequate for any of the following I.B.M. computers with virtually no modifications: the 704, 709, 7090, 7994, 7040, 7044; continued compatibility may be assured in future computers which are upward program compatible with these series. The primary reason this statement may be made, of course, is the assumption that input and output will be handled by subroutine calls, which removes the only really essential differences from the listed computers in terms of the stored program.

A further attempt to impart a certain amount of machine independence may be seen in the stressing of basic concepts which in fact carry over from one computer to another, and, it is hoped, from one generation to another. Ideas such as loops, branches, and program modification will turn up on any computer, and many of the techniques discussed are quite general use of the capability of the computer. One aim was to produce a skeleton of ideas, which could, for example, be turned into a workbook for some other computer with a minimum of text editing. One would like to have a skeleton text which could be fed into a text compiler along with the syntax table describing the particular computer configuration desired, and a text for that computer would result. While this workbook is certainly a long way from that goal, the thoughts involved did shape the course of its design.

In a very real sense the need for this text is temporary; both from the point of view that it describes this generation of computers, not the next; and that it fills a gap in this generation of languages which may not exist in the next. Yet the basic bit-manipulation capabilities of a digital computer will remain and it is their description (here framed in terms of machine language) that is the real subject of the workbook.

Appendix: Review of four recent programming texts.

In this appendix four textbooks in the programming field which have been published within the last two years are critically reviewed as possible texts for the machine language programming section of a programming techniques course. The primary subject of discussion in each of the reviews is the content of the book, rather than the presentation.

Programming and Utilizing Digital Computers, by Robert S. Ledley (National Biomedical Research Foundation and Johns Hopkins University) McGraw -Hill, New York, 1962

The best description of this book is "all-inclusive, but still very good." It takes a beginner from zero knowledge, through almost everything he might want to know about modern digital computer usage. Covered are algebraic languages (ALGOL), data processing languages (COBOL), machine languages (no particular language, except in appendix; four, three, two, and one-address schemes are discussed without a specific computer, but with examples.) In addition, the last third of the book, entitled "data-processing techniques" covers numerical analysis, boolean algebra, searching and sorting methods, coding and decoding. A real bonus is the discussion of methods of translation of algebraic languages into machine languages, and a complete description of a syntax directed compiler. A brief survey is given of some of the techniques of heuristic programming and an introduction to the general problem solver of Newell, Shaw, and Simon. Extensive examples are given of each of the subjects discussed, and each chapter is followed by exercises, problems, and an excellent list of references.

The text is extremely lucid and a bright college student should have not any trouble assimilating virtually everything discussed. (Certain descriptions of heuristic techniques move a little fast.) An interesting sidelight is that many of the examples of computer usage are medically oriented, reflecting the author's connection with and interest in such things. The best thing about all the examples used is that they offer real illustrations of problems best done with the aid of the computer; no simple but quite useless examples appear anyplace in the text.

An Introduction to Digital Computing, by Bruce W. Arden (University of Michigan) Addison-Wesley, 1963 389 pages.

This book is a comprehensive introduction to the aspects of digital computers most likely to be needed by a beginner planning to use the computer in engineering work. It begins with a brief introduction to computing ideas, with appeal to Turing and Wang machines. From here it rapidly develops the MAD language, and gives examples of its use.

The rest of the book concerns itself with techniques, and illustrates each idea with programs in the MAD language. About half of the book is devoted to numerical techniques. The last two chapters are devoted to non-numerical techniques, searching and sorting, etc., and description of a simple syntax directed compiler. (The compiler is simpler than the one discussed by Ledley.)

The radix sort is illustrated in a quite difficult to follow MAD program which is specially adapted to sorting numbers rather than symbols; its efficient extension to arbitrary symbols is not immediately clear. Arden does not attempt at any time to utilize the full bit-manipulating capability of the computer (his discussion of the machine language is minimal) and in fact does not indicate which problems are likely candidates for a machine language representation. (Except for a special note about the radix sort.) He does point out that programs which are not efficient in MAD are still useful for their lack of ambiguity in communication of an algorithm.

Arden enjoys using push-down lists and many programs are written with their aid which normally are seen with conventional arrays. He discusses recursive procedures extensively (without any indication of the traps laying wait in the MAD language) but gives a slightly vague definition of a "recursive prodedure" which might well leave a student slightly adrift.

The appendices include a brief reference manual for the MAD language (apparently included so that students will not have to purchase a MAD manual in addition to their textbook) and a description of the University of Michigan operating system which would be of interest only to a University of Mich. student or someone interested in an example of an operating system.

The book concludes with a set of very good problems for each of the chapters. Many of the problems require some real thought on the part of a student, and can aid in the instruction process.

In general, the book is well written, easy to understand and the examples chosen are interesting and useful. It appears (from the description of course MATH 373) that the text is ideally suited to the particular course taught at the University of Michigan, but that it does not cover as broad a range of subject matter as might be wanted in a general purpose reference on computer techniques. On the other hand, as a reference for numerical techniques only, it does very well.

Computer Programming Fundamentals, by Herbert D. Leeds and Gerald M. Weinberg. McGraw-Hill, New York, 1961.

This is a book on machine language, from beginning to end. It is a little hard to understand how a book which claims to introduce the reader to the complete range of computer uses can not mention anyplace the use of automatic programming languages. Rather, the authors introduce the student to such things as input-output, subroutines, fixed and floating point arithmetic with all the attendant detail and difficulty of the 7090 machine language without ever indicating that people have found ways of largely simplifying each of these coding tasks by using algebraic compilers. On the other hand, the examples done nowhere require the full bit-processing capability of the computer. If you wish to ask the question "why should I learn machine language", do not expect to find the answer here, as every problem solved could be done almost equally well in MAD or FORTRAN.

Apart from the examples used, and the attempt to cover an extensive amount of material all from the machine language context, the actual description of the language is reasonably interesting, punctuated with numerous amusing analogies and quotations from such sources as Alice in Wonderland. Much to the book's credit is the lucid section on traps and trap programming.

The authors seem to have one bone to pick and they do so extensively. Several pages are devoted to the mechanics of subroutines (apart from the concept of a subroutine) and such things as the importance of subroutine writeups and usefulness of the subroutines to other programmers. Pages spent complaining about the large number of bad subroutines in subroutine libraries could better be spent describing one virtue of subroutines completely ignored by the authors: the ability to segment and work on small parts of a big problem; perhaps creating subroutines which are only used once in the course of solution of a problem. In the authors' view, the only reason for subroutines is to allow one person to use another's work. (This attitude is no doubt shaped by the choice of the SOS language as a vehicle for the book.)

Programming and Coding Digital Computers, by Philip M Sherman. John Wiley and Sons Inc., New York, 1963. (Mr. Sherman is from the Bell Telephone Laboratories.)

This book inspires considerable comment and criticism. It has a number of minor faults and one larger difference of opinion with this writer. However, on the whole, it treats its subject quite well, and it particularly shines in part three, where techniques are discussed.

The major difference in opinion alluded to is the emphasis on the language of the machine as the beginning and end of all computer programming. Use of a compiler is considered as a "technique" with the same emphasis as use of macro-instructions or debugging. The point of view seems to be that some people will have a use for this technique . . . This viewpoint may be considered controversial in some quarters. In any case, it requires that he discuss items such as loops and branching, push-down lists, input-output, switches, and subroutines from the point of view of a machine language programmer, with all the attendant detail of a machine language program in each of the examples accompanying the explanation.

The unfortunate part of the attendant detail is the use of a hypothetical computer, too similar to a 7090, and a hypothetical assembly program, too similar to BEFAP. Even though it is intended that the book be provided with a supplement for each particular machine to which it may be applied, one suspects that a beginner will find the approach confusing. (For example, Sherman's hypothetical computer has the Multiplier register to the left of the AC, for long shifting purposes. When the student attempts to transfer his knowledge to the 7090, with very similar instructions and layout, except that the MQ is to the right of the AC, he may throw up his hands in despair.)

Some other minor faults noted include a lack of perspective and importance of the various items under discussion. For example, in a diagram of the assembly-execution process, the operation of putting cards on tape is given as much (apparent) emphasis as the assembly of the symbolic program. The relative importance of these operations is not emphasized either in the diagram, or in the text, where a lack of redundancy makes it difficult for a beginner to discern the relative importance of all the things discussed.

In places, the discussion tends to be slightly dogmatic, in the sense that no alternatives are mentioned. (e.g., the statement: "When a monitor is used, it remains in memory at all times." (p. 202.) No hint is dropped that perhaps some monitors, somewhere, do not remain in memory at all times.)

A very minor criticism is the terminology for one of his monitor control cards. The "load assembler card" causes loading of the assembled object program, not the assembly program as its name might suggest. Since Sherman has modified the 7090 instruction set to eliminate some of the difficulties in learning it, as well as the BEFAP language, he might as well attempt to make the monitor control cards for his "hypothetical" monitor system as mnemonic as possible.

On the plus side is part three of the volume, covering a variety of techniques. These techniques have been avoided in beginning books, until recently (e.g. Arden or Ledley) and do have a place in a full volume. Similarly, an excellent discussion of the assembly program accompanies the description of symbolic coding. Another interesting highlight is a loader program example. With these examples the beginner ought not be afraid about getting into or using a "system program." Examples discussed in section three include interpreters, list structures, searching and sorting methods, and data processing techniques for business.

Two omissions are notable in the list of techniques covered. First is the complete lack of discussion of overlapped input and output techniques. This could be excused by noting that the beginning programmer will not be overly interested in this level of sophistication. However, a more important omission for an otherwise comprehensive book is that of traps of any kind, except for a meta-bit instruction trap which is brought in for debugging purposes. The problem of accumulator overflow and the requirement that checks be made is mentioned, without pointing out that a floating point trap can relieve the programmer of much coding. Since overlapped input-output is not discussed, data channel traps are not needed, of course, but the ability to handle remote direct data with the aid of a trap is not mentioned. Similarly, an interval timer could have found much application in the book, particularly under the extensive discussion of debugging procedures. Trapping techniques are of sufficient interest to merit a larger inclusion in a book which devotes a third of its space to techniques.

Interesting also is that one out of ten practice problems were quite subjective; much more so that the book; coupled with an occasionally dogmatic point of view, subjective problems seem out of place or possibly misleading. Sample: "What characteristics are required of a good programmer?" after discussion of sequencing (p. 180.)

It is not clear that the author has ever taught a class or that the text has derived from or been used in a class, although it is intended to be a textbook.

<u>Bibliography</u> (See also appendix)

Corbató, F.J. et al., <u>The Compatible Time Sharing System, a Programmer's Guide</u>, M.I.T. Press, Cambridge, 1963.

Galler, Bernard A., <u>The Language of Computers</u>, Mcgraw-Hill, New York, 1962.

McCracken, Daniel D., <u>A Guide to ALGOL Programming</u>, John Wiley and Sons, Inc., New York, 1962.

Organick, Elliot I., <u>A Computer Primer for the MAD Language</u>, Cushing-Malloy Inc., Ann Arbor, Michigan, May, 1962.

University of Michigan Staff, <u>Final Report: The Use of Computers in Engineering Education</u>, alloy Litho. Co., Inc., Ann Arbor, Michigan, 1963.

PART II

SYMBOLIC MACHINE LANGUAGE PROGRAMMING

FOR THE I.B.M. 7090/7094

COMPUTER

A Laboratory Workbook

TABLE OF CONTENTS

FORWARD

How to use this workbook.

The reader will find that this workbook is somewhat unlike other texts on computer programming for several reasons. The material is presented from a different point of view, which it is hoped will be slightly more machine independent than earlier texts, even though it covers machine language and the I.B.M. 7090/7094 computer. The language developed, while not the complete language of the 7090/7094 is adequate to write correct, efficient programs, and represents most of the types of facilities available.

A more important departure is that practice in the laboratory is considered an integral part of study of this workbook, and each lesson is followed by a problem which is to be solved by writing a computer program and testing it.

The development which makes both these departures possible is a large-scale time-sharing system for the 7090/7094 computer. With reference to the first the closer interaction between the user and the computer permits sophisticated support programs to give the user a completely symbolic view of the programs he is running and testing, whether they be algebraic language or machine language.

Secondly, the ability to debug a program completely and rapidly from the computer console, again with the aid of sophisticated support programs, permits us to take the attitude that each of the assigned programs is of the nature of an experiment (not unlike a chemistry laboratory) which is to be prepared, then carried out at the time-shared computer console.

It is expected that the work-book will be studied as follows: the reader should study carefully the material in one lesson. After spending perhaps three to four hours gaining a thorough understanding of the concepts presented and examples discussed, he should then attempt to write out a solution to the assigned problem. With proposed solution in hand, he should sign up for one-half to one hour of console time. (On early lessons, an hour is recommended, until familiarity with the console is attained.) During the period at the typewriter console, he should type in his solution program, and have it tested by one of the supporting testing programs. If it fails, he can run it under control of a symbolic debugging program to help find the difficulty. He will be able then to watch his program in action, and thus easily discover where he has made an error. The reader is certain to have some questions about the material, even after careful, thoughtful study, and he should jot them down for later referal to reference manuals or to experienced programmers.

Mechanics of reader-computer interaction.

Each of the chapters in the workbook ends with a problem to be solved by writing and debugging a computer program in the machine language at a time-sharing console. The reader will notice, however, that no detailed information is given as to the use of this console. The reason for this omission is that the detailed characteristics of the consoles and the support programs which enable their efficient use are presently and will for some time in the future be the subject of much experimentation. The reader should, therefore, write out his solutions on a sheet of paper in a format similar to the examples given in the text, and later obtain information on how the detailed interaction with the computer console is to take place. It is hoped that this information will be available in the form of a brief memo; in any case the information can be obtained from the book "The Compatible Time-Sharing System, A Programmer's guide" by F.J. Corbató, et al.

What is covered.

This workbook discusses only the symbolic assembly language of the I.B.M. 7090/7094 computer. The actual language used is that acceptable as input to the FØRTRAN Assembly Program (FAP), although that language is little different than other assembly languages for the 7090. Only a subset of the available computer instructions is discussed, with the intention of introducing the reader to all important aspects and ideas of machine language programming, but leaving him to "window-shop" through the computer instruction manual if he wishes to pick up the details of more instructions and facilities of the computer.

Similarly, only certain aspects of the assembly program are covered, and the reader is again left to explore, although this time with the warning that sophisticated assembly programs are difficult to use without a good knowledge of the internal workings and techniques of the assembly program itself. (This statement is not so true of the 7090/7094 computer.)

While written primarily for the computer user who has experience with algebraic languages and wishes to learn a little more about his computer so he can tackle a more sophisticated program, the material is complete and with a technical introduction which will require some extra study by a rank beginner.

