systems programming

JOHN J. DONOVAN

TATA McGRAW-HILL EDITION
FOR SALE IN INDIA ONLY

contents

52599

PREFACE xiii

ACKNOWLEDGEMENTS xvi

NOTE TO THE STUDENT xviii

1. BACKGROUND 1
1. Machine Structure 2
2. Evolution of the Components of a Programming System 4
   1. Assemblers 4
   2. Loaders 4
   3. Macros 6
   4. Compilers 7
   5. Formal Systems 7
3. Evolution of Operating Systems 8
4. Operating System User Viewpoint: Functions 10
5. Operating System User Viewpoint: Batch Control Language 11
6. Operating System User Viewpoint: Facilities 14
7. Summary 14
   Questions 16

2. MACHINE STRUCTURE, MACHINE LANGUAGE, AND ASSEMBLY LANGUAGE 21
1. General Machine Structure 21
   1. General Approach to a New Machine 23
   2. Machine Structure–360 and 370 25
      1. Memory 25 2. Registers 25 3. Data 27
      4. Instructions 29 5. Special Features 35
2. Machine Language 35
   1. Long Way, No Looping 36
   2. Address Modification Using Instructions as Data 38

3. Address Modification Using Index Registers 40
4. Looping 41
3 Assembly Language 43
1. An Assembly Language Program 43
2. Example Using Literals 45
4. Summary 47
Questions 48

3. ASSEMBLERS 59
1. General Design Procedure 60
2. Design of Assembler 60
1. Statement of Problem 60
2. Data Structure 62
3. Format of Data Bases 65
4. Algorithm 73
5. Look for Modularity 77
3. Table Processing: Searching and Sorting 80
1. Linear Search 81
2. Binary Search 82
3. Sorting 84
1. Interchange Sort 84 2. Shell Sort 86
3. Bucket Sort 86 4. Radix Exchange Sort 88
5. Address Calculation Sort 88
6. Comparison of Sorts 90
4. Hash or Random Entry Searching 91
4. Summary 95
5. Epilog 95
Questions 98

4. MACRO LANGUAGE AND THE MACRO PROCESSOR 111
1. Macro Instructions 112
2. Features of a Macro Facility 114
1. Macro Instruction Arguments 114
2. Conditional Macro Expansion 117
3. Macro Calls within Macros 119
4. Macro Instructions Defining Macros 121
3. Implementation 122
1. Implementation of a Restricted Facility: A Two-Pass Algorithm 123
2. A Single-Pass Algorithm 127
3. Implementation of Macro Calls within Macros 133
4. Implementation within an Assembler 136
4. Summary 142
Questions 143

5. LOADERS 149
1. Loader Schemes 150
1. "Compile-and-Go" Loaders 150
2. General Loader Scheme 151
3. Absolute Loaders 152
4. Subroutine Linkages 154
5. Relocating Loaders 156
6. Direct-linking Loaders 160
7. Other Loader Schemes–Binders, Linking Loaders, Overlays, Dynamic Binders 164
2. Design of an Absolute Loader 167
3. Design of a Direct-Linking Loader 168
1. Specification of Problem 169
2. Specification of Data Structures 176
3. Format of Data Bases 178
4. Algorithm 181
4. Summary 186
Questions 188

6. PROGRAMMING LANGUAGES 199
1. Importance of High Level Languages 200
2. Features of a High Level Language 201
3. Data Types and Data Structures 202
1. Character String 203
2. Bit String–Boolean 204
3. Data Structures 205
4. Storage Allocation and Scope of Names 206
1. Storage Classes 206
2. Block Structure 206
5. Accessing Flexibility 207
1. Pointers 208
2. Label Variables and Label Arrays 208
6. Functional Modularity 210
1. Procedures 211
2. Recursion 212
7. Asynchronous Operation 212
1. Conditions 212
2. Signals 213
3. Multitasking 215
8. Extensibility and Compile-Time Macros 215
9. Miscellaneous 216
10. Summary 216
Questions 217

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.

