systems programming

JOHN J. DONOVAN

TATA McGRAW-HILL EDITION
FOR SALE IN INDIA ONLY

Tata McGraw-Hill

SYSTEMS PROGRAMMING

Tata McGraw-Hill Edition 1991

46th reprint 2009
RBLARDLXRZLRD

Reprinted in India by arrangement with The McGraw-Hill Companies, Inc., New York

For Sale in India Only

Library of Congress Catalog Card Number 79-172263

ISBN-13: 978-0-07-460482-3
ISBN-10: 0-07-460482-1

Published by Tata McGraw-Hill Publishing Company Limited, 7 West Patel Nagar, New Delhi 110 008, and printed at Pushp Print Services, Delhi 110 053

The McGraw-Hill Companies

contents

preface

SCOPE

In this book we address ourselves to the full spectrum of systems programming endeavors, including the use and implementation of assemblers, macros, loaders, compilers, and operating systems. We present each of these components in detail, exposing the pertinent design issues. The issues are discussed within the context of modern computer languages and advanced operating systems; it is recognized that in addition to the traditional compiler problem of syntax and semantics, we now have storage allocation and accessing methods to contend with, and that file systems, multiprocessing, and multiprogramming are now commonplace in operating systems. To introduce the more formal aspects of computer science, we have included a presentation of formal systems and their application to programming languages.

The book is written as a text, with problems and exercises, with particular emphasis on the problems and examples. We have assumed that the reader has had experience in some high level language.

An attempt has been made to keep the book as machine-independent as possible; the text has, in fact, been used in conjunction with several different types of machines. However, to add substance to the book, we have taken specific examples from an IBM 360/370 type machine and, in our discussion of compilers, from languages with features like those exhibited in PL/I.

The book covers material contained in six courses of Curriculum 68 as described by the Association of Computing Machinery (ACM) Curriculum Committee in Computer Science.[1] The basic course, Computer Organization and

[1]As documented in the Communications of the Association of Computing Machinery (CACM), vol. 11, no. 3, p. 151 (March 1968).

Programming (B2), is covered in Chapters 1 through 5; Programming Languages (12) and Compiler Construction (15), are covered in Chapters 6, 7, and 8; Systems Programming (14); Advanced Computer Organization (A2); and Large Scale Information Systems (A8) are covered in Chapter 9.

MAIN USES

We feel that the book has three major uses: (1) as an undergraduate text in a one- or two-semester course on systems programming; (2) as a book for professionals; and (3) as a reference for graduate students.

More specifically, the book has been used to meet the needs of the following types of courses:

1. A first course in the undergraduate computer science curriculum (following an introductory programming course, e.g., FORTRAN, PL/I).
2. A general Institute service course for non-computer scientists.
3. An advanced course in software.
4. A software engineering course emphasizing practical issues.
5. An extensive review or introductory course for graduate students in computer science.

At M.I.T. the book is used in the undergraduate course 6.251, Digital Computer Programming Systems. There is a tradition and excitement associated with the course, and it is one of the most highly subscribed elective courses, having as many as 350 students per semester. I am also told that it is one of the most challenging. At M.I.T. the course is used to meet all of the above needs.

We have also used the material as a two-semester graduate course. In the first semester we dealt with the topics of machine organization, assemblers, macros, loaders, I/O programming, and operating systems (Chapter 1, 2, 3, 4, 5, and parts of 9). In the second we discussed programming languages, design of compilers, formal systems, and other aspects of operating systems not covered in the first semester (Chapters 6, 7, 8, 9).

The course has been given at several industrial firms: Honeywell, U. S. Underwater Systems Center, Martin Marietta, and others, where it focused mainly on the design issues of these system components, and omitted the formal systems aspects (Chapter 7).

This text was used to give an intensive course in programming at SESA in France, and also for the sequence of computer courses included in a two-year technical program at the Lowell School in Boston.

At Texas Tech University in Lubbock, Texas the programming course was first taught by using the video tapes of the lectures at M.I.T., which were sent to Texas for replay in the subsequent week. This method of transferring the course proved to be effective, since the undergraduates in Texas took the same quizzes and exams as did the M.I.T. students, and there was no appreciable difference in

grades. Video tapes of the course are available, and may be obtained by writing the author at M.I.T.

At M.I.T. the machine used in conjunction with this course was the IBM 360/370 type computer. At U. S. Underwater Sound Laboratories a UNIVAC 1108 was employed, while at Honeywell various Honeywell machines were used.

It is helpful for students to have had experience with assembly language and PL/I, though we have found that in many cases they have had neither. Some students have been able to use Chapters 2 and 6 as an introduction to assembly language and PL/I, especially in conjunction with reference manuals and lectures.

If used by professionals or graduate students, the book is self-sufficient in that there are enough details for 370 and PL/I to support the rest of the material.

In addition to discussing the traditional system components of assemblers and macros, the book gives special emphasis to important features of systems programming presently not covered in many texts–compilers, the advanced problems of storage allocation, recursion, operating systems, and I/O programming.

The problems are designed to be expository and solutions are available in the Teacher's Manual. Also included in the Teacher's Manual are sample syllabuses, quizzes, and helpful hints in presenting material.

acknowledgments

This book itself is an acknowledgment to the intensity, drive, and technical competence of the many individuals who have contributed to it. I list here only a few of the contributors.

One individual, Mr. Stuart Madnick, stands apart from all other contributors to this book. Mr. Madnick is the most competent systems programmer I know. He worked with me for five years, both as a teaching assistant and lecturer, on the M.I.T. course (Digital Computer Programming Systems) that uses this book. He is responsible for the presentation of the material file systems and the view of operating systems, and he has read and contributed to every chapter. I wish to thank him for both his technical contributions and his loyalty.

The M.I.T. course on Digital Computer Programming Systems has evolved from the contributions that every lecturer associated with it has made. We list these lecturers here.

1959	Professor F. Verzuh
1960	Professor F. J. Corbató
1961	Professor W. Poduska, N. Haller
1962	Professor J. Saltzer
1963	Professors C. L. Liu, D. Kuck, F. J. Corbató, D. Thornhill
1964	Professors J. Saltzer, D. Kuck, R. Fabry, T. Stockham
1965	Professors W. Poduska, L. Hatfield, R. Baecker, R. M. Graham
1966	Professor R. M. Graham
1967 - Present	Professor J. J. Donovan

It is difficult to state the exact contributions of each lecturer, as course content has evolved from assemblers to include loaders, compilers, searching techniques, debugging techniques, and recently, operating systems. Professor Corbató set the philosophy and introduced the case study method, and my immediate predecessor, Professor Graham, greatly influenced my thinking, as he taught me many of the fundamentals.

During the past five years, with the help of many teaching assistants, we reconstructed the course around the IBM 360, introducing the direct-linking loading schemes, PL/I, advanced compiling techniques, memory, processor management, and most recently, I/O programming.

Each year an excellent group of teaching assistants has worked with me on the course, which has enrollments as high as 350 students a semester. We owe much to these assistants both for their administrative help and for their ingenuity in handling the diverse needs of the students. The head teaching assistants in the past few years were Stuart Madnick, Steven Zilles, Richard Mulhern, William Michels, and Glen Brunk, but many others helped and contributed in countless ways, especially in problem sets. To name a few: David Quimby, Jerry Johnson, Michael Hammer, Benjamin Ashton, Orville Dodson, Steven Halfich, Norman Kohn, Martin Jack, Al Moulton, Chander Ramchandani, and Harris Berman. Mrs. Muriel Webber did the diagrams, and Max Byer contributed in the administration of the course at M.I.T.

A tremendous team effort by the following individuals has given this book its quality: William Michels contributed greatly to the compiler chapter; Norman Kohn to the macro and formal systems chapters and to the compilation of the references; Sterling Eanes proofread the formal systems and compilers chapters, and David Quimby assisted in assembling the problems and commenting on early drafts of the book. Leonard Goodman proofread the entire book–I thank him for the errors that are not here. Mr. David M. Jones edited the manuscript. Ellen Nangle supervised the typing and preparation of the manuscript–she cared. She further edited the manuscript, problems, solutions and teacher's manual.

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01), and in part by the Electrical Engineering Department of M.I.T.

Finally–and I hesitate to say this because it seems like a formality when I read it, which it is not–I simply and sincerely thank my wife and five children. I thank them for their patience, support, and endurance of my working essentially twenty-four hours a day, seven days a week for the past two years and for the long hours previous to that in preparation of this manuscript.

John J. Donovan

note to the student

This book does not presuppose extensive knowledge of assembly language although students with some experience in this language will find the material easier to approach. Chapter 2 contains basic information on assembly-language programming, Chapter 6 on high level languages, PL/I in particular. Students may wish to refer to additional sources on assembly language programming and PL/I.

Chapters 8 and 9 (Compilers and Operating Systems) are long and dense. (Entire courses have been built around the material in these chapters.) While both these chapters, like all the others in the book, are logical entities, we have divided them, for your convenience, into parts that are more manageable.

To be competent in any technical field, and particularly in one developing as rapidly as that of computer science, it is important to be familiar with the current literature. Chapter 10 contains references. Use them.

The book does presuppose familiarity with programming. The inexperienced reader should not therefore be discouraged if he finds the material difficult because of its depth and density.

This material is basic to computer science, and we believe that the student who explores it in depth will find it both relevant and exciting. I know of no other formal way of acquiring this material.

systems programming

1

background

This book has two major objectives: to teach procedures for the design of software systems and to provide a basis for judgement in the design of software. To facilitate our task, we have taken specific examples from systems programs. We discuss the design and implementation of the major system components.

What is systems programming? You may visualize a computer as some sort of beast that obeys all commands. It has been said that computers are basically people made out of metal or, conversely, people are computers made out of flesh and blood. However, once we get close to computers, we see that they are basically machines that follow very specific and primitive instructions.

In the early days of computers, people communicated with them by *on* and *off* switches denoting primitive instructions. Soon people wanted to give more complex instructions. For example, they wanted to be able to say $X = 30 * Y$; given that $Y = 10$, what is X? Present day computers cannot understand such language without the aid of systems programs. Systems programs (e.g., compilers, loaders, macro processors, operating systems) were developed to make computers better adapted to the needs of their users. Further, people wanted more assistance in the mechanics of preparing their programs.

Compilers are systems programs that accept people-like languages and translate them into machine language. Loaders are systems programs that prepare machine language programs for execution. Macro processors allow programmers to use abbreviations. Operating systems and file systems allow flexible storing and retrieval of information (Fig. 1.1).

There are over 100,000 computers in use now in virtually every application. The productivity of each computer is heavily dependent upon the effectiveness, efficiency, and sophistication of the systems programs.

In this chapter we introduce some terminology and outline machine structure and the basic tasks of an operating system.

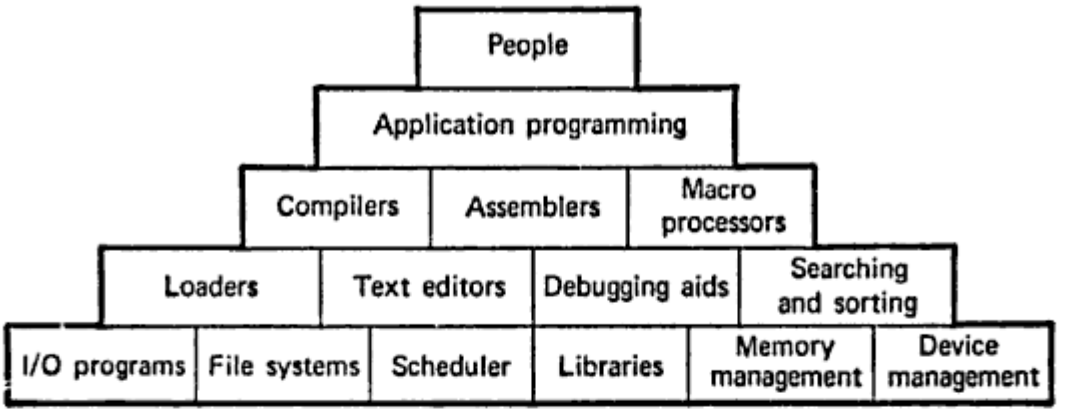


FIGURE 1.1 Foundations of systems programming

1.1 MACHINE STRUCTURE

We begin by sketching the general hardware organization of a computer system (Fig. 1.2).

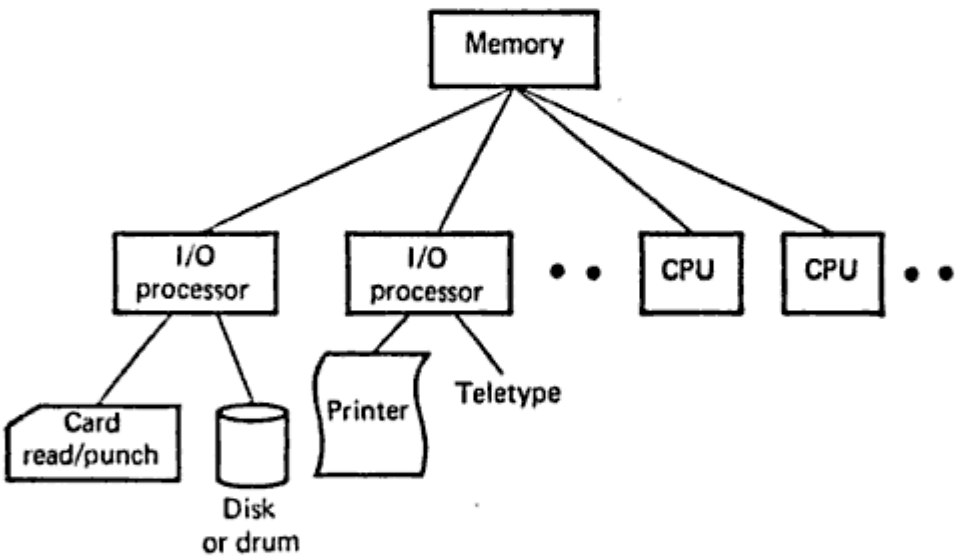


FIGURE 1.2 General hardware organization of a computer system

Memory is the device where information is stored. *Processors* are the devices that operate on this information. One may view information as being stored in the form of ones and zeros. Each one or zero is a separate binary digit called a *bit*. Bits are typically grouped in units that are called words, characters, or *bytes*. Memory locations are specified by *addresses*, where each address identifies a specific byte, word, or character.

The contents of a word may be interpreted as *data* (values to be operated on) or *instructions* (operations to be performed). A processor is a device that performs a sequence of operations specified by instructions in memory. A *program* (or procedure) is a sequence of instructions.

Memory may be thought of as mailboxes containing groups of ones and zeros. Below we depict a series of memory locations whose addresses are 10,000 through 10,002.

Address	*Contents*
10,000	0000 0000 0000 0001
10,001	0011 0000 0000 0000
10,002	0000 0000 0000 0100

An IBM 1130 processor treating location 10,001 as an instruction would interpret its contents as a "halt" instruction. Treating the same location as numerical data, the processor would interpret its contents as the binary number 0011 0000 0000 0000 (decimal 12,288). Thus instructions and data share the same storage medium.

Information in memory is coded into groups of bits that may be interpreted as characters, instructions, or numbers. A *code* is a set of rules for interpreting groups of bits, e.g., codes for representation of decimal digits (BCD), for characters (EBCDIC, or ASCII), or for instructions (specific processor operation codes). We have depicted two types of processors: *Input/Output* (I/O) processors and *Central Processing Units* (CPUs). The I/O processors are concerned with the transfer of data between memory and peripheral devices such as disks, drums, printers, and typewriters. The CPUs are concerned with manipulations of data stored in memory. The I/O processors execute I/O instructions that are stored in memory; they are generally activated by a command from the CPU. Typically, this consists of an "execute I/O" instruction whose argument is the address of the start of the I/O program. The CPU interprets this instruction and passes the argument to the I/O processor (commonly called I/O channels).

The I/O instruction set may be entirely different from that of the CPU and may be executed *asynchronously* (simultaneously) with CPU operation. Asynchronous operation of I/O channels and CPUs was one of the earliest forms of *multiprocessing*. Multiprocessing means having more than one processor operating on the same memory simultaneously.

Since instructions, like data, are stored in memory and can be treated as data, by changing the bit configuration of an instruction – adding a number to it – we may change it to a different instruction. Procedures that modify themselves are

called *impure* procedures. Writing such procedures is poor programming practice. Other programmers find them difficult to read, and moreover they cannot be shared by multiple processors. Each processor executing an impure procedure modifies its contents. Another processor attempting to execute the same procedure may encounter different instructions or data. Thus, impure procedures are not readily reusable. A *pure* procedure does not modify itself. To ensure that the instructions are the same each time a program is used, pure procedures (*re-entrant code*) are employed.

1.2 EVOLUTION OF THE COMPONENTS OF A PROGRAMMING SYSTEM

1.2.1 Assemblers

Let us review some aspects of the development of the components of a programming system.

At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of ones and zeros (machine language), place them into the memory of the machine, and press a button, whereupon the computer would start to interpret them as instructions.

Programmers found it difficult to write or read programs in machine language. In their quest for a more convenient language they began to use a *mnemonic* (symbol) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an *assembly language*. Programs known as *assemblers* were written to automate the translation of assembly language into machine language. The input to an assembler program is called the *source program*; the output is a machine language translation (*object program*).

1.2.2 Loaders

Once the assembler produces an object program, that program must be placed into memory and executed. It is the purpose of the loader to assure that object programs are placed in memory in an executable form.

The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be executed.

However, this would waste core[1] by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome the problems of wasted translation time and wasted memory, systems programmers developed another component, called the loader.

A *loader* is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary storage device and a loader is placed in core. The loader places into memory the machine language version of the user's program and transfers control to it. Since the loader program is much smaller than the assembler, this makes more core available to the user's program.

The realization that many users were writing virtually the same programs led to the development of "ready-made" programs (packages). These packages were written by the computer manufacturers or the users. As the programmer became more sophisticated, he wanted to mix and combine ready-made programs with his own. In response to this demand, a facility was provided whereby the user could write a main program that used several other programs or subroutines. A *subroutine* is a body of computer instructions designed to be used by other routines to accomplish a task. There are two types of subroutines: closed and open subroutines. An *open subroutine* or *macro definition* is one whose code is inserted into the main program (flow continues). Thus if the same open subroutine were called four times, it would appear in four different places in the calling program. A *closed subroutine* can be stored outside the main routine, and control transfers to the subroutine. Associated with the closed subroutine are two tasks the main program must perform: transfer of control and transfer of data.

Initially, closed subroutines had to be loaded into memory at a specific address. For example, if a user wished to employ a square root subroutine, he would have to write his main program so that it would transfer to the location assigned to the square root routine (SQRT). His program and the subroutine would be assembled together. If a second user wished to use the same subroutine, he also would assemble it along with his own program, and the complete machine language translation would be loaded into memory. An example of core allocation under this inflexible loading scheme is depicted in Figure 1.3, where core is depicted as a linear array of locations with the program areas shaded.

---

[1]Main memory is typically implemented as magnetic cores; hence *memory* and *core* are used synonymously.

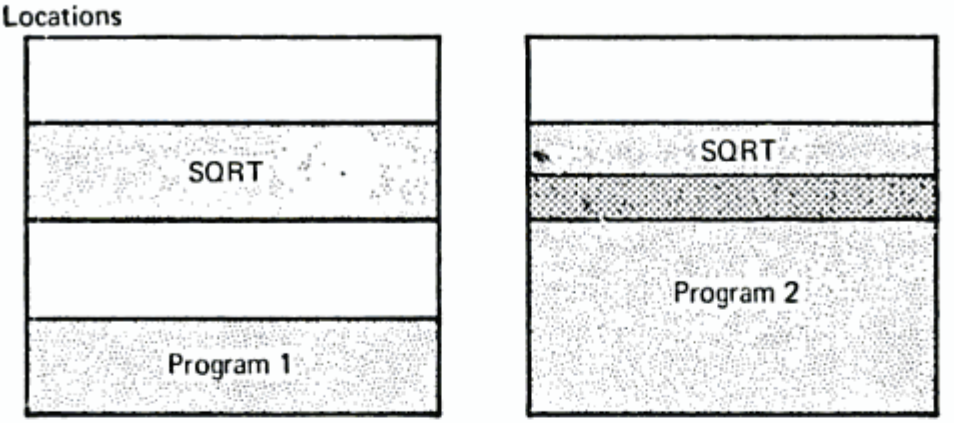


FIGURE 1.3 Example core allocation for absolute loading

Note that program 1 has "holes" in core. Program 2 *overlays* and thereby destroys part of the SQRT subroutine.

Programmers wished to use subroutines that referred to each other symbolically and did not want to be concerned with the address of parts of their programs. They expected the computer system to assign locations to their subroutines and to substitute addresses for their symbolic references.

Systems programmers noted that it would be more efficient if subroutines could be translated into an object form that the loader could "relocate" directly behind the user's program. The task of adjusting programs so they may be placed in arbitrary core locations is called *relocation*. *Relocating* loaders perform four functions:

1. Allocate space in memory for the programs (*allocation*)
2. Resolve symbolic references between object decks (*linking*)
3. Adjust all address-dependent locations, such as address constants, to correspond to the allocated space (*relocation*)
4. Physically place the machine instructions and data into memory (*loading*).

The various types of loaders that we will discuss ("compile-and-go," absolute, relocating, direct-linking, dynamic-loading, and dynamic-linking) differ primarily in the manner in which these four basic functions are accomplished.

The period of execution of a user's program is called *execution time*. The period of translating a user's source program is called *assembly* or *compile time*. *Load time* refers to the period of loading and preparing an object program for execution.

1.2.3 Macros

To relieve programmers of the need to repeat identical parts of their program,

operating systems provide a macro processing facility, which permits the programmer to define an abbreviation for a part of his program and to use the abbreviation in his program. The macro processor treats the identical parts of the program defined by the abbreviation as a *macro definition* and saves the definition. The macro processor substitutes the definition for all occurrences of the abbreviation (macro call) in the program.

In addition to helping programmers abbreviate their programs, macro facilities have been used as general text handlers and for specializing operating systems to individual computer installations. In specializing operating systems (systems generation), the entire operating system is written as a series of macro definitions. To specialize the operating system, a series of macro calls are written. These are processed by the macro processor by substituting the appropriate definitions, thereby producing all the programs for an operating system.

1.2.4 Compilers

As the user's problems became more categorized into areas such as scientific, business, and statistical problems, specialized languages (*high level languages*) were developed that allowed the user to express certain problems concisely and easily. These high level languages – examples are FORTRAN, COBOL, ALGOL, and PL/I – are processed by compilers and interpreters. A *compiler* is a program that accepts a program written in a high level language and produces an object program. An *interpreter* is a program that appears to execute a source program as if it were machine language. The same name (FORTRAN, COBOL, etc.) is often used to designate both a compiler and its associated language.

Modern compilers must be able to provide the complex facilities that programmers are now demanding. The compiler must furnish complex accessing methods for pointer variables and data structures used in languages like PL/I, COBOL, and ALGOL 68. Modern compilers must interact closely with the operating system to handle statements concerning the hardware interrupts of a computer (e.g. conditional statements in PL/I).

1.2.5 Formal Systems

A formal system is an uninterpreted calculus. It consists of an alphabet, a set of words called axioms, and a finite set of relations called rules of inference. Examples of formal systems are: set theory, boolean algebra, Post systems, and Backus Normal Form. Formal systems are becoming important in the design, implementation, and study of programming languages. Specifically, they can be

used to specify the *syntax* (form) and the semantics (meaning) of programming languages. They have been used in syntax-directed compilation, compiler verification, and complexity studies of languages.

1.3 EVOLUTION OF OPERATING SYSTEMS

Just a few years ago a FORTRAN programmer would approach the computer with his source deck in his left hand and a green deck of cards that would be a FORTRAN compiler in his right hand. He would:

1. Place the FORTRAN compiler (green deck) in the card hopper and press the load button. The computer would load the FORTRAN compiler.
2. Place his source language deck into the card hopper. The FORTRAN compiler would proceed to translate it into a machine language deck, which was punched onto red cards.
3. Reach into the card library for a pink deck of cards marked "loader," and place them in the card hopper. The computer would load the loader into its memory.
4. Place his newly translated object deck in the card hopper. The loader would load it into the machine.
5. Place in the card hopper the decks of any subroutines which his program called. The loader would load these subroutines.
6. Finally, the loader would transfer execution to the user's program, which might require the reading of data cards.

This system of multicolored decks was somewhat unsatisfactory, and there was strong motivation for moving to a more flexible system. One reason was that valuable computer time was being wasted as the machine stood idle during card-handling activities and between jobs. (A *job* is a unit of specified work, e.g., an assembly of a program.) To eliminate this waste, the facility to *batch* jobs was provided, permitting a number of jobs to be placed together into the card hopper to be read. A *batch operating system* performed the task of batching jobs. For example the batch system would perform steps 1 through 6 above retrieving the FORTRAN compiler and loader from secondary storage.

As the demands for computer time, memory, devices, and files increased, the efficient management of these resources became more critical. In Chapter 9 we discuss various methods of managing them. These resources are valuable, and inefficient management of them can be costly. The management of each resource has evolved as the cost and sophistication of its use increased.

In simple batched systems, the memory resource was allocated totally to a

single program. Thus, if a program did not need the entire memory, a portion of that resource was wasted. Multiprogramming operating systems with *partitioned core memory* were developed to circumvent this problem. *Multiprogramming* allows multiple programs to reside in separate areas of core at the same time. Programs were given a fixed portion of core (*Multiprogramming with Fixed Tasks* (MFT)) or a varying-size portion of core (*Multiprogramming with Variable Tasks* (MVT)).

Often in such partitioned memory systems some portion could not be used since it was too small to contain a program. The problem of "holes" or unused portions of core is called *fragmentation*. Fragmentation has been minimized by the technique of relocatable partitions (Burroughs 6500) and by paging (XDS 940, HIS 645). *Relocatable partitioned core* allows the unused portions to be condensed into one continuous part of core.

Paging is a method of memory allocation by which the program is subdivided into equal portions or pages, and core is subdivided into equal portions or *blocks*. The pages are loaded into blocks.