The presentation is intended to contrast sharply with many available texts which discuss machine language programming with the point of view that it is the only usable or practical form of communication with a computer. Such a viewpoint normally forces a text to treat all the varied applications and techniques of computer usage from the highly detailed viewpoint of the machine language.

Instead, it is hoped that the reader will tend more toward the opinion that the preferred method of communication with a computer is the simplest one available for a job, and that for many routine tasks the algebraic or other highly symbolic languages should perform very well. The machine language is left only for those problems which would run into limitations of time or space because of inadequate freedom of expression or utility of the basic machine operations provided by present forms of more highly symbolic languages.

The entire workbook is written from this viewpoint, and the emphasis will be observed in the lack, for example, of numberical problems. Most of the problems discusses and examples given would be difficult to perform with any efficiency without the ability to describe the operations in terms of the machine's own orders.

<u>How far does it go?</u>

The workbook stops with a description of a subset of the 7090/7094 language. The material covered is complete only in the sense of an introduction. Virtually no techniques are discussed, except as necessary to illustrate the language description. The reader will probably wish to use many facilities of the computer described only in its reference manual. The intent of this workbook is to bring the reader to a level where he may continue to learn on his own those techniques applicable to his particular problem.

The reader should be able to write subprograms in the FAP symbolic language with little difficulty when he is finished, but he will know little or nothing about the actual binary language into which his program is translated, or, for that matter, of the intermediate (BSS) language which his programs go through on the way to becoming complete programs.

The study of the characteristics (but not the details) of this intermediate language and the final binary form would be the next logical step for the reader wishing to acquire a complete understanding of the detailed techniques of computer usage.

INTRODUCTION TO MACHINE LANGUAGE

The Computer.

A computer, basically, is an electronic device which follows instructions. These instructions are provided by a programmer, and are done (executed) by the computer one at a time and one after another, in sequence. The programmer, then, must learn to express himself in the available instruction set of the computer, the machine's language.

A characteristic of the language of present-day computers is that the instruction set is quite primitive in terms of the amount of work accomplished by each instruction, and a fairly simple job may require many dozens or hundreds of machine language instructions to accomplish. On the other hand, the basic nature of each instruction permits extreme flexibility of expression.

The instructions act upon pieces of data in the form of patterns of binary digits*. The instructions and the

---

* A binary digit (abbreviated "bit") is one of the two numbers "0" or "1", and is used to represent one of the two states of a device within the computer. If three devices within the computer could be described as "on",

data, both, are stored as patterns of binary digits in the computer memory, a bank of 32,768 words, in which each word has room for one instruction, or one 36 digit binary number, or one (approximately) ten digit decimal number. (See figure 1.) The instructions, as patterns of bits, may also be interpreted as numbers, and by looking at the number stored in one of the 32,768 locations it is not possible to tell whether it is an instruction or a data word. The computer cannot tell, either, and it is part of the programmer's job to write a program of instructions which does not instruct the computer to interpret a data word as an instruction unless that is specifically intended.

An important consequence of having the instructions stored in the memory of the computer in a form indistinguishable from data words is that the program may act on other instructions, as if they were data, perhaps modifying them before asking the computer to execute them as instructions. This modification of an instruction by the program itself is called program modification, and is one of the sources of the great flexibility and ability of a computer programmed in the machine language.

"on", and "off", a description of this situation in terms of binary digits would be "110", the "1" representing the state "on" and the "0" the state "off".

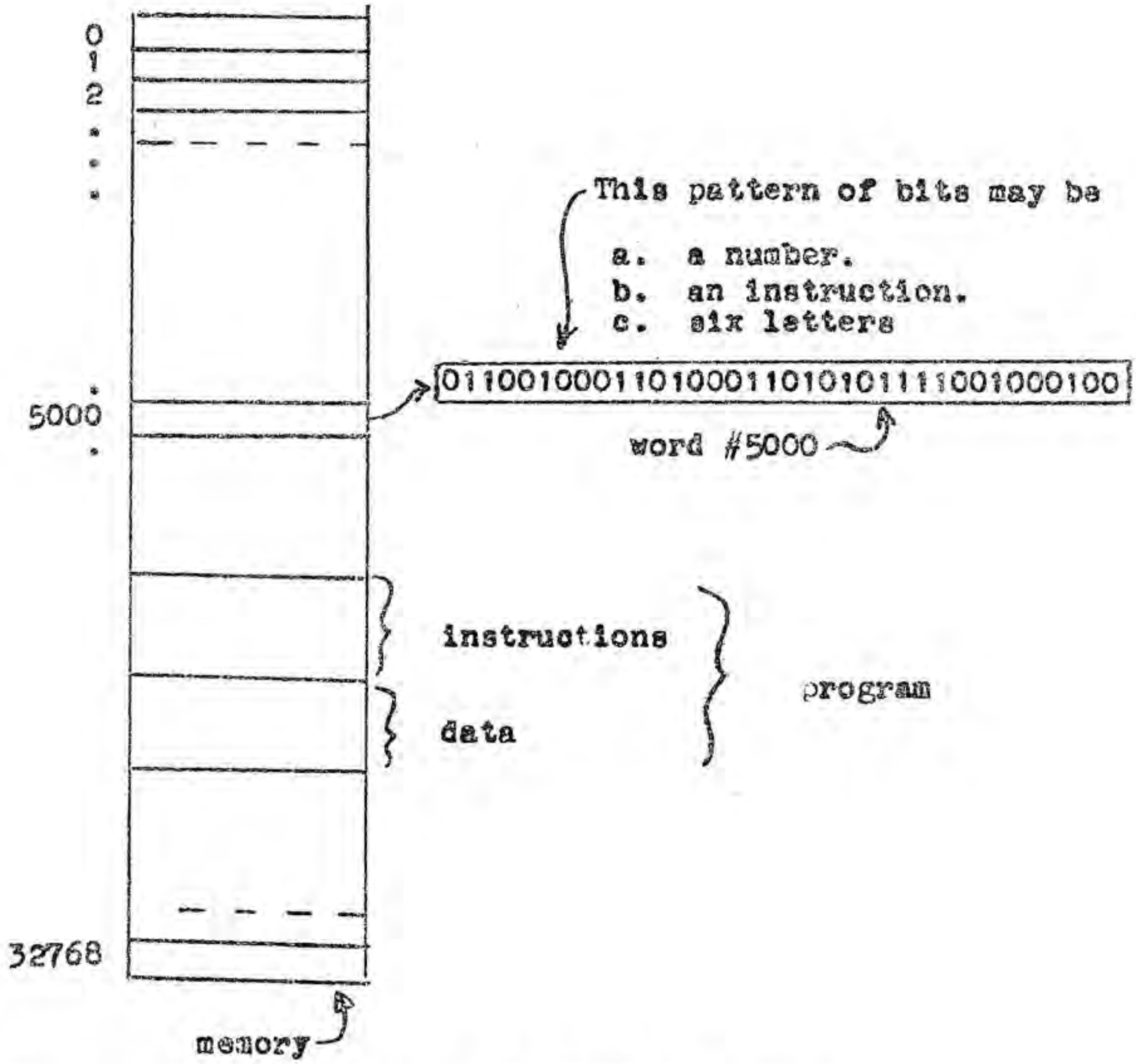

Figure 1. Visualization of 7090/7094 memory.

Program modification may range from inserting an address into a single instruction through making extensive modifications to several instruction bit patters, up to the extreme example of completely writing a computer program, which is what a compiler program does.

The locations of words in the memory are internally numbered, and words may be referred to by their location numbers, but this is rarely done when programming in machine language. Instead, to avoid commitments about which location an instruction should be placed into, the location of a piece of data or an instruction is given a <u>name</u> and the piece of data or instruction may be referred to by giving the name of its location.

Then, if a new instruction is later inserted between two others, the locations of later instructions or data may change, but the names, by which they are obtained, remain invariant.

In a computer, most instructions make reference not to a piece of data directly, but to the location of the piece of data. For example, the instruction to add 10 to a number would be

```
        ADD     CØNST
```

and elsewhere in the program would be the data word:

```
    CØNST  DEC     10
```

This illustration shows how instructions or data locations are named, and how reference is made to the names. (Note that the letters DEC simply indicate that the number 10 is to be interpreted as a decimal integer and not, for example, as a binary number.)

Since we do not specify the location of each instruction in terms of a location number, we must make the convention that instructions and data words are sequentially assigned to consecutive memory locations in the order they are found in the program.

Instruction sequencing: Loops.

Normally, after executing the instruction in one memory location, the computer goes to the next location in memory to obtain its next instruction. Thus a sequence of instructions written one after another will be executed in the order written. This normal sequencing may be interrupted, however, if one of the instructions is an instruction for the computer to take its next instruction from a new location. Such an instruction is known as a transfer instruction.

With the aid of the transfer instruction, we can write a loop, a sequence of instructions which performs some pattern of operations, followed by an instruction to transfer back to the first instruction in the loop. Coupled with program modification, to allow a minor change to the instructions within the loop each time through, the programmer has an extremely powerful tool in his hands. As an example, consider the following hypothetical English language program (since we do not yet know any other language.)

location name	instruction
	insert the location name DATA in the blank part (address) of the instruction in location GET.
GET	get the number in location ____.
	.. (here would be instructions
	.. to do something with the
	.. piece of data obtained.)
	increase the address of the instruction in location GET by one.
	transfer control to location GET.

In this example, the first instruction changes the instruction in location GET to read

get the number in location <u>DATA</u>.

After doing the desired operations with this piece of data, the next to last instruction says to change instruction in location GET to read

get the number in location <u>DATA+1</u>.

which will cause a different piece of data to be operated on. The transfer instruction then causes the loop to be repeated. The piece of data in location DATA+1 will be processed on the second "pass" through the loop, and the loop will be set up to work on location DATA+2 on the third pass.

Thus, by writing just a few instructions, and supplying several data words, the computer will perform substantially the same computation many times; it may perform many thousands of instructions even though the programmer wrote only a dozen. Virtually every computer program written makes use of at least one loop, and usually many, for the ability to loop is perhaps one of the most important abilities of a computer.

An observant reader should have noted that the program example given has one difficulty: it goes on forever and ever, processing successive pieces of data. While the operator could turn off the computer after it has done all the useful work necessary, a more flexible and again very powerful capability is added to the repertoire of the computer's instructions. We can write in place of the last instruction which was

> transfer control to location GET.

the <u>conditional</u> instruction

> if the address of GET is less than DATA+100, transfer to GET.

This instruction has the all-important word "if" attached to it; the computer has been effectively given the ability and authority to make a decision as to whether or not to repeat the loop. This decision is based on the contents of the address of the instruction in location GET in our particular example, but remember that this address is calculated by the computer program. Thus a program can direct its own course, depending on the results of previous computations. (See figure 2.) A properly written program may actually control its own course; it may proceed with an iteration (loop) until some intermediate result is smaller than a certain value; or on the basis of some piece of data decide to use one of two different methods of solution for an arithmetic problem. The ability to loop, and to use a conditional branch are probably the most important properties of a digital computer, and virtually all programs take advantage of both of these properties.

An important difference between the methods by which a human solves a problem by hand, and with the aid of a computer becomes evident when loops and conditional branches are considered. Consider, for example, the way in which a differential equation may be numerically integrated. Performing this calculation by hand requires that each step of each iteration be carefully thought out. After performing one iteration, it is necessary to think out and grind out the details of the second. Also, when the next numerical integration is encountered, the entire process must be again thought out and carried through explicitly, step by step.

On the other hand, if a computer is available, the programmer need work out the routine only <u>once</u>, and write

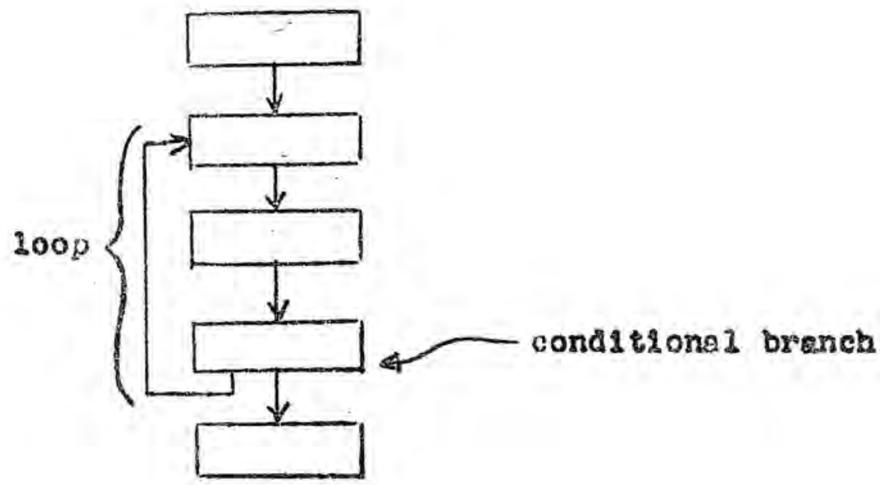

Figure 2. The loop and the conditional branch.
(each box represents one instruction.)

a program to perform that routine. His program involves an iteration, which again he only need work out once, and make into a loop. Thus the programmer has in a sense a strong amplification of his efforts in being able to program loops and also to use a program again, for a different set of data.

Symbolic programming.

A conflict should be apparent if the preceding paragraphs are read carefully. First, it was said that instructions for the computer are stored in the computer memory as patterns of binary digits. Instead, the actual instruction would look more like:

0001000000000000000000000000101110

assuming that the number ten were stored in location 101110. It is clear that a programmer would prefer to write instructions in the former language, rather than as binary numbers. If he is allowed to do so, however, at some point his symbolic instructions must be translated into the actual patterns of binary digits required by the computer. Similarly his data words such as

CØNST DEC 10

must be turned into patterns of binary digits which are properly interpreted by the computer.

Since the translation into the binary language of the computer is clearly a (difficult but) precise and well defined task, a computer program can do the job. This so-called <u>assembly</u> program reads the symbolic cards and translates them into the correct patterns of binary digits. Then, and only after a successful translation, can the resulting binary program be run on the computer.

An assembly program belongs to the class of programs known as system programs, that is, it is a program commonly used to aid in operating or programming the computer. Its purpose is to take as input a shorthand symbolic notation for a machine language program, and produce as output the binary machine language program for which the symbolic notation was a shorthand. (Note the similarity between figure 3 and figure 4.)

In the early days of computers, assembly programs were not available, and programmers had to write out long strings of numbers to represent the instructions they were using. Since a string of numbers has very little mnemonic value for most people, the programmers of those days actually invented mnemonic names for their instructions and programmed in terms of these symbols. When they had completely written the program and were satisfied with it, they then rewrote it in terms of the binary numbers required by the computer, and these binary numbers were punched into paper tape or cards so that they could be read into the computer.