Draw a neat sketch of the foundations of system programming

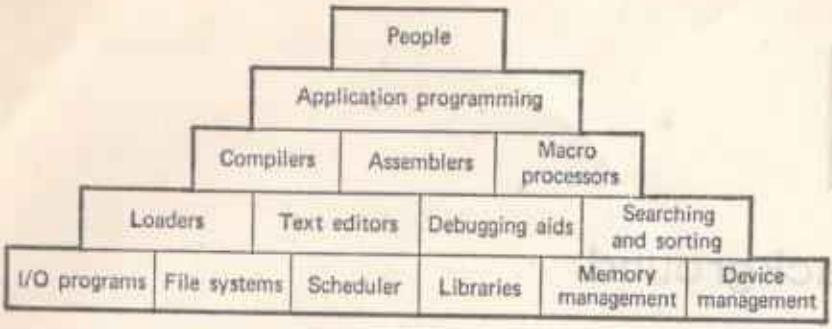


FIGURE 1.1 Foundations of systems programming

1.1 MACHINE STRUCTURE

explain

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

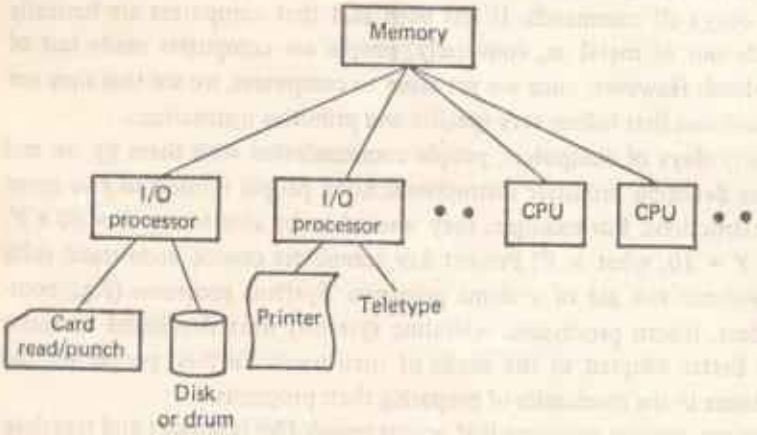


FIGURE 1.2 General hardware organization of a computer system

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.

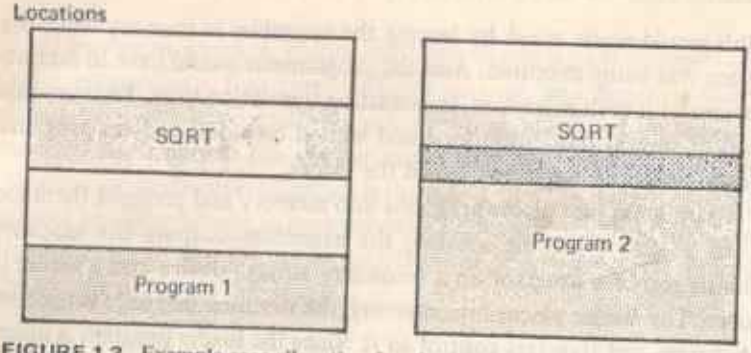


FIGURE 1.3 Example core allocation for absolute loading

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.

Describe the evolution of the components of a programming system

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.

Define paging. state its types also

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.

Explain various functions of an operating system in brief.

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.

Draw flowchart for simple batch monitor system also explain the different classes of control cards used in simple batch monitor systems with suitable ex