There are two paging techniques: simple and demand. In *simple paging* all the pages of a program must be in core for execution. In *demand paging* a program can be executed without all pages being in core, i.e., pages are fetched into core as they are needed (demanded).

The reader will recall from section 1.1 that a system with several processors is termed a multiprocessing system. The *traffic controller* coordinates the processors and the processes. The resource of processor time is allocated by a program known as the *scheduler*. The processor concerned with I/O is referred to as the *I/O processor*, and programming this processor is called *I/O programming*.

The resource of files of information is allocated by the *file system*. A *segment* is a group of information that a user wishes to treat as an entity. *Files* are segments. There are two types of files: (1) directories and (2) data or programs. *Directories* contain the locations of other files. In a hierarchical file system, directories may point to other directories, which in turn may point to directories or files.

Time-sharing is one method of allocating processor time. It is typically characterized by interactive processing and time-slicing of the CPU's time to allow quick response to each user.

A *virtual memory* (*name space, address space*) consists of those addresses that may be generated by a processor during execution of a computation. The *memory space* consists of the set of addresses that correspond to physical memory locations. The technique of *segmentation* provides a large name space and a good

protection mechanism. Protection and sharing are methods of allowing controlled access to segments.

1.4 OPERATING SYSTEM USER VIEWPOINT: FUNCTIONS

From the user's point of view, the purpose of an operating system (monitor) is to assist him in the *mechanics* of solving problems. Specifically, the following functions are performed by the system:

1. Job sequencing, scheduling, and traffic controller operation
2. Input/output programming
3. Protecting itself from the user; protecting the user from other users
4. Secondary storage management
5. Error handling

Consider the situation in which one user has a job that takes four hours, and another user has a job that takes four seconds. If both jobs were submitted simultaneously, it would seem to be more appropriate for the four-second user to have his run go first. Based on considerations such as this, job scheduling is automatically performed by the operating system. If it is possible to do input and output while simultaneously executing a program, as is the case with many computer systems, all these functions are scheduled by the traffic controller.

As we have said, the I/O channel may be thought of as a separate computer with its own specialized set of instructions. Most users do not want to learn how to program it (in many cases quite a complicated task). The user would like to simply say in his program, "Read," causing the monitor system to supply a program to the I/O channel for execution. Such a facility is provided by operating systems. In many cases the program supplied to the I/O channel consists of a sequence of closely interwoven interrupt routines that handle the situation in this way: "Hey, Mr. I/O Channel, did you receive that character?" "Yes, I received it." "Are you sure you received it?" "Yes, I'm sure." "Okay, I'll send another one." "Fine, send it." "You're sure you want me to send another one?" "*Send* it!"

An extremely important function of an operating system is to protect the user from being hurt, either maliciously or accidentally, by other users; that is, protect him when other users are executing or changing their programs, files, or data bases. The operating system must insure inviolability. As well as protecting users from each other, the operating system must also protect itself from users who, whether maliciously or accidentally, might "crash" the system.

Students are great challengers of protection mechanisms. When the systems

programming course is given at M.I.T., we find that due to the large number of students participating it is very difficult to personally grade every program run on the machine problems. So for the very simple problems – certainly the first problem which may be to count the number of A's in a register and leave the answer in another register – we have written a grading program that is included as part of the operating system. The grading program calls the student's program and transfers control to it. In this simple problem the student's program processes the contents of the register, leaves his answer in another register, and returns to the grading program. The latter checks to find out if the correct number has been left in the answer register. Afterwards, the grading program prints out a listing of all the students in the class and their grades. For example:

```
VITA KOHN          —  CORRECT
RACHEL BUXBAUM     —  CORRECT
JOE LEVIN          —  INCORRECT
LOFTI ZADEH        —  CORRECT
```

On last year's run, the computer listing began as follows:

```
JAMES ARCHER       —  CORRECT
ED MCCARTHY        —  CORRECT
ELLEN NANGLE       —  INCORRECT
JOHN SCHWARTZ      —  MAYBE
```

(We are not sure how John Schwartz did this; we gave him an A in the course.)

Secondary storage management is a task performed by an operating system in conjunction with the use of disks, tapes, and other secondary storage for a user's programs and data.

An operating system must respond to errors. For example, if the programmer should overflow a register, it is not economical for the computer to simply stop and wait for an operator to intervene. When an error occurs, the operating system must take appropriate action.

1.5 OPERATING SYSTEM USER VIEWPOINT: BATCH CONTROL LANGUAGE

Many users view an operating system only through the batch system control cards by which they must preface their programs. In this section we will discuss a simple monitor system and the control cards associated with it. Other more complex monitors are discussed in Chapter 9.

Monitor is a term that refers to the control programs of an operating system. Typically, in a batch system the jobs are stacked in a card reader, and the monitor system sequentially processes each job. A job may consist of several separate programs to be executed sequentially, each individual program being called a *job step*. In a *batch monitor system* the user communicates with the system by way of a control language. In a simple batch monitor system we have two classes of control cards: execution cards and definition cards. For example, an execution card may be in the following format:

```
// step name  EXEC   name of program to be executed, Argument 1, Argument 2
```

The job control card, a definition card, may take on the following format:

```
// job name  JOB      (User name, identification, expected time use, lines to
                      be printed out, expected number of cards to be printed
                      out.
```

Usually there is an end-of-file card, whose format might consist of /*, signifying the termination of a collection of data. Let us take the following example of a FORTRAN job.

```
//EXAMPLE    JOB      DONOVAN,    T168,1,100,0
//STEP1      EXEC     FORTRAN, NOPUNCH
           READ 9100,N
           DO 100 I = 1,N
           I2 = I*I
           I3 = I*I*I
      100  PRINT 9100, I, I2, I3
     9100  FORMAT (3I10)
           END
/*
//STEP2      EXEC LOAD
/*
//STEP3      EXEC OBJECT
        1D
/*
```

The first control card is an example of a definition card. We have defined the user to be Donovan. The system must set up an accounting file for the user, noting that he expects to use one minute of time, to output a hundred lines of output, and to punch no cards. The next control card, EXEC FORTRAN, NOPUNCH, is an example of an execution card; that is, the system is to execute the program FORTRAN, given one argument – NOPUNCH. This argument allows the monitor system to perform more efficiently; since no cards are to be punched, it need not utilize the punch routines. The data to the compiler is the FORTRAN program shown, terminated by an end-of-file card /*.

The next control card is another example of an execution card and in this

case causes the execution of the loader. The program that has just been compiled will be loaded, together with all the routines necessary for its execution, whereupon the loader will "bind" the subroutines to the main program. This job step is terminated by an end-of-file card. The EXEC OBJECT card is another execution card, causing the monitor system to execute the object program just compiled. The data card, 10, is input to the program and is followed by the end-of-file card.

The simple loop shown in Figure 1.4 presents an overview of an implementation of a batch monitor system. The monitor system must read in the first card, presumably a job card. In processing a job card, the monitor saves the user's name, account number, allotted time, card punch limit, and line print limit. If the next control card happens to be an execution card, then the monitor will load the corresponding program from secondary storage and process the job step by transferring control to the executable program. If there is an error during processing, the system notes the error and goes back to process the next job step.

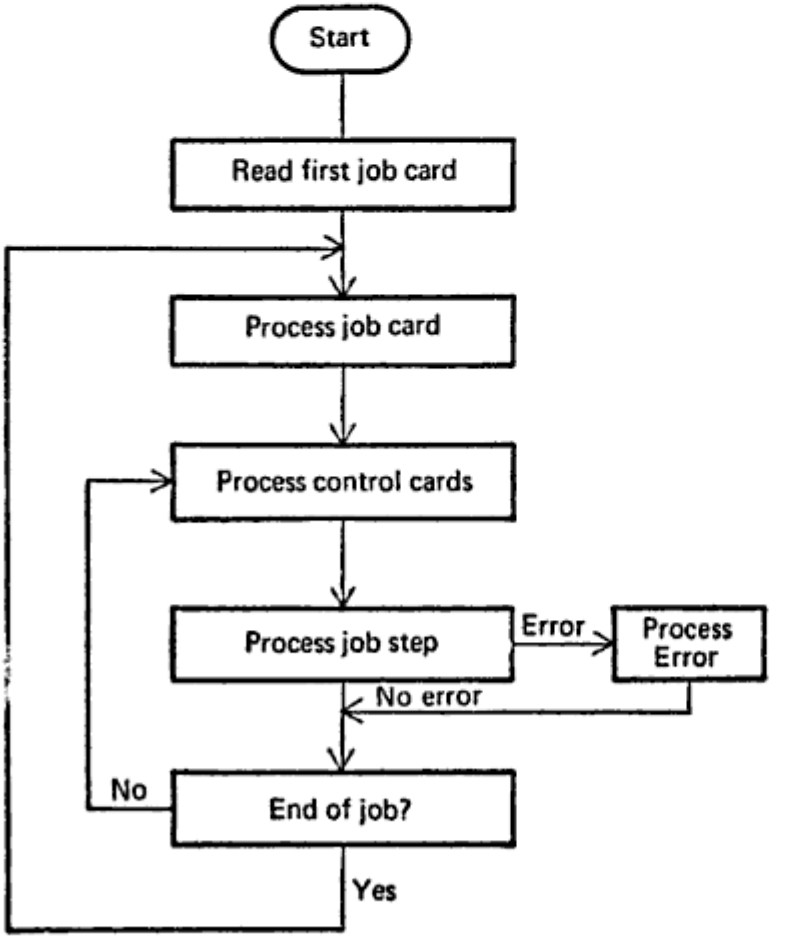


FIGURE 1.4 Main loop of a simple batch monitor system

1.6 OPERATING SYSTEM USER VIEWPOINT: FACILITIES

For the applications-oriented user, the function of the operating system is to provide facilities to help solve problems. The questions of scheduling or protection are of no interest to him; what he is concerned with is the available software. The following facilities are typically provided by modern operating systems:

1. Assemblers
2. Compilers, such as FORTRAN, COBOL, and PL/I
3. Subroutine libraries, such as SINE, COSINE, SQUARE ROOT
4. Linkage editors and program loaders that bind subroutines together and prepare programs for execution
5. Utility routines, such as SORT/MERGE and TAPE COPY
6. Application packages, such as circuit analysis or simulation
7. Debugging facilities, such as program tracing and "core dumps"
8. Data management and file processing
9. Management of system hardware

Although this "facilities" aspect of an operating system may be of great interest to the user, we feel that the answer to the question, "How many compilers does that operating system have?" may tell more about the orientation of the manufacturer's marketing force than it does about the structure and effectiveness of the operating system.

1.7 SUMMARY

The major components of a programming system are:

1. Assembler

Input to an assembler is an *assembly language program*. Output is an object program plus information that enables the loader to prepare the object program for execution.

2. Macro Processor

A *macro call* is an abbreviation (or name) for some code. A *macro definition* is a sequence of code that has a name (macro call). A *macro processor* is a program that substitutes and specializes macro definitions for macro calls.

3. Loader

A loader is a routine that loads an object program and prepares it for execution.

There are various loading schemes: absolute, relocating, and direct-linking. In general, the loader must *load*, *relocate*, and *link* the object program.

4. Compilers

A compiler is a program that accepts a source program "in a high-level language" and produces a corresponding object program.

5. Operating Systems

An operating system is concerned with the allocation of resources and services, such as memory, processors, devices, and information. The operating system correspondingly includes programs to manage these resources, such as a *traffic controller*, a *scheduler*, *memory management module*, *I/O programs*, and a *file system*.

QUESTIONS

1. What is the difference between (processor, procedure); (procedure, program); (processor, I/O channel); (multiprocessing, multiprogramming); and (open subroutine, closed subroutine)?

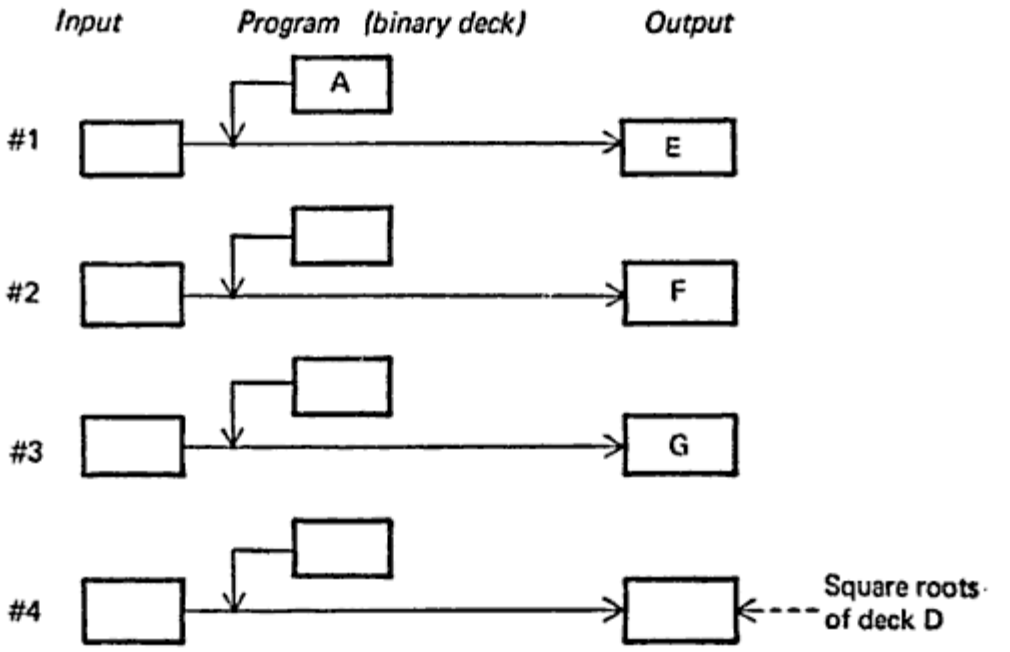
2. Bits in memory may represent data or instructions. How does the processor "know" whether a given location represents an instruction or a piece of data?

3. Assume that you have available to you a 360-type computer and the following available input decks:

   Deck A: A Basic Assembly Language (BAL) assembler written in *binary* code (machine language)
   Deck B: A FORTRAN to BAL translator, written in BAL
   Deck C: A FORTRAN program that will read data cards and print the square roots of the data
   Deck D: A data deck for the square root program of deck C

   In order to compute the square roots you will have to make four computer runs. The runs are described schematically below. Each run involves (reading from left to right) an input deck that will be operated on by a program to produce an output deck. Of course, the output deck produced by one run may be used as either the input deck or the program of a subsequent run.

   In the figure below identify the unlabelled decks with the letters A, B, C, D, E, F, G.



4. There is a distinction between hardware (physical devices made of nuts, bolts, transistors, etc.) and software (information stored as a binary pat-

tern on cards, tape, disc, with the ultimate purpose of being loaded into core memory to be used as a program or data). Label each of the following as hardware or software:

a. Compiler
b. Processor
c. Operating system
d. Loader
e. I/O channel
f. Core memory
g. Assembler
h. File
i. Monitor
j. Disk drive

5. A simple batch monitor was discussed in section 1.5. In this problem, we will increase its flexibility and look more closely at its structure. To give the user more control over resource allocation, we introduce the concept of a *data set*; a data set is simply a source or repository for data, which can take the physical form of a disk, printer, card reader, etc. The user defines his data sets by means of a new control card:

```
// Logical data set name  DD  parameter list
```

For instance, in the FORTRAN compilation of section 1.5, SYSPRINT is the logical name for the source language listing data set, and

```
// SYSPRINT  DD  UNIT = 00E
```

says that the listing is to be made on the device numbered 00E (printer). Other data sets used by the compiler are:

```
SYSIN     (Source language input; usually card reader)
SYSLIN    (Object code output; usually disk)
```

The loader would use:

```
SYSLIN    (Output of translator)
SYSPRINT  (Messages--usually printer)
SYSLIB    (Library subroutines; usually on disk)
```

And the user program:

```
SYSIN     (Data, usually from card reader)
SYSPRINT  (Printer, Same as above)
```

To include data cards in the same deck as control cards, an asterisk (*) is put in the parameter field of the DD card, meaning "follows in input stream." The data cards must be ended with a /* (end of data set) card,

and a DD * data set must be the last data set in a job step.

On a system with a disk numbered 141 and printer numbered 00E, the job from section 1.5 with all data sets defined would be:

```
// EXAMPLE      JOB      DONOVAN,T168,1,100,0
// STEP1        EXEC     FORTRAN,NO PUNCH
// SYSPRINT     DD       UNIT = 00E
// SYSLIN       DD       UNIT = 141
// SYSIN        DD       *
         READ       9100,N
         DO    100  I=1,N
         I2 = I*I
         I3 = I*I*I
100      PRINT      9100,I,I2,I3
9100     FORMAT     (3I10)
         END
/*
// STEP2        EXEC     LOADER
// SYSPRINT     DD       UNIT = 00E
// SYSLIB       DD       DSNAME = FORTLIB
// SYSLIN       DD       FROM STEP1.SYSLIN
// STEP3        EXEC     OBJECT
// SYSPRINT     DD       UNIT = 00E
// SYSIN        DD       *
         10
/*
```

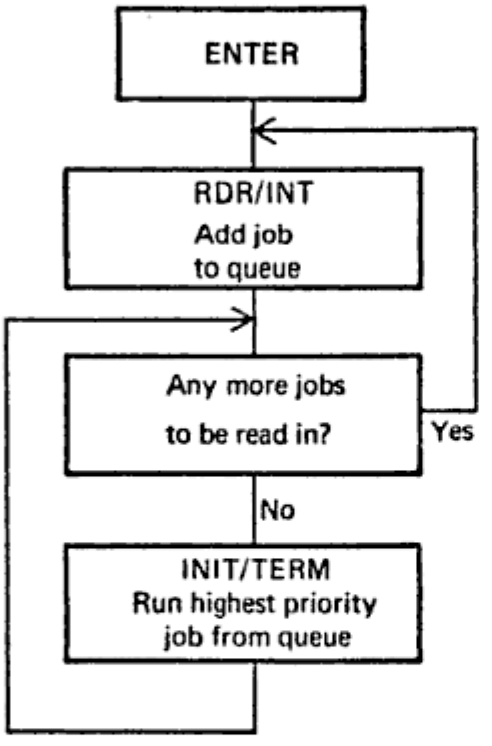
The monitor must read the user's control cards, allocate resources as requested, and initiate execution of the requested job steps. Consider it to be broken down into two sections:

1) The reader/interpreter (RDR/INT) reads control cards, interprets them, and builds data bases from the information on the control cards.
2) The initiator/terminator (INIT/TERM) schedules resources and initiates and terminates job steps, using the data bases created by the RDR/INT.

a. Why is it useful to have logical names that can be assigned by the user to any physical unit he chooses?
b. What data base(s) must the RDR/INT build for the INIT/TERM to support the DD control card feature?
c. What other data base(s) are needed?
d. As the RDR/INT reads cards sequentially, at what point does it transfer control to the INIT/TERM to actually perform the job step? (Remember that data cards in a DD * data set are not read by the RDR/INT, but as data by the program being executed.)
e. What would happen to each data base of the RDR/INT at the end of each job step?
f. What would happen to each data base at the end of each job?
g. What errors might be recognized by RDR/INT? By INIT/TERM? What action could be taken?

6. To increase the flexibility of our batch monitor system even more, we want to add a new parameter called PRIORITY to the JOB card. This would be a number from 0 to 9, with higher numbers being run first and, of course, charged more. Obviously, we can no longer read in one job at a time, so we add a facility called a *queue.*

   The data bases created by the RDR/INT for each job will be placed on the queue until there are no more jobs to be read, and then INIT/TERM will be allowed to run the highest priority job. A check will be made afterwards to see whether any new jobs must be added to the queue, and then the remaining job with the highest priority will be run:



   a. What addition(s) must be made to the data bases created by the RDR/INT?
   b. In the model of section 1.5 we could ignore data cards in DD * data sets. What, if anything, is done about them now?
   c. Why is the PRIORITY parameter useful?

2

machine structure, machine language, and assembly language

The purpose of this chapter is to discuss machine structure, machine language, and assembly language.

We have taken examples from the IBM Systems/360 and 370.[1] Our purpose is not to teach specific assembly languages, and we present only enough material to illustrate the design of assemblers (and later the design of compilers). The introduction to 370 assembly language afforded by our discussion should be supplemented by further reading (see Chapter 10 (References) – machine structure). We have written this section primarily for two classes of people: those who know assembly language programming well and want to become somewhat familiar with the 370; and those who have not programmed in any assembly language and who may use this chapter as an introduction to the manuals. The approach and examples can be easily translated to other machines.

2.1 GENERAL MACHINE STRUCTURE

Almost all conventional modern computers are based upon the "stored program computer" concept, generally credited to the mathematician John von Neumann (1903-1957). Figure 2.1 illustrates the structure of the CPU for a typical von Neumann machine, such as the IBM System/360.

The CPU consists of an instruction interpreter, a location counter, an instruction register and various working registers and general registers. The *instruction interpreter* is a group of electrical circuits (*hardware*), that performs the intent of instructions fetched from memory. The *Location Counter* (LC), also called

[1]The IBM System/360 (or just 360) is the name of a series of IBM computers in production since 1964, all of which have compatible instruction sets and manuals. The IBM 370, a revised version of the 360, was introduced in 1970. The *370 is compatible with the 360.*

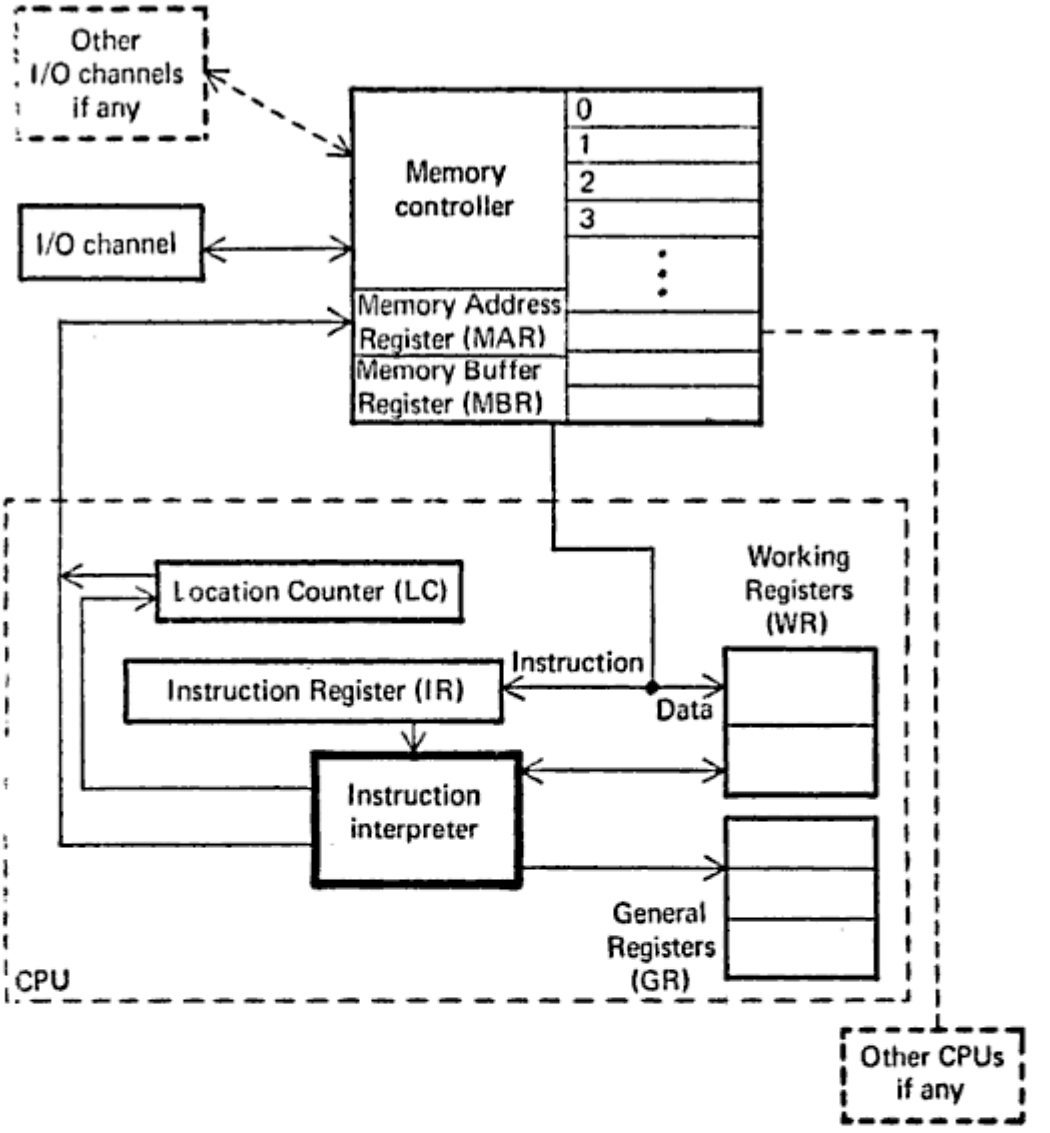


FIGURE 2.1 General machine structure

Program Counter (PC) or *Instruction Counter* (IC), is a hardware memory device which denotes the location of the current instruction being executed. A copy of the current instruction is stored in the *Instruction Register* (IR). The *working registers* are memory devices that serve as "scratch pads" for the instruction interpreter, while the *general registers* are used by the programmer as storage locations and for special functions.

The primary interface between the memory and the CPU is via the memory address register and the memory buffer register. The *Memory Address Register* (MAR) contains the address of the memory location that is to be read from or stored into. The *Memory Buffer Register* (MBR) contains a copy of the designated memory location specified by the MAR after a "read," or the new contents of the memory location prior to a "write." The *memory controller* is hardware

that transfers data between the MBR and the core memory location the address of which is in the MAR.

The *I/O channels* may be thought of as separate computers which interpret special instructions for inputting and outputting information from the memory.