The assembly program, then, takes over this tedious, errorprone second step of writing a program. It reads in

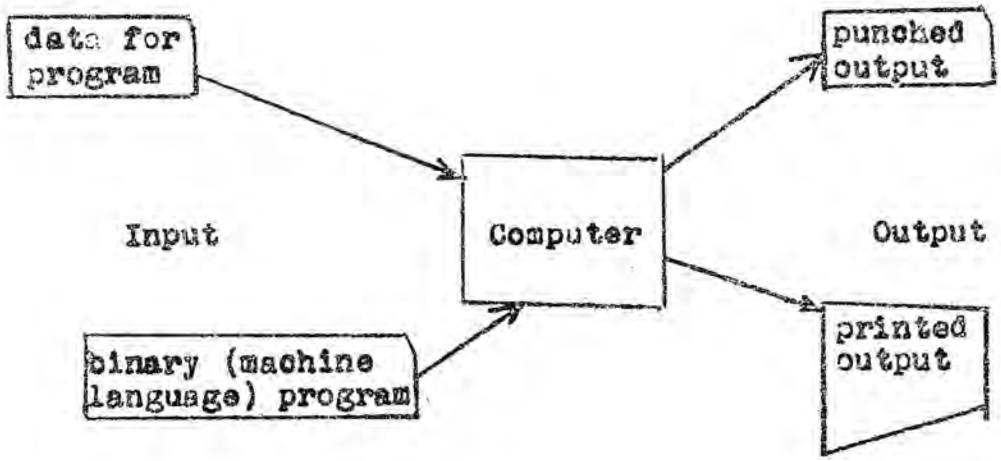

Figure 3. General computer use.

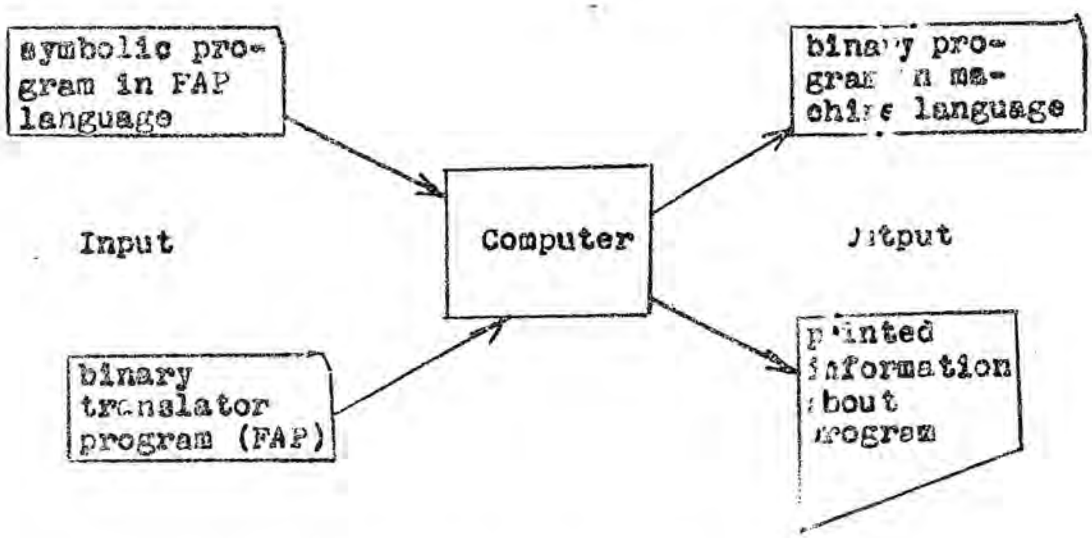

Figure 4. Use of computer for assembly.

symbols which have a mnemonic value to the programmer, and translates them into the binary machine instructions required by the computer. Also, once an assembler is available it can do other things beyond the simple substitution of binary machine codes for symbolic mnemonics.

The programmer can also leave the problem of assigning memory locations to the assembly program. The programmer simply writes his instructions down, one after the other in the order he desires them. Similarly he places his data in the program in the desired order with respect to the instructions. When an instruction is to make reference to a piece of data (or another instruction) the programmer (since he now has no idea what the location of the piece of data will be) invents a symbol, and <u>names</u> the location of the piece of data with this symbol. He then uses this symbol as the address of the instruction which is making reference to the piece of data. The <u>value</u> of the symbol is unknown to the programmer, and it will remain unknown until the assembler begins working on the program.

The assembly program, then, is given the additional task of assigning each of the instructions and pieces of data to a memory location, thereby establishing the values of the symbols which may appear as names of the locations of various instructions and data. Then it may proceed with the process of replacing the operation mnemonic with the correct binary machine code, and it may evaluate the address of each instruction in terms of the values of the symbols it has previously established.

One more special feature of assembly programs will complete our discussion of them. The assembly program can, while in the process of looking up the proper binary machine code for each of the machine instruction mnemonics check for certain special mnemonics intended to convey information to the assembler itself, rather than to be translated into a binary machine instruction. For example, the programmer may type the letters END into the operation field of an instruction and make this "pseudo-instruction" the last one in his program deck. The assembler then will examine each instruction it processes, to check for this special one. When it finds the END pseudo-instruction the assembler knows that there are no more instructions to follow in the program. The END pseudo-instruction itself does not cause any binary instructions to generated in the object program, it simply acts as a "note" to the assembler.

END is but one example of a pseudo-operation mnemonic. Seven other pseudo-operations are described in the reference appendix; their effects on the assembly process are noted there. Some of these pseudo-operations do cause the generation of words in the assembled program in some special format; others more similar to the END card are simply notes to the assembler on some particular aspect of the assembly.

We have seen, then, that the assembly program does three jobs for the programmer. First, it assigns his instructions locations in storage and defines symbols he has used. Second, using these symbol definitions and a standard table of operation mnemonics, it translates each of his symbolic instructions into binary machine instructions and punches them out on cards in a format suitable for reading directly into the computer. Third, it looks for and recognizes several pseudo-operation codes which appear in the operation field, and considers these to be special notes to itself from the programmer; its operation is modified accordingly.

Now that we know what operations an assembly program is expected to perform, we may proceed with a discussion of how programs are written in a form suitable for FAP translation. In what follows we will talk only about the symbolic language which the FORTRAN Assembly Program is prepared to accept and translate. We must remember at all times, however, that the instructions and data words we write must be ultimately translated by FAP into binary digit patterns before the computer can interpret them as instructions or data.

Introductory exercise.

This introductory exercise is intended to allow the reader to see some of the ideas mentioned in the last few pages in action, and to gain practice in the use of the time-sharing console.

A program in the FAP language has been written and is available for the reader to assemble and run. The following steps should be taken:*

1. Print out the instructions to see a typical FAP language program. Save this printout, as it contains examples of the things which will be discussed in later chapters.
2. Assemble the FAP language program with the aid of the FAP translator. This will produce a binary machine language version of the symbolic program.
3. Now, run this program on the coputer.

---

* See note on mechanics of console usage at the end of the Forward.

LESSON ONE: SHIFTING AND COUNTING

Registers.

Most machine instructions act to change in some way or manipulate the pattern of bits stored in one of the registers of the computer. Some instructions merely test the number, others cause addition to it, while still others may cause the number to be stored in one of the 32,768 storage locations for later use.

The simplest register in the computer is the index (sometimes called a counting or tally) register. As its names imply, there are special instructions to make counting a natural job for this register. For example, it may be desired to go through a loop a certain number of times. We can put that number into an index register with the AXT instruction; for example

```
        AXT     15,4            (Address to Index True)
```

puts 15 into index register four. (On the 7090 there are three* identical index registers, numbered one, two, and four.) We may then count down by one each time we go through the loop, watching for the index register to reach

---

* On a 7094 computer there are seven index registers, numbered one through seven.

zero. When it does, we know that the loop has been done the required number of times. Since this counting down and testing operation is associated so often with loops, a special instruction is available which both counts down, and tests, and in addition causes the computer to transfer back to the beginning of the loop. This instruction is TIX; it is used as follows:

TIX START,4,1 (Transfer If Index)

The four indicates which index register is being counted; the one indicates the amount of the count. (Sometimes it is more desirable to count by two's, or three's, etc.) START is the name of the location containing the first instruction of the loop. By making the TIX the last instruction of the loop, the computer will execute each of the instructions of the loop, lower the index register, and return to do the instructions of the loop again.

It was stated that the TIX instruction would also perform a test; how is this done? When executing the TIX instruction, the computer first checks to see whether lowering the index register would cause the register to contain zero or a negative number. If it would, the register is unchanged; and the transfer is not executed, instead the next instruction after the TIX is taken. We have, therefore, a form of a conditional branch; depending on the number in the index register, one of two possible paths of program are taken, one being a repetition of the loop, the other being the next instruction after the loop.

A brief reference was made above to the name of a location, giving that term to the symbol START. It would be well now to stop a minute and examine the special role of symbols in our language.

Symbols.

In the FAP language, a symbol may stand for one of two things:

1. The name of a location.

2. The name of the distance between two locations.

The second use of a symbol is only needed in more sophisticated applications, so we will primarily be concerned with symbols as they stand for the names of locations. There is no way of attaching a name to the contents of a storage location, but since there are no instructions available in present computers which would make such a naming desirable, it is not a serious limitation. Of course, a storage location may contain as a piece of data the name of another storage location.

The convention is made that if the first of two consecutive locations is named (say, NAME) the second one may be referred to by an expression such as

NAME+1

Similar expressions can be used to refer to instructions or data words farther from the named instruction, although care must be used if instructions are inserted in the program later. It is good programming practice to always allow for future addition of instructions to a program; therefore such expressions should be limited to cases where the locations in question are related by context, as two consecutive memory locations containing related pieces of data. Since the location names will be translated into numbers later, and since successive instructions and data will occupy successively numbered locations in core storage, one may also use the algebraic addition instructions of the computer to perform address arithmetic at execution time.

Operands.

A characteristic of most instructions of the 7090/7094 computer is that they specify a single operand by giving a location in memory in which that operand may be found. For example the instruction

```
        ADD     CØNST
```

specifies that the single operand may be found in the location which we have named CØNST.

Since most arithmetic operations require two operands, we must ask where is the other one, and this question brings us to another component of the computer, the accumulator register (AC). The accumulator is used to hold the other operand in addition, and also to contain the result of the addition. Its name follows from its ability to accumulate results of addition and subtraction operations. The special properties of the accumulator register do not stop with its ability to act as an aid to arithmetic. In addition, one may shift patterns of bits right or left within the accumulator, and in or out of the accumulator into the MQ register, which may for this purpose be considered to be a 36 bit right extension of the AC.

This shifting of bit patterns around gives rise to considerable bit manipulating abilities in a computer, and since such bit manipulation is difficult to specify conveniently and efficiently in current algebraic languages, it is often done with machine language programs.

The AC is a 38-bit register. One of these bits, named "s", is placed away from the register and does not usually take part in shifting operations; it contains a bit giving the sign of the addend in addition. The other 37 bits are named (and numbered) "p", "q", and 1-35. See figure 5. The shifting instructions of the AC cause the bit pattern (except for bit "s") to move to the right or left, depending on the particular instruction. For example,

ARS   6   (AC Right Shift)

causes the pattern of bits in the AC to be shifted to the right six places. The bit in position 1 will now be in position 7; the bit in position "p" will be in position 5.

figure 5. The Accumulator register. (AC)

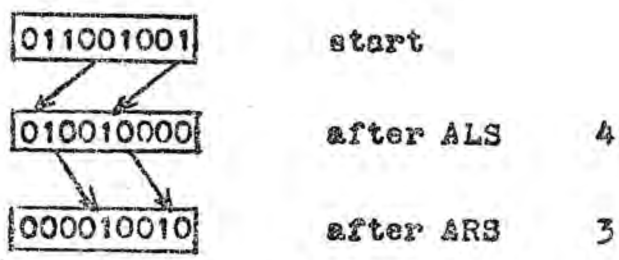

figure 6. Effect of shifting on the bit pattern in a nine-bit register.

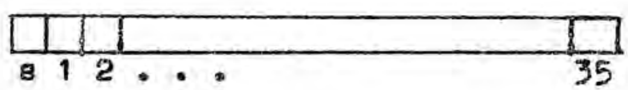

figure 7. The MQ register.

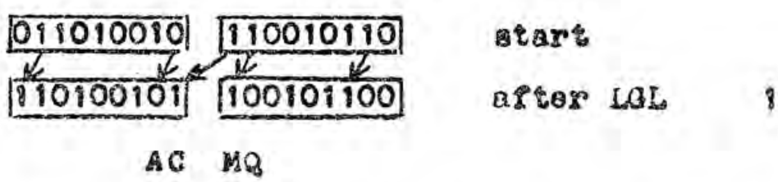

figure 8. Shifting between the AC and MQ register.

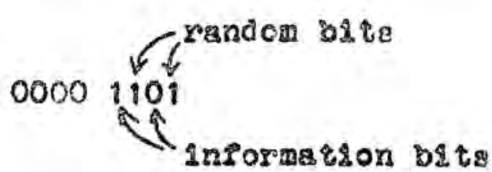

figure 9. Coded information in 4-bit AC and MQ.

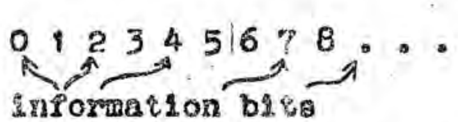

figure 10. More complex information coding.

etc. On the left end, the first six positions will be filled in with zeros, on the right end of the AC some bits may have been shifted beyond position 35; they have been lost forever. (See figure 6.) Similarly, the left shift instruction

ALS 10 (AC Left Shift)

causes the pattern to shift to the left ten positions.

The MQ register can be lined up to the right of the AC for purposes of some shifting instructions, and the two registers considered as one long register. The MQ contains 36 bits, the same as a word in memory, and these bits are numbered "s", 1-35, as in figure 7. In the MQ, the "s" bit does take part in shifting. The instruction

LGL 4 (Logical Left Shift)

will cause the AC and MQ, considered as one long register, to be shifted together left four places. Bits leaving position "s" of the MQ enter position 35 of the AC. There is a similar right shift instruction, LGR. (See figure 8.)

The MQ has a special shifting instruction, called a rotate instruction, in which bits leaving the left end of the MQ re-enter the right end, making the MQ a sort of a circular register. Considering the bit "s" and 35 to be next to each other, the instruction

RQL 7 (Rotate MQ Left)

would cause such a seven-bit rotation to take place.

A program.

We can now write a program which uses these instructions. Consider a code word problem; the word in the MQ register contains information, but the information has been encoded in the following fashion: 18 information-carrying bits were placed in consecutive even numbered positions of the MQ: the odd numbered positions are filled in with random ones and zeros. The problem in decoding such a word is to separate the information carrying bits from the random bits. Let us suppose we desire to have the answer end up in the AC register. The general procedure will be to shift one bit into the AC (an even numbered bit) and then rotate the next odd numbered bit out of the way. When we have gone through this procedure (loop) 18 times the entire 18 information bits will have been pushed into the AC, and the random bits will remain in the MQ in their original positions, in fact.)