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
        10
/*
```

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.

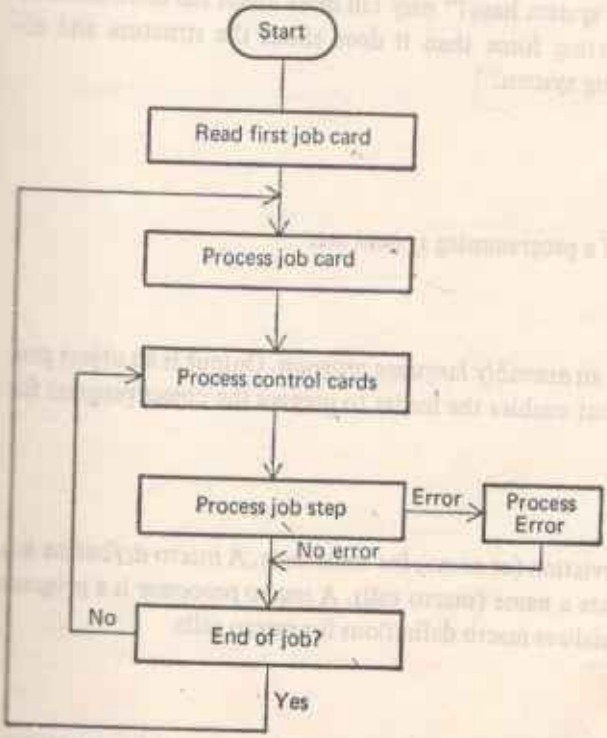


FIGURE 1.4 Main loop of a simple batch monitor system

Explain various facilities provided by O.S. to the user.

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*. Define operating systems

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.*

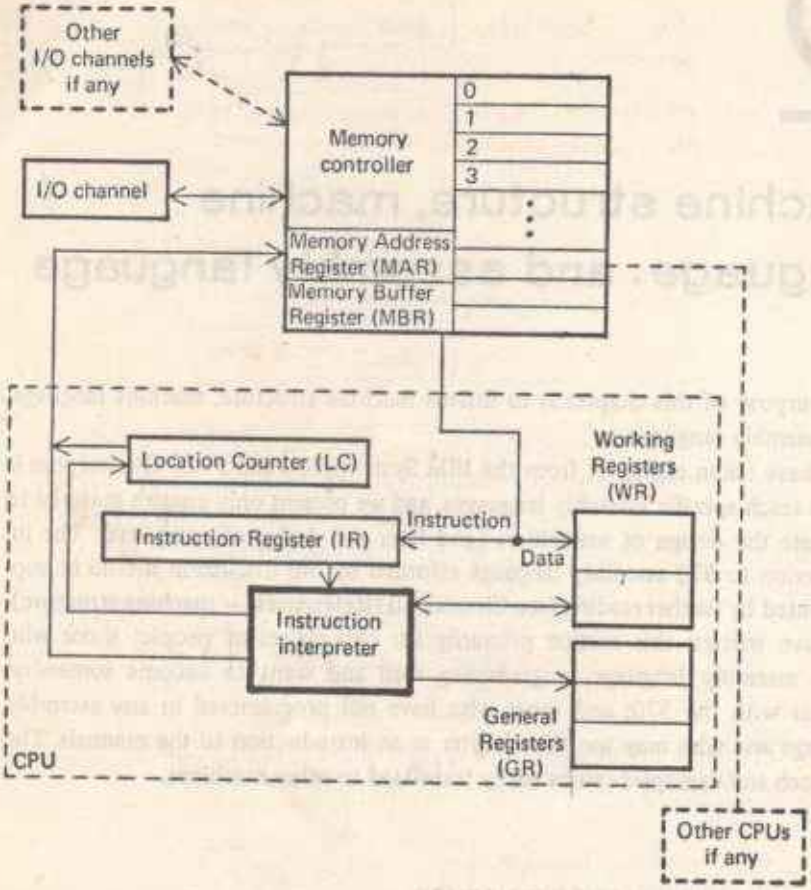


FIGURE 2.1 General machine structure

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?