To illustrate how these components of the machine structure interact, let us consider a simple computer (SC-6251). The SC-6251 has four general registers, designated 00, 01, 10, and 11 in binary rotation. The basic instruction format is as follows:

Operation code	Register number	Memory location
(op)	(reg)	(addr)

For example, the instruction

```
ADD     2,176
```

would cause the data stored in memory location 176 to be added to the current contents of general register 2. The resulting sum would be left as the new contents of register 2. The micro-flowchart in Figure 2.2 illustrates the sequence of hardware operations performed within the instruction interpreter to execute an instruction.

Although the specific details vary from computer to computer, this example of machine structure is representative of all conventional machines.

2.1.1 General Approach to a New Machine

Outlined in this section is an approach that may be taken to become familiar with a new machine. It consists of finding answers to a series of questions that we ask if we wish to program the machine.

We first list these questions and then answer them for the IBM 360 and 370. In section 9.1 we will ask these same questions regarding I/O channels (which may be considered as separate computers).

1. MEMORY

What is the memory's basic unit, size, and addressing scheme?

2. REGISTERS

How many registers are there? What are their size, function, and interrelationship?

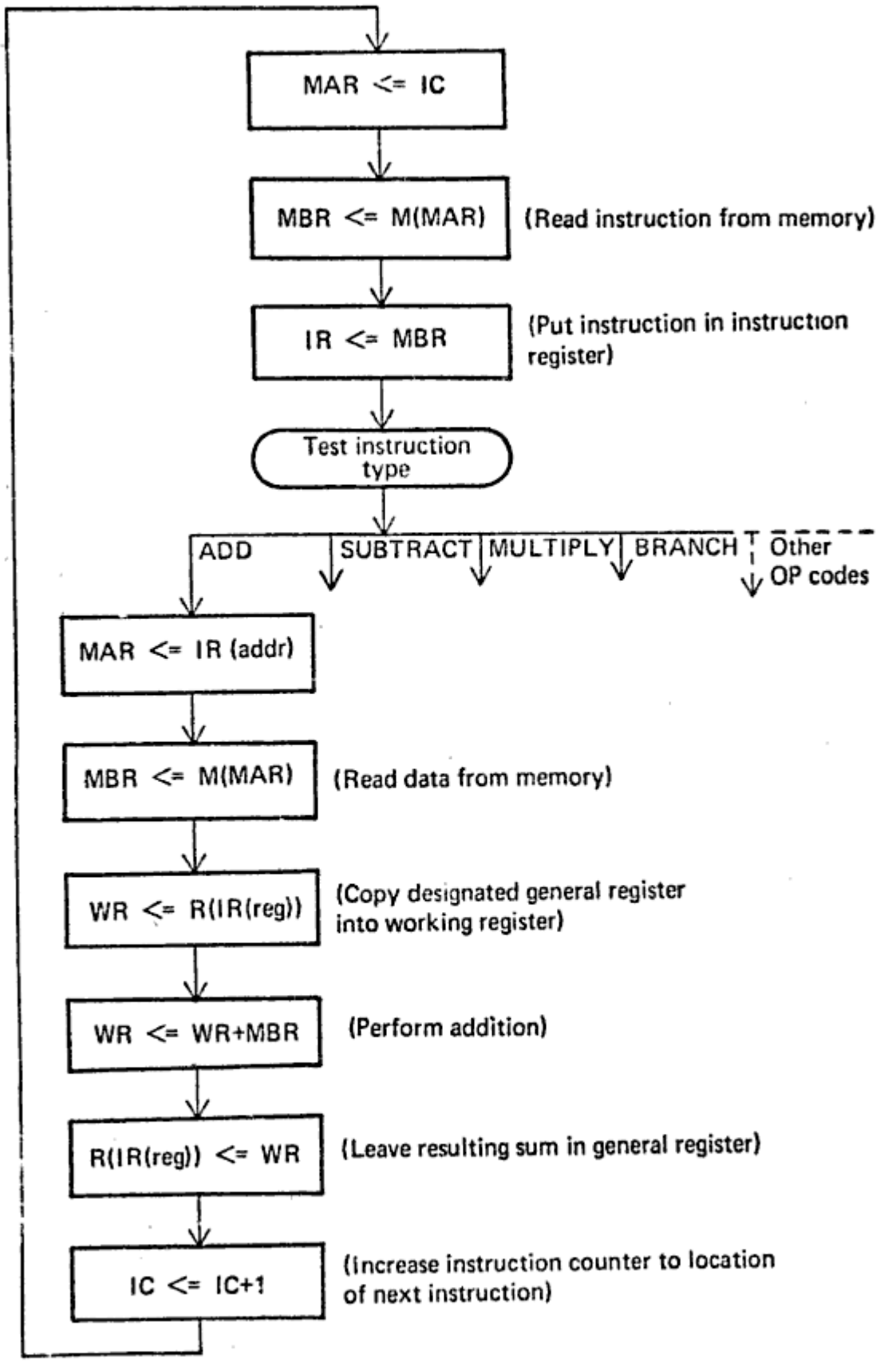


FIGURE 2.2 Example micro flowchart for ADD instruction

3. DATA

What types of data can be handled by the computer? Can it handle characters, numbers, logical data? How is this data stored?

4. INSTRUCTIONS

What are the classes of instructions on the machine? Are there arithmetic instructions, logical instructions, symbol-manipulation instructions? What are their formats? How are they stored in memory?

5. SPECIAL FEATURES

What is the interrupt structure of the machine? What sort of protection mechanism is available to the user?

2.1.2 Machine Structure – 360 and 370

In this section we will answer these questions in the context of the IBM 360. The material is equally applicable to the 370.

1. MEMORY

The basic unit of memory in the 360 is a byte – eight bits of information. That is, each addressable position in memory can contain eight bits of information. There are facilities to operate on contiguous bytes in basic units. The basic units are as follows:

Unit of memory	*Bytes*	*Length in bits*
Byte	1	8
Halfword	2	16
Word	4	32
Doubleword	8	64

A unit of memory consisting of four bits is sometimes referred to as a *nibble*

The size of the 360 memory is up to 2^{24} bytes (about sixteen million).

The addressing on the 360 memory may consist of three components. Specifically, the value of an address equals the value of an offset, plus the contents of a base register, plus the contents of an index register. We will give examples of this addressing later.

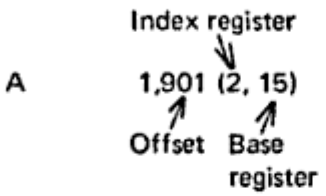
In general, operations on units of memory are specified by the low-order byte address. For example, when addressing a word (four bytes) the address of the word is that of the low order byte.

2. REGISTERS

The 360 has 16 general-purpose registers consisting of 32 bits each. In addition there are 4 floating-point registers consisting of 64 bits each. It has a 64-bit

Program Status Word (PSW) that contains the value of the location counter, protection information, and interrupt status.

The general-purpose registers may be used for various arithmetic and logical operations and as base registers. When the programmer uses them in arithmetic or logical operations, he thinks of these registers as scratch pads to which numbers are added, subtracted, compared, and so forth. When used as base registers, they aid in the formation of the address. Take for example the instruction



It is interpreted as an add instruction. A number is to be added to the contents of register 1.

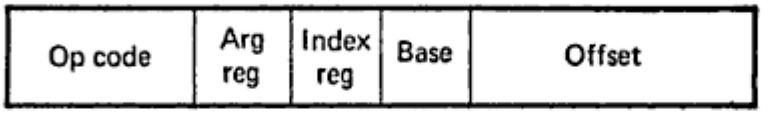
The location of the number is 901 (*offset*) plus the contents of register 2 (*index*) plus the contents of register 15 (*base*). That is, if those three numbers were added together, the result would be the address of the memory location whose contents we wish to add to the contents of register 1.

One may ask why such complexity in the formation of addressing is necessary. The motivation is twofold. First, a base register aids in the process of relocation of a program. As we will see, an entire program may be moved from one series of locations to another by changing the contents of a base register.[2] A major motivation for employing base registers, however, is to promote efficient addressing of core. For example, in order to address all possible core locations (16 million) in the 360 without the use of a base register, we would need 24 bits for every address. By way of illustration, if the preceding add instruction were formed in core as depicted in the following diagram, we would need a total of 40 bits to store it: 8 bits for the op code, 4 bits to specify one of 16 possible registers to which the number is added, an additional 4 bits to specify one of 16 possible index registers, and lastly, 24 bits to specify the address of the number we wish to add

---

[2]Base registers do not completely solve the problem of relocation. The difficult problem of *address constants* must also be resolved. An address constant is a feature by which a programmer may specify that a certain location in memory contains an address of a specified memory location.

If we use a base register, we can store the instruction in the following format. We could specify any one of 16 possible registers as the base register, using 4 bits, and employ an additional 12 for an offset. The total number of bits for an add instruction would be 32, a savings of 8 bits per address reference.



The disadvantages of this shorter form are the overhead associated with the formation of the address during execution and the fact that the offset, which is 12 bits long, can only specify a number from 0 to 4,095. Thus, it may be difficult to "reach" the data. That is, without using an index register and without changing the contents of the base register, the core location we wish to address cannot be any further than 4,095 locations away from the core location to which the base register is pointing.

3. DATA

The 360 may store several different types of data as is depicted in Figure 2.3. That is, groups of bits stored in memory are interpreted by a 360 processor in several ways. If a 360 interprets the contents of two bytes as an integer (Fig. 2.3a), it interprets the first bit as a sign and the remaining 15 as a binary number (e.g., 0000 0010 0001 1101 is interpreted as the binary number equivalent to the decimal number +541).[3] If a 360 interprets the contents of two bytes as a packed decimal (Fig. 2.3c), it would interpret the first byte as two BCD coded digits, the first four bits of the second byte as a BCD digit, and the last four as a sign (e.g., $\underbrace{0000}_{0}\ \underbrace{0010}_{2}\ \underbrace{0001}_{1}\ \underbrace{1101}_{\text{Sign}}$ is interpreted as the decimal number -021).

All data and instructions are physically stored as sequences of binary ones and zeros. Thus, a 16-bit fixed-point halfword with decimal value +300 would be

[3]See Appendix A for binary to decimal conversions.

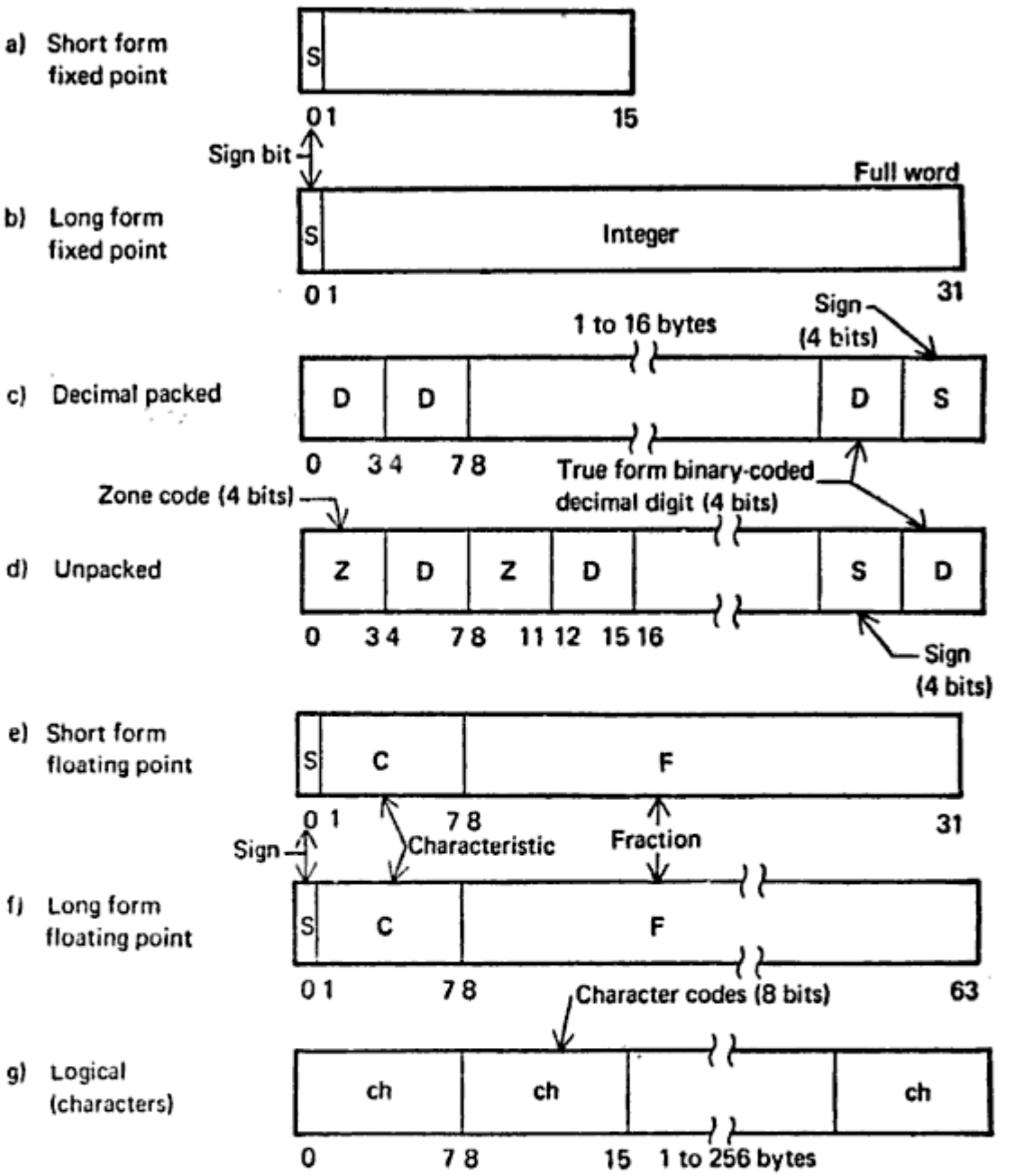


FIGURE 2.3 Data formats for the system/360 and 370

stored in binary as '0000 0001 0010 1100'. For convenience, binary numbers are usually written in the hexadecimal (base 16) number system rather than the binary (base 2) number system. The *hexadecimal* digits are shown in Figure 2.4. Note that every hexadecimal digit can be replaced by exactly four binary digits and vice versa. Thus when we have the number +300 in decimal, which equals

B'0000 0001 0010 1100' in binary
X' 0 1 2 C in hexadecimal

Hexadecimal	*Binary*	*Decimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

FIGURE 2.4 Hexadecimal-binary-decimal conversion

The prefixes X and B indicate mode of representation (hexadecimal, binary).

Fixed-point numbers may be stored in either a halfword or a fullword (Figs. 2.3a and 2.3b).

The 360 allows the storage of numbers in decimal form (Figs. 2.3c and 2.3d). That is, numbers may be stored not as binary numbers but in a format closely approximating the decimal representation. For example, the number 12 could appear in one byte where the first four bits would contain a decimal 1 (0001) and the second four bits would contain a decimal 2 (0010). Decimal forms are useful in business data processing.

The 360 allows floating-point numbers, logical data, and character strings (Fig. 2.3g) to be represented in memory as depicted in Figure 2.3.

There are instructions to operate on all these types of data.

4. INSTRUCTIONS

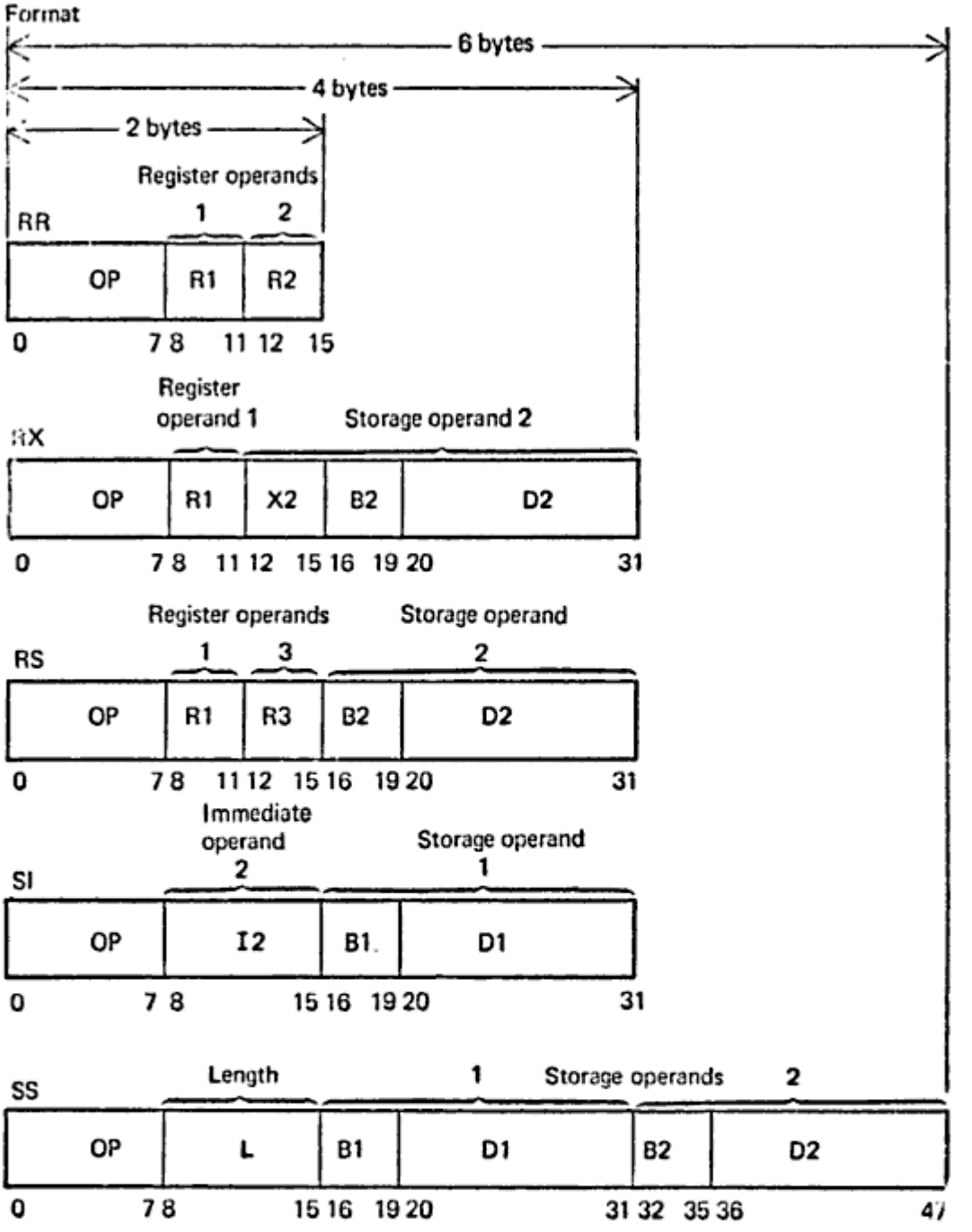
The 360 has arithmetic, logical, control or transfer, and special interrupt instructions.

The formats of the 360 instructions are depicted in Figure 2.5.

These five types of instructions differ basically in the types of operands they use.

Register operands refer to data stored in one of the 16 general registers (32 bits long), which are addressed by a four-bit field in the instruction. Since registers are usually constructed of high-speed circuitry, they provide faster access to data than does core storage.

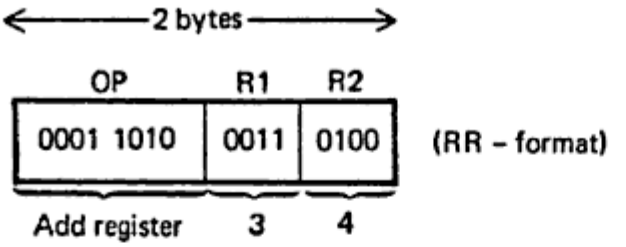
For example, the instruction *Add register 3, 4*



Mnemonics used:

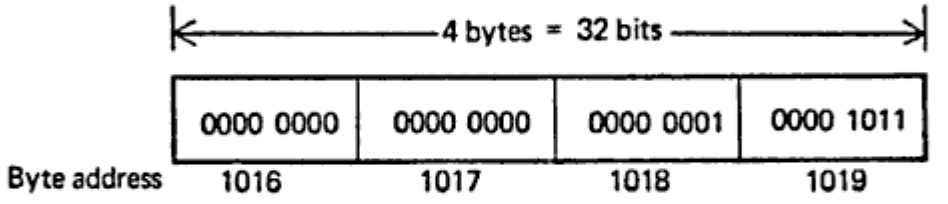
OP	~	operation code
Ri	~	contents of general register used as operand
Xi	~	contents of general register used as index
Bi	~	contents of general register used as base
Di	~	displacement
Ii	~	immediate data
L	~	operand length

FIGURE 2.5 Basic 360 instruction formats



causes the contents of general register 4 (32 bits) to be added to the contents of general register 3 (32 bits) and the resulting sum to be left in general register 3.

Storage operands refer to data stored in core memory. The length of the operand depends upon the specific data type (as illustrated in Figure 2.3). Operand data fields that are longer than one byte are specified by the address of the lowest-address byte (logically leftmost). For example, the 32-bit binary fixed-point full word with value +267 (in hexadecimal X'00 00 01 0B'), stored in locations 1016, 1017, 1018, and 1019 as depicted below is said to be "located at address 1016."

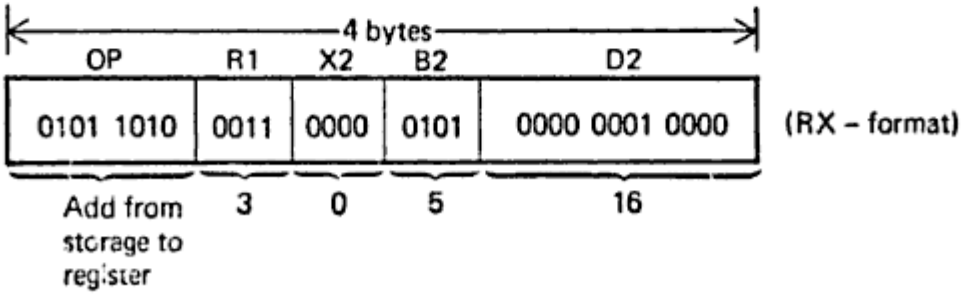


The address of the i^{th} storage operand is computed from the instruction in the following way:

Address = c(Bi) + c(Xi) + Di (RX format)
or c(Bi) + Di (RS, SI, SS format)

where c(Bi) and c(Xi) denote the contents of general registers Bi and Xi respectively. Exception: if Xi=0, then c(Xi) are treated as 0; likewise for Bi=0.

For example, if we assume that general register 5 contains the number 1000, the following instruction:

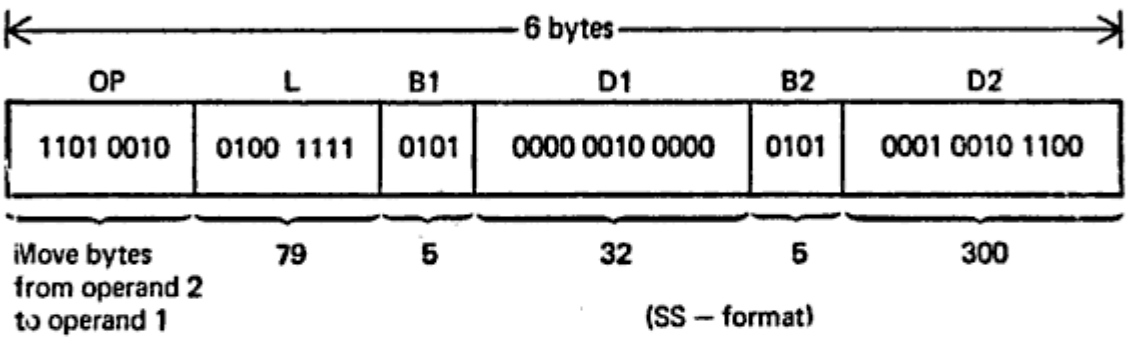


causes the contents of the word (32 bits) located at address

```
= c(B2) + c(X2) + D2
= c(5)  + c(0)  + 16
= 1000  + 0     + 16
= 1016
```

to be added to the contents of general register 3 (32 bits), with the resulting sum left in general register 3.

Another example, assuming again that general register 5 contains 1000, is the following instruction: (Note: in SS instructions the length is always one less than the data moved, e.g., length = 0 means move one byte.)



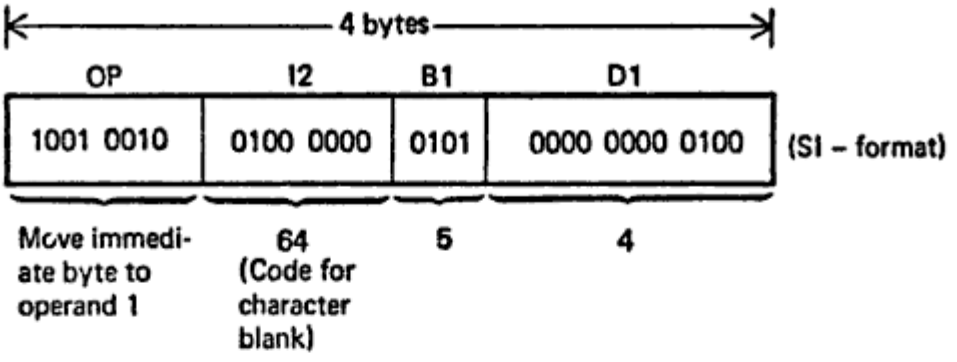
This instruction involves two storage operands:

```
Storage operand 1 address = c(B1) + D1 = c(5) + 32
                          = 1000  + 32 = 1032
Storage operand 2 address = c(B2) + D2 = c(5) + 300
                          = 1000  + 300 = 1300
```

This instruction copies (moves) the 80 bytes from locations 1032 - 1111 to locations 1300 - 1379. Since a character is stored as a byte (see Fig. 2.3g), this instruction could be viewed as copying an 80-character "card image" from one area to another.

Immediate operands are a single byte of data and are stored as part of the instruction.

Again assuming register 5 to contain 1000, the following SS instruction, causes the byte 0100 0000 (bits 8 through 15 of instruction) to be stored at location 1004.