We will assume that the AC is filled with zeros when we start. The sequence which shifts one information bit into the AC and skips one random bit is

```
        LGL     1           get even numbered bit
        RQL     1           skip odd numbered bit
```

We wish to write a loop about these instructions, so let us use index register two to count our loop. The program then looks like:

```
         AXT     18,2          set to loop 18 times
SHIFT    LGL     1
         RQL     1
         TIX     SHIFT,2,1     test and lower index
```

Note that the location of the LGL instruction has been given a name, SHIFT, so that the TIX instruction will have a place to transfer control back to. Our program will run through the loop 18 times, causing the correct decoding to take place. To illustrate, suppose we start with the four bit AC and MQ in figure 9. The code message is 10, the result of looking at only the even digits. (The first bit is bit zero.) We LGL once, giving

	0001 1010
then RQL	0001 0101
then LGL	0010 1010
then RQL	0010 0101

The result is the code message 10 in the AC, the random bits left in the MQ where they started.

Problem.

Write a program which decodes the message in the MQ, assuming it has been encoded as follows: The first six bits are coded as described above, even bits containing information and odd bits random numbers. The next six bits are encoded the other way; odd bits containing information and even bits containing junk, as in figure 10. The next six bits are coded like the first, and so on across the word. You may assume that the AC is empty when you start, and that the MQ contains the word to be decoded. Leave the result in the right half of the AC. (Hint: how would you have to change the program written before, if the odd-numbered bits had contained the information?)

LESSON TWO: SYMBOLS

The BCD code.

Although special languages, such as COMIT and LISP have been developed to allow description of symbol manipulation, these special purpose languages are not usually adaptable to doing a small part of a larger program written, say, in an algebraic language. When, as part of a larger problem, some symbol manipulation is required, a machine language program is often used. In addition, when large quantities of data are involved, efficient manipulation often requires at least some knowledge of the detailed structure of the symbols.

As has been mentioned several times before, the computer works with patterns of binary digits, not with letters or symbols. For a program to manipulate symbols, they must first be encoded into patterns of bits. A standard code, known as BCD (for binary-coded-decimal) is often used for the letters of the alphabet, and the special characters appearing in text, as well as the digits. This code uses a basic block of six binary digits for each character being decoded. For example, the letter "A" is encoded as 010001. Each of the 26 letters of the alphabet, the ten digits, and 12 special characters, including one for a blank space, are included in the BCD set. The entire BCD code is shown in figure 11.

A programmer does not usually need to know anything about the details of the code, except for perhaps two things about its overall structure. First, as can be seen from figure 11, the succeeding letters of the alphabet are encoded as successive larger binary numbers, if the bit patterns are interpreted as numbers. Thus, by sorting a symbolic table into numerical order, with algebraic instructions, it will then be in alphabetical order, when considered as a set of symbols.

The secnd piece of information needed is that the digit "zero" is encoded as the six bit pattern 000000. We can then predict how the zero portions of a word will be interpreted as BCD. Note that the "blank space" is considered a character distinct from the digit zero, with its own BCD code. Since six bits are used for each coded character, six such coded characters may be stored in one 36 bit computer word in storage.

Most input-output devices on the computer are designed to work with these BCD codes; if a word containing six BCD characters is sent to a printer, the printer will print a line consisting of those six characters. In fact, if any binary word is sent to a printer, the printer

symbol	BCD code	symbol	BCD code	symbol	BCD code
BLANK	110000	F	010110	V	110101
0	000000	G	010111	W	110110
1	000001	H	011000	X	110111
2	000010	I	011001	Y	111000
3	000011	J	100001	Z	111001
4	000100	K	100010	+	010000
5	000101	L	100011	-	100000
6	000110	M	100100	/	110001
7	000111	N	100101	=	001011
8	001000	Ø	100110	'	001100
9	001001	P	100111	.	011011
A	010001	Q	101000	)	011100
B	010010	R	101001	$	101011
C	010011	S	110010	*	101100
D	010100	T	110011	,	110011
E	010101	U	110100	(	111100

figure 11. The BCD code.

symbolic instruction		data pattern generated					
BCI	1,HELLØ,	011000	010101	100011	100011	100110	111011
required by syntax	data	H	E	L	L	Ø	,

figure 12. Operation of BCI pseudo-instruction.

will print the characters corresponding to the six six-bit patterns found in that word. Note that as always, the printer cannot tell for sure that the word sent it is supposed to be six characters, but it can always interpret the word that way, even if the programmer intended that particular word to represent a decimal constant, or a machine instruction.

One small difficulty should be resolved before we attempt to write any programs to move about arbitrary strings of bits considered as symbols. The AC register, it will be remembered, contains 38 bits, two more than a word in storage (see figure 5.) The "s", "p", and "q" bits must be considered individually. To make explicit which bits are affected by any particular instruction, we may use a special notation, in which the symbol C(X) means the contents of the register or memory location named X; and a subscript refers to certain bits of the register or memory location X. For example, the so-called "logical" accumulator is bits "p", and 1-35. We might denote the contents of the logical AC by

$$C(AC)_{p,1-35}$$

In describing the operation of an instruction which affects the AC one must always be careful to discern exactly which bits are affected. For example, there are two instructions to load the AC with a word from storage; these are CLA and

CAL. Their descriptions differ as follows:

CAL	X	$C(X)$ replace $C(AC)_{p,1-35}$
CLA	Y	$C(Y)$ replace $C(AC)_{s,1-35}$

In both cases, bits of the AC not mentioned are set to zeros.

Compare instructions.

The 7090/7094 instruction set includes instructions which allow the programmer to compare the word in the AC to a word in storage. Depending on whether the AC is larger, smaller, or identical to the word in storage, one of three alternate instructions is taken as the next step. One compare instruction is used as follows:

LAS Y (Compare Logical AC with Storage)

In the notation introduced above, if the $C(AC)_{p,1-35}$ are greater than the $C(Y)$, the computer takes the next instruction after the LAS instruction. If the $C(AC)_{p,1-35}$ are identical with the $C(Y)$, the computer skips the first instruction after the LAS, and takes the second. If the $C(AC)_{p,1-35}$ are smaller than the $C(Y)$, the computer skips the first instruction after the LAS, and the second one, taking the third. The instructions placed in the three locations after an LAS instruction are often transfer instructions which send the computer to the proper section of the program. (The instruction

TRA Y (Transfer to Y)

causes the computer to take its next instruction from the location named Y.)

Let us write a program using this compare instruction. Suppose that the MQ contains six BCD characters, some of which may be the BCD code for the comma. It is desired to count the number of commas in the word in the MQ.

We will again have to write a loop. The procedure will be to shift one six-bit BCD character into the right end of the AC, and then compare the AC with a word in memory which contains a BCD code for a comma. If it is the same, we will increase a counter (one of the index registers) with the instruction TXI (Transfer with index Incremented) The basic instructions appearing in the center of the loop will be:

```
LGL    6          get character from MQ
LAS    CØMMA      compare with comma
TRA    END        not a comma
TXI    END,4,1    comma, increase counter
TRA    END        not a comma
```

where END is the name of the location of the next instruction after these, and we have assumed that the AC was all zeros before starting. Also, elsewhere in the program, at the location named CØMMA there must be stored the BCD pattern for a comma. This is done by using the data-generating pseudo-instruction "BCI" as

```
CØMMA   BCI    1,00000,
```

BCI is called a pseudo-instruction because it is not an instruction to the 7090, but rather an indication to the assembly program that we would like it to set up a word containing a certain bit configuration that we interpret to be a BCD code. The number "1" and the first comma are required by the syntax of the BCI pseudo-instruction; the six characters following this comma are taken to be the six to be inserted into the word in the BCD representation. (See figure 12.) We have taken advantage of the fact that the digit zero is represented by six binary zeros, in writing

00000,

as the six desired characters. Then, when a comparison with the contents of the AC is made, the left-most 30 bits of the word will all be zeros, as will the left-most bits of the AC, assuring correct comparison of those parts of the two words. Note that if we had written

```
     CØMMA  BCI    1,     ,
```

indicating five blanks and a comma, the comparison would not have worked correctly, as the blank is not six zeros in the BCD code.

Let us now write the complete program, with a loop, and being careful to clear the AC each time before going through the loop. Note that index register four, used for counting commas, must also be set to zero at the start.

```
        AXT    0,4           reset comma counter
        AXT    6,2           ready for six passes
BEGIN   CAL    ZERØ          clear out AC.
        LGL    6             shift in next char.
        LAS    CØMMA         is it a comma
        TRA    END           no
        TXI    END,4,1       yes, increase counter
        TRA    END           no
END     TIX    BEGIN,2,1     count passes thru loop
```

and elsewhere in the program away from the instructions would be the data words

```
CØMMA   BCI    1,00000,      comma for comparison
ZERØ    DEC    0             zero to clear AC.
```

Problem.

On older keypunches there were two different minus signs, represented by different BCD codes within the computer. This second BCD code is now used for the apostrophe, but some people with older keypunches still use this code for a minus sign. Write a program which will scan the six BCD characters in the MQ, and replace any apostrophes with minus signs. The resulting word should be in the MQ when you finish.

LESSON THREE: NUMBERS

Arithmetic instructions and number representation.

So far, we have seen that a pattern of bits in a word may be interpreted as an instruction by the computer, or as six BCD characters. We can also, of course, attach a numerical value to the pattern of bits by interpreting the pattern as a number in the binary number system. That is, to bit 35 we attach the value one; to bit 34 the value two, to bit 33 the value four, and so on, each bit having twice the value of its neighbor on the right.

To determine the value of a complete 36 bit word we simply add up the values of the bits which are ones. For example, in the four bit number

1011

the rightmost bit is on, having value one. The next one is on also, having value two. The third bit being off, we skip value four; the left-most one has value eight. Adding eight, two, and one, we obtain eleven for the value of this pattern of bits. With 36 bits at our disposal, we can of course obtain much larger numbers if we desire. We make the convention that if the zero bit (s-bit) is a one, we will consider the number to be negative.

One exception should be noted here, however, in the index registers. The index registers are only 15 bits long, and therefore the largest number one can place there is 32,767 ($2^{15}-1$). Also, negative numbers are represented in index registers in a slightly different fashion than in the AC called complement form. This difference only becomes important when trying to compare a positive with a negative number. Lesson seven contains details of the complement representation.

The computer has several instructions available to allow us to add, subtract, multiply, and divide numbers which are interpreted as above. That is, if we have two words containing patterns of bits to which we attach values 'a' and 'b' by the above procedure, if the computer adds these two patterns together with the ADD instruction, the resulting pattern of bits will have value 'a+b' when interpreted by the same procedure.

A parity checking program.

Let us write a short program which uses arithmetic instructions. Suppose that the 36 bit word in location DATA has been read into the computer from a telephone line, from a remote laboratory experiment. To check the correct transmission of the data word, only 35 bits of the word contain information. The 36th bit is a check bit, which has been chosen to make the total number of ones in the word odd.

(This is known as a parity check.) We are to write a program which checks whether or not the number of one bits in the word are indeed odd; if so, transfer to location GØØD; if not transfer to location BAD.

We will set aside a storage location in which to place the count of the number of one bits in the word; when we start, we should set this location to zero with the instruction STZ. We can take advantage of the fact that if we shift one of the bits of the data word into the AC, it will have value one if on, zero if off; thus we can add it directly to the count with an ADD instruction. The procedure will then be: first clear out the AC (set it to zeros); shift in one bit from the MQ; add this bit to the count, then store the result in a location named CØUNT so it will be available for the next time through the loop. We will go through this loop 36 times; once for each of the 36 bits in the MQ. Here is the basic pattern:

```
CLA     ZERØ        set AC to zero
LGL     1           get next bit from MQ
ADD     CØUNT       add in to count
STØ     CØUNT       replace count
```

Thus in location CØUNT we will build up the number of ones in the word. We must still face the issue of whether the result is even or odd. This can be determined easily if we remember the representation of numbers in the binary system, described above. The right-most bit has value one; the rest have values of two, four, etc., all multiples of two.

Since this is the case, if the number is odd, its rightmost (sometimes called low-order) bit must be a one; if the number is even the low-order bit must be a zero. We need merely check the low-order bit with the LBT instruction. This instruction is a skip-type instruction (as was LAS.) If the low order bit of the AC is a one, the computer skips the next instruction; if a zero, it instead takes the next instruction. The sequence is then,

```
        CLA     CØUNT          get count
        LBT                    is it odd or even
        TRA     BAD            even, parity wrong
        TRA     GØØD           odd, go on
```

The complete program is, then, after adding the instructions to make a loop:

```
        STZ     CØUNT          set count to zero
        LDQ     DATA           get data word into MQ
        AXT     36,2           go thru loop 36 times
LØØP    CLA     ZERØ           zero AC.
        LGL     1              count
        ADD     CØUNT          one
        STØ     CØUNT          bits.
        TIX     LØØP,2,1       index
        CLA     CØUNT          get final count
        LBT                    check parity
        TRA     BAD            even, parity wrong
        TRA     GØØD           odd, go to GOOD
CØUNT                          storage for bit count
ZERØ    DEC     0              constant zero
```

An observant reader might note that following the TIX instruction it is not really necessary to

```
        CLA     CØUNT
```

as the count is still in the AC from the last pass through the loop. An even more observant reader might note that since the accuracy of the count is not important, but only its oddness or evenness (parity) the instruction to clear the AC could be dispensed with. (Why?)

Problem.

In location DATA is a 36 bit word which contains four positive integers; laboratory data which have been read into the computer. The rightmost nine bits contain one number; the next nine the next, etc. Unfortunately, each of the numbers is higher by two than desired; write a program which will get the contents of location DATA, lower each of the four numbers found there by two, and return the resulting data word to location data. Important: if subtracting two from any one of the integers causes it to go negative, set that integer to zero rather than a negative number.

This program will require use of either the TPL or TMI instructions, which test for the presence of the sign bit of the AC; the reference manual describes these instructions. You are encouraged from now on to "window shop" in the reference manual for instructions which may help solve a problem more easily. It should be emphasized, however, that if a way works, it should be considered satisfactory, and time should not be wasted looking for a more elegant solution, using a more sophisticated instruction. (A perfectly good way of picking up ideas, if the opportunity presents itself, is to read over programs written by others.

Different programmers will attack a problem in diverse ways, much as different authors will handle the same plot. The difference is best described as one of style.)

LESSON FOUR: PROGRAM MODIFICATION

Program modification.

One of the interesting features of having instructions stored as patterns of bits in the computer memory along with the data is that the computer program can then itself modify instruction words. Since this modification can be under control of loops and conditions which may depend on data being processed, very sophisticated procedures can be developed.