* Microflowchart for "Rx" type instruction of IBM 360

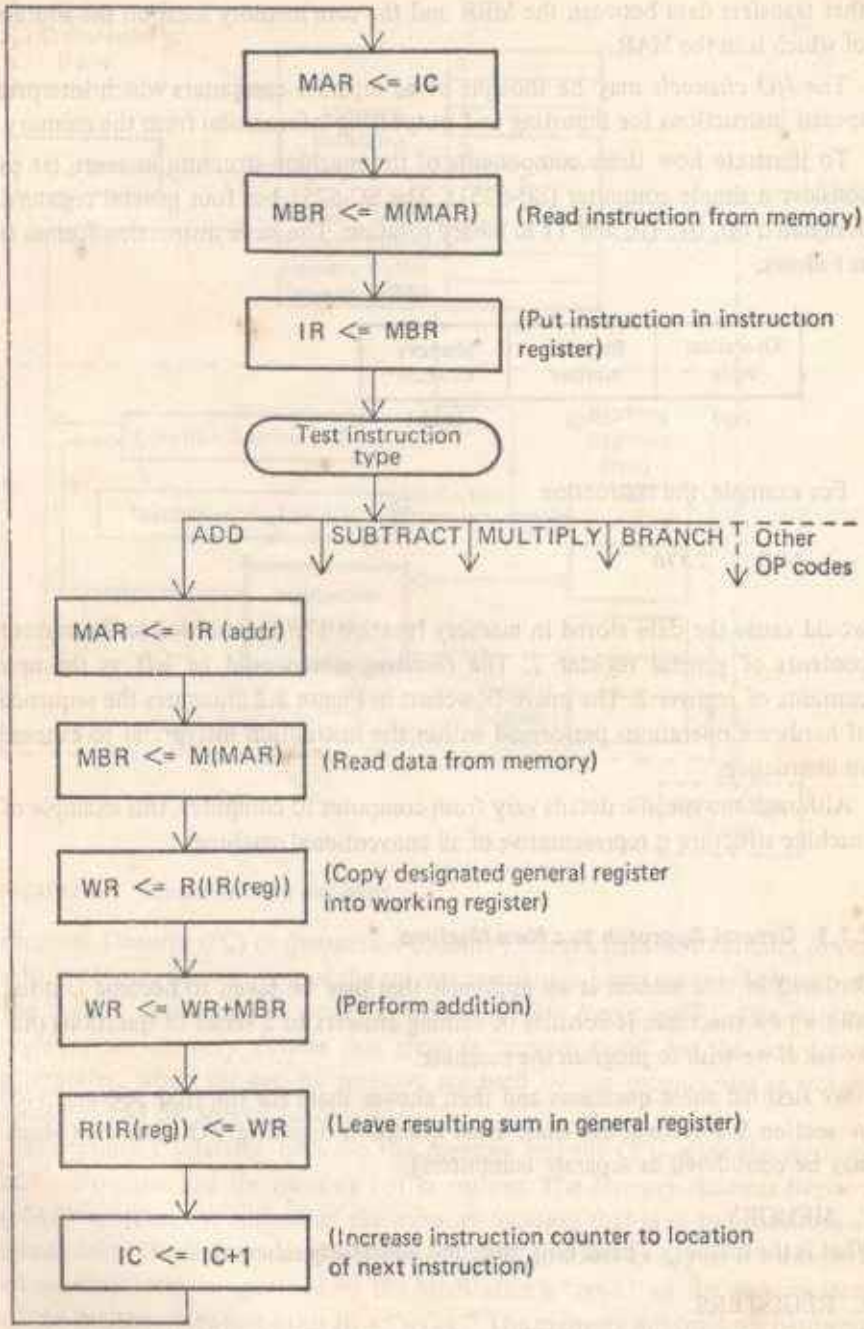


FIGURE 2.2 Example micro flowchart for ADD instruction

* Explain giving example the use of base reg. & index reg. in IBM 360 machines.

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

What is the memory's basic unit, size and addressing scheme for IBM 360 and 370?

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

How many registers are there in IBM 360 and 370? What are there size, function & interrelationship

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