Representative 360/370 instructions

Various 360 instructions will be used throughout this book in examples and as needed in problem sets and machine problem assignments. The following subset is particularly relevant to our purpose and should be studied in the appropriate reference manual. (See Appendix A for complete set of instructions.)

	Hexadecimal op code	*Mnemonic*	*Meaning (format)*
		Load group	
Load-store-register	58	L	Load (RX)
	48	LH	Load halfword (RX)
	98	LM	Load multiple (RS)
	18	LR	Load (RR)
	12	LTR	Load and test (RR)
		Store group	
	50	ST	Store (RX)
	40	STH	Store halfword (RX)
	90	STM	Store multiple (RS)
		Add-group	
Fixed-point arithmetic	5A	A	Add (RX)
	4A	AH	Add halfword (RX)
	1A	AR	Add (RR)
		Compare-group	
	59	C	Compare (RX)
	49	CH	Compare halfword (RX)
	19	CR	Compare (RR)
		Divide-group	
	5D	D	Divide (RX)
	1D	DR	Divide (RR)

	Hexadecimal op code	Mnemonic	Meaning (format)
		Multiply-group	
Fixed-point arithmetic	5C	M	Multiply (RX)
	4C	MH	Multiply halfword (RX)
	1C	MR	Multiply (RR)
		Subtract-group	
	5B	S	Subtract (RX)
	4B	SH	Subtract halfword (RX)
	1B	SR	Subtract (RR)
		Compare-group	
Logical	55	CL	Compare logical (RX)
	D5	CLC	Compare logical (SS)
	95	CLI	Compare logical (SI)
	15	CLR	Compare logical (RR)
		Move-group	
	D2	MVC	Move (SS)
	92	MVI	Move (SI)
		And-group	
	54	N	Boolean AND (RX)
	D4	NC	Boolean AND (SS)
	94	NI	Boolean AND (SI)
	14	NR	Boolean AND (RR)
		Or-group	
	56	O	Boolean OR (RX)
	D6	OC	Boolean OR (SS)
	96	OI	Boolean OR (SI)
	16	OR	Boolean OR (RR)
		Exclusive-or group	
	57	X	Exclusive-or (RX)
	D7	XC	Exclusive-or (SS)
	97	XI	Exclusive-or (SI)
	17	XR	Exclusive-or (RR)
		Shift	
	8D	SLDL	Shift left (double logical) (RS)
	89	SLL	Shift left (single logical) (RS)
	8C	SRDL	Shift right (double logical) (RS
	88	SRL	Shift right (single logical) (RS)

	Hexadecimal op code	Mnemonic	Meaning (format)
		Linkage group	
Transfer	45	BAL	Branch and link (RX)
	05	BALR	Branch and link (RR)
		Branch group	
	47	BC	Branch on condition (RX)
	07	BCR	Branch on condition (RR)
	46	BCT	Branch on count (RX)
	06	BCTR	Branch on count (RR)
		Miscellaneous	
Miscellaneous	9E	HIO	Halt I/O (RX)
	41	LA	Load address (RX)
	9C	SIO	Start I/O (RX)
	0A	SVC	Supervisor call (SI)
	9D	TIO	Test I/O (RX)
	43	IC	Insert character (RX)
	91	TM	Test under mask (SI)
	42	STC	Store character (RX)

5. SPECIAL FEATURES

The 360 has hardware protection in blocks of 2,048 bytes and has an elaborate interrupt structure discussed in Chapter 9.

2.2 MACHINE LANGUAGE

In this section we will discuss machine language (the actual code executed by a computer). Again, our examples are taken from a 360-type computer. However, they are easily applied to other machines.

In this section we will start the reader on his way to learning machine language. After reading this section, the reader is referred to one of the many books or manuals that discuss the machine language of the particular machine that he will be using.

We will not write machine language in ones and zeros, nor will we use hexadecimal numbers. Rather, we will use a mnemonic form of machine language.

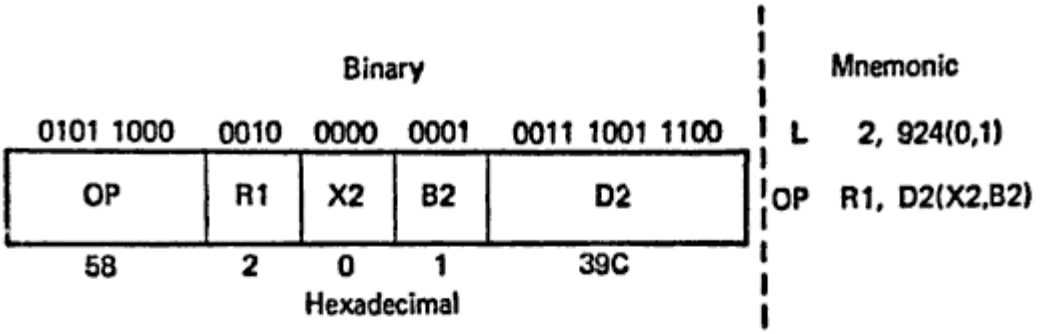


FIGURE 2.6 Mnemonic form of machine language

Figure 2.6 depicts a series of ones and zeros that may be interpreted by the CPU as a load instruction, and the mnemonic form that we shall employ to represent this instruction.

The following simple example will be used several times in this chapter to demonstrate features of machine language:

Write a program that will add the number 49 to the contents of 10 adjacent fullwords in memory, under the following set of assumptions:

Assumption 1.	The 10 numbers that are to be added to are in contiguous fullwords beginning at absolute core location 952.
Assumption 2.	The program is in core starting at absolute location 48.
Assumption 3.	The number 49 is a fullword at absolute location 948.
Assumption 4.	Register 1 contains a 48.

Core may be thought of as shown in Figure 2.7.

2.2.1 Long Way, No Looping

Figure 2.8 illustrates a program to accomplish this addition.

The first instruction L 2,904(0,1) loads the first number into register 2. Register 2 will be used as the accumulator. As was explained in section 2.1.3, the 360 addresses are made up of an offset plus the contents of an index register, plus the contents of a base register. In this instruction we have denoted the index register as being 0. There is a zero register. However, when it is used as an index, base, or branch register, it is assumed to have zero contents. Therefore, the address specified in the first load instruction above is equal to 904 plus the contents of register 1 (which contains a 48), i.e., 952. This is the absolute address of the first data element, DATA1.

The next instruction in the program adds the contents of absolute location 948 to register 2. Absolute location 948 contains a 49. Next comes a store instruction that stores the contents of register 2 back into absolute location 952,

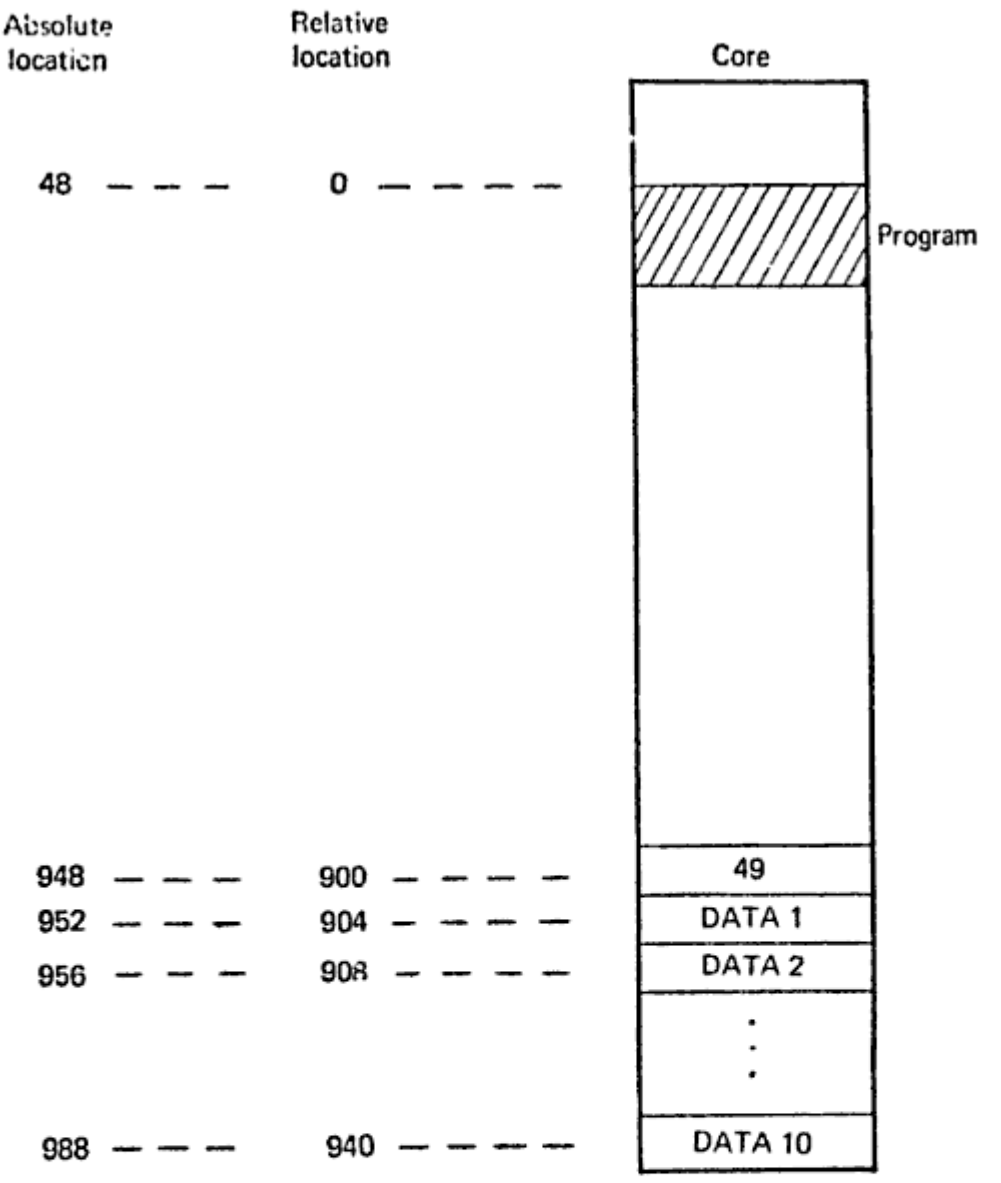


FIGURE 2.7 Diagram of core setup for addition problem

destroying the original first data item and replacing it by a new one that is equal to DATA1 plus 49. Similarly, the next three instructions add 49 to DATA2. An identical set of three instructions is used for each data item.

The preceding program will work; however, there are some potential problems. For example, if we wanted to process 300 data items rather than just 10, the storage needed for the instructions would be (3 instructions) x (length of each instruction) x (number of data items) = 3,600 bytes. Thus the instructions would overlay our data in core. Furthermore, the distance from the first instruction to the last piece of data would be 4,800 bytes, since the data itself occupies 4 x 300 = 1,200 bytes. Using register 1 as a base register, it would be impossible to

Absolute address	*Relative address*	*Hexadecimal*	*Instructions*	*Comments*
48	0	58201388	L 2,904(0,1)	Load reg 2 from loc 904+C(reg 1) = 952
52	4	5A201384	A 2,900(0,1)	Add 49 (loc=900+c(reg 1)) = 948
56	8	50201388	ST 2,904(0,1)	Store back
60	12	5820138C	L 2,908(0,1)	Load next data
64	16	5A201384	A 2,900(0,1)	Add 49
68	20	5020138C	ST 2,908(0,1)	Store back
⋮	⋮	⋮	⋮	
948	900	00000031	49	
952	904	⋮	⋮	

FIGURE 2.8 Program for addition problem – straightforward approach

access both the first data item and the last. The programmer must be aware of this type of problem if he is writing in machine language, and should bear in mind the fact that the largest possible value of an offset is 2^{12}-1 or 4,095, which may not reach all his data. (It is, of course, possible to use more than one base register.)

Note that if the preceding program were loaded into location 336 instead of location 48, it would still execute correctly if the content of register 1 was 336. Moving the program to a different location is a process called *relocation*. The use of base registers facilitates this process.

2.2.2 Address Modification Using Instructions As Data

Our example may be analogous to the "program" depicted in Figure 2.9. If an M.I.T. student had a date with a girl, he might write a program to do the following.

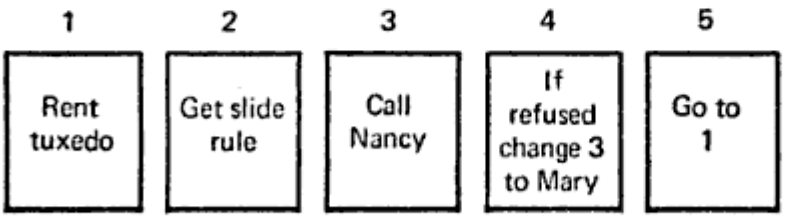


FIGURE 2.9 Situation

The preceding boxes represent locations, and the words in those boxes represent instructions to the processor, in this case, the M.I.T. student. The program would have the student rent a tuxedo, get a slide rule (this is M.I.T.), and call Nancy. But if Nancy refuses, the M.I.T. student does not want to write a new

program, so he writes an instruction that simply changes the contents of location 3 from Nancy to Mary and then repeats 1 through 4. The execution of the instruction in location 4 changes the instruction in location 3. Of course, in the preceding program he could end up renting many tuxedos, and if Mary refuses, she will receive an awful lot of telephone calls.

This M.I.T. student, however, has grasped two basic programming techniques. The first is that the instruction in location 3 may be treated as data. The second is looping, which he accomplished with the transfer rule in step 5. In this section and the next we will see how these two techniques can simplify our previous program.

Observe that in the program in Figure 2.8 we were merely using three instructions over and over again. The only element that changed was the offset of the load and store commands. In the first set of three instructions, the offset was 904. In the next set it was 908, then 912. An alternate technique is to write a machine language program consisting only of those three instructions, followed by a sequence of commands that would change the offset of the load and store instructions by adding 4 to them. The computer would execute the three instructions, change two of them so that the offset was increased by 4, and loop back to re-execute the set of three instructions. Of course, they would now have a different offset and therefore would refer to a different address.

The program in Figure 2.10 depicts a sequence of instructions that will perform the operation of adding the number 49 to 10 locations in core by modifying the instructions themselves.

Aosolute address	*Relative address*	*Instructions*	*Comments*
48	0	L 2,904(0,1)	Add 49 to a number
52	4	A 2,900(0,1)	
56	8	ST 2,904(0,1)	
60	12	L 2,0(0,1)	Increase displacement of load instruction by 4
64	16	A 2,896(0,1)	
68	20	ST 2,0(0,1)	
72	24	L 2,8(0,1)	Increase displacement of store instruction by 4
76	28	A 2,896(0,1)	
80	32	ST 2,8(0,1)	
		Branch to relative location 0 nine times	
⋮	⋮		
944	896	4	
948	900	49	
952	904	Numbers	
⋮	⋮		

FIGURE 2.10 Program for addition problem using instruction modification

ADDITIONAL ASSUMPTIONS:

Assumption 5. Relative location 896 contains a 4.

To see how the program operates we must keep in mind that the contents of location 48 (Fig. 2.11) are not L 2,904(0,1), but rather

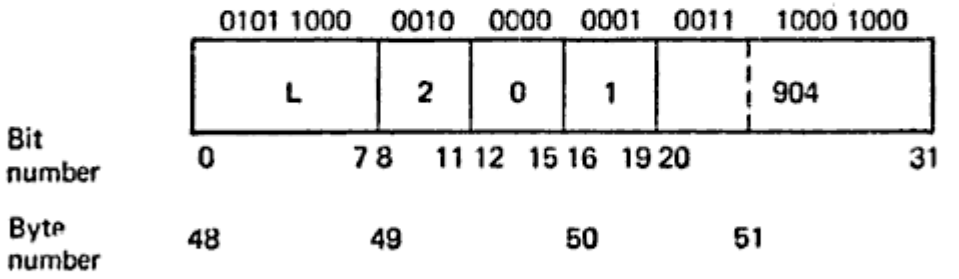


FIGURE 2.11 Contents of location 48

The offset of the instruction is the last and rightmost part of the number stored in that location. This instruction may be interpreted as a piece of data, and adding the number 4 to it updates the offset.

Treating instructions as data is not a good programming technique because in maintaining the program over a period of time, it may become difficult to understand what the original programmer was doing. In the case of multiprocessing systems it would violate all the rules of pure procedures (*re-entrant code*), which are procedures that do not modify themselves. We are including this example merely to exemplify instruction formats and explicitly demonstrate that instructions are stored as a type of data.

2.2.3 Address Modification Using Index Registers

Perhaps the most elegant way to solve this example is to use the index registers for address modification. Recall that an address equals the offset plus the contents of the base register plus the contents of an index register. We use the same three main instructions: the load instruction, add 49, and the store instruction. We simply loop through those three instructions, updating the contents of an index register by 4 during each pass and so updating the value of the address specified in the load and store instructions. The following program section uses this technique:

Absolute address	*Relative address*	*Instructions*	*Comments*
48	0	SR 4,4	Clear register 4
50	2	L 2,904(4,1)	Load data element of array
54	6	A 2,900(0,1)	Add 49
58	10	ST 2,904(4,1)	Replace data element
62	14	A 4,896(0,1)	Add 4 to index register
		Branch back to relative location 2, nine times	

The first instruction in this program is a subtract instruction. It is a register-to-register instruction, subtracting register 4 from register 4, thereby initializing its contents to 0. Note that this instruction is only two bytes long. As we discussed in the previous section, the 360/370 has different length instructions. The SR instruction starts at absolute location 48. The next instruction starts at location 50. Notice also that the load and store instructions now specify register 4 as an index register. The first time through the loop, register 4 will contain 0, the next time, 4, etc. This will point the load and store instructions at a different data item each time.

Now that we have seen how to modify instruction addresses, we will proceed to add the instructions that will carry out the actual looping.

2.2.4 Looping

In this section we will discuss two looping methods using machine language. We will make the additional assumptions:

Assumption 6. Relative location 892 contains a 10
Assumption 7. Relative location 888 contains a 1 (first method only)

Figure 2.12 depicts one looping scheme.

After the first four basic instructions are executed, there is a sequence of instructions that subtracts one from the temporary location and detects whether or not the result is positive. If positive, control loops back to the main program at relative location 2.

There is one instruction, Branch on Count (BCT, shown in Fig. 2.13) that accomplishes the work of the last four instructions in Figure 2.12.

The reader is referred to the manuals for the explanation of the BCT instruction. Essentially, register 3 is decremented by 1 until a 0 is reached. While the content of register 3 is positive, we transfer to the address specified in the address field, in this case, 6 plus the contents of register 1. When zero is reached, no branch occurs. Most computers have a similar branching instruction.

Absolute address	*Relative address*	*Instructions*		
48	0	SR	4,4	
50	2	L	2,904(4,1)	Add 49 to a number
54	6	A	2,900(0,1)	
58	10	ST	2,904(4,1)	
62	14	A	4,896(0,1)	Add 4 to index register
66	18	L	3,892(0,1)	Load temp into register 3
70	22	S	3,888(0,1)	Subtract 1
74	26	ST	3,892(0,1)	Store temp
78	30	BC	2,2(0,1)	Branch if result is positive (2 denotes a condition code)
—	—	—		
—	—	—		
—	—	—		
936	888	1		
940	892	(Initially 10 - decremented by 1 after each loop)		
944	896	4		
948	900	49		
952	904	Numbers		

FIGURE 2.12 Program for addition problem showing looping

Absolute address	*Relative address*	*Instructions*		
48	0	L	3,892(0,1)	Load register 3 with 10
52	4	SR	4,4	Clear register 4
54	6	L	2,904(4,1)	
58	10	A	2,900(0,1)	Add 49 to a number
62	14	ST	2,904(4,1)	
66	18	A	4,896(0,1)	Add 4 to index register
70	22	BCT	3,6(0,1)	Subtract one from register 3 and branch to relative location 6 when positive
⋮	⋮	⋮		
940	892	10		
944	896	4		
948	900	49		
952	904	Numbers		
⋮	⋮			
992	unused			
⋮				

FIGURE 2.13 Final version of example

We now have reduced the program to 26 bytes of instructions and 52 bytes of data, in contrast to the 120 bytes of instructions and 44 of data utilized in our first attempt. This is a savings of 86 bytes. Note: all of the preceding programs could be placed elsewhere in core, at location 400 rather than 48, for example, and only register 1 need be changed.

2.3 ASSEMBLY LANGUAGE

When the user wishes to communicate with the computer, he has available to him a spectrum of languages:

So far we have discussed the two lowest members of this spectrum. We will now go into assembly language, which is the most machine-dependent language used by programmers today.

There are four main advantages to using assembly language rather than machine language.

1. It is mnemonic; e.g., we write ST instead of the bit configuration 01010000 for the store instruction
2. Addresses are symbolic, not absolute
3. Reading is easier
4. Introduction of data to program is easier

A disadvantage of assembly language is that it requires the use of an assembler to translate a source program into object code. Many of the features of 360 or 370 assembly language exist in assembly languages for other machines (if the reader is using another machine). These examples may be easily translated into the machine he is using.

2.3.1 An Assembly Language Program

Let us rewrite the program discussed in the previous section in assembly language (shown in Fig. 2.14). In doing so, the assumptions that were made when written in machine language are eliminated. One of the assumptions is that the program's

starting address was absolute core location 48. We, as programmers, cannot presume to know into what location our program will be loaded in core. Thus there must be a way for us to load the base register with the address of the program in core just prior to execution time. That is, execution time is the only time in which the program, the programmer, or anyone else can be certain as to where in core the loader will load the user's program. The BALR instruction is one mechanism for loading the base register.

If the assembler is to automatically compute the displacement field of instructions, it must know what register can be used as a base register and what that register will contain. The USING instruction tells both of these things to the assembler and thus makes it possible for it to produce correct code. The USING instruction is a *pseudo-op*. A pseudo-op is an assembly language instruction that specifies an operation of the assembler; it is distinguished from a *machine-op* which represents to the assembler a machine instruction. The Define Constant (DC) and Define Storage (DS) instructions are pseudo-ops that instruct the assembler to place a 10, a 4, and a 49 in 3 consecutive fullwords ("F") in memory and leave 10 more for data. A number before the F would cause multiple allocations, e.g. DS 100F causes the assembler to set aside a 100 full word area.

	Program		*Comments*
TEST	START		Identifies name of program
BEGIN	BALR	15,0	Set register 15 to the address of the next instruction
	USING	BEGIN+2,15	Pseudo-op indicating to assembler register 15 is base register and its content is address of next instruction
	SR	4,4	Clear register 4 (set index=0)
	L	3,TEN	Load the number 10 into register 3
LOOP	L	2,DATA(4)	Load data (index) into register 2
	A	2,FORTY9	Add 49
	ST	2,DATA(4)	Store updated value of data (index)
	A	4,FOUR	Add 4 to register 4 (set index = index+4)
	BCT	3,LOOP	Decrement register 3 by 1, if result non-zero, branch back to loop
	BR	14	Branch back to caller
TEN	DC	F'10'	Constant 10
FOUR	DC	F'4'	Constant 4
FORTY9	DC	F'49'	Constant 49
DATA	DC	F'1,3,3,3,3, 4,5,8,9,0'	Words to be processed
	END		

FIGURE 2.14 An assembly language program

CLARIFICATION

1. *USING* is a pseudo-op that indicates to the assembler which general register to use as a base and what its contents will be. This is necessary because no special registers are set aside for addressing, thus the programmer must inform the assembler which register(s) to use and how to use them. Since addresses are relative, he can indicate to the assembler the address contained in the base register. The assembler is thus able to produce the machine code with the correct base register and offset.
2. *BALR* is an instruction to the computer to load a register with the next address and branch to the address in the second field. When the second operand is register 0, as it is here, execution proceeds with the next instruction. It is important to see the distinction between the BALR, which loads the base register, and the USING, which informs the assembler what is in the base register. The distinction will be clearer after we study the assembler, but for now, note that a USING only provides information to the assembler but does not load the register. Therefore, if the register does not contain the address that the USING says it should contain, a program error may result.
3. *START* is a pseudo-op that tells the assembler where the beginning of the program is and allows the user to give a name to the program. In this case the name is TEST.
4. *END* is a pseudo-op that tells the assembler that the last card of the program has been reached.
5. Note that instead of addresses in the operand fields of the instructions as in the example of Figure 2.8, there are symbolic names. The main reason for assemblers coming into existence was to shift the burdens of calculating specific addresses from the programmer to the computer.
6. BR 14, the last machine-op instruction, is a branch to the location whose address is in general register 14. By convention, calling programs leave their return address in register 14.

2.3.2 Example Using Literals

Here we will repeat the same example using *literals*, which are mechanisms whereby the assembler creates data areas for the programmer, containing constants he requests.

In the program of Figure 2.15 the arguments =F'10', =F'49', =F'4' are literals which will result in the creation of a data area containing 10,49,4 and replacement of the literal operand with the address of the data it describes.

The assembler translates the instruction L 3, = F'10' so that its address portion points to a full word that contains a 10. Normally, the assembler will construct

```
TEST     START   0
BEGIN    BALR    BASE,0
         USING   BEGIN+2,BASE
         SR      4,4
         L       3,=F'10'
LOOP     L       2,DATA(4)
         A       2,=F'49'
         ST      2,DATA(4)
         A       4,=F'4'
         BCT     3,*-16
         BR      14
         LTORG
DATA     DC      F'1,3,3,3,3,4,5,8,9,0'
BASE     EQU     15
         END
```

FIGURE 2.15 Assembly language program using literals

a "literal table" at the end of the program. This table will contain all the constants that have been requested through the use of literals. However, the pseudo-op LTORG can tell the assembler to place the encountered literals at an earlier location. This pseudo-op is used in the case where we have a very long program. For example, if our DC instruction contained 10,000 pieces of data, it would have been impossible for the offset of the load instruction to reach the literals at the end of our program. In this case, we would want to force the literals into the program before the DC instruction.