As a simple example, let us consider a program in which the address part of one instruction must be changed each time we go through a loop. This kind of manipulation occurs whenever we wish to work with an array of numbers. Suppose some statistical data has been compressed into the form of 36 bit words; each word representing one questionnaire and each bit within the word representing a person's yes or no answer to a certain question. Several, say 100, of these words are stored in memory in adjacent locations starting at location LIST. It is desired to count the number of questionnaires with yes answers to question three (i.e., the number of words in the array with bit two on.)

If we have an entry from the array in the MQ there will be no difficulty determining whether bit two is on; we have seen programs similar to this before. The interesting problem is how, after processing the first word in the array, can we get to the second one. If the first instruction in the loop is

```
        LDQ     LIST
```

we will be able to get the first word of the array into the MQ. We can then examine the third bit and if it is a one, add it to a counter. To get the next item, we need an instruction which says

```
        LDQ     LIST+1
```

in effect. Rather than writing the above instruction (and the 98 others which would be needed if we continued on this tack) we will attempt to modify the original instruction which said

```
        LDQ     LIST
```

First, let us give the location in which the LDQ instruction is found a name, e.g. PCKP. The instruction now reads:

```
PCKP    LDQ     LIST
```

Now, at the end of the loop, we write the sequence:

```
        CAL     PCKP        get pickup instruction
        ADD     ØNE         increase address
        STA     PCKP        insert new address
```

We have taken the old instruction and added one to it, making the instruction in the AC read LDQ LIST+1;

the last instruction in the sequence inserts the new address part (LIST+1) into the instruction at location PCKP. (Why was the instruction CAL instead of CLA used? Hint: what do we know about the sign bit of the word in location PCKP?) If we could examine the instruction in location PCKP it would now be an instruction to load the MQ from location LIST+1. Thus we have modified the LDQ instruction, to permit it to be used several times for slightly different purposes.

As a final step, we should, after completing the loop, fix up the LDQ instruction we have modified so it looks like it did originally, in case the program is used again. Even better, we can do this step at the very beginning of the program. We can do this with the sequence:

```
        CLA     KEY          get address of list
        STA     PCKP         reset pickup
```

and elsewhere in the program would be the data word containing the address of the first item in the array:

```
KEY             LIST
```

Note that this instruction has a blank operation field; this is taken to mean that no instruction code (i.e. all zeros) is desired, but that a word should be created with the given address, (LIST) and its location should be given the specified name (KEY).

In the above sequence then, the CLA instruction brings into the AC the word at location KEY: this word is nothing but the location of the first entry in the array. The STA instruction then stores this location in the address part of the LDQ instruction so it will be ready to start working on the array.

Let us now look at the entire program:

```
        STZ    CØUNT       clear count register
        CLA    KEY         reset pickup in case pro-
        STA    PCKP        gram has been used before
        AXT    100,1       count 100 words in array
PCKP    LDQ    LIST        get current word
        CLA    ZERØ        clear out AC
        RQL    2           skip first two bits
        LGL    1           get bit three
        ADD    CØUNT       count it
        STØ    CØUNT       save count
        CLA    PCKP        fix pickup instruction to
        ADD    ØNE         get next word from
        STA    PCKP        array
        TIX    PCKP,1,1    count, and return
        .
        .
ØNE     DEC    1           constant one
ZERØ    DEC    0           zero
KEY            LIST        to reset pickup
CØUNT                      storage for statistic count
```

The array at location LIST is not shown; presumably it is somewhere else in the program.

Instruction parts.

When using program modification, it is necessary to know how to build up an instruction in some cases, in others the internal structure of an instruction word is needed to know where to insert a vital part of the instruction. For example, in the previous program, we inserted an address into the address part of an instruction with the STA instruction. A word can be divided into four parts, the prefix, tag, decrement, and address, as shown in figure 13.

The address part of the instruction always contains the location reference for that instruction. For those instructions referring to an index register the tag part contains the number of the index register. Those instructions (such as TXI or TIX) which have a count field to indicate the amount of an index register change have this field stored in the decrement part of the instruction word. The prefix then contains the operation code. (Most instructions use both the decrement and the prefix as part of the operation code.) To aid in program modification, there are a set of instructions available which store the various parts of the AC into the corresponding parts of a word in memory. STA was one of these.

Since these special program modification instructions also are available to a programmer who is not doing program modification, he may wish to divide his data words into a similar format to take advantage of these instructions.

Problem.

A special kind of a list, called a string pointer has been stored in the computer. The address part of the AC contains the location of the first word in the list. This first word contains in its address part the location of the

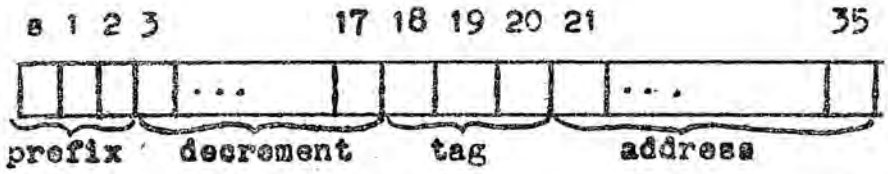

figure 13. Division of word into instruction parts.

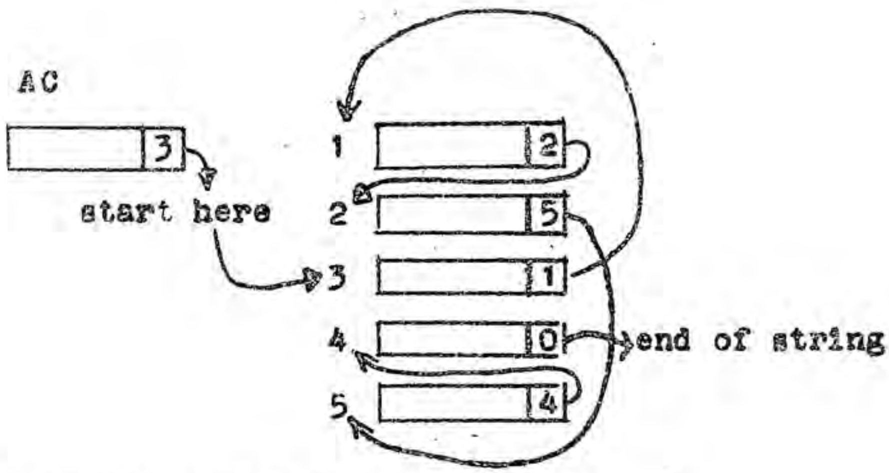

figure 14. A string pointer list.

second word (which may not be stored sequentially after the first); the second item contains in its address part the location of the third, etc. For example, a five word string pointer list starting at location three and ending with a zero word might look like figure 14. We see that the address part of location three is one; this means that the next item in the list is in location one, etc. This list ends at location four, which contains a zero.

Only the address part of a word is used for the string. The rest of the word is used for data. You are to write a program which puts the tenth item in the string pointer list into the logical AC.

LESSON FIVE: EFFECTIVE ADDRESSES

Index registers as address counters.

In the previous lesson we saw a typical situation requiring the use of an array, a group of data words stored one after another. Referring to successive elements in the array, even though they were adjacent in storage, required program modification, and the addition of several instructions to the loop. Since operations with arrays are quite common, a procedure for handling them more easily is available. Since most loops have an index register as a counter for the number of times through the loop, it would be handy if that same index register could also be used as an address counter to determine which of the elements of the array is to be used. This is exactly what happens if an _effective address_ is used. The procedure is as follows: if we write

```
LDQ     DATA,2
```

specifying one of the three index registers in a second subfield after the address DATA, the computer will calculate an effective address for the LDQ instruction; this effective address is defined to be the given address minus the contents of the specified index register, as in figure 15. Thus in the above example, if index register 2 contained a

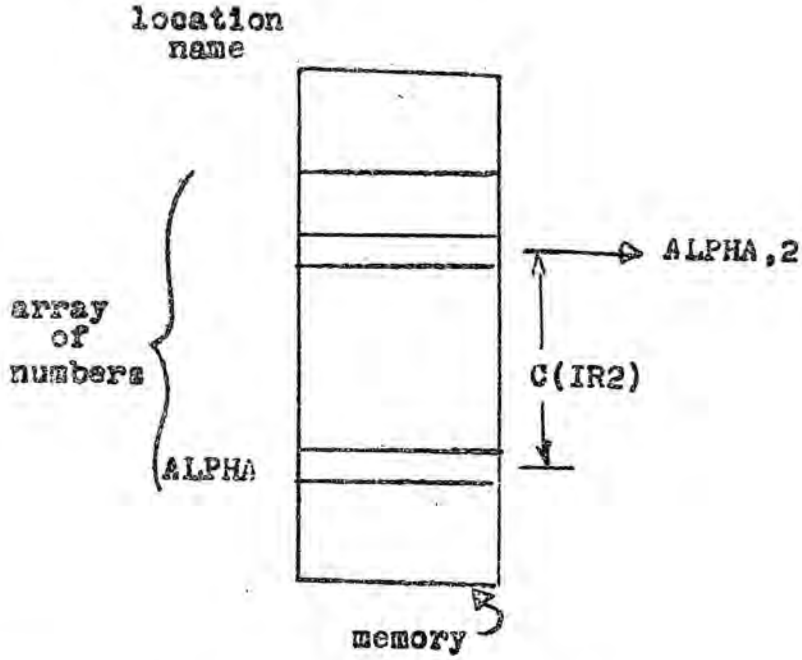

figure 15. Effective addressing.

seven, the contents of location DATA-7 would be loaded; if the index register contained six, location DATA-6 would be used, and so on.

Although the index register contents are subtracted from the given address, (rather than added, which might seem more convenient*) it is still possible to make much use of effective addresses to simplify programs which would otherwise require program modification. For example, consider recoding the previous program to use an effective address:

```
        STZ    CØUNT
        AXT    100,2
PCKP    LDQ    LIST+100,2
         .
         .
         .
        TIX    PCKP,2,1
```

The instruction at location PCKP now loads the MQ with

---

* The fact that effective addresses are computed by subtraction is a result of an economy decision in designing the I.B.M. 704 computer several years ago, and does not stem from increased utility to the programmer. Present computers (7090/7094) retain this design for compatibility with older programs (and programmers.) Since this is the case, the simplest procedure for learning to count backwards is to learn "clichés" for writing loops such as the AXT--LIST+100--TIX combination used here.

LIST+100-100 = LIST, the first time; after IR2 has been lowered by the TIX instruction, it loads location LIST+100-99 = LIST+1; on successive passes through the loop successive items from the array are obtained.

Note that effective addresses do not eliminate the need for program modification. The string pointer problem posed in the previous lesson still requires program modification.

In writing a program to sort a table of numbers into increasing numerical order, one needs a section of coding which does the following: scan the table from the top, looking at one word at a time. Examine, say, the third bit of each word, looking for a word with a one in the third bit. When one is found, stop scanning. Then, perform a similar scan starting at the bottom of the table, looking at the same bit, but this time looking for a zero bit. If one is found, stop scanning and interchange this word with the one found previously.

Let us write a section of code which does the first scan, and make it general enough that it can perform the scan on any bit within the words. Assume, for example, that index register one contains the number of the bit to be examined when the program starts. If IR1 contains 0, bit zero (the sign bit) is examined. If IR1 contains 4, bit four of the word is the one in question. Since the entire scan is done on the same bit, we will simply need a couple of initialization instructions to make the program scan the correct bit. One way to do this is to assume that there will be someplace in the scan program an instruction

ALS n

where "n" is the number of the bit to be examined. Then, having shifted the bit in question into the p-bit of the AC, we can use the instruction PBT (P-Bit Test). The shift instruction will even work for the case of the zero bit, since if we write

ALS 0

the computer will not shift the AC at all. To make the program flexible, we will have the program itself decide how much shifting to do; the address of the ALS will be left for program modifications. We indicate this by writing it thus:

```
SHIFT ALS **
```

The double star is given value zero by the assembly program. We could have written zero, or anything else, but the ** is used primarily to indicate to the reader that some other value will be planted here by some program modification instruction.

Now, we must modify the ALS instruction before we use it. This can be done easily in this case, since the bit number, and consequently the length of the shift, is contained in index register one. We can write as our first instruction

```
        SXA     SHIFT,1         (Store index in Address)
```

storing the index register in the address of the shift instruction. If IR1 had contained seven, the shift instruction would now read

```
        ALS     7
```

Enough for preliminaries, let us write the program. Assume that there are 100 numbers stored in the array, whose first entry is at location ARRAY. The program is:

```
        SXA     SHIFT,1         initialize shifter
        AXT     100,2           get set for 100 looks
SCAN    CAL     ARRAY+100,2     get data item
SHIFT   ALS     **              shift to position
        PBT                     is bit on
        TRA     INDEX           no, go index
        TRA     YES             this word has bit on
INDEX   TIX     SCAN,2,1        index, and get next word
        TRA     NØ              no words with bit on
```

The CAL instruction is used to get the sign bit of the word into the p-bit of the AC, in case bit zero is specified.

Problem.

Write the similar program segment which scans the 100 entry ARRAY from the bottom, looking for the first word, if any, which has a zero bit in the bit position specified by index register one. If one is found, transfer to location CHANGE, with IR2 containing the index of the word found, as in the problem above. If no such word is found, transfer to location NØ.

LESSON SIX: INPUT AND OUTPUT

Computer-user communication.

We come now to the subject of the communication between the computer and its user. In most problems data of some sort must be read into the computer as a source of computation and when finished, the results must be printed out in a form useful to the user of the computer. These operations are known as input and output, and quite often occupy considerably more than their share of attention in the programming of any particular problem.

We have seen before that symbols (letters and numbers) can be represented within the computer by special codes, known as BCD. The computer input and output equipment is designed around the BCD codes, that is, if a word of BCD information is sent to a typewriter or printer, the six letters printed will correspond to the six BCD codes found in the word. In studying input and output, then, we will concern ourselves with sending blocks of BCD information to an output unit, or receiving such blocks from an input unit. We will not worry about the other just as important problem of converting a number stored, say, as an integer, in a word, into the series of symbols (digits, decimal point, plus or minus sign, etc.) needed to represent the number in BCD. We will assume that that job has been done and that a string of characters is available in a buffer waiting to be printed. (Or alternately, we will assume that such a string is to be obtained from a card or an input typewriter.)

As has already been implied, the information flow between an input or output unit and the computer naturally occurs in packages, such as a six bit character. Another natural package is the six character word, and most output devices are more nearly oriented about either a single word or a variable number of words. (See figure 16.)