```
                Index register
                      ↘
A          1,901 (2, 15)
            ↑      ↑
         Offset  Base
                 register
```

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.

Op code	Arg reg	Index reg	Address
8 bits	4 bits	4 bits	24 bits

Total: 40

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.

Op code	Arg reg	Index reg	Base	Offset
8 bits	4 bits	4 bits	4 bits	12 bits

Total: 32

limitation of using 12 bit offset in IBM 360

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

what types of data can be handled by the IBM 360? can it handle character, no. logical data? How is data stored

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.

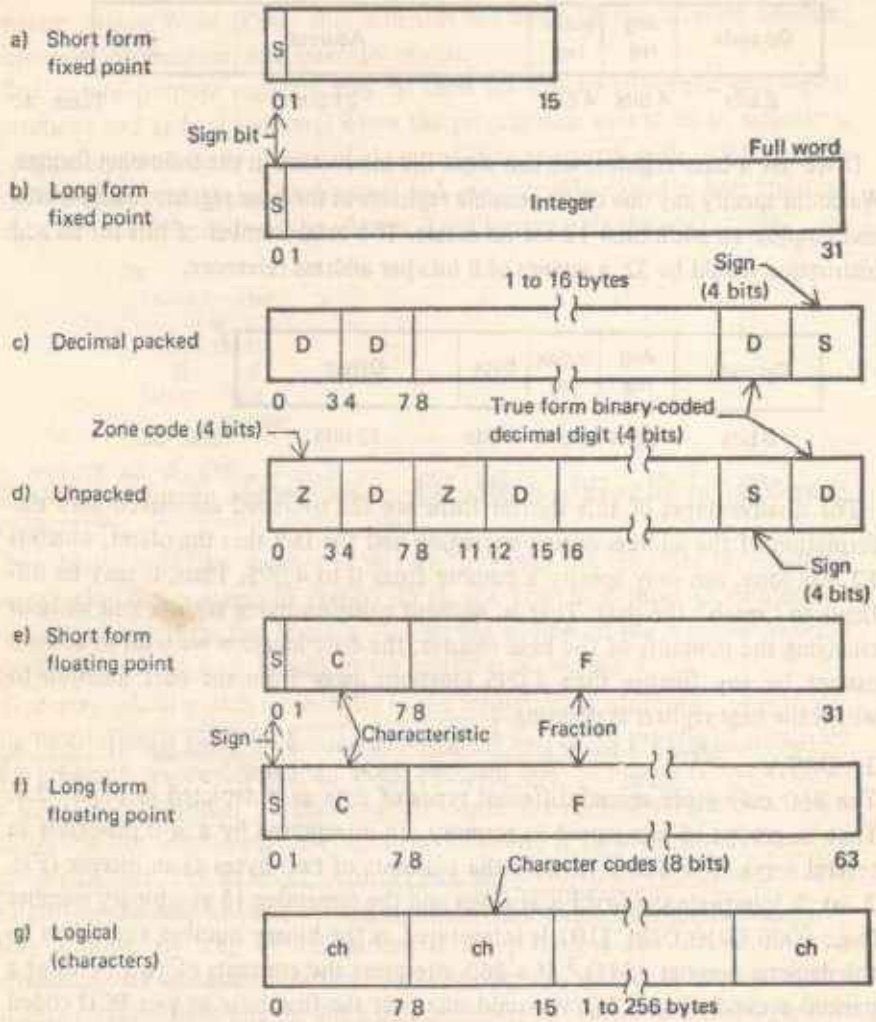


FIGURE 2.3 Data formats for the system/360 and 370

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

$$\left.\begin{array}{llll} \text{B'}\underbrace{0000} & \underbrace{0001} & \underbrace{0010} & \underbrace{1100} \\ \text{X'}\ 0 & 1 & 2 & C \end{array}\right\} \begin{array}{l}\text{in binary}\\ \text{in hexadecimal}\end{array}$$

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

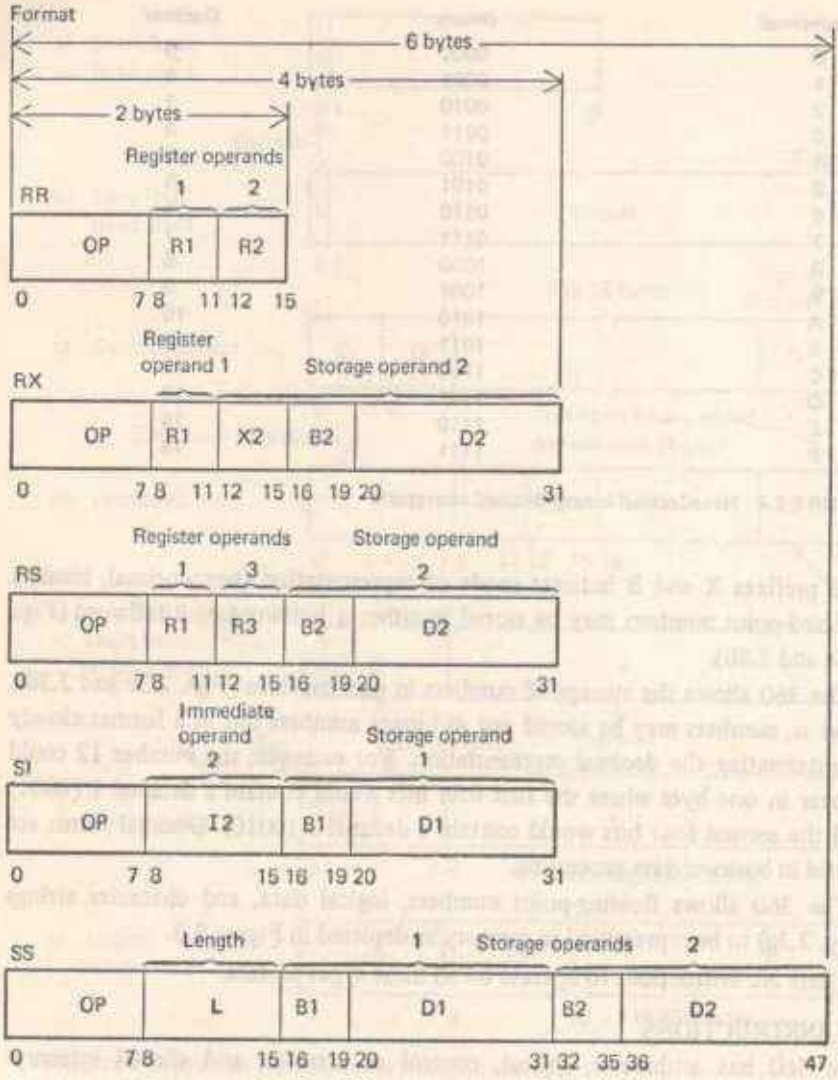