In the BCT instruction in the same program we have used as an address *-16. The star is a mnemonic that means "here." The expression *-16 refers to the address of the present instruction minus 16 locations, which is LOOP. (This type of addressing is not usually good practice: should it become necessary for the programmer to insert other statements in-between LOOP and BCT, he would have to remember to change the 16.)

The statement BASE EQU 15 assigns this value 15 to the symbol BASE; BASE will be everywhere evaluated as 15. The EQU pseudo-op allows the programmer to define variables. Here, for example, if he wished to use a different base register, he would need only to change the EQU statement. Any sort of valid arithmetic expression may appear as the operand of the EQU statement.

Depicted in Figure 2.16 is the assembled version (that is, the output of the assembler) of the preceding program.

Observe that some pseudo-ops (e.g., START and USING) do not generate machine code. Note also that the BR 14 instruction has been translated to BCR 15,14. This is because BR is a member of the assembler mnemonic group of instructions that allow the programmer to use a more mnemonic op code in place of BC followed by a particular mask value (see Appendix A).

2.4 SUMMARY

We have presented a general approach to understanding a new machine, and applied this approach in the context of the IBM 360 and 370. We have evolved a machine language example illustrating base register use, storage of instructions and data, indexing and looping. Some features of 360 Basic Assembly Language (BAL) were introduced, e.g., symbolic addressing, literals and pseudo-ops.

Assembly language program			*Relative location*	*Assembled mnemonic program*	
TEST	START				
BEGIN	BALR	15,0	0	BALR	15,0
	USING	BEGIN+2,15			
	SR	4,4	2	SR	4,4
	L	3, =F'10'	4	L	3,30(0,15)
LOOP	L	2, DATA (4)	8	L	2,42(4,15)
	A	2, =F'49'	12	A	2,34(0,15)
	ST	2, DATA (4)	16	ST	2,42(4,15)
	A	4, =F'4'	20	A	4,38(0,15)
	BCT	3, *-16	24	BCT	3,6(0,15)
	BR	14	28	BCR	15,14
	LTORG				
			32	10	
			36	49	
			40	4	
DATA	DC	F'1,3,3,3,3,4,5,9,0'	44	1	
			48	3	
			52	3	
				.	
				.	
				.	
	END				

FIGURE 2.16 Assembled version of example program

QUESTIONS[4]

1. All data is stored as ones and zeros, taking on meaning only when interpreted according to some rule. Order the following sets of strings according to the specified conditions:

   a.

OP1	1001	0111	1001	1100
OP2	0110	1001	0010	1100
OP3	1000	0011	0110	1101

   1) Halfword integers
   2) Packed decimals

   b.

OP1	0100	1110	1111	0110
OP2	0110	0000	1111	0001
OP3	1111	0101	1111	0010

   1) Halfword integers
   2) Characters (EBCDIC)

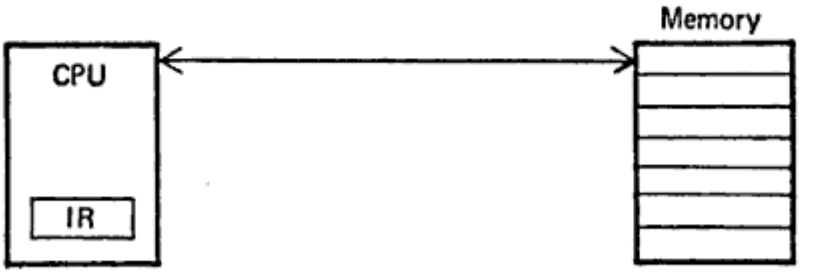
2. Below are several hexadecimal strings representing fullwords and halfwords from core.

   (1) 052C (2) 452C
   (3) 4528367D (4) 5914973C

   For each of the above,

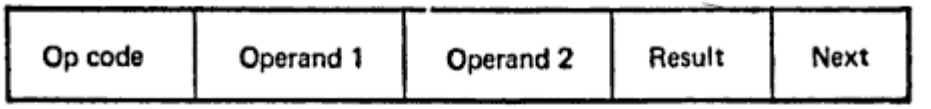
   a. Write the binary equivalent.
   b. Assume each to be a halfword/fullword integer, and write its value in decimal.
   c. Assume each to be a packed decimal number, and write its value in decimal.
   d. If it represents a legitimate 360 machine instruction, give the mnemonic representation; if not, tell why not.

3. This problem will investigate basic properties of machine architecture. The simple machine in our example consists of a location-addressed memory where instructions and data are stored, and a Central Processing Unit (CPU) that interprets instructions and performs the specified operations. The CPU contains an Instruction Register (IR) which holds the present instruction being interpreted.

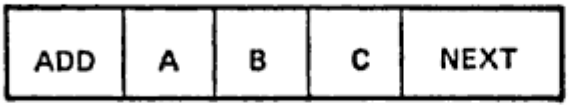


[4] An * denotes that the question may require the use of IBM 360 manuals. A † denotes that the question is a machine problem.

An instruction on this machine has the form



In questions a - d you are to redesign this basic machine with the objective of shortening the instruction length. For these questions consider the following instruction



which adds the number at location A to that at location B and stores the result at C. The next instruction fetched for interpretation is located at NEXT.

a. Give a method by which the NEXT field of the instruction could be eliminated. What additional instruction would be required in this new machine that was not required in the original? What additional register must be added to the CPU?
b. Give a method by which the RESULT field of the instruction would be eliminated. (Do not add any registers to the CPU.) What additional instruction would be required for this new machine that was not required in the original?
c. Another way to eliminate the RESULT field is to add a single register to the CPU that is the same size as a memory word. Such a register is usually referred to as an accumulator (AC). The additional instruction that is required in this case is a store instruction which stores the contents of the accumulator into a specified word in memory. All other instructions take two operands and place their result into the accumulator. In addition to the RESULT field, one of the operand fields can also be eliminated. Describe how this can be done. What additional instruction is required?
d. The IBM 360 and many other computers use the two-address instruction format that you obtained from questions 2 and 3, rather than the four-address format of our basic machine. What advantages are to be found by shortening the instruction length? What are the disadvantages of this scheme?

4. a. What are the advantages of the 360's multiple register scheme over machines with fewer or specialized registers?
*b. On the 360, register 0 operates differently in three contexts from the other registers. Give an example of each and explain how register 0 is being used in each case.

5. *a. What is the difference in function between the BALR and USING

instructions? What happens to each at assembly time? At execution time?

b. For each of the following program segments show the equivalent mnemonic machine language and determine the value placed in register 1 by the instruction LH 1,DATA2.

```
OCT15   START   0           OCT15   START   0
        BALR    15,0                USING   *,15
        USING   *,15                BALR    15,0
        LR      10,15               LA      10,DATA2
        USING   *,10                USING   DATA2,10
        LH      1,DATA2             DROP    15
        BR      14                  LH      1,DATA2
DATA1   DC      H'1'        DATA1   DC      F'1'
DATA2   DC      H'2'        DATA2   DC      F'2'
DATA3   DC      H'3'        DATA3   DC      F'3'
        END                         END
```

6. *a. The following program is supposed to multiply 3 times 2 and store the result into location 1000. Will it? (Note: The address of 1000 refers to location 1000 in core; no base or index register is used.)

```
L       3,=F'2'
M       3,=F'3'
ST      3,1000
```

b. Will the following divide 10 by 2? Justify.

```
L       3,=F'10'
D       2,=F'2'
ST      3,1000
```

7. *a. What is the difference between

```
INDEX   EQU     5
INDEX   DC      F'5'
```

b. What is the difference between the CR instruction and the CLR instruction?

8. *a. What will be in register 3 after each instruction in the following sequence of instructions:

```
        LA      3,=A(XYZ)
        LR      3,3
        L       3,=F'5'
XYZ     LCR     3,3
        LNR     3,3
```

b. Which instruction will be executed after the BE SAME. Why?

```
        CLI     =F'3',3
        BE      SAME
        LR      3,5
SAME    AR      5,5
```

9. *a. When do 'SRDA 0,5' and 'SRDL 0,5' execute differently?
   b. When do 'SLL 1,1' and 'LA 1,0(1,1)' execute differently?
   c. When do 'MVC TEMP(0), DATA' and 'MVC TEMP(1), DATA' execute differently?
   d. Is 'LA 1,0(1)' equivalent to a no-operation? Is 'SRA 1,0'? If not, explain.
   e. Assume STOMP is defined by:

```
STOMP   DC    C' ERASURE'
```

   How will the following instructions execute individually?

```
MVC     STOMP+1(8), STOMP
MVC     STOMP(8), STOMP+1
```

10. *Do all the 360's instructions set or check the CC (Condition Code)? What is the role of the CC in the 360's instruction set?

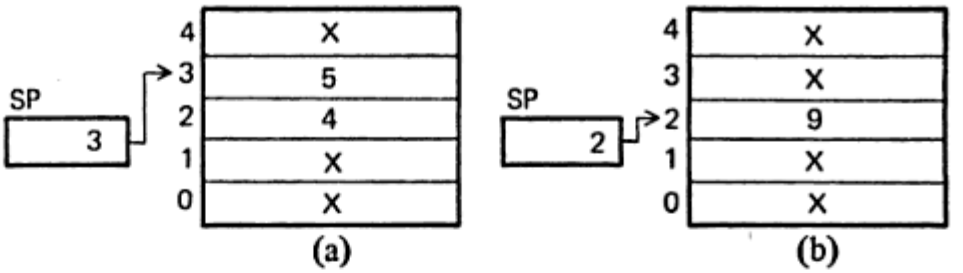
11. Draw micro-flowcharts for the following IBM 360 instructions (see Fig. 2.2).
    a. S (Subtract, RX form)
    b. BCR (Branch on Condition, RR form)
    c. BXLE (Branch on Index Less than or Equal, RS form).

12. Consider the following computer system organization, referred to as a "stack" computer.

    There are two memories: one memory is directly addressable, that is, it is possible to load from or store into any location at any time. The other memory has an associated register, called the "stack pointer;" it is only possible to load from or store into the location currently specified by the "stack pointer." We will refer to the "Stack Pointer" as SP.

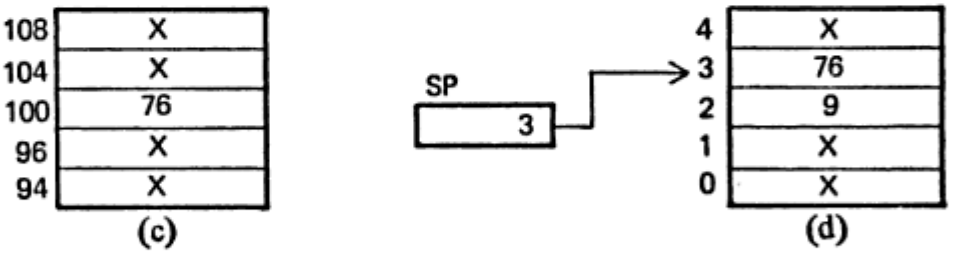
    There are three types of instructions: (1) instructions that transfer data between the two memories, (2) instructions that operate on the "stack" memory only, and (3) transfer instructions. Note that there are *no* registers such as an AC (accumulator) or GPRs (General Purpose Registers) as on the IBM 7094 or 360.

    We will first discuss the ADD instruction. Assume the "stack" memory has the contents specified in Figure (a) below:

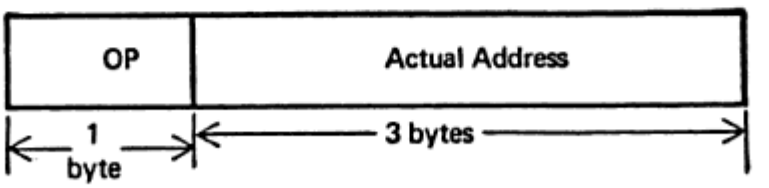


The ADD instruction will change the "stack" memory to the contents of Figure (b). Memory locations marked X are not affected by the instruction and may be ignored. Note that the following two actions occurred: (1) the contents of the locations specified by C(SP) and C(SP-1) are added and the result stored in the location specified by C(SP-1) and (2) SP is decreased by 1.

a. Draw a machine organization diagram for this computer (see Fig. 2.1).
b. Draw a micro-flowchart for the ADD instruction. It should be similar in complexity to the preceding flowchart in Figure 2.2, but the actual registers etc., will not necessarily be the same. You should *clearly* specify your notation. Assume the instruction is already in the decoding register. (Hopefully, you noticed that the ADD instruction does *not* have an operand, only an op-code. Assume that the ADD instruction requires only one byte of op-code.)
c. The LOAD instruction operates as described below. Assume that the regular memory has its contents illustrated in Figure (c) and the "stack" memory is still as in Figure (b). The instruction LOAD 100 would result in Figure (d).



Note that the instruction loads into the location specified by C(SP+1) and then increases C(SP) by 1. Assume that the LOAD instruction is four bytes long, i.e.



Draw a micro-flowchart for the LOAD instruction.

13. The following 360 assembly language program computes the function:

$$A = 2 \cdot B + 2 \cdot C - 1$$

```
 1) COMPUTE   START
 2)           USING   *,15
 3)           L       1,B
 4)           SLA     1,1
 5)           L       2,C
 6)           SLA     2,1
 7)           AR      1,2
 8)           S       1,=F'1'
 9)           ST      1,A
10)           BR      14
11) A        DC      F'0'
12) B        DC      F'5'
13) C        DC      F'7'
14)          END
```

a. Verify that the preceding program works correctly by simulating the instructions one by one and filling in the table below. Indicate the contents of each register and memory location *after* each instruction is executed.

Instruction	*Register 1*	*Contents of register 2*	*Location A*
3			
4			
5			
6			
7			
8			
9			

b. The above program is reasonably efficient; it only requires 44 bytes and cleverly avoids the use of the slow multiply instruction and the literal 2. It is known that by rewriting statements 3 through 9 only, it is possible to reduce the entire program to only 12 statements that require 32 bytes. This new program must compute the same function though not necessarily in the same way.

Fill in the blank spaces in the following program to represent a new program that is more efficient than the preceding one. It need not be optimal nor use all the statements 3 through 9.

```
1) COMPUTE   START
2)           USING   *,15
3)
4)
5)
```

```
6)
7)
8)
9)
10)                 BR          14
11)  A              DC          F'0'
12)  B              DC          F'5'
13)  C              DC          F'7'
                    END
```

14. Obtain the manufacturer's manual for any digital computer and answer the five basic questions of section 2.1.1.

15. †Write a subroutine in 360 assembly language that does the following.
    a. When control is passed to your program, registers 2 and 3 will contain 8 EBCDIC characters (four eight-bit characters in each register).
    b. The subroutine should *count* the number of EBCDIC commas and return the count in register 1.

```
Example:                Register 2              Register 3
          INPUT:        A , B C                 D E , F  (EBCDIC)
                        Register 1
          OUTPUT:       00000002      (HEXADECIMAL)
```

A reasonable solution to this problem will probably require from 8-20 assembler cards.

There will be a grader[5] to evaluate your program. The format for each program is:

```
      GRADER                          STUDENT
Loads registers 2,3        STUDENT    START 0
L      15, =V(STUDENT)                 ENTRY STUDENTN
BALR 14,15                             USING *,15
Checks for right answer                .                    Body of program
                                       .
                                       .
                                       BR      14           DC and DS
                                       .                    pseudo-ops
                                                            (if any)
                                       LTORG
                           STUDENTN    EQU     *
                                       END
```

[5]The problem can be done without a grader. The student will have to supply data and a main program. A grading program calls student programs and keeps a record of the results. We specify the grader only as an aid to the instructor or student who wishes to use such a system.

The student's program *must not change* the contents of registers 14 or 15.

16. †All data forms are read by the assembler into the machine in character format (EBCDIC), so the assembler or compiler must provide routines for converting the data into the internal representation specified by the programmer.

Your assignment is to write a 360 assembly language subroutine that simulates the data processing performed by an assembler in handling a 'DC' pseudo-op for C, X, P, and F formats. You will be given an EBCDIC character string of specified byte-length in storage and asked to convert it to a specified internal data format.

a. *Communication with the Grading[5] Program* When your program is entered, the general registers will contain the following information.

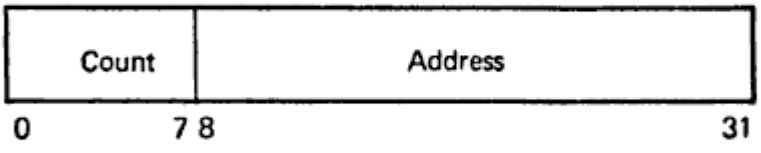
Register 0:

The contents of register zero indicate the type of conversion to be performed, as follows:

0	Character
1	Hexadecimal
2	Packed decimal
3	Fullword binary

Register 1:

The contents of register one indicate the length and location of the data to be converted. Bits 0-7 contain the byte count of the data (including any optional sign as described below) and bits 8-31 contain the absolute address of the data:



Registers 2-12:

The contents of registers 2-12 are not significant, but must be saved and restored by your program.

Register 13:

Register 13 contains the address of an 18 fullword area where you may save general registers.

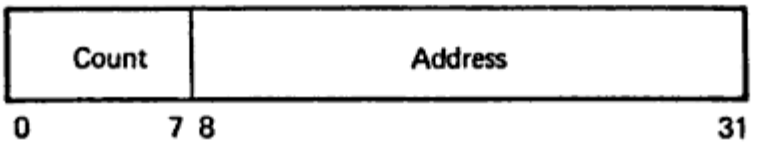
Register 14:

Register 14 contains the address of the return point in the grading program. You should branch to this address when you have completed processing.

Register 15:

Register 15 contains the address of the entry point to your program.

When your program returns to the grading program, the contents of registers 2-12 should be the same as when you were called, and register 1 should contain the *address* of a fullword area in your program with the following structure:



The "count" will be the length in bytes of the converted data, and the "address" will be the absolute address of an area within your program containing the converted data.

The following restrictions will be placed on the input and output data:

Character data will have a maximum byte count of 40.

Data to be converted to hexadecimal will have a maximum byte count of 40, yielding a maximum answer count of 20, and the byte count will be even. (Thus, you will not have to pad with zeros.)

Data to be converted to packed decimal will have, at most, 16 significant decimal digits not including an optional prefixed plus or minus sign. When you return your answer, it should be as short as possible, that is, with leading zeros removed. You must return the answer with the EBCDIC preferred sign codes C and D.

Data to be converted to fullword binary will also have an optional prefixed plus or minus sign, and a maximum significant digit count of 16. Your answer should be aligned to a fullword boundary.

You must not modify the input strings in any way.

b. *Examples* If you were given the string '110', with a count of 3, your answer would be:

Conversion	*Internal code (hexadecimal)*	*Final count*
C	F1F1F0	3
X	0110	2
P	110C	2
F	0000006E	4

Note the zero padding and justification in the X, P, F conversions.

17. †Write a 360 assembly language subroutine that will perform addition and subtraction of mixed base numbers. For example:

```
      1 days 22 hours  3 min
-(          22 hours 20 min    )
-------------------------------
      0 days 23 hours 43 min
```

Register	Contents (hex)			
R1	+	01	16	03
R2	−	00	16	14
R3	+	00	17	28

Registers 1 and 2 contain the operands; register 0 will contain your answer. For each register the first byte is the sign, the second is the number of days, third number of hours, fourth number of minutes. The student program must have the same form as problem 16.

3

assemblers

An assembler is a program that accepts as input an assembly language program and produces its machine language equivalent along with information for the loader (Fig. 3.1). In this chapter we will discuss the design of an assembler.

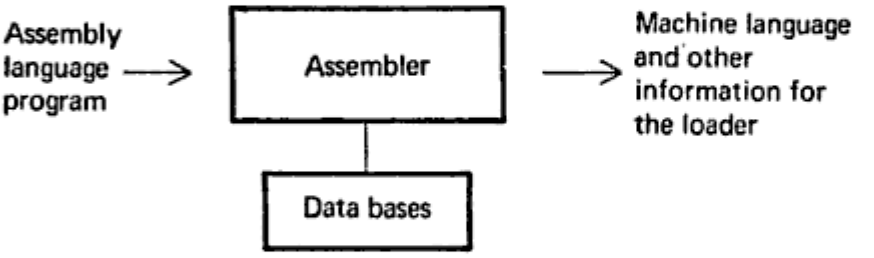


FIGURE 3.1 Function of an assembler

We focus on procedures for producing the machine language. However, the reader must keep in mind that (in all but the most primitive of loading schemes) the assembler must also produce other information for the loader to use. For example, externally defined symbols must be noted and passed on to the loader; the assembler does not know the address (value) of these symbols and it is up to the loader to find the programs containing them, load them into core, and place the values of these symbols in the calling program. Loading is discussed in Chapter 5.

In this chapter we are primarily concerned with the production of machine language. The illustrative examples use a 360-type assembler and mnemonic machine language. In our design of an assembler, and later in our design of a macro processor, many possible algorithms could have been used. We have chosen a way which we feel demonstrates the basic tasks of such programs.

Throughout this book, we will be referring to "decks" and "programs." At one time a "deck" always meant a deck of cards. Today, with the widespread use of

other forms of secondary storage and of typewriter terminals, many programs never actually take the form of card decks. A "card" is a convenient unit of information; other devices offer similar divisions into units, or *records*, often of variable lengths. These different forms of storage are essentially interchangeable; a "statement" may be a card or other record, and a "card" may be a record on tape or drum. The term "deck," as used in this book and throughout the computer industry, has become a commonplace for every form of program used as input or output to a computer.

3.1 GENERAL DESIGN PROCEDURE

Before discussing the detailed design of an assembler, let us examine the general problem of designing software. Listed below are six steps that should be followed by the designer:

1. Specify the problem
2. Specify data structures
3. Define format of data structures
4. Specify algorithm
5. Look for modularity (i.e., capability of one program to be subdivided into independent programming units)
6. Repeat 1 through 5 on modules

In this book we have followed this procedure in the design of the assembler, loader, and compiler.

3.2 DESIGN OF ASSEMBLER

3.2.1 Statement of Problem

Let us pretend that we are the assembler trying to translate the program in the left column of Figure 3.2. We read the START instruction and note that it is a pseudo-op instruction (to the assembler) giving JOHN as the name of this program; the assembler must pass the name on to the loader. The next instruction is the USING pseudo-op. This tells the assembler (us) that register 15 is the base register and at execution time will contain the address of the first instruction of the program.

Source program			First pass			Second pass		
			Relative address	Mnemonic instruction		Relative address	Mnemonic instruction	
JOHN	START	0						
	USING	*,15						
	L	1,FIVE	0	L	1,-(0,15)	0	L	1,16(0,15)
	A	1,FOUR	4	A	1,-(0,15)	4	A	1,12(0,15)
	ST	1,TEMP	8	ST	1,-(0,15)	8	ST	1,20(0,15)
FOUR	DC	F'4'	12	4		12	4	
FIVE	DC	F'5'	16	5		16	5	
TEMP	DS	1F	20	-		20	-	
	END							

FIGURE 3.2 Intermediate steps in assembling a program

There is no BALR instruction. This program was presumably called by another program that left the address of the first instruction in register 15 (see standard subroutine linkage conventions in Appendix B). Next comes a Load instruction: L 1, FIVE. We can look up the bit configuration for the mnemonic in a table and put the bit configuration for the L in the appropriate place in the machine language instruction. Now we need the address of FIVE. At this point, however, we do not know where FIVE is, so we cannot supply its address. Because no index register is being used, we put in 0 for the index register. We know that register 15 is being used as the base register, but we cannot calculate the offset. The base register 15 is pointing to the beginning of this program, and the offset is going to be the difference between the location FIVE and the location of the beginning of the program, which is not known at this time. We maintain a *location counter* indicating the relative address of the instruction being processed; this counter is incremented by 4 (length of a Load instruction).

The next instruction is an Add instruction. We look up the op-code, but we do not know the offset for FOUR. The same thing happens with the Store instruction. The DC instruction is a pseudo-op directing us to define some data; for FOUR we are to produce a '4'. We know that this word will be stored at relative location 12, because the location counter now has the value 12, having been incremented by the length of each preceding instruction. The first instruction, four bytes long, is at relative address 0. The next two instructions are also four bytes long. We say that the symbol "FOUR" has the value 12. The next instruction has as a label FIVE and an associated location counter value of 16. The label on TEMP has an associated value of 20. We have thus produced the code in the center column of Figure 3.2

As the assembler, we can now go back through the program and fill in the offsets as is done in the third column of Figure 3.2.

Because symbols can appear before they are defined, it is convenient to make two passes over the input (as this example shows). The first pass has only to define the symbols; the second pass can then generate the instructions and addresses. (There are one-pass assemblers and multiple-pass assemblers. Their design and implications are discussed in this chapter). Specifically, an assembler must do the following:

1. Generate instructions:
   a. Evaluate the mnemonic in the operation field to produce its machine code.
   b. Evaluate the subfields – find the value of each symbol, process literals, and assign addresses.
2. Process pseudo ops

We can group these tasks into two *passes* or sequential scans over the input; associated with each task are one or more assembler modules.

Pass 1: Purpose – define symbols and literals
1. Determine length of machine instructions (MOTGET1)
2. Keep track of Location Counter (LC)
3. Remember values of symbols until pass 2 (STSTO)
4. Process some pseudo ops, e.g., EQU, DS (POTGET1)
5. Remember literals (LITSTO)

Pass 2: Purpose – generate object program
1. Look up value of symbols (STGET)
2. Generate instructions (MOTGET2)
3. Generate data (for DS, DC, and literals)
4. Process pseudo ops (POTGET2)

Figures 3.3 and 3.4 outline the steps involved in pass 1 and pass 2. The specifics of the data bases and a more detailed algorithm are developed in the following sections.

3.2.2 Data Structure

The second step in our design procedure is to establish the data bases that we have to work with.