Another important aspect of communication with input-output devices is that they are usually mechanical contrivances, which tend to run much more slowly and less reliably than the electronic computer, and the programmer often finds himself involved in problems such as attempting to time his program to synchronize with rotation of a print wheel, or checking to make sure that the correct set of letters was written. Since such programs are difficult, and the programmer doesn't usually want this detailed a control over his output anyway, the job of writing the detailed machine language program to communicate with a particular input-output device is usually left to an expert, and most users perform input or output by calling on a <u>subroutine</u>, a package of instructions which actually

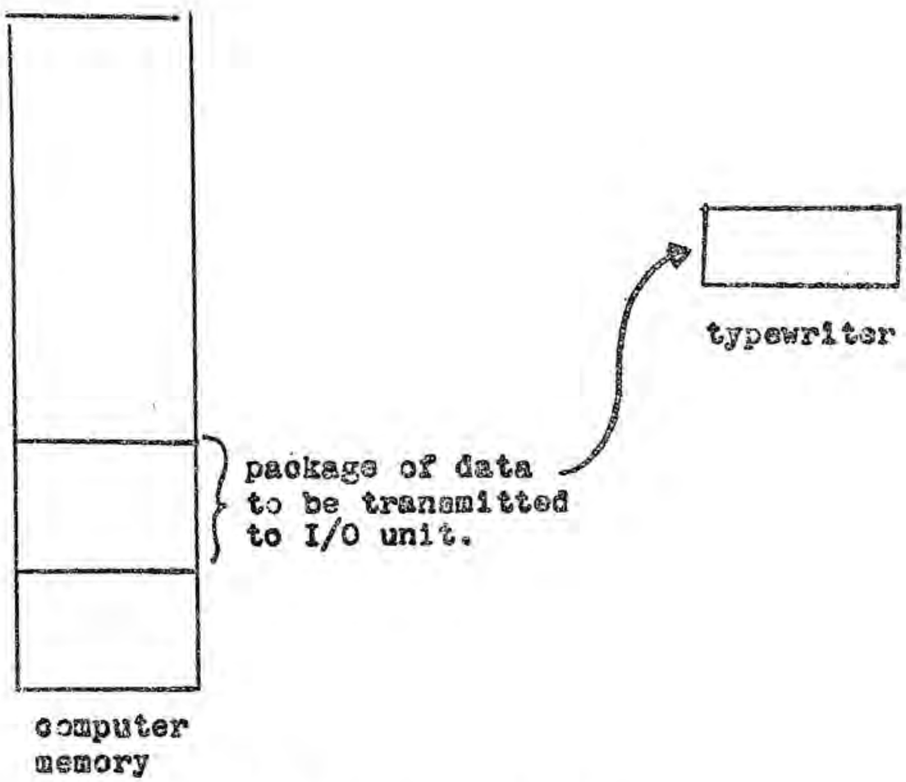

figure 16. Input-output communication.

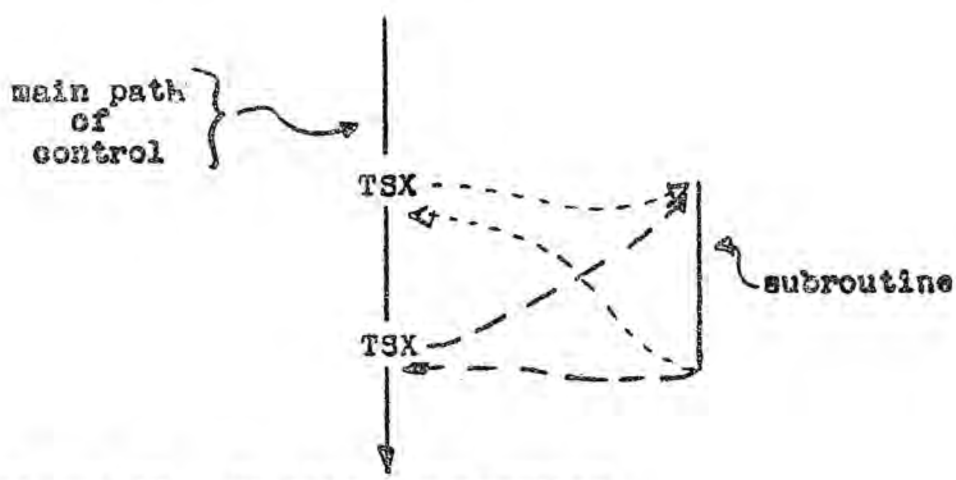

figure 17. Calling a subroutine.

perform the manipulation of the input-output device.

Subroutines.

From the point of view of the programmer using these subroutines, however, the instructions to call the subroutines can be considered to be the input or output instructions themselves, as when they are performed the desired function occurs. The programmer need only learn the calling sequence to a subroutine and he can use it as if he himself had written it.

Let us look at one of these subroutines, and its calling sequence. The operation attempted is to type out a line of characters. The characters are stored six to a word in a buffer, at location BUFF; the buffer is ten words long, which means we wish to type out a total of 60 characters. The subroutine calling sequence which we use is:

```
TSX     $WRFLX,4
        BUFF,0,10
```

Two adjacent words in storage are used for this calling sequence. The first contains a special subroutine calling instruction and the name of the subroutine. The second contains the address of the buffer and the number of words in the buffer. This second word, not an instruction but really a piece of data, is known as a parameter of the subroutine.

In this case, the name of the subroutine is WRFLX; the dollar sign indicates that WRFLX is indeed an external subroutine name and not an ordinary symbol standing for a location within this program. Subroutine WRFLX will take the 10 words located in the buffer at location BUFF and type them out on the user's typewriter. If a different parameter word had been used, of course a different buffer of a different length may have been printed.

As might be guessed from the tag of four on the TSX instruction, index register four is used in the subroutine call, to tell the subroutine where to return when finished. See figure 17. One must be careful, therefore, not to be using index register four when making calls to subroutines. The index register can of course be stored before the TSX instruction is encountered, and reloaded after the subroutine is finished. The convention is made that the subroutine will not change the contents of the other index registers.

When WRFLX has finished writing out the necessary characters it transfers control of the computer back to the instruction following the parameter word, the second location after the TSX instruction. It determines this location by inspection of the number in index register four, which was set by the TSX instruction. We will see how this is done later.

There is a complementary subroutine available which "listens" for something to be typed in at the typewriter. It collects the typed-in letters in a specified buffer until the typist returns the carriage; then it returns control to the program which called it to process the data. This subroutine is called by the sequence:

```
        TSX     $RDFLX,4
                INBUF,0,14
```

In this case, subroutine $RDFLX is to accept characters from the typewriter and store them in the 14-word buffer which starts at location INBUF.

Let us use these two subroutines in a simple program which asks the typist to type his name, listens for his reply, th n types a response. We start by typing a request; then a listen, then type another line. There will be three subroutine calls in all. Here is the program:

```
        TSX     $WRFLX,4
                CØM1,,4
        TSX     $RDFLX,4
                INPUT,,14
        TSX     $WRFLX,4
                THKS,,2
         .
         .
```

Else where in the program are the input and output buffers, which might appear as:

```
CØM1    BCI     4,PLEASE TYPE YØUR NA E.
THKS    BCI     2,THANK YØU.
INPUT   BSS     14
```

When the program is finished, the input buffer at INPUT will contain the characters that the typist typed. The program may now use them in any fashion it wishes.

A new pseudo-operation (BSS) appeared in the last sample program. This pseudo-operation causes the assembly program to leave unassigned a number of adjacent storage words for future input data. The name in the symbolic location field of the BSS pseudo-operation is taken as the name of the location of the first of the block of words. In this case, it was desired to leave a 14 word space (buffer) for the input letters; the pseudo-instruction

```
INPUT  BSS    14
```

causes 14 locations to be set aside and the first one to be named INPUT. BSS may also be used in applications unrelated to input and output operations, for example, when a block of numbers is copied over into another empty block, the empty block may be reserved by a BSS pseudo-instruction.

Other I/O devices.

Although we have only discussed the specific problem of communication with a typewriter, the general procedure followed is the same no matter what input-output device is used. One need merely to learn the details of the subroutine calling sequence to be able to use a new device.

Problem.

In checking out remote console equipment, it often is necessary to test the reliability of the transmission of the characters between the remote console and the computer. For this purpose, one may write a program which listens for console input, then types back on the console the same sequence of characters typed in, for comparison. Write a checking program which first types a "go-ahead" message, then listens for input, then types the input back out, and starts over with the "go-ahead" message.

LESSON SEVEN: SUBROUTINES

Writing subroutines.

In the previous lesson we saw how to use a subroutine which someone else (in that case an experienced system programmer) had written. We used a special instruction, TSX, and gave his subroutine a parameter which it used as a piece of data.

It quite often happens that in writing a large program a certain task has to be done several times, but not in the repetitious nature of a loop. To use an earlier example, it may be necessary to count commas in a word at different points within the program. Rather than writing the count procedure into the program each time it is needed, it will save space (and the programmer's time) to write the count procedure as a subroutine, and call for it whenever it is needed. We might, for example, write a subroutine named CØMMA, which will count the commas in the word in the MQ and place the answer in the AC. The calling sequence to the subroutine might be:

```
        TSX     $CØMMA,4
```

and it would be understood that the word to be checked is in the MQ when CØMMA is called. CØMMA is to place the answer in the AC when it is finished and return control to the first location following the TSX.

It is a characteristic of external subroutines, which are the only kind we will study, that they are rigidly independent of the programs which call them, except for the linking through the specified calling sequence. In addition, they may be translated (assembled) into binary machine language independently of the main program. This means, for example, that subroutine CØMMA cannot refer to any of the symbols in the calling program. As review, let us look again at the instructions needed to count commas; this example was discussed in lesson two.

```
        AXT   0,4          reset comma counter
        AXT   6,2          ready for six passes.
BEGIN   CAL   ZERØ         clear out AC
        LGL   6            get next char.
        LAS   CØMMA        compare with comma
        TRA   END          no
        TXI   END,4,1      yes, count comma
END     TIX   BEGIN,2,1    count passes
          .
          .
CØMMA   BCI   1,00000,     comma constant
ZERØ    DEC   0            zero constant
```

Let us now make a subroutine out of our comma counting program. First, consider the operation of the TSX instruction which was earlier referred to as a special subroutine linking instruction. The TSX instruction causes a transfer to the location given in its address, but in addition index register four is set to contain the negative of the location of the TSX instruction itself. (Note that negative numbers are represented differently in index registers than they are in the AC; this different representation is sometimes called complement representation.*) For example, if the instruction

```
HERE   TSX   THERE,4
```

were executed, the computer would take its next instruction from the location named THERE, and index register four would contain (-HERE). Thus the subroutine at location THERE can look in index register four and determine the location from which it was called. If the subroutine needs a piece of data (parameter) stored in location HERE+1 it need merely form the address HERE+1 with the aid of

---

* Since the largest number an index register can hold is 32,767, we may ask "what if we add 1 to this number?" The effect is to return the register to zero, much as the odometer in a car returns to zero after 999999 miles have been traveled. This effect can be used to define a negative (complement) number, as the number added to 1 which produces zero is usually called -1. Thus 32767 can for addition purposes be considered to be -1; 32766 can be considered as -2, etc. Obviously, for comparison, our special equivalent of -2 is actually larger than, say, +3, so we must be careful when making comparisons with complement form numbers.

index register four. For example, we could write

```
        CLA     1,4
```

an effective address which is interpreted as location (1-C(IR4)) but since IR4 contains (-HERE), (1-(-HERE)) = 1+HERE; so the CLA instruction has the proper address to pick up the parameter word. Similarly, if we wish to transfer to the second location after the TSX instruction, we may use the instruction

```
        TRA     2,4
```

which transfers to location 2-(-HERE) = HERE+2

Similarly, if later the instruction

```
NEW     TSX     THERE,4
```

is executed, this time (-NEW) is placed in index register four, and when the subroutine at location THERE does a

```
        CLA     1,4
```

it will pick up the parameter in location (NEW+1), and later return to (NEW+2) with the

```
        TRA     2,4.
```

Thus the subroutine may be called from several places in the program, but each time it will use the correct parameter word, and each time it will return to the instruction after the one that called it. (See figure 17.)

Our subroutine, CØMMA, will have to do something similar to this, although it does not have a parameter word in location 1,4. The one other thing we must do is name our subroutine so that someone else will be able to refer to it, and also we must indicate where the first instruction in our subprogram is. These two operations are taken care of by the ENTRY pseudo-operation which does nothing more than these two operations. The complete program would then be:

```
        ENTRY   CØMMA         this is a subroutine
CØMMA   SXA     IR4,4         first instruction; save
        SXA     IR2,2         index registers
        AXT     0,4           go into comma counting loop
        AXT     6,2           ..
BEGIN   CAL     ZERØ          ..
        LGL     6             ..
        LAS     CØMMAB        compare with comma
        TRA     END           ..
        TXI     END,4,1       ..
END     TIX     BEGIN,2,1     ..
        PXA     0,4           get answer into AC
IR2     AXT     **,2          restore index registers
IR4     AXT     **,4          ..
        TRA     1,4           return to caller
CØMMAB  BCI     1,00000,      storage and constants
ZERØ    DEC     0
        END
```

We here see the ENTRY pseudo-operation in use. Note that the instruction in the location labeled CØMMA is the first one (logically) in the program, and the ENTRY pseudo-operation indicates this fact.

One other interesting aspect may be noted, that the subroutine saves and restores the contents of index registers two and four before using them. It is commonly considered the responsibility of a subroutine to do this operation, as the program calling may have some useful piece of information stored in the index register.

A link with other languages.

The TSX instruction and the calling sequence give us the ability to link together programs written in two different languages, such as FORTRAN and FAP. To make such a link, it is merely necessary to learn what calling sequence is provided. For example, the FORTRAN compiler generates a calling sequence for the statement

```
        CALL ALPHA(A,B,C)
```

which is, in FAP symbolic language,

```
        TSX     $ALPHA,4
                A
                B
                C
```

e.g., location 1,4 contains the location of the first argument, location 2,4 the location of the second, etc. FORTRAN may insert other pieces of information in the decrement portions of the parameter words, which have nothing to do with the subroutine call.

Similar calling sequences exist for other languages, and the programmer writing a subroutine for use with a FORTRAN program, for example, does not need any information about the FORTRAN compiler, except the nature of the calling sequence.

Problem.

A number of moderately difficult problems which are to be written as subroutines are given in lesson eight.

LESSON EIGHT: MORE DIFFICULT PROBLEMS

More problems.

At the end of this chapter are presented several more difficult problems which provide good examples of problems often done in the machine language. The first three are easily solvable with what has been learned in this workbook; the rest are somewhat more difficult and may require some outside research. The reader should pick one and write a solution. Testing programs for each are available.

Epilogue.

Having made his way to the last chapter, the reader may well wish to ask "How much to the collected wisdom in the field of computer programming has been discussed here, and how much more will I have to learn in order to solve my particular problem?" This is an appropriate spot to stop and answer this question.