The 360 has arithmetic, logical, control or transfer, and special interrupt instructions.

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.

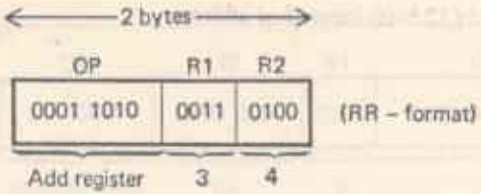
For example, the instruction *Add register 3, 4*



Mnemonics used:

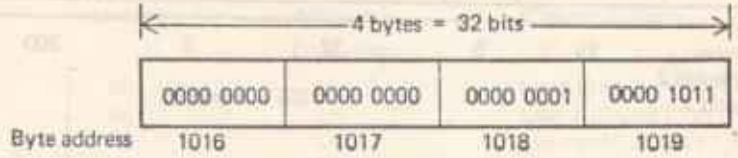
OP ~ operation code
Ri ~ contents of general register used as operand
Xi ~ contents of general register used as index
Bi ~ contents of general register used as base
Di ~ displacement
Ii ~ immediate data
L ~ operand length

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."

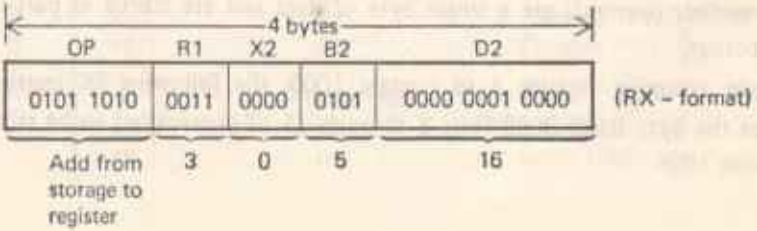


The address of the ith storage operand is computed from the instruction in the following way:

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:

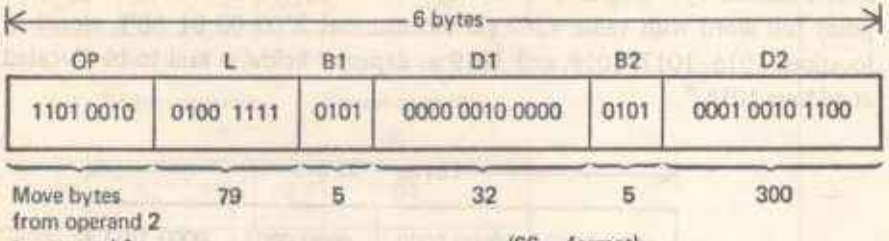


causes the contents of the word (32 bits) located at address

= 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:

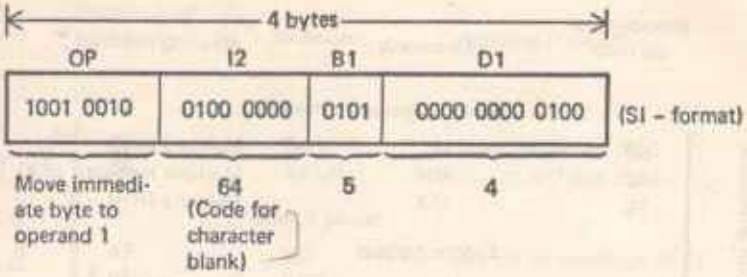
Storage operand 1 address = c(B1) + D1 = c(5) + 32
= 1000 + 32 = 1032

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) 4
	48	LH	Load halfword (RX) 4
	98	LM	Load multiple (RS) 4
	18	LR	Load (RR) 2
	12	LTR	Load and test (RR) 2
		Store group	
	50	ST	Store (RX) 4