Pass 1 data bases:
1. Input source program.
2. A Location Counter (LC), used to keep track of each instruction's location.
3. A table, the Machine-Operation Table (MOT), that indicates the symbolic

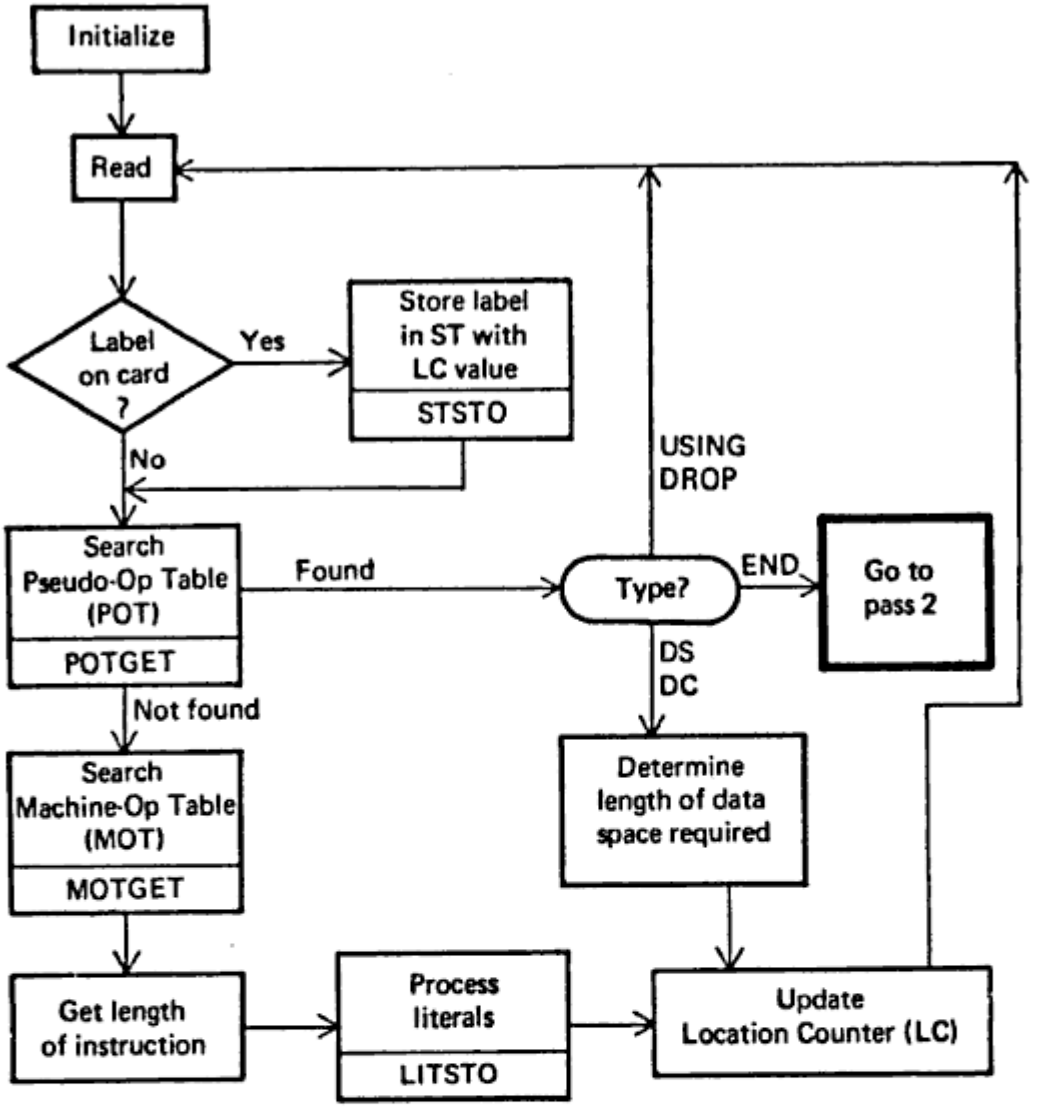


FIGURE 3.3 Pass 1 overview: define symbols

mnemonic for each instruction and its length (two, four, or six bytes).

4. A table, the Pseudo-Operation Table (POT), that indicates the symbolic mnemonic and action to be taken for each pseudo-op in pass 1.
5. A table, the Symbol Table (ST), that is used to store each label and its corresponding value.
6. A table, the Literal Table (LT), that is used to store each literal encountered and its corresponding assigned location.
7. A copy of the input to be used later by pass 2. This may be stored in a secondary storage device, such as magnetic tape, disk, or drum, or the original source deck may be read by the assembler a second time for pass 2.

Pass 2 data bases:

1. Copy of source program input to pass 1.
2. Location Counter (LC).

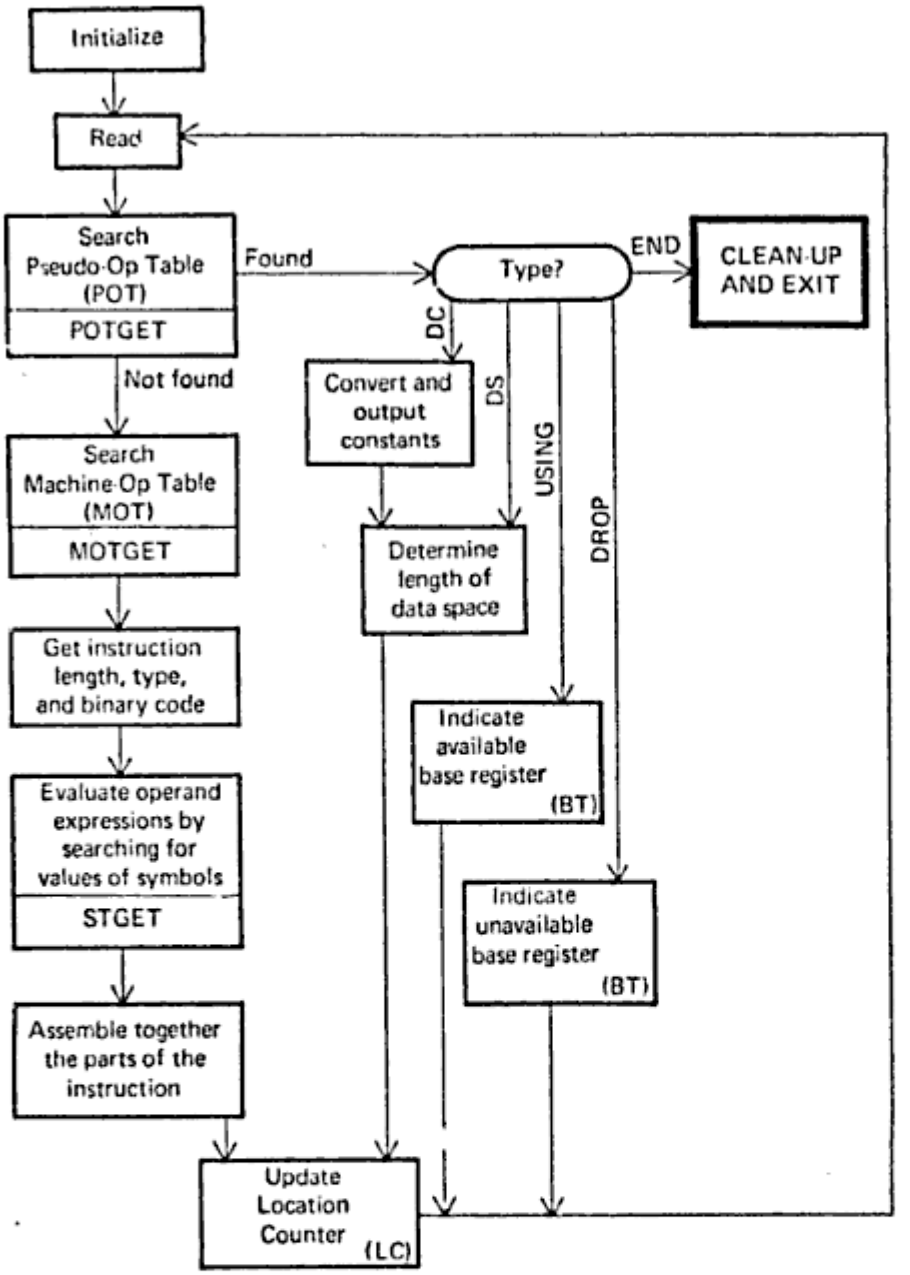


FIGURE 3.4 Pass 2 overview: evaluate fields and generate code

3. A table, the Machine Operation Table (MOT), that indicates for each instruction: (a) symbolic mnemonic; (b) length; (c) binary machine op-code, and (d) format (e.g., RS, RX, SI).
4. A table, the Pseudo-Operation Table (POT), that indicates for each pseudo-op the symbolic mnemonic and the action to be taken in pass 2.
5. The Symbol Table (ST), prepared by pass 1, containing each label and its corresponding value.
6. A table, the Base Table (BT), that indicates which registers are currently

specified as base registers by USING pseudo-ops and what are the specified contents of these registers.

7. A work-space, INST, that is used to hold each instruction as its various parts (e.g., binary op-code, register fields, length fields, displacement fields) are being assembled together.
8. A workspace, PRINT LINE, used to produce a printed listing.
9. A workspace, PUNCH CARD, used prior to actual outputting for converting the assembled instructions into the format needed by the loader.
10. An output deck of assembled instructions in the format needed by the loader.

Figure 3.5 illustrates the interrelation of some of the data bases and the two passes of the assembler.

3.2.3 Format of Data Bases

The third step in our design procedure is to specify the format and content of each of the data bases – a task that must be undertaken even before describing the specific algorithm underlying the assembler design. In actuality, the algorithm, data base, and formats are all interlocked. Their specification is in practical designs, circular, in that the designer has in mind some features of the format and algorithm he plans to use and continues to iterate their design until all cases work.

Pass 2 requires a Machine-Operation Table (MOT) containing name, length, binary code, and format; pass 1 requires only name and length. We could use two separate tables with different formats and contents or use the same table for both passes; the same is true of the Pseudo-Operation Table (POT). By generalizing the table formats, we could combine the MOT and POT into one table. For this particular design, we will use a single MOT but two separate POTs.

Once we decide what information belongs in each data base, it is necessary to specify the format of each entry. For example, in what format are symbols stored (e.g., left justified, padded with blanks, coded in EBCDIC or ASCII) and what are the coding conventions? The Extended Binary Coded Decimal Interchange Code (EBCDIC) is the standard IBM 360 coding scheme for representing characters as eight-bit bytes. The character "A," for instance, is represented in EBCDIC as 1100 0001 binary (C1 in hexadecimal).

The Machine-Op Table (MOT) and Pseudo-Op Tables (POTs) are examples of *fixed tables*. The contents of these tables are not filled in or altered during the assembly process. Figure 3.6 depicts a possible content and format of the machine-op table. The op code is the key and its value is the binary op-code equivalent, which is stored for use in generating machine code. The instruction

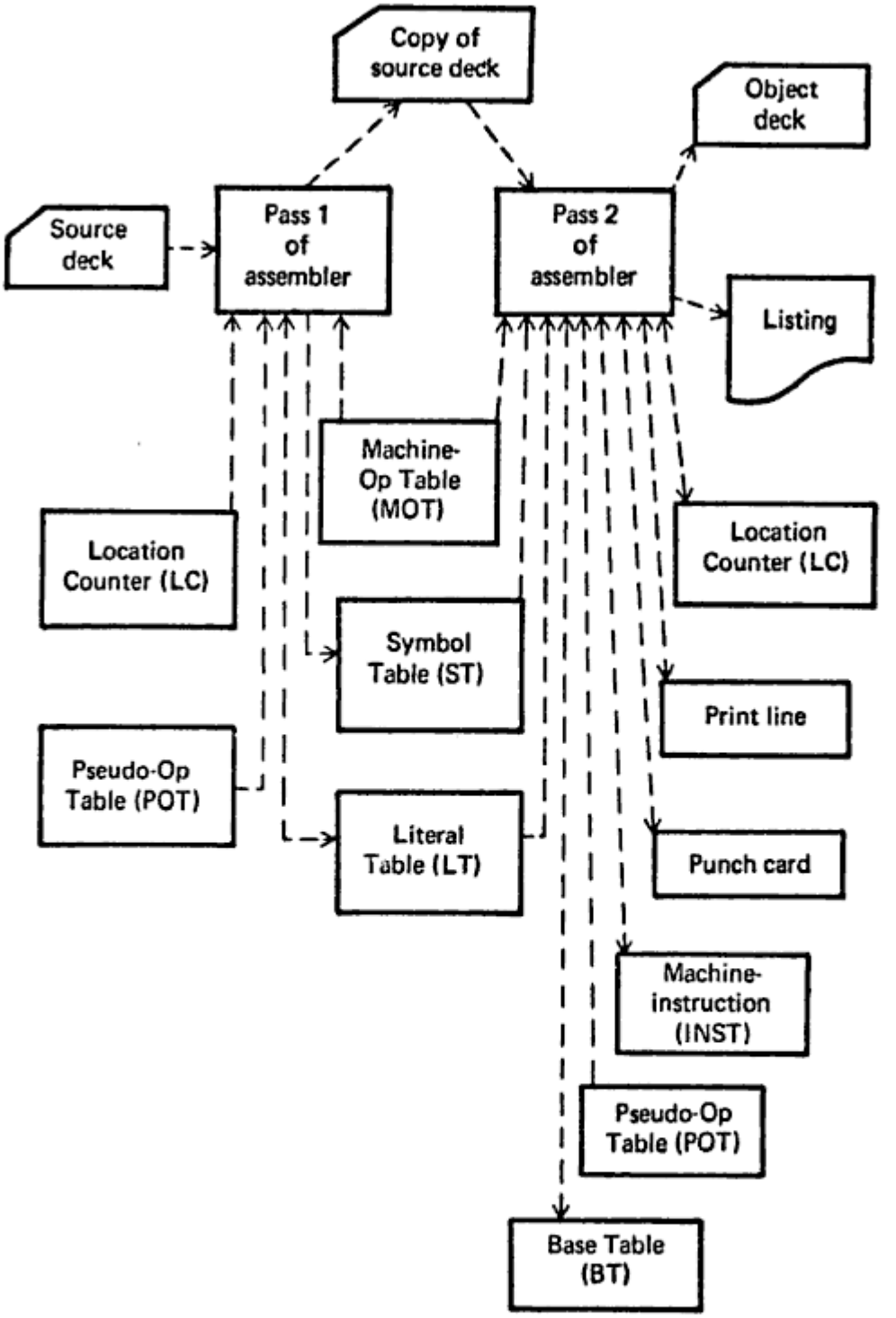


FIGURE 3.5 Use of data bases by assembler passes

length is stored for use in updating the location counter; the instruction format for use in forming the machine language equivalent.

Figure 3.7 depicts a possible pseudo-op table. Each pseudo-op is listed with an associated pointer to the assembler routine for processing the pseudo-op.

6-bytes per entry

Mnemonic op-code (4-bytes) (characters)	Binary op-code (1-byte) (hexadecimal)	Instruction length (2-bits) (binary)	Instruction format (3-bits) (binary)	Not used in this design (3-bits)
"A*bbb*"	5A	10	001	
"AH*bb*"	4A	10	001	
"AL*bb*"	5E	10	001	
"ALR*b*"	1E	01	000	
"AR*bb*"	1A	01	000	
...	...	...	...	
"MVC*b*"	D2	11	100	
...	...	...	...	

b ~ represents the character "blank"

Codes:

Instruction length

01 = 1 half-words = 2 bytes
10 = 2 half-words = 4 bytes
11 = 3 half-words = 6 bytes

Instruction format

000 = RR
001 = RX
010 = RS
011 = SI
100 = SS

FIGURE 3.6 Machine-Op Table (MOT) for pass 1 and pass 2

8-bytes per entry

Pseudo-op (5-bytes) (character)	Address of routine to process pseudo-op (3-bytes = 24 bit address)
"DROP*b*"	P1DROP
"END*bb*"	P1END
"EQU*bb*"	P1EQU
"START"	P1START
"USING"	P1USING

These are presumably labels of routines in pass 1; the table will actually contain the physical addresses.

FIGURE 3.7 Pseudo-Op Table (POT) for pass 1 (similar table for pass 2)

The Symbol Table and Literal Table (Fig. 3.8) include for each entry not only the name and assembly-time value fields but also a length field and a relative-location indicator. The length field indicates the length (in bytes) of the instruc-

tion or datum to which the symbol is attached. For example, consider

```
COMMA   DC    C','
F       DS    F
AD      A     1,F
WORD    DC    3F'6'
```

The symbol COMMA has length 1; F has length 4; AD has length 4 (an add instruction is four bytes long); and WORD has length 4 (the multiplier, 3, is not considered in determining length). The symbol * (current value of location counter) always has a length of 1. If a symbol is equivalent (via EQU) to another, its length is made the same as that of the other. The length field is used by the assembler to calculate length codes used with certain SS-type instructions.

← 14-bytes per entry →

Symbol (8-bytes) (characters)	Value (4-bytes) (hexadecimal)	Length (1-byte) (hexadecimal)	Relocation (1-byte) (character)
"JOHN*bbbb*"	0000	01	"R"
"FOUR*bbbb*"	000C	04	"R"
"FIVE*bbbb*"	0010	04	"R"
"TEMP*bbbb*"	0014	04	"R"

FIGURE 3.8 Symbol Table (ST) for pass 1 and pass 2

The relative-location indicator tells the assembler whether the value of the symbol is absolute (does not change if the program is moved in core), or relative to the base of the program. The assembler can use a simple heuristic to decide into which class a symbol falls. If the symbol is defined by equivalence with a constant (e.g., 6) or an absolute symbol, then the symbol is absolute. Otherwise, it is a relative symbol. The relative-location field in the symbol table will have an "R" in it if the symbol is relative, or an "A" if the symbol is absolute. In the actual assembler a substantially more complex algorithm is generally used.

Figure 3.9 depicts a possible base table that is used by the assembler to generate the proper base register reference in machine instructions and to compute the correct offsets. Basically, the assembler must generate an address (offset, a base register number, an index register number) for most symbolic references. The symbol table contains the address of the symbol relative to the beginning of the program. When generating an address, the assembler may use the base register table to choose a base register that will contain a value closest to the symbolic

reference. The address is then formulated. Base register number = the base register containing a value closest to the symbolic reference. Offset = (value of symbol in symbol table) - (contents of base register).

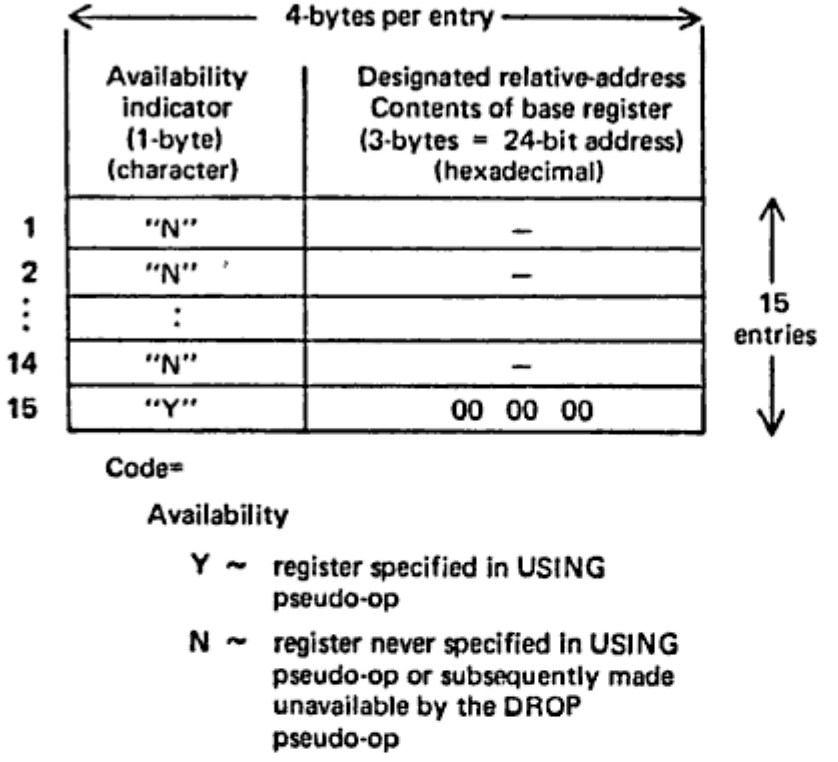


FIGURE 3.9 Base Table (BT) for pass 2

The following assembly program is used to illustrate the use of the *variable tables* (symbol table, literal table, and base table) and to demonstrate the motivation for the algorithms presented in the next section. We are only concerned with the problem of assembling this program; its specific function is irrelevant.

Sample Assembly Source Program

Statement no.			
1	PRGAM2	START	0
2		USING	*,15
3		LA	15,SETUP
4		SR	TOTAL, TOTAL
5	AC	EQU	2
6	INDEX	EQU	3
7	TOTAL	EQU	4
8	DATABASE	EQU	13

Sample Assembly Source Program (continued)

Statement no.			
9	SETUP	EQU	*
10		USING	SETUP, 15
11		L	DATABASE, = A(DATA1)
12		USING	DATAAREA, DATABASE
13		SR	INDEX, INDEX
14	LOOP	L	AC, DATA1 (INDEX)
15		AR	TOTAL, AC
16		A	AC, = F'5'
17		ST	AC, SAVE (INDEX)
18		A	INDEX, = F'4'
19		C	INDEX, = F'8000'
20		BNE	LOOP
21		LR	1, TOTAL
22		BR	14
23		LTORG	
24	SAVE	DS	2000F
25	DATAAREA	EQU	*
26	DATA1	DC	F'25, 26, 97, 101 . . . .' [2000 numbers]
27		END	

In keeping with the purpose of pass 1 of an assembler (define symbols and literals), we can create the symbol and literal tables shown below.

Variable Tables

Symbol Table

Symbol	*Value*	*Length*	*Relocation*
PRGAM2	0	1	R
AC	2	1	A
INDEX	3	1	A
TOTAL	4	1	A
DATABASE	13	1	A
SETUP	6	1	R
LOOP	12	4	R
SAVE	64	4	R
DATAAREA	8064	1	R
DATA1	8064	4	R

Literal Table

A(DATA1)	48	4	R
F'5'	52	4	R
F'4'	56	4	R
F'8000'	60	4	R

As in the flowchart of Figure 3.3, we scan the program above keeping a location counter. For each symbol in the label field we make an entry in the symbol table. For the symbol PRGAM2 its value is its relative location. By IBM convention its length is 1.

We update the location counter, noting that the LA instruction is four bytes long and the SR is two. Continuing, we find that the next five symbols are defined by the pseudo-op EQU. These symbols are entered into the symbol table and the associated values given in the argument fields of the EQU statements are entered. The location counter is further updated noting that the L instruction is four bytes and the SR is two. (None of the pseudo-ops encountered affect the location counter since they do not result in any object code.) Thus the location counter has a value 12 when LOOP is encountered. Therefore, LOOP is entered into the symbol table with a value 12. It is a relocatable variable and so noted. Its length is 4 since it denotes a location that will contain a four-byte instruction. All other symbols are entered in like manner.

In the same pass all literals are recognized and entered into a literal table. The first literal is in statement 11 and its value is the address of the location that will contain the literal. Since this is the first literal, it will have the first address of the literal area. The LTORG pseudo-op (statement 23) forces the literal table to be placed there, where the location counter is updated to the next double word boundary which equals 48. Thus the value of '=A(DATA1)' is its address, 48. Similarly, the value of the literal F'5' is the next location in the literal table, 52 and so on.

Base table (showing only base registers in use)

1) After statement 2:

base	*contents*
15	0

2) After statement 10:

base	*contents*
15	6

3) After statement 12:

base	*contents*
13	8064
15	6

The literal table and the symbol table being completed, we may initiate pass 2 (Fig. 3.4), whose purpose is to evaluate arguments and generate code. To generate proper address in an instruction, we need to know the base register. To compute the offset, we need to know the content of the base register. The assembler of course does not know the execution time value of the base register, but it does know the value relative to the start of the program. Therefore, the assembler enters as "contents" its relative value. This value is used to compute the offset. Processing the USING pseudo-ops produces the base table shown above.

For each instruction in pass 2, we create the equivalent machine language instruction as shown below. For example, for statement 3 we:

1. Look up value of SETUP in symbol table (which is 6)
2. Look up value of op code in machine op table (binary op code for LA)
3. Formulate address
   a. Determine base register — pick one that has content closest to value of SETUP (register 15)
   b. Offset = value of symbol — content of base register = 6-0 = 6
   c. Formulate address: *offset* (*index register, base register*) = 6(0,15)
4. Average output code in appropriate formula

Similarly, we generate instructions for the remaining code as shown below.

Generated "machine" code

Corresponding statement no.	*Location*	*Instruction/datum*	
3	0	LA	15,6 (0,15)
4	4	SR	4,4
11	6	L	13,42 (0,15)
13	10	SR	3,3
14	12	L	2,0 (3,13)
15	16	AR	4,2
16	18	A	2,46 (0,15)
17	22	ST	2,58 (3,15)
18	26	A	3,50 (0,15)
19	30	C	3,54 (0,15)
20	34	BC	7,6 (0,15)
21	38	LR	1,4
22	40	BCR	15,14
23	48	8064	
	52	X'00000005'	
	56	X'00000004'	
	60	8000	

Generated "machine" code (continued)

Corresponding statement no.	*Location*	*Instruction/datum*
24	64	.
	.	.
	.	.
25	8064	X'00000019'
		.
		.
		.

3.2.4 Algorithm

The flowcharts in Figures 3.10 and 3.11 describe in some detail an algorithm for an assembler for an IBM 360 computer. These diagrams represent a simplification of the operations performed in a complex assembler but they illustrate most of the logical processes involved.

PASS 1: DEFINE SYMBOLS

The purpose of the first pass is to assign a location to each instruction and data-defining pseudo-instruction, and thus to define values for symbols appearing in the label fields of the source program. Initially, the Location Counter (LC) is set to the first location in the program (relative address 0). Then a source statement is read. The operation-code field is examined to determine if it is a pseudo-op; if it is not, the table of machine op-codes (MOT) is searched to find a match for the source statement's op-code field. The matched MOT entry specifies the length (2,4 or 6 bytes) of the instruction. The operand field is scanned for the presence of a literal. If a new literal is found, it is entered into the Literal Table (LT) for later processing. The label field of the source statement is then examined for the presence of a symbol. If there is a label, the symbol is saved in the Symbol Table (ST) along with the current value of the location counter. Finally, the current value of the location counter is incremented by the length of the instruction and a copy of the source card is saved for use by pass 2. The above sequence is then repeated for the next instruction.