A complete, self-consistent subset of the language of the I.B.M. 7090/7094 computer has been described by example. This subset is adequate for writing correct, moderately efficient programs in the machine language. There are, of course, a number of more exotic techniques often used by experienced programmers. These techniques range from more sophisticated approaches to simple problems, to simple approaches to very sophisticated problems. A bibliography of related recent books on the subject of programming techniques and languages is given at the end of this chapter, and the reader is urged to explore among these for ideas. By now, also, the reader should be competent enough to explore within the I.B.M. reference manuals and understand the descriptions of the computers and systems described there.

Taking the longer view the reader will note that the ideas discussed here, apart from the details of their implementation, have been present in the computer field almost from its beginning and may be reasonably expected to be still present through the next few generations of computers. Ideas such as loops, conditional brances, and program modification are significant in that they make the computer what it is, a powerful assistant to the programmer. Details such as word size, or whether effective addresses are computed by addition or subtraction will, of course, change from computer to computer, but the fundamental concepts emphasized here in fact carry over in large part to all computers programmed in the machine language.

One other comment is for the reader who may have started with the discussion of the machine language as his introduction to computer programming. With modern algebraic and other highly symbolic languages many programs can be handled adequately without ever resorting to the detail of the machine language. One should keep proper perspective and only attempt to apply machine language formulations to those problems which, because of considerations of time, space, or intricacy, are going to require the detailed flexibility of expression available, but also necessary when programming in the machine language.

He should also realize that in a certain sense, an independent formulation of machine language programming such as this one is quite temporary, awaiting development of common languages which can describe the bit manipulating capabilities of a digital computer with an efficiency comparable to present machine language programs, at the same time allowing the programmer to write less exacting sections of his program in, for example, algebraic equation form.

Six problems.

Problem one:

In translator programs one must often count the number of commas in the variable field of a card. Write a subroutine named CØMMA which counts the number of commas in a variable length buffer specified by a parameter word, and adds this count to whatever number was in index register one when subroutine CØMMA was called. The calling sequence to your program will be

```
        TSX     $CØMMA,4
                BUFF,,n
```

where the address of the parameter word contains the name of the first location of the buffer, and the decrement of the parameter word contains the number of words in the buffer. You are to start scanning the word in location BUFF and continue to location BUFF+n.

Problem two:

Oil pipeline data is being read into the computer in the following format: six six-bit numbers, each between zero and sixty-three are packed into each 36-bit word; ten words come into the computer at a time. These numbers are to be unpacked and placed in sixty consecutive core storage locations. In addition, they must be decoded into readings of gallons per minute by the following transformation:

```
63 becomes +31
62 becomes +30
     etc.
33 becomes +1
32 becomes +0
31 becomes -0
30 becomes -1
     etc.
 1 becomes -30
 0 becomes -31
```

Note that there is a distinction between plus and minus zero. The calling sequence to your program will be the following:

```
          TSX     $CØN,4
                  DATA
                  GALNS
```

where location DATA is the first of the ten packed words anddlocation GALNS is the first of sixty storage spaces left for your output. The word in location DATA is to be unpacked into locations GALNS, GALNS+1, etc.; the leftmost data item in a word appearing as the first one to be stored in array GALNS.

Problem three:

The number

A modulo B

is defined as the remainder left after division of A by B. Write a program which computes A modulo B assuming that A is in the accumulator and B is in the MQ at the time the subroutine is called. The calling sequence to your program will be:

```
          TSX     $MØDULØ,4
```

You should leave the answer in the AC. (Note that this problem does not require use of division instructions, although they may be used.) Assume that both A and B are positive.

Problem four:

Searching a table may be done quite efficiently in a machine language program. Consider a table made up of pairs of entries (arguments and values) in successive words as in the example:

```
TABLE  Arg 1
       Value 1
       Arg 2
       Value 2
       Arg 3
        etc.
         .
         .
```

TABLE is the location of the first argument in the table, alternate entries after this item are values and arguments as illustrated. Write a subprogma which searches this table for the argument identical to that one in the AC and replaces the AC with the value corresponding to the argument. The calling sequence to your program will be

```
        TSX     $SEARCH,4
                TABLE,,n
```

where TABLE is the location of the first argument, and "n" is the number of arguments in the table.

If you wish, you may assume that the arguments are in ascending numerical order, and write a logarithmic search program.

The logarithmic search.

If it is known in advance that a table is in order, advantage can be taken of this fact to shorten the search time. An ordinary search, examining each entry of the table, requires a number of steps proportional to N, the length of the table. If, on the other hand, one picks out the middle entry of an ordered table, and compares it with the key, it will be evident which half of the table should be examined. This half of the table can then be split into two parts by a similar check of its middle entry. Carrying this procedure to its logical conclusion results in location of the symbol in question in a number of steps proportional to ($\mathrm{Log}_2 N$). The proportionality constant is usually larger, but for fairly large N, (Log N) is so much smaller than N that the log search is preferable.

Problem five:

A television or facsimile picture can be converted into binary digits and stored by considering each position of the television scan to be light or dark, and recording a binary one or zero for that position. The next position of the scan is used to derive the next binary digit. In scanning across one line of a TV picture, 1024 separate positions are examined, and 1024 binary digits are stored. The scan then proceeds to the next line, where 1024 more bits are derived. If these words were stored in the 7094, 29 words would be required for each line.

To cut down on the amount of storage needed for a picture, (and on the number of bits needed to transmit such a picture) one can take advantage of the simple redundancy of most line scan pictures. Generally, positions within a picture will not change from light to dark at every transition between digits. In fact, if a light (one) digit is found, it is likely that there will be a long string of identical digits before a dark (zero) digit is encountered. Rather than storing each of the digits of the string, we can simply count them and store the count. If for example, a particular string of 1024 bits contained 10 strings of about 100 bits each, we would only have to store 10 seven-bit numbers. (Seven bits will allow us to store a number up to 128.) Thus the 1024 bits have been condensed to only 70 bits of storage, and the transmission of the smaller number of bits can be done more quickly.

Obviously this procedure will not work for any arbitrary set of 1024 digits, as it depends on the existence of strings of similar digits. This assumption is fulfilled for the particular situation at hand, however, so the procedure can be used. Write a FAP subroutine which scans the 1024 binary digits in the 29 words starting with PICT, PICT+1, etc. (The 29th word only contains 16 digits left-adjusted within the word.) Your subroutine is to count the lengths of strings of zeros and ones, and store the resulting counts (which must be less than 1024) as 12-bit integers packed three to a word in the output buffer starting at BUFF, BUFF+1, etc. When finished, IR1 should contain the number of string lengths stored in the buffer at BUFF.

The calling sequence to your subroutine is

```
        TSX     $CØMPRS,4
                PICT,0,BUFF
```

The locations PICT and BUFF are to be obtained from the calling sequence. Since it is not known in advance whether the first string will be a string of "zeros" or of "ones" make the convention that the first word in BUFF will be of negative sign if the first string is a string of "ones."

Problem six:

In the modernization of large-scale freight train operations the computer is playing a growing role as a data-coordinator. In this problem, you are asked to automate a typical task which might be reassigned from a dispatcher to a computer. The techniques needed to solve the problem are of general interest as they are also used in such diverse areas as cryptanalysis, information transmission, language translation, information retrieval, and heuristic compilers.

The problem is the following: freight cars are brought from loading platforms throughout the city at various times and are switched as they arrive onto sidings corresponding to their destination. Simultaneous with their arrival, information about their identification, contents, and destination are given to the dispatcher. He then assigns the cars positions in a train, so that the cars to be removed at each stop are together in the train.

The dispatcher finds it fairly easy to keep this list of cars in order as he can keep a separate page for each stop of the train. He may then add a new car to the page corresponding to the correct destination and, if necessary, insert extra pages for some stops. In a computer, however, memory space would have to be saved for a full train going to each destination unless only one list were kept for the whole train. Then if an item must be inserted in the middle of the list, other items must be moved up or down to make room. This moving is time consuming and difficult to program. The string pointer technique described in the appendix is an efficient way to hanle this kind of list. To make a simplified, though realistic problem, assume that a train is being made up for a trip between Boston and Miami. This train will stop in Hartford, New York City, Philadelphia, Washington, D.C., Richmond, Raleigh, Savannah and Miami, and may drop off cars in each city. Assume furthur that the only information of interest about a freight car is an 18 digit binary identification number, and its destination which is encoded as three BCD letters.

Your program is to have two entry points, START1 and ADD1. The first is called by:

```
        TSX     $START1,4
```

This is the signal for your program to begin assembling a new train. (If it was working on a train before, it is to forget about it and start a new one. Storage used for previous trains may be reused.) When you feel you have been adequately notified that a new train is being constructed, return to the main program with the location of the beginning of the list you will construct in the address portion of the AC.

The main program will then send your program detailed information about the cars in the following manner:

```
        TSX     $ADD1,4
```

The $AC_{p,1-17}$ will contain the 3 character BCD destination name, the $MQ_{s,1-17}$ an 18-bit car number. You are to insert the car number into your string pointer list in the proper place so that the car will be in order in the train. The order of the cars to be dropped at any one destination does not matter, as long as all the cars for one destination are listed under that destination. ADD1 may be called any number of times, and the list it makes may be used between calls. Since the calls will be in no particular order, ADD1 must determine where to put each car number in the list as it arrives.

To insert cars into the train with ease, a string pointer list is to be used. (See below) List continuity is to be maintained through the address portion of the word, with data stored in the left half of the word. If the tag position of the word is zero, the left half of the word is interpreted as a car number. If the tag position of the word contains a seven, the left half is interpreted as a 3-character BCD destination code. The destinations should read from the top of the list in the same order as indicated below.

Following each name in the list should be all the car numbers destined for that city, in any order. Then comes another destination name, and more car numbers. If no cars are destined for any one city, the name of that city is followed by the name of the next city in order. The list is terminated by a word of all zeros. Note that a subroutine that generates the correct train for any number of cars must also generate a correct train with no cars (i.e. just the destinations followed by a word of all zeros.) You may assume that fewer than 100 cars will be attached to your train in the test of your program, and that no "unknown" destinations will be given ADD1.

Order of stops:

destination	code
Hartford	HFD
New York City	NYC
Philadelphia	PHD
Washington	WSH
Richmond	RMD
Raliegh	RAL
Savannah	SAV
Miami	MMI

String pointer lists:

In translation, sorting, and information transmission problems, one often starts out with some sequence of symbols, and ends up with another sequence of symbols, after performing some operations on that sequence. These operations may include rearranging the order of individual symbols, inserting and deleting symbols, and replacing some symbols with others. One way of simplifying these manipulations is to place the symbols in a string pointer list. This is a list which, although intended to be in some order, is not necessarily stored in that order within core memory. Instead, for each piece of data there exists a pointer word containing two location references. The first reference tells where the piece of data is stored, the second tells where the pointer word for the next piece of data in the string may be found. The location of the pointer word to the first item in the list may be stored in another string pointer list.

Inserting an item in the middle of such a list now involves simply changing a location reference in one pointer word, not moving all later symbols down one place in storage as in conventional ordered lists. See figures 18a and 18b.

A simpler type of string pointer list, used when the pieces of data do not require a full word of storage space, is constructed as follows: each word in the string contains in its left half the piece of data; the address portion of

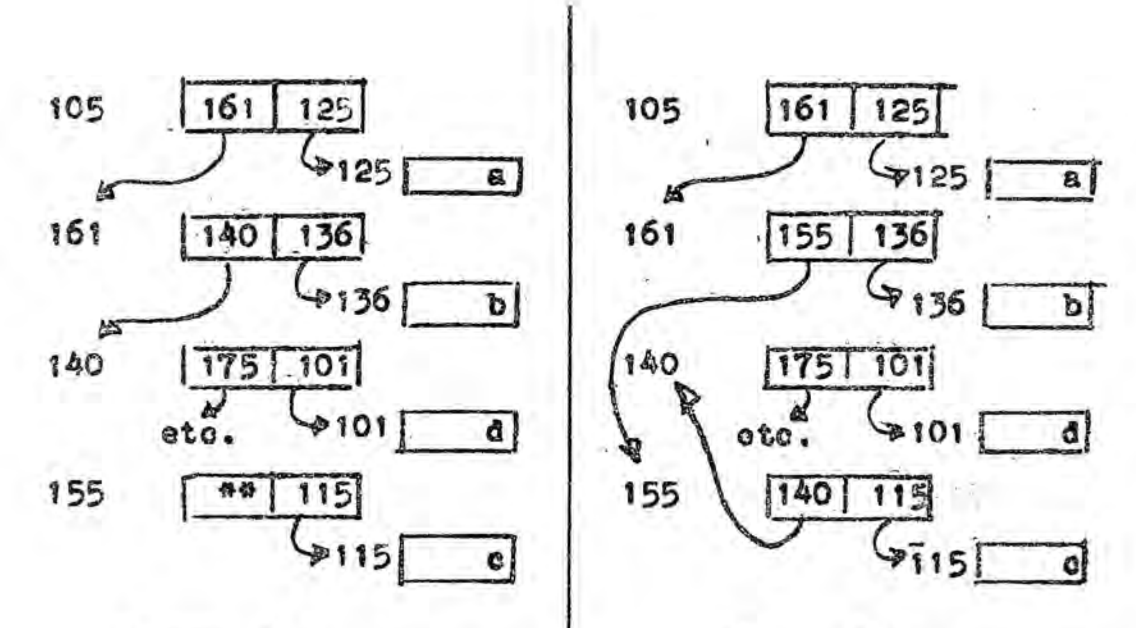

figure 18a.

figure 18b.

Figure 18. Inserting an item "c" into a string pointer list between items "b" and "d". Continuity is maintained in this list through the left half of the pointer words.

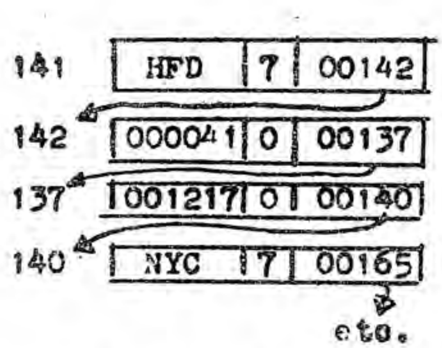

figure 19. A typical train list.

the word contains the location of the next item in the string. See figure 19. It is clear that there are many possible variations on the basic list structure.

You are to use this simpler type of string for your train list. Each eighteen bit number or destination name should be stored in the left half of a word, and the address part of the word should contain the location of the next element in the string. The tag part of the word contains a zero or a seven as the left half should be interpreted as a car number or destination code respectively. The end of the string is denoted by a word of all zeros.

REFERENCE APPENDIX

Introduction.

The body of this workbook is intended primarily as a workbook, and will not suffice as a reference for details of instruction and pseudo-instruction operations; partly because its layout makes reference difficult, and partly because some instructions or pseudo-operations are in the text only used in representative situations, not in their full generality. For this reason, the I.B.M. 7090 or 7094 reference manual should be used whenever a question arises as to the exact operation of one of the machine instructions. A list of the instructions most likely to be of use to a beginning programmer is given on the next page.