	40	STH	Store halfword (RX) 4
	90	STM	Store multiple (RS) 4
		Add group	
Fixed-point arithmetic	5A	A	Add (RX) 4
	4A	AH	Add halfword (RX) 4
	1A	AR	Add (RR) 2
		Compare-group	
	59	C	Compare (RX) 4
	49	CH	Compare halfword (RX) 4
	19	CR	Compare (RR) 2
		Divide-group	
	5D	D	Divide (RX) 4
	1D	DR	Divide (RR) 2

	Hexadecimal op code	Mnemonic	Meaning (format)
Fixed-point arithmetic		*Multiply-group*	
	5C	M	Multiply (RX) 4
	4C	MH	Multiply halfword (RX) 4
	1C	MR	Multiply (RR) 2
		Subtract-group	
	5B	S	Subtract (RX) 4
	4B	SH	Subtract halfword (RX) 4
	1B	SR	Subtract (RR) 2
Logical		*Compare-group*	
	55	CL	Compare logical (RX) 4
	D5	CLC	Compare logical (SS) 6
	95	CLI	Compare logical (SI) 4
	15	CLR	Compare logical (RR) 2
		Move-group	
	D2	MVC	Move (SS) 6
	92	MVI	Move (SI) 4
		And-group	
	54	N	Boolean AND (RX) 4
	D4	NC	Boolean AND (SS) 6
	94	NI	Boolean AND (SI) 4
	14	NR	Boolean AND (RR) 2
		Or-group	
	56	O	Boolean OR (RX) 4
	D6	OC	Boolean OR (SS) 6
	96	OI	Boolean OR (SI) 4
	16	OR	Boolean OR (RR) 2
		Exclusive-or group	
	57	X	Exclusive-or (RX) 4
	D7	XC	Exclusive-or (SS) 6
	97	XI	Exclusive-or (SI) 4
	17	XR	Exclusive-or (RR) 2
		Shift	
	8D	SLDL	Shift left (double logical) (RS) 4
	89	SLL	Shift left (single logical) (RS) 4