The loop described is physically a small portion of pass 1 even though it is the most important function. The largest sections of pass 1 and pass 2 are devoted to the special processing needed for the various pseudo-operations. For simplicity, only a few major pseudo-ops are explicitly indicated in the flowchart (Fig. 3.10); the others are processed in a straightforward manner.

We now consider what must be done to process a pseudo-op. The simplest pro-

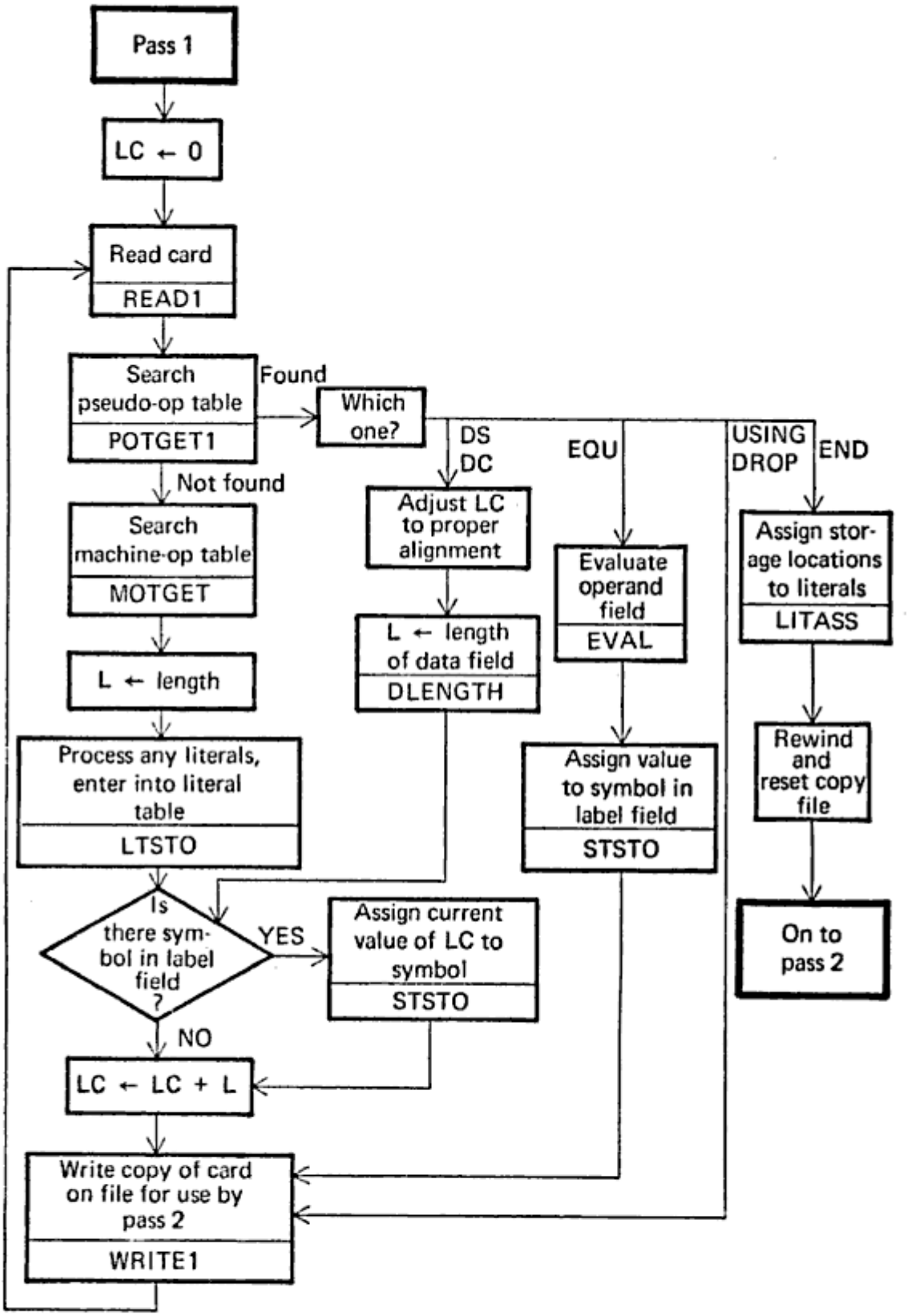


FIGURE 3.10 Detailed pass 1 flowchart

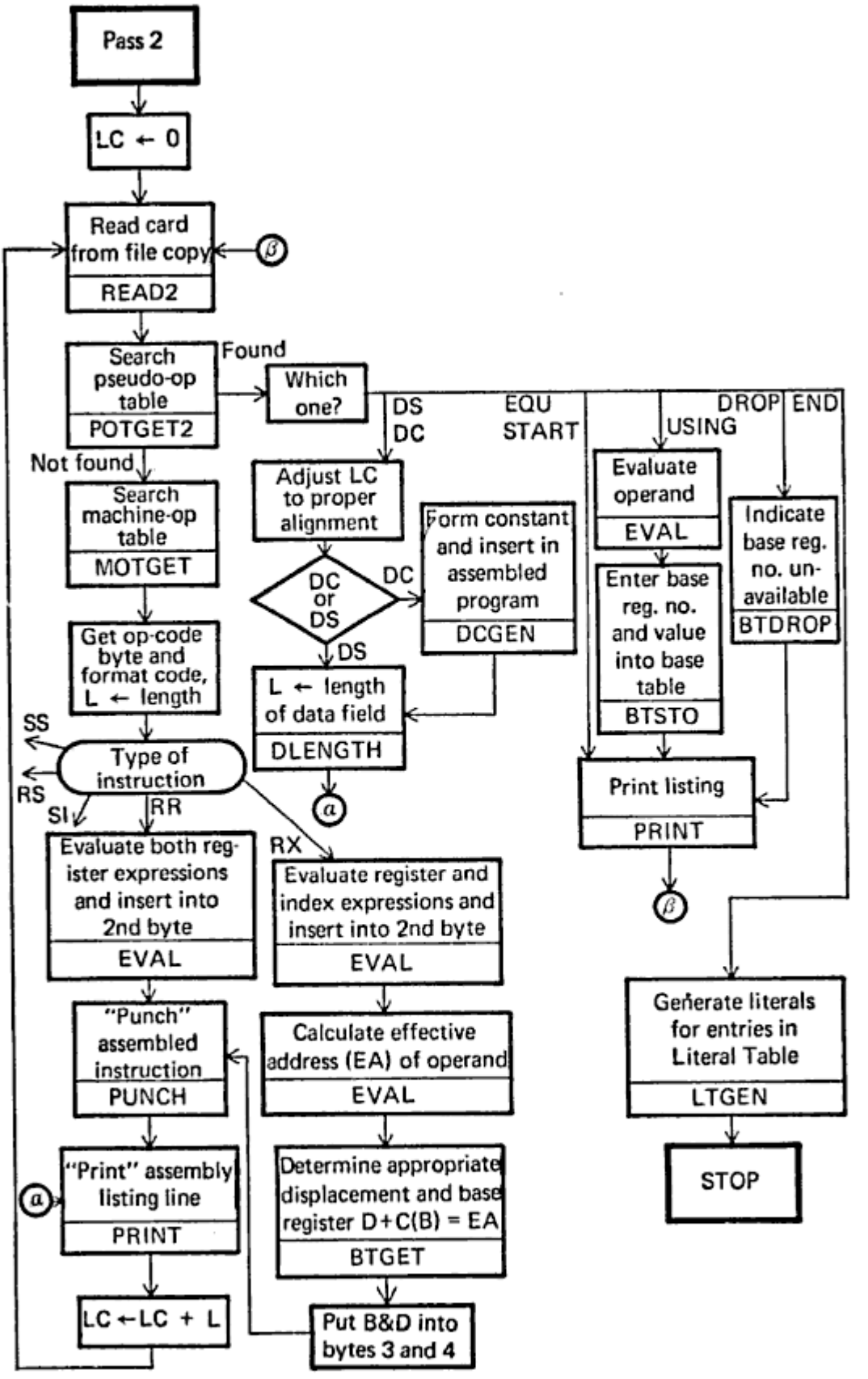


FIGURE 3.11 Detailed pass 2 flowchart

cedure occurs for USING and DROP. Pass 1 is only concerned with pseudo-ops that define symbols (labels) or affect the location counter; USING and DROP do neither. The assembler need only save the USING and DROP cards for pass 2.

In the case of the EQU pseudo-op during pass 1, we are concerned only with defining the symbol in the label field. This requires evaluating the expression in the operand field. (The symbols in the operand field of an EQU statement must have been defined previously.)

The DS and DC pseudo-ops can affect both the location counter and the definition of symbols in pass 1. The operand field must be examined to determine the number of bytes of storage required. Due to requirements for certain alignment conditions (e.g., fullwords must start on a byte whose address is a multiple of four), it may be necessary to adjust the location counter before defining the symbol.

When the END pseudo-op is encountered, pass 1 is terminated. Before transferring control to pass 2, there are various "housekeeping" operations that must be performed. These include assigning locations to literals that have been collected during pass 1, a procedure that is very similar to that for the DC pseudo-op. Finally, conditions are reinitialized for processing by pass 2.

PASS 2: GENERATE CODE

After all the symbols have been defined by pass 1, it is possible to finish the assembly by processing each card and determining values for its operation code and its operand field. In addition, pass 2 must structure the generated code into the appropriate format for later processing by the loader, and print an assembly listing containing the original source and the hexadecimal equivalent of the bytes generated. The Location Counter is initialized as in pass 1, and the processing continues as follows.

A card is read from the source file left by pass 1. As in pass 1, the operation code field is examined to determine if it is a pseudo-op; if it is not, the table of machine op-codes (MOT) is searched to find a match for the card's op-code field. The matching MOT entry specifies the length, binary op-code, and the format-type of the instruction. The operand fields of the different instruction format types require somewhat different processing.

For the RR-format instructions, each of the two register specification fields is evaluated. This evaluation may be very simple, as in:

```
AR      2,3
```

or more complex, as in:

```
MR      EVEN , EVEN + 1
```

The two fields are inserted into their respective four-bit fields in the second byte of the RR-instruction.

For RX-format instructions, the register and index fields are evaluated and processed in the same way as the register specifications for RR-format instructions. The storage address operand is evaluated to generate an Effective Address (EA). Then the base register table (BT) must be examined to find a suitable base register (B) such that D = EA - c(B) < 4096. The corresponding displacement can then be determined. The 4-bit base register specification and 12-bit displacement fields are then assembled into the third and fourth bytes of the instruction. Only the RR and RX instruction types are explicitly shown in the flowchart (Fig. 3.11). The other instruction formats are handled similarly.

After the instruction has been assembled, it is put into the necessary format for later processing by the loader. Typically, several instructions are placed on a single card (see Chapter 5 for a more detailed discussion). A listing line containing a copy of the source card, its assigned storage location, and its hexadecimal representation is then printed. Finally, the location counter is incremented and processing is continued with the next card.

As in pass 1, each of the pseudo-ops calls for special processing. The EQU pseudo-op requires very little processing in pass 2, because symbol definition was completed in pass 1. It is necessary only to print the EQU card as part of the printed listing.

The USING and DROP pseudo-ops, which were largely ignored in pass 1, require additional processing in pass 2. The operand fields of the pseudo-ops are evaluated; then the corresponding Base Table entry is either marked as available, if USING, or unavailable, if DROP. The base table is used extensively in pass 2 to compute the base and displacement fields for machine instructions with storage operands.

The DS and DC pseudo-ops are processed essentially as in pass 1. In pass 2, however, actual code must be generated for the DC pseudo-op. Depending upon the data types specified, this involves various conversions (e.g., floating point character to binary representation) and symbol evaluations (e.g., address constants).

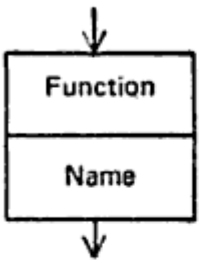
The END pseudo-op indicates the end of the source program and terminates the assembly. Various "housekeeping" tasks must now be performed. For example, code must be generated for any literals remaining in the Literal Table (LT).

3.2.5 Look for Modularity

We now review our design, looking for functions that can be isolated. Typically,

such functions fall into two categories: (1) multi-use and (2) unique.

In the flowcharts for pass 1 (Fig. 3.10) and pass 2 (Fig. 3.11), we examine each step as a candidate for logical separation. Likely choices are identified in the flowcharts by the shape



where "name" is the name assigned to the function (e.g., MOTGET, EVAL, PRINT).

Listed below are some of the functions that may be isolated in the two passes.

PASS 1:

1. READ1	—	Read the next assembly source card.
2. POTGET1	—	*Search* the pass 1 Pseudo-Op Table (POT) for a match with the operation field of the current source card.
3. MOTGET1	—	*Search* the Machine-Op Table (MOT) for a match with the operation of the current source card.
4. STSTO	—	*Store* a label and its associated value into the Symbol Table (ST). If the symbol is already in the table, return error indication (multiply-defined symbol).
5. LTSTO	—	*Store* a literal into the Literal Table (LT); do not store the same literal twice.
6. WRITE1	—	Write a copy of the assembly source card on a storage device for use by pass 2.
7. DLENGTH	—	*Scan* operand field of DS or DC pseudo-op to determine the amount of storage required.
8. EVAL	—	*Evaluate* an arithmetic expression consisting of constants and symbols (e.g. 6, ALPHA, BETA + 4 * GAMMA).
9. STGET	—	*Search* the Symbol Table (ST) for the entry corresponding to a specific symbol (used by STSTO, and EVAL).
10. LITASS	—	Assign storage locations to each literal in the literal table (may use DLENGTH).

PASS 2:

1. READ2	—	Read the next assembly source card from the file copy.
2. POTGET2	—	Similar to POTGET1 (search POT).
3. MOTGET2	—	Same as in pass 1 (search MOT).
4. EVAL	—	Same as in pass 1 (evaluate expressions).
5. PUNCH	—	Convert generated instruction to card format; punch card when it is filled with data.
6. PRINT	—	Convert relative location and generated code to character format; print the line along with copy of the source card.
7. DCGEN	—	Process the fields of the DC pseudo-op to generate object code (uses EVAL and PUNCH).
8. DLENGTH	—	Same as in pass 1.
9. BTSTO	—	Insert data into appropriate entry of Base Table (BT).
10. BTDROP	—	Insert "unavailable" indicator into appropriate entry of BT.
11. BTGET	—	Convert effective address into base and displacement by searching Base Table (BT) for available base registers.
12. LTGEN	—	Generate code for literals (uses DCGEN).

Each of these functions should independently go through the entire design process (problem statement, data bases, algorithm, modularity, etc.). These functions can be implemented as separate external subroutines, as internal subroutines, or as sections of the pass 1 and pass 2 programs. In any case, the ability to treat functions separately makes it much easier to design the structure of the assembler and each of its parts. Thus, rather than viewing the assembler as a single program (of from 1,000 to 10,000 source statements typically), we view it as a coordinated collection of routines each of relatively minor size and complexity.

We will not attempt to examine all of these functional routines in detail since they are quite straightforward. There are two particular observations of interest: (1) several of the routines involve the scanning or evaluation of fields (e.g., DLENGTH, EVAL, DCGEN); (2) several other routines involve the processing of tables by storing or searching (e.g., POTGET1, POTGET2, MOTGET1, MOTGET2, LTSTO, STSTO, STGET). The section of this book dealing with the implementation of compilers (Chapter 8) will discuss techniques for parsing fields and evaluating arithmetic expressions, many of which are also applicable to the functional modules of the assembler.

Table processing, as discussed in regard to assembler implementation, is found in almost every type of system program, including compilers, loaders, file systems and operating systems, as well as in many application programs. The general topic of processing data structures and data organizations plays a crucial role in systems programming. Since storing and searching for entries in tables often represent the largest expenditures of time in an assembler, the next section examines some techniques for organizing these tasks.

3.3 TABLE PROCESSING: SEARCHING AND SORTING

THE PROBLEM

It is often necessary to maintain large tables of information in such a way that items may be moved in and out quickly and easily. Let us consider the restricted case of a table whose entries are made on the basis of a keyword, such as the *symbol table* maintained by an assembly program.

The assembler symbol table is composed of multiple-word entries in a fixed format. In the table is the symbol name, its value, and various attributes, such as relocatability. The symbol name is the *key*, the string distinguishing each entry from the others that is matched during a search. Each symbol has a corresponding location, its *value*. (Analogously, in a telephone directory a subscriber's name is the key and his telephone number is the value.) There are two important things to notice about the assembler symbol table:

1. The symbols are placed in the table in the order in which they are gathered, so the table is unlikely to be ordered.
2. The symbols and their associated data are placed in consecutive locations in the table. They are all packed starting at one end of the table.

These two statements are true of most tables constructed one entry at a time without an encoding of addresses.

SEARCHING A TABLE

The problem of *searching* is as follows: given a keyword, find an entry in the table that matches, and return its value. The special problems – more than one entry with the same keyword, and no entry found – require individual treatment depending on the function of the table. In the case of an assembler's symbol table, these special cases correspond to multiply defined symbols and undefined symbols.

3.3.1 Linear Search

For a table in which the items have not been ordered (by sorting or otherwise), one way to look for a given keyword is to compare exhaustively every entry in the table with the given keyword. This is known as a *linear search* and is demonstrated in Figure 3.12.

```
            LA      4,SYMTBL          Start of table
LOOP        CLC     0(8,4),SYMBOL     Compare symbols
            BE      SYMFOUND          Equal
            A       4,=F'14'          Move to next symbol
            C       4,LAST            Are we at end of table
            BNE     LOOP              Loop back if not
NOTFOUND            (Symbol not found)
              :
SYMFOUND            (Symbol found)
              :
SYMBOL      DS      CL14              Symbol to be searched for, character
                                            string of length 14
SYMTBL      DS      100CL14           Symbol table space (14 bytes per entry)
LAST        DC      A(----)           Address of current end of symbol table
```

FIGURE 3.12 Sample linear search program

Here the symbols and values are stored in adjacent locations in an array named SYMTBL and defined by a DS. The word LAST contains the location of the current "end of table."

The loop described will compare the keyword (in the location SYMBOL) with each successive item in the table. When a match is found, exit is made via SYMFOUND; if no match is found by the end of the table, execution will go to location NOTFOUND.

On the average we would search half the table, using a linear search, before finding an entry. Therefore, the average length of time to find an entry is

$$T(\text{avg}) = [\text{overhead associated with entry probe}] \times \frac{N}{2}$$

Such a *linear search* procedure is fine for short tables and has the great virtue of simplicity, but for long tables it can be very slow. It is comparable to looking up a word in a dictionary whose contents are not ordered. It would provide little comfort to know that on the average you only have to search *half* of the dictionary.

3.3.2 Binary Search

When consulting a dictionary, we don't search every page for the definition of a word. We make a vague estimate of the location of our word in the dictionary (i.e., page number) and we open to that page. If the word is not on it, we go either to the right or left a number of pages and check again. We know which way to go because we are aware of an important property possessed by the dictionary, namely, it is ordered (B follows A, and S comes way after G). Such ordering of letters is called a *lexicographical* order.

A more systematic way of searching an *ordered* table is: Start at the middle of the table and compare the keyword with the middle entry. The keyword may be equal to, greater than, or smaller than the item checked. The next action taken for each of these outcomes is as follows:

1. If equal, the symbol is found.
2. If greater, use the top half of the given table as a new table to search.
3. If smaller, use the bottom half of the table.

This method, effectively divides the table in half on every probe, systematically bracketing the item searched for. The search is terminated with a 'not found' condition when the length of the last subtable searched is down to 1 and the item has not been found.

As an example of this type of search, consider a table of 15 items (Fig. 3.13). Suppose for example that we are searching for item IF (for convenience the values are not shown). We first compare IF with the middle item LO and find that IF must be in the top half of the table; a second comparison with the middle item in this half of the table, FU, shows IF to be in the second quarter; a third comparison with IW shows IF to be in the third eighth of the table (i.e., between items 4 and 6); and a final comparison is made with the item in position 5. A comparison failure on the fourth probe would have revealed that the item did not exist in the table.

This bracketing method of searching a table should be clear in principle although its implementation may be a little more complicated. It is known as a *binary search* or a *logarithmic search*, and it should be clear that since the effective table is halved on each probe, a maximum of about $\log_2(N)$ probes is required to search it.

Comparing the times of the linear search with those of the binary search, where A and B are overhead times associated with each table probe, we obtain

$$T(\text{lin}) = A \cdot N$$
$$T(\text{bin}) = B \cdot \log_2(N);$$

Since the binary search is more complicated, we can expect the constant B to

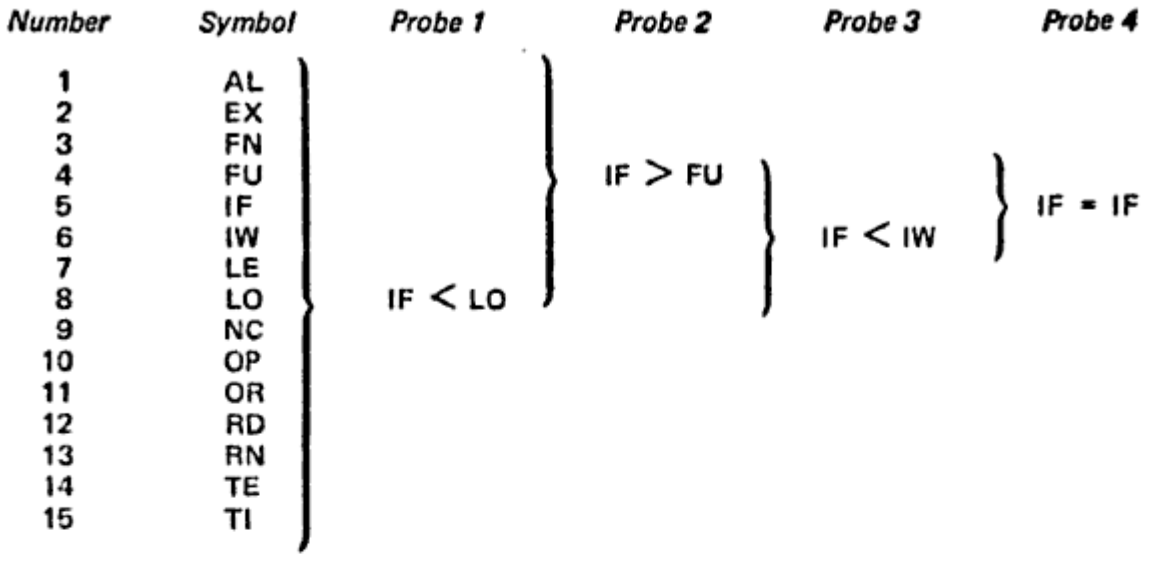


FIGURE 3.13 Illustration of binary search

be considerably larger than A. Thus, a plot of T versus N for the two searching methods might look like Figure 3.14.

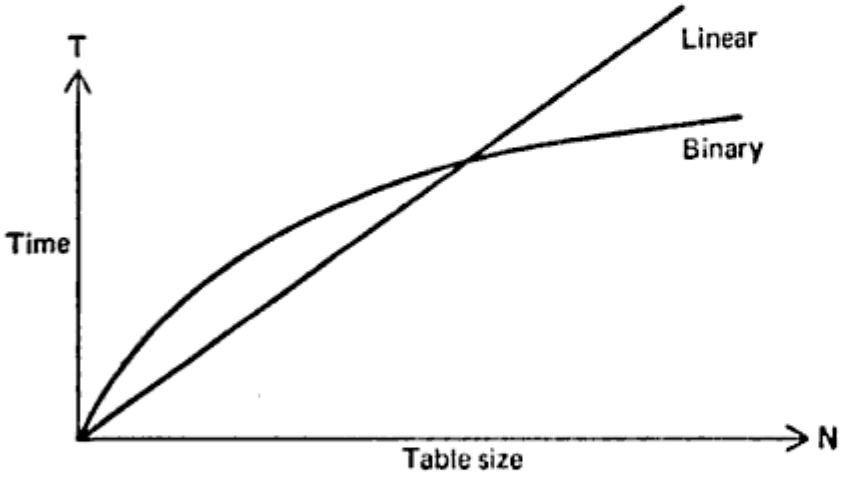


FIGURE 3.14 Search time versus N

Thus for small N we should use linear search while for large N we should use a binary search. The crossover point is generally around 50 - 100 items for machines like the 360. For other computers the number might vary from 10 to 1,000 depending on the available hardware.

Figure 3.15 depicts a sample binary search program. Since the binary process continually divides by 2, for efficiency and simplicity we assume table size is a power of 2 (e.g., 2,4,8,16, . . . etc.). This condition is easily attained by merely

adding sufficient "dummy" entries to the end of the table (e.g., entries for the symbol ZZZZ ZZZZ).

```
                L       5,SIZE          Set table size (2^N*14 bytes)
                SRL     5,1             Divide by 2 by shifting
                LR      6,5             Copy into register 6
LOOP            SRL     6,1             Divide table size in half again
                LA      4,SYMTBL(5)     Set address of table entry
                CLC     0(8,4),SYMBOL   Compare with symbol
                BE      FOUND           Symbols match, entry found
                BH      TOOHIGH         SYMTBL entry > SYMBOL
TOOLOW          AR      5,6             Move higher in table
                B       TESTEND
TOOHIGH         SR      5,6             Move lower in table
TESTEND         LTR     6,6             Test if remaining size is 0
                BNZ     LOOP            No, look at next entry
NOTFOUND        (Symbol not found)
                  :
FOUND           (Symbol found)
```

FIGURE 3.15 Sample binary search program

3.3.3 Sorting

It seems clear that for some purposes a binary search is more efficient than a linear one; but such a search requires an ordered table, which may not be easily obtainable. The Machine-Op Table (MOT) and Pseudo-Op Table (POT) of the assembler are fixed tables that can be manually organized so as to be ordered. Normally, however, a table is not generated in an ordered fashion; indeed, the symbol table created by an assembler is usually far from ordered, the symbols being stored in the exact order in which they appear in label fields.