The situation with respect to the pseudo-operations of FAP is much the same, with the exception that the FAP language has many features of interest only to an experienced programmer, and the reference manual descriptions are quite often impossibly difficult for a beginner to follow. For that reason, a brief review of the FAP language and description of nine of its pseudo-operations are given in this appendix for reference by the beginner.

Guide to the 7094 manual.

Details of the 7094 are set out in the following sections:

Instruction descriptions

Arithmetic instructions

CLA	LDQ	ALS	TOV	ADD	STQ	ARS
CAS	SUB	MPY	LLS	XCA	STO	DVP
LRS						

Instructions for manipulating words as bit patterns

CAL	LGL	RQL	SLW	LGR	LAS

Test and branch instructions

TMI	TZE	TRA	PBT	TPL	TNZ	LBT

Indexing instructions

LXA	PAX	TIX	TXH	SXA	PXA	TXI
TXL	TSX					

Other useful instructions

CHS	SSP	STA	STD	STL

Look at the appendices to see what material is there.

A brief review of the FAP language.

FAP is an assembly language for the 7090/7094 computer developed at the Western Data Processing Center, at UCLA. It was originally conceived as an aid in writing FORTRAN-compatible machine language programs, and the FAP assembly program works within the FORTRAN monitor system. However, it is a complete assembly program in its own right, and has most of the features of modern-day assembly programs, including the independent subroutine ability which has proved valuable even when not writing FORTRAN subroutines.

Throughout this writeup it will be assumed that the reader is unacquainted with any assembly language, but is familiar with the operation of the 7090/7094 computer and some of its instructions.

Only an essential subset of the full FAP language is discussed here, but enough is said to permit writing complete, accurate programs.

FAP--the language.

In this section we will discuss the details of the FAP language, and the format of instructions written in the FAP language.

Symbolic card format. Instructions are punched one to a card (or typed one to a line) in the format of figure 20. Columns 1-6 comprise the symbolic location field. Column

1 6	7	8 14	15	16	72	73 80
symbolic location field		oper- ation field		variable field	comment field	sequence number field

figure 20. FAP card format.

seven is always blank. Columns 8-14 are known as the operation field. Column 15 is blank, and the variable field starts in column 16. The variable field continues from column 16 until the first blank column is reached. After this first blank column may appear an arbitrary comment extending up to column 72. Columns 73-80 are commonly used for labeling and sequence numbering programs.

The contents of each symbolic card are copied onto the output assembly listing, along with the octal equivalent of any binary word generated by that card and the location assigned the binary word. Note that not all of the symbolic cards in a FAP program generate binary words; In case no binary word is generated, only the symbolic card is listed on the assembly listing.

The symbolic location field. The location of an instruction or a piece of data may be named by placing a symbol in the symbolic location field of some card in the symbolic program. If a symbol appears on a card containing a machine instruction, its value will be the location to which that machine instruction has been assigned by the assembler. If it appears in the location field of a pseudo-operation, the discussion of the pseudo-operation must be read to determine which location has been named.

A symbol consists of one to six characters, which may include letters, numbers, parentheses, and the period. At least one of the characters must bot be a number. The programmer is free to invent any names he likes within these restrictions. (Names are usually chosen for their mnemonic value.) He must be careful, however, to make sure he does not attempt to define the same symbol twice by having it appear in the symbolic location field of two different instructions.

<u>The operation field</u>. The operation field may contain any one of the mnemonic codes corresponding to machine instructions described in the 7090 or 7094 manual. Or, it may contain any of the pseudo-operation mnemonic codes described below. If it contains a 7090/7094 instruction mnemonic, the assembler will look up the proper binary operation code and insert it into the word assembled for that instruction. The operation of the assembler for the pseudo-operations should be checked in the description of the appropriate pseudo-operation.

The assembler recognizes a blank operation field as equivalent to the 7090/7094 machine instruction "HTR" and assembles a word with a zero operation code. The other fields are treated as in any other machine instruction. In this connection note that a blank card will cause the generation of a word of all zeros in the assembled program.

<u>The variable field</u>. As its name implies, both the contents and the interpretation of the variable field change from instruction to instruction. For example, the variable field of a 7090/7094 instruction mnemonic is interpreted as

the name of a location in core storage. On the other hand the variable field of some pseudo-operations is interpreted as a piece of data for inclusion in the program. In most cases the variable field contains an <u>expression</u> and is intended to be interpreted as the name of a core storage location. (The interpretation of the variable field for the pseudo-operations is described in the sections on the individual pseudo-operations.) An expression consists of either a symbol, a decimal integer, a symbol plus a decimal integer or a symbol minus a decimal integer. The symbol, if present, must be the name of the location of some instruction. An expression such as

ALPHA+5

is interpreted as the fifth location after the location named ALPHA. An expression such as

4

is interpreted as the fourth location in the computer.

The special symbol "**" has value zero, and is used primarily for the convenience of another person reading the program, so he may recognize those parts of the program which may be changed by the program itself.

<u>Tags and decrements</u>. Numbers may be inserted into the tag and decrement fields of those instructions which may have tags and decrements, by adding subfields to the variable field. A subfield is indicated by typing a comma at the end of the variable field, followed by an integer. This

integer is inserted into the tag part of the instruction being assembled. A second subfield may be indicated by a comma following the first one. Again, an integer (or the symbol **) may appear, and it will be evaluated and inserted in the decrement part of the instruction being assembled. Examples:

```
CLA    ARRAY,1      tag is 1
TXI    LØØP,4,1     tag is 4, decrement is 1
```

It should be noted that FAP permits somewhat more complicated expressions for the variable field address, tag, and decrement; however the correct construction of these expressions is somewhat difficult. Since the simpler expressions described here will suffice for almost all situations, the more general facility of FAP may be left for future study, or the advanced reader.

Assembly. FAP begins assembling the program as though it would start in location zero. It assigns instructions, data words, and space for arrays to ascending locations in core storage in the order they appear in the symbolic deck. The resulting binary instructions are punched in a relocatable column binary format suitable for loading into the computer by the FORTRAN monitor system and the BSS loader.

The pseudo-operations

One of several pseudo-operation mnemonics may appear in the operation field of a card. These pseudo-operations can be placed in one of five classes: list control, data-generating, storage allocating, symbol defining, and organisational. These classes will be discussed in order.

List control pseudo-operations. The list control pseudo-operations have no effect on the assembled program. Instead they are used to control the printed assembly listing, to make it more understandable to the reader. Only one list control pseudo-op is of general enough interest to mention.

REM (remark)

1. The letters REM in the operation field.
2. An arbitrary remark starting after column 11.

The REM pseudo-op is used to introduce a remark into the assembly listing. The entire card, with the exception of the operation field is printed on the output listing. No binary instructions are assembled, and no symbols are defined.

Example:

```
        REM     SECTIØN TØ CALCULATE CØRRELATIØN.
```

Data generating pseudo-operations. The data generating pseudo-ops are used to introduce into the program registers containing those constants which are needed by the program.

DEC (Decimal data item)

1. A name may appear in the symbolic location field.
2. The letters DEC in the operation field.
3. An integer or real constant in the variable field.

DEC is used to introduce decimal integer and floating point (real) constants into a program. If the variable field contains a floating point (real) constant a word will be assembled which contains that floating point number, in the proper format for floating point machine operations. If the variable field contains an integer constant, a word will be assembled which contains that integer, in binary form. The definitions of integer and real constants are the same as in the MAD and FORTRAN languages. Examples:

```
TEN     DEC     10
CØNST   DEC     10.425
TWØ     DEC     2.0
```

BCI (Binary coded decimal information)

1. A name may appear in the symbolic location field.
2. The letters BCI in the operation field.
3. The digit "1" followed by a comma, followed by six characters of information in the variable field.

The BCI pseudo-op is used to encode letters and numbers in the standard BCD code, and insert these codes into the assembled program. The six characters (including blanks and commas) following the comma are converted to BCD and the resulting word is inserted into the assembled program. A symbol, if present is the name of the location of the BCD word. Examples:

```
TØWN    BCI     1,NYC
NAME    BCI     1, JIMMY
```

<u>Symbol defining pseudo-operations</u>. In addition to the usual procedure for assigning names to locations by placing them in the symbolic location field of some instruction, a name may be assigned by the SYN pseudo-operation.

SYN (Define synonymous symbol)

1. A symbol in the symbolic location field.
2. The letters SYN in the operation field.
3. An expression in the variable field.

The expression in the variable field of the SYN pseudo-op is assumed to be the name of some location in the computer. The symbol in the symbolic location field is defined as the name of that location. No binary words are generated or inserted in the program. SYN is commonly used to make two names (perhaps provided by different programmers) synonymous. Thus the same location may have two or more names. Example:

```
A      SYN    B
Q      SYN    ALPHA+14
```

Restriction: Any symbol appearing in the variable field of an SYN pseudo-operation must be "previously defined". That is, it must appear in the symbolic location field of a card (instruction) earlier in the program.

Discussion: Note carefully the difference in the following situations:

```
       CLA    ALPHA              CLA    BETA
        .                         .
ALPHA  DEC    5            BETA  SYN    5
```

In the first ALPHA is the name of the location of the integer 5. At execution time, the CLA instruction will cause the integer 5 to be brought into the AC. In the second, BETA is the name of location five, and if used as an address will cause reference to location 5. The CLA instruction will therefore bring the contents of location 5 into the AC. This difference illustrates that one must carefully distinguish between the name of a storage location and the name of the contents of a storage location.

Storage allocating pseudo-operations. In some programs it is desirable to set aside a section of core storage for an array of numbers to be read in or computed by the program. If this storage space is desired, some way is needed to inform the assembler that it should not place any data or instructions in the area. The storage allocating pseudo-operations are used to accomplish this.

BSS (Block of storage started by symbol)

1. A name may appear in the symbolic location field.
2. The letters BSS in the operation field.
3. A decimal integer in the variable field.

The BSS pseudo-operation causes a block of storage cells equal in length to the value of the integer in the variable field to be set aside. Any symbol in the symbolic location field is the name of the location of the first cell in the block. "Setting aside" of a block of storage is evidenced by the fact that the next instruction after the BSS will be assigned a location after the block; the cells in between will have no particular binary number assigned to them. Example:

```
ARRAY  BSS    19
```

Organizational pseudo-operations. The organizational pseudo-operations are used to indicate important features of the program to the assembler and pertain to the entire program rather than a single instruction. As such, they must appear in specific places within the program.

END (End of the program)

1. The letters END in the operation field.

The END pseudo-operation marks the physical end of the program, and therefore, must be the last card (or instruction) in it.

ENTRY (Entry point)

1. The letters ENTRY in the operation field.
2. A symbol in the variable field.

In a subprogram which is to be referred by another program, the ENTRY pseudo-operation indicates the first instruction which is to be executed in the subprogram when it is called. The ENTRY card has three functions:

a. It defines this program to be a subroutine.
b. It defines the name of the subroutine to be the symbol in its variable field.
c. It indicates the location within the program at which the first instruction to be executed may be found.

The symbol appearing in the variable field must be a name which appears in the symbolic location field of some instruction within the program. If the ENTRY pseudo-operation appears, it must be placed at the beginning of a program. Two or more entry points to the same program can be indicated by two or more ENTRY cards together at the beginning of the program. Example:

```
ENTRY   CØS
ENTRY   CØMMA
```

Error messages.

One bonus which may be obtained when using an assembly program is that the assembler can look for certain standard types of errors and inform the programmer of them. FAP distinguishes between two types of errors, those which make it impossible to assemble the program correctly, and those which can be assembled, but are probably slips by the programmer. All such errors are indicated to the programmer by the presence of a letter at the left edge of his assembly listing adjacent to the instruction in question.

Fatal error indicators:

letter used	error made
U	An undefined name has been used in this instruction in the variable field. The assembler does not know what location corresponds to the name.
M	This instruction uses (or defines) a symbol which has been defined more than once in the program. The assembler does not know which definition to use.

Ø The operation field of this instruction contains a mnemonic unknown to FAP.

E The address field of this data-generating pseudo-operation contains an error.

Non-fatal error indicators:

F This SYN pseudo-operation contains a symbol which has not yet appeared in a symbolic location field. That is, it is not previously defined. The SYN has been ignored.

A This instruction is expected to have an address and the programmer has not provided one. (Or it is not expected to have an address, and the programmer has provided one.)

T Same as A, but applies to the tag field.

D Same as A, but applies to the decrement field.

Certain of the more sophisticated features of the FAP language are also carefully checked, and appropriate error indicators are printed. Occasionally, an error when using a simple feature will appear to the assembler to be an error in use of one of its bells or whistles, and some rather obscure indication may be made. In these cases, the difficulty is usually obvious from an inspection of the instruction in question.

BIBLIOGRAPHY

1. Languages

_______, Reference Manual, I.B.M. 7094 Data Processing System, I.B.M. Corp., White Plains, New York, 1963.

_______, Reference Manual, I.B.M. 709/7090 Programming Systems, FORTRAN Assembly Program (FAP), I.B.M. Corp., White Plains, New York, 1963.

_______, I.B.M. 7090/7094 Programming Systems, FORTRAN II Programming, I.B.M. Corp., White Plains, N.Y., 1963.

_______, I.B.M. 7090/7094 Programming Systems, FORTRAN IV Language, I.B.M. Corp., White Plains, N.Y., 1963.

_______, I.B.M. 7090/7094 Programming Systems, MAP (Macro Assembly Program) Language, I.B.M. Corp., White Plains, N. Y., 1963.

Arden, B.W., B.A. Galler, and R.M. Graham, Michigan Algorithm Decoder (MAD), Malloy Litho. Co., Ann Arbor, Mich. 1963.

Galler, Bernard A., The Language of Computers, Mcgraw-Hill, New York, 1962.

McCracken, Daniel D., A Guide to ALGOL Programming, John Wiley and Sons, New York, 1962.

Organick, Elliot I., A Computer Primer for the MAD Language, Cushing-Malloy, Inc., Ann Arbor, Michigan, May, 1962.

2. Techniques

Arden, Bruce W., An Introduction to Digital Computing, Addison Wesley, Reading, Mass., 1963.

Corbató, F. J., et al., The Compatible Time-Sharing System: A Programmer's Guide, M.I.T. Press, Cambridge, 1963.

Corbató, F. J., Poduska, J.W., and Saltzer, J.H., Advanced Computer Programming, M.I.T. Press, Cambridge, 1963.

Ledley, Robert S., Programming and Utilizing Digital Computers, McGraw-Hill, New York, 1963.

Sherman, Philip M., Programming and Coding Digital Computers, John Wiley and Sons, New York, 1963.

Ware, Willis H., Digital Computer Technology and Design, John Wiley and Sons, New York, 1963.