3.3.3.1 INTERCHANGE SORT

We now address the problem of how to sort a table. There are a number of ways of doing this, some simple and some complicated. Figure 3.16 is a section of coding that performs an *interchange sort* (also known as a *bubble sort*, a *sinking sort* or a *sifting sort*). This simple sort takes adjacent pairs of items in the table and puts them in order (interchanges them) as required. Such a sorting algorithm is not very efficient, but it is simple. Let us take an example to see how it works. Consider the table of 12 numbers shown in Figure 3.17; each column represents one pass over the numbers interchanging any two adjacent numbers that are out of order. This particular table is completely sorted in only seven passes over the data. In the worst case, N-1 (here, 11) passes would be necessary. The inter-

change sort is thus able to exploit whatever natural order there may be in the table. Moreover, on each pass through the data at least one item is added to the bottom of the list in perfect order (in this case the 31 first, then 27, then 26, etc.). Hence, the sort could be made more efficient by (1) shortening the portion of the sorted list on each pass; and (2) checking for early completion. Such an optimized sort should require roughly N*(N-1)/2 comparisons and thus should take a time roughly proportional to N^2.

```
         L       5,LAST
         LA      4,SYMTBL
LOOP     CLC     0(8,4),14(4)      Compare adjacent symbols — 8 bytes
         BNH     OK                Correct order
         MVC     TEMP(14),0(4)     Switch entries
         MVC     0(14,4),14(4)     . . .
         MVC     14(14,4),TEMP     . .
OK       A       4,=F'14'          Move to next entry
         C       4,LAST            Is it last entry
         BNE     LOOP              No
          ⋮
SYMTBL   DS      0F                Symbol table
         DS      100CL14           14 bytes per entry
TEMP     DS      CL14              Temporary entry
LAST     DC      A(-----)          Location of next free entry in table
```

FIGURE 3.16 Interchange sort example in 360 assembly code

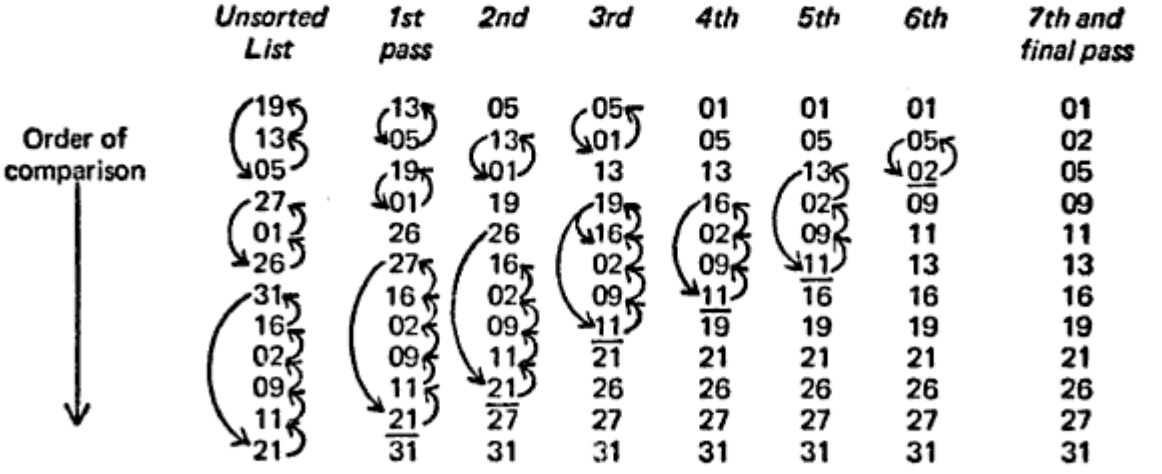


FIGURE 3.17 Illustration of interchange sort

We would like an even better sorting method which requires less time. Sorting methods fall into one of three basic types: (1) *distributive* sorts which sort by examining entries one digit at a time; (2) *comparative* sorts which sort by

comparing keywords two at a time; and (3) *address calculation* sorts that transform a key into an address close to where the symbol is expected to end up.

3.3.3.2 SHELL SORT

A fast comparative sort algorithm is due to D.L. Shell (see bibliography, Chapter 10) and is referred to as a *Shell sort*. It approaches optimal performance for a comparative type of sort. The Shell sort is similar to the interchange sort in that it moves data items by exchanges. However, it begins by comparing items a distance "d" apart. This means that items that are way out of place will be moved more rapidly than a simple interchange sort. In each pass the value of d is decreased usually;

$$d_{i+1} = \frac{d_i+1}{2}$$

l every pass, each item is compared with the one located d positions further in the vector of items. If the higher item has a lower value, an exchange is made. The sort continues by comparing the next item in the vector with an item d locations away (if one exists). If an exchange is again indicated, it is made and the comparison is tried again with the next entry. This proceeds until no lower items remain. This process is called bubbling; if you think of low valued items floating to the top, the behavior in the subprocess is like that of a bubble in a water tank. After bubbling can no longer occur with a fixed value of d, the process begins again with a new d.

It is difficult to predict the time requirements of the Shell sort since it is hard to show the effect of one pass on another. It should be obvious that if the above method of calculating d is used, the number of passes will be approximately $\log_2(d)$ since bubbling when d=1 will complete the sort. Empirical studies have shown that the Shell sort takes approximately $B*N*(\log_2 N)^2$ units of time for an N element vector. The constant of proportionality B is fairly small, so for small N the Shell sort will out-perform the radix exchange sort described in section 3.3.3.4 (N up to 1,000). An example of a Shell sort is given in Figure 3.18.

3.3.3.3 BUCKET SORT

One simple distributive sort is called the *radix sort* or *bucket sort*. The sort involves examining the *least* significant digit of the keyword first, and the item is then assigned to a bucket uniquely dependent on the value of the digit. After all items have been distributed, the "buckets" items are merged in order and then the process is repeated until no more digits are left. A number system of base P requires P buckets.

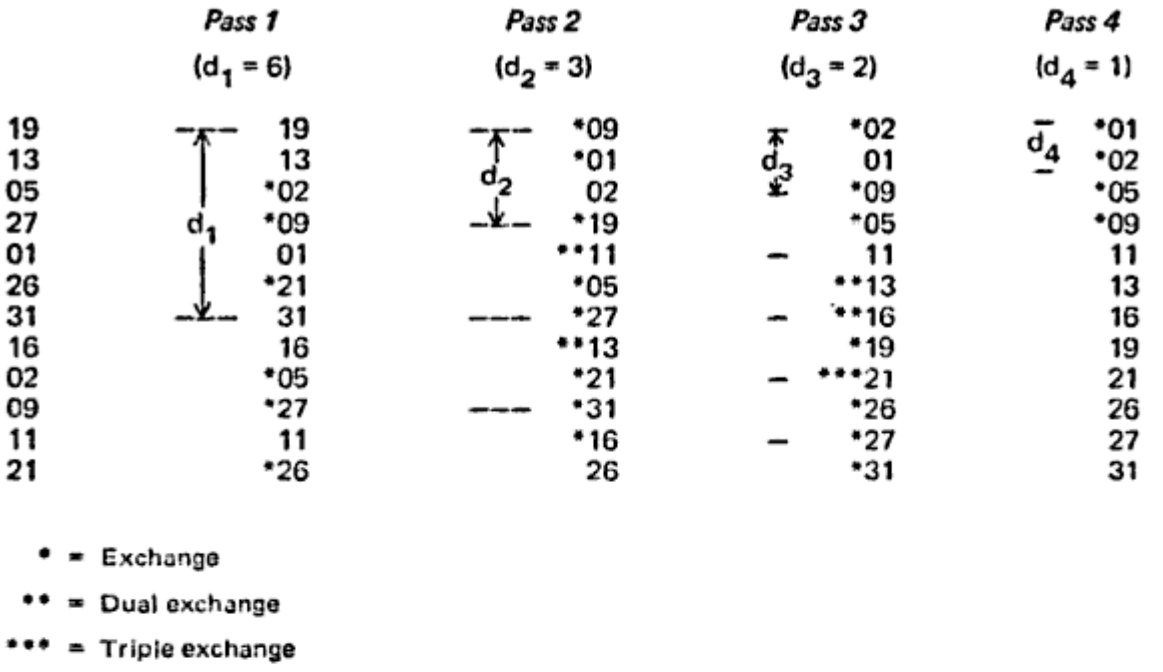


FIGURE 3.18 Example of a Shell sort

Consider for example the radix sorting of the numbers as shown in Figure 3.19. You should be able to figure out rather quickly how this sort works. In fact, this is precisely the method used on a card sorting machine. However, there are serious disadvantages to using it internally on a digital computer (or on tape sorts for that matter): (1) it takes two separate processes, a separation and a merge; and (2) it requires a lot of extra storage for the buckets. However, this

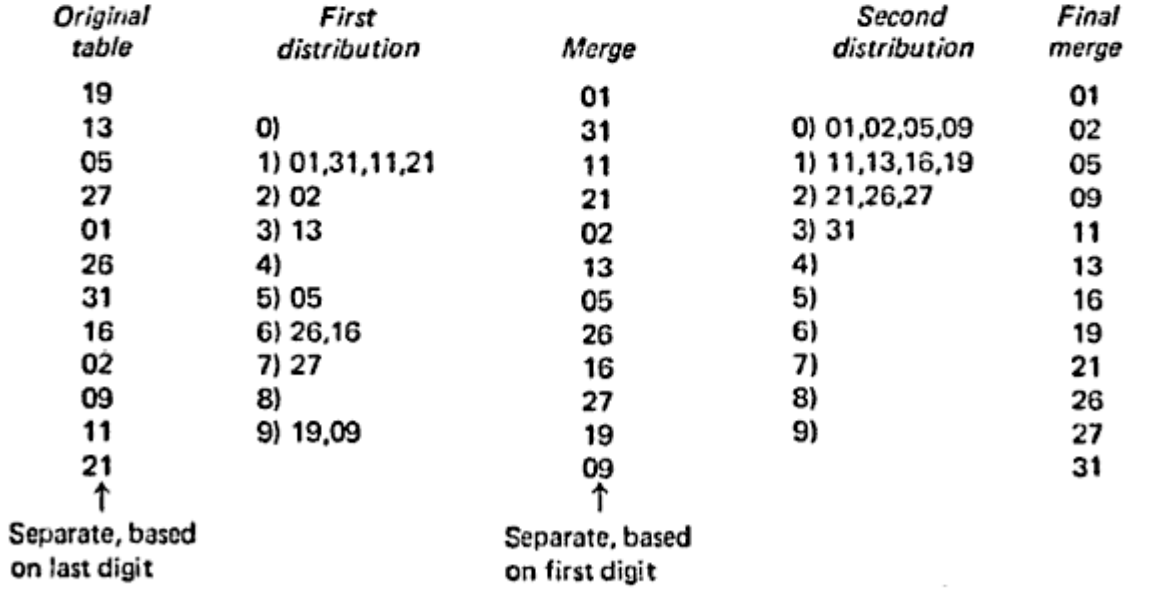


FIGURE 3.19 Demonstration of radix sorting

last disadvantage can be overcome by chaining records within a logical "bucket" rather than pre-allocating maximum size buckets.

3.3.3.4 RADIX EXCHANGE SORT

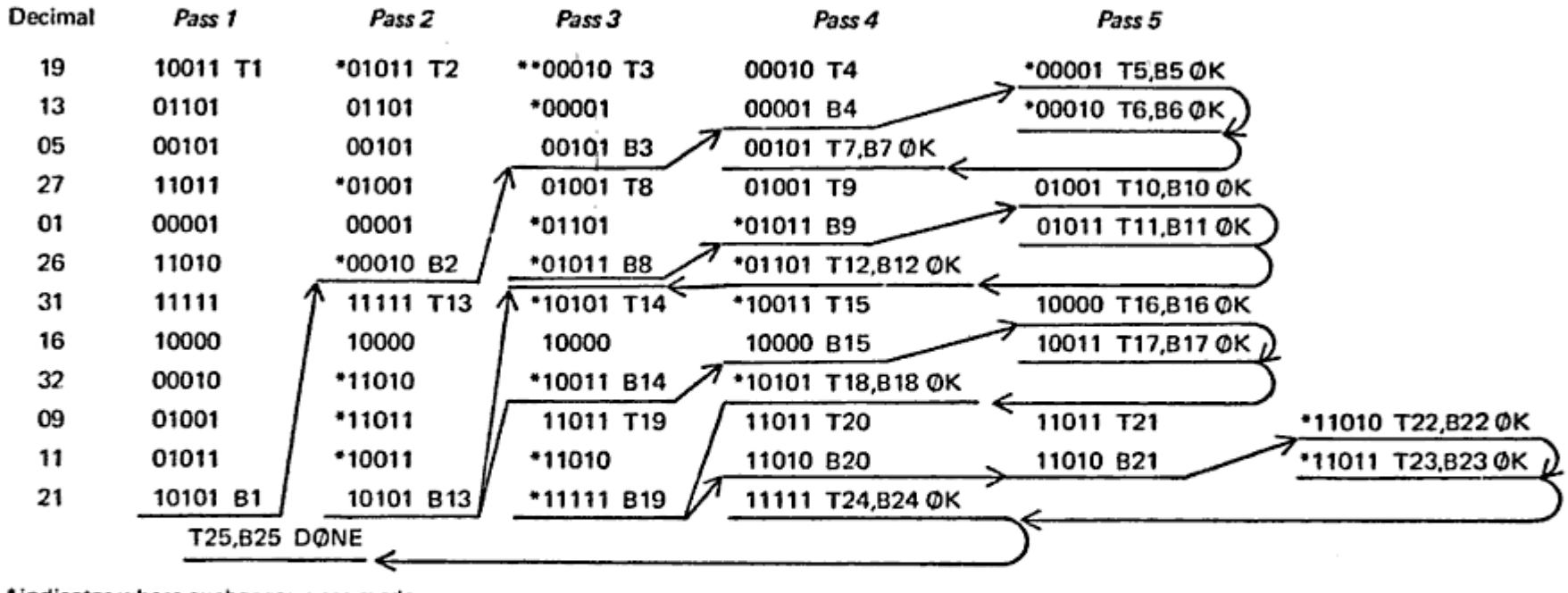
A considerably better distributive sort is the *radix exchange* sort which is applicable when the keys are expressed (or are expressable) in binary. Sorting is accomplished by considering groups with the same (M) first bits and ordering that group with respect to the (M+1)st bit. The ordering of a group on a given bit is accomplished by scanning down from the top of the group for a one bit and up from the bottom for a zero bit; these two are exchanged and the sort continues. This algorithm requires the program to keep up with a large number of groups and, coded in a bad form, could require an additional table N long. However, with optimal coding it is possible to keep track of the groups by simply monitoring the top of the table and a list of break points, one for each bit of the keyword. (Thus with 32 bit words a table of only 33 entries is required.) An example of the radix exchange sort is shown in Figure 3.20. It is a rather complicated example and somewhat difficult to understand – qualities characteristic of most distributive sorts.

If the sort algorithm is programmed to quit sorting when a group contains only one item, then the time required for the radix exchange sort is proportional to N*log(N) as compared to $N*\log_p(K)$ for the bucket sort (here K is the maximum key size and p is the radix). Note that the radix exchange sort does not require extra table space for "buckets."

3.3.3.5 ADDRESS CALCULATION SORT

The last example is of the *address calculation sort*. This can be one of the fastest types of sorts if enough storage space is available. The sorting is done by transforming the key into an address in the table that "represents" the key. For example, if the key were four characters long, one method of calculating the appropriate table address would be to divide the key by the table length in items, multiply by the length of an item and add the address of the table. If the table length is a power of 2, then the division reduces to a shift. This sort would take only N* (time to calculate address) if it were known that no two keys would be assigned the same address. However, in general, this is not the case and several keys will be reduced to the same address.

Therefore, before putting an item at the calculated address it is necessary to first check whether that location is already occupied. If so, the item is compared with the one that is already there, and a linear search in the correct direction is performed to find the correct place for the new item. If we are lucky, there will be an empty space in which to put the items in order. Otherwise, it will be necessary to move some previous entries to make room. It is the search and



*indicates where exchanges were made

FIGURE 3.20 Example of radix exchange sort

moves that increase the time required for this type of sort.

The time required for the sort can be decreased by making the table bigger than the number of values it will be required to hold. This will provide more open spaces in the table and create less likelihood of address conflicts, long searches or long moves. With a table about 2.2 times the size of the data to sort, this approach (allowing for a final pass to compact the table) uses time proportional to N, making it the fastest type of sort.

An example of an address calculation sort is given in Figure 3.21. The table is of size 12; since it is known that the maximum key is less than 36, the address transformation is to divide the key by 3 and take the integer part (e.g., 19/3 = 6 + 1/3 so use 6). A "*" indicates a conflict between keys, and an arrow indicates when a move is necessary and in which direction. The associated addresses calculated are given in the second row.

3.3.3.6 COMPARISON OF SORTS

We have discussed five different kinds of sorts: interchange, radix, radix-exchange, Shell, and address calculation. The characteristics for each of these sorts are presented below:

Type	*Average time (approx)*	*Extra storage (wasted space)*
Interchange	$A*N^2$	None
Shell	$B*N*(\log_2(N))^2$	None
Radix	$C*N*\log_p(K)$	N*p
Radix exchange	$D*N*\log_2(N)$	k+1
Address calculation	E*N	2.2*N (approximate)

where N is the table size, K is the maximum key size (32 generally, on the 360), and p is the radix of the radix sort. A, B, C, D, and E are the constants of proportionality.

Comparative sorts can take a time from roughly proportional to N^2 down to roughly proportional to N*log(N). They are insensitive to the distribution of magnitudes; they take good advantage of any natural order in the data; and they generally require no extra storage.

Distributive sorts usually require a time roughly proportional to $N*\log_p(K)$ (where p is the radix of the number system employed and K is the maximum size of the key) because there are N numbers with digits to check and there are $\log_p(K)$ digits per number. However, the distributive sorts are sensitive to the distribution of magnitudes of the entries; they can make scant use of any natural order in the data; and they quite often require considerable additional storage.

Address calculation sorts usually operate quite fast on the first elements entered in the table since address conflicts are unlikely. However, as the table

Data number =	1	2	3	4	5	6	7	8	9	10	11	12
Data =	19	13	05	27	01	26	31	16	02	09	11	21
Calculated address =	6	4	1	9	0	8	10	5	0	3	3	7
Table =												
0	—	—	—	—	01	01	01	01	*01	01	01	01
1	—	—	05	05	05	05	05	05	02	02	02	02
2	—	—	—	—	—	—	—	—	05	*05	05	05
3	—	—	—	—	—	—	—	—	—	09	*09	09
4	—	13	13	13	13	13	13	13	13	13	11	11
5	—	—	—	—	—	—	—	16	16	16	13	13
6	19	19	19	19	19	19	19	19	19	19	16	16
7	—	—	—	—	—	—	—	—	—	—	19	*19
8	—	—	—	—	—	26	26	26	26	26	26	21
9	—	—	—	27	27	27	27	27	27	27	27	26
10	—	—	—	—	—	—	31	31	31	31	31	27
11	—	—	—	—	—	—	—	—	—	—	—	31

FIGURE 3.21 Example of address calculation sort

fills up, the time to add a new entry increases exponentially.

In summary, the *interchange sort* is by far the simplest and should on that account be used whenever speed is not crucial. The *radix* sort is efficient in time but requires an inordinate amount of space so that it is seldom used on a computer; for card sorting, where space is no problem, this is a good sort. The *radix-exchange* sort is very fast and requires very little extra storage (roughly 32 words on the 360), but it is difficult to program and debug. The *address calculation* sort requires the most space to be efficient, but if such space is available, it is faster than any other method.

3.3.4 Hash or Random Entry Searching

Binary search algorithms, while fast, can only operate on tables that are *ordered* and *packed*, i.e., tables that have adjacent items ordered by keywords. Such search procedures may therefore have to be used in conjunction with a sort algorithm which both orders and packs the data.

Actually, it is unnecessary for the table to be ordered and packed to achieve good speed in searching. As we shall presently see, it is possible to do considerably better with an unpacked, unordered table, provided it is sparse, i.e., the number of storage spaces allocated to it exceeds the number of items to be stored.

We have already observed that the address calculation sort gives good results

with a sparse table. However, having to put elements in order slows down the process. A considerable improvement can be achieved by inserting elements in a random (or pseudo-random) way. The random entry-number K is generated from the key by methods similar to those used in address calculation. If the Kth position is void, then the new element is put there; if not, then some other cell must be found for the insertion.

The first problem is the generation of a random number from the key. Of course, we don't really want a random number in the sense that a given keyword may yield one position today and another tomorrow. What we want is a procedure that will generate pseudo-random, consistent table positions for keywords. One fairly good prospect for four character EBCDIC keywords is to simply divide the keyword by the table length N and use the remainder. This scheme works well as long as N and the key size (32 bits in our case) have no common factors. For a given group of M keywords, the remainders should be fairly evenly distributed over 0···(N-1). Another method is to treat the keyword as a binary fraction and multiply it by another binary fraction:

```
L    1,SYMBOL
M    0,RHO
```

The result is a 64-bit product in registers 0 and 1. If RHO is chosen carefully, the low order 31 bits will be evenly distributed between 0 to 1, and a second multiplication by N will generate a number uniformly distributed over 0···(N-1). This is known as the *power residue* method. It has the advantage that the 31-bit first result can be used to generate *another* uniformly distributed number (by multiplying again by RHO) in the event that the first probe of the table is unsuccessful.

The second problem is the procedure to be followed when the first trial entry results in a filled position. There are a number of methods of resolving this problem, three of which will be covered here. These are:

1. *Random entry with replacement* A sequence of random numbers is generated from the keyword (such as by the power residue method). From each of these a number between 1 and N is formed and the table is probed at that position. Probings are terminated when a void space is found. Notice that the random numbers generated are independent and it is perfectly possible (but not likely) to probe the same position twice.

2. *Random entry without replacement* This is the same as above except that any attempt to probe the same position twice is bypassed. This method holds advantage over the above only when *probes* are expensive, e.g., for files on tape or drum.

3. *Open addressing* If the first probe gives a position K and that position is filled, then the next location K+1 is probed, and so on until a free position is found. If the search runs off the bottom of the table, then it is renewed at the top (i.e., the table is considered to be cyclic).

Of these three perhaps the open addressing scheme is the simplest. An example here should serve to illustrate this method.

Consider a table of 17 positions (N=17) in which the following twelve numbers are to be stored: 19, 13, 05, 27, 01, 26, 31, 16, 02, 09, 11, 21. These items are to be entered in the table at a position defined by the remainder after division by 17; if that position is filled, then the next position is examined, etc. Figure 3.22 shows the progress of entry for the 12 items; notice the resolution of conflicts on items 02, 09, and 11. The column 'Probes to find' gives the number of probes necessary to find the corresponding items in the table; thus it takes 3 probes to find items 09, and 1 to find item 26. The column 'Probes to find not' gives the number of probes necessary to determine that an item is not in the table; thus the search for the number 54 would give an initial position of 3 and it would take 4 probes to find that the item is not present. The item is known not to be present when a void position is encountered (position 6 in this case). Notice here that the following figures hold·

Length of table	$N = 17$
Items stored	$M = 12$
Density	$\rho = 12/17 = 0.705$
Probes to store	$T_s = 16$
Average probes to find	$T_p = 16/12 = 1.33$
Average probes to find not	$T_n = 54/16 = 3.37$

The comparative times for a packed table, using radix exchange sort and binary search, are as follows:

Probes to store and sort	$T_s = M+M \cdot \log_2(M) = 55$
Average probe to find	$T_p = \log_2(M) = 3.58$
Average probe to find not	$T_n = \log_2(M) = 3.58$

Thus, it would appear that the open addressing scheme holds considerable advantage in speed, but it pays for this by having a table nearly 50 percent longer than necessary. Furthermore, the table cannot be compressed after its initial assignment nor can the assigned area be easily shared among several tables. One final very serious disadvantage is that it is very difficult to *delete* any item from the table – one cannot simply zero out that location because this might break the addressing chain.

It is interesting to consider the expected probe time, etc., for the random entry

Position	*Item*	*Probes to find*	*Probes to find not*
0			1
1	01	1	6
2	19, 02*	1	5
3	02	2	4
4	21	1	3
5	05	1	2
6			1
7			1
8			1
9	26, 09*	1	7
10	27, 09*	1	6
11	09, 11*	3	5
12	11	2	4
13	13	1	3
14	31	1	2
15			1
16	16	1	1
		16	54

FIGURE 3.22 Example of open addressing

methods. The simpler method to evaluate is *random entry with replacement.* Consider a table of N positions with K-1 elements already inserted. We define density $\rho = (K\text{-}1)/N$.

$$\text{Probes to store K}^{\text{th}}\text{: } L_K = \frac{1}{1\text{-}\rho}$$

$$\text{Probes to search: } T_P = \frac{1}{\rho} \log_e \frac{1}{1\text{-}\rho}$$

These figures are very interesting and illustrate well the tradeoff between search time and table density.

For open addressing the figures are considerably different. As the table becomes denser, the probability of long strings becomes greater so that more probes are required. In fact, it can be shown that the number of probes to find is roughly

$$T_P(\rho) = 1 + \frac{\rho}{2} \cdot \frac{1}{1\text{-}\rho}$$

and that the time to determine that an item is not in the table is roughly

$$T_N(\rho) = \frac{1}{1\text{-}\rho}$$

OTHER BOOKS OF INTEREST

DHAMDHERE	:	Systems Programming and Operating Systems, 2/e
DHAMDHERE	:	Operating Systems: A Concepts – Based Approach

The McGraw-Hill Companies

Tata McGraw-Hill
Publishing Company Limited
7 West Patel Nagar, New Delhi 110 008

Visit our website at : www.tatamcgrawhill.com

ISBN-13: 978-0-07-460482-3
ISBN-10: 0-07-460482-1

9 780074 604823