	8C	SRDL	Shift right (double logical) (RS 4
	88	SRL	Shift right (single logical) (RS) 4

	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.

* Write an assembly lang. prog. which will add the no. 49 to the contents of 10 adjacent full words in memory

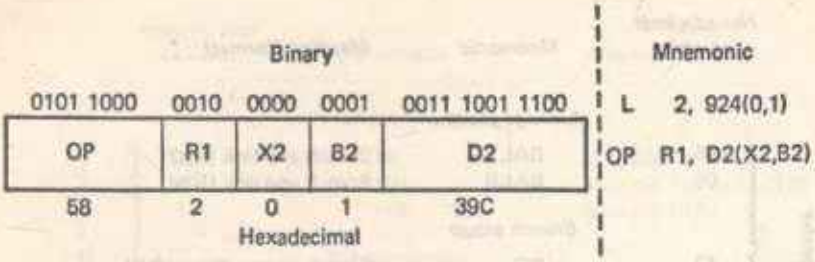


FIGURE 2.6 Mnemonic form of machine language

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,

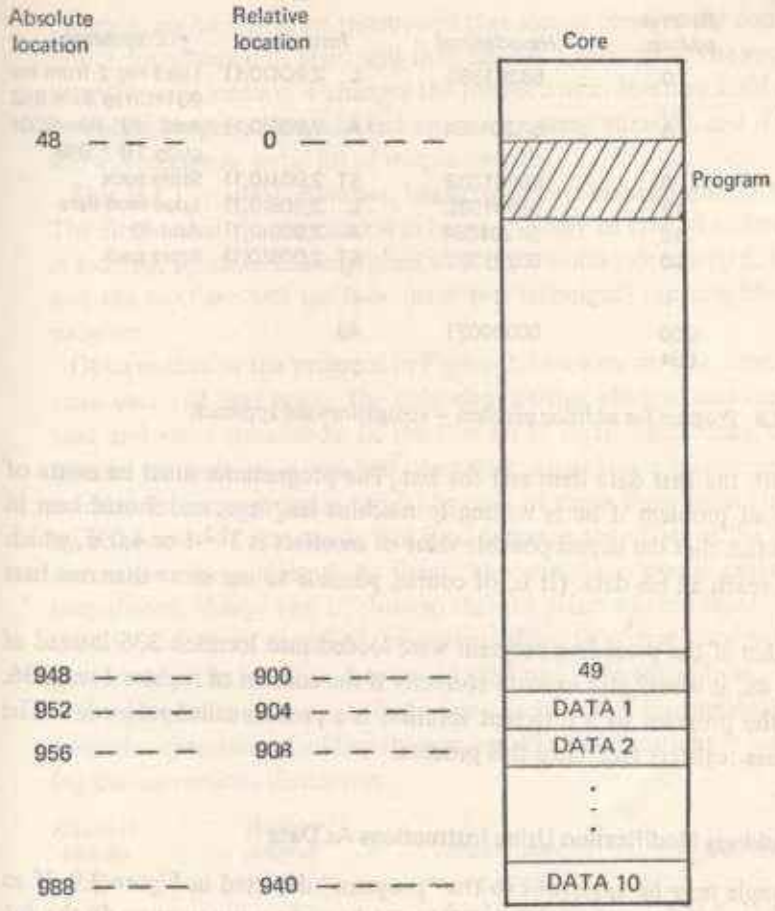


FIGURE 2.7 Diagram of core setup for addition problem

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.

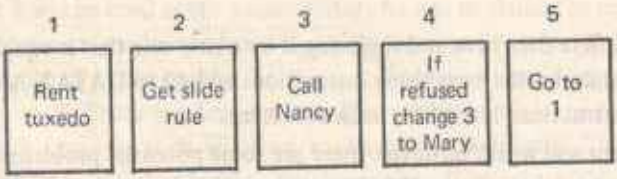


FIGURE 2.9 Situation

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.

Absolute 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

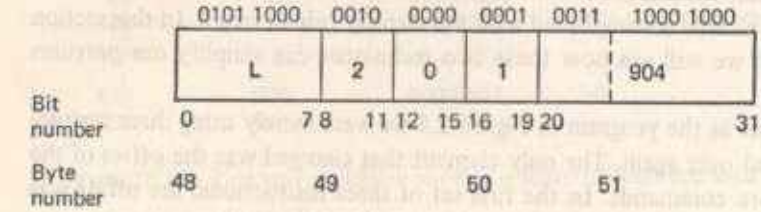


FIGURE 2.11 Contents of location 48

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.

with the help of suitable example, explain the following features of machine language

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

Explain use of following pseudo-op

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 2.15 Assembled version of example program

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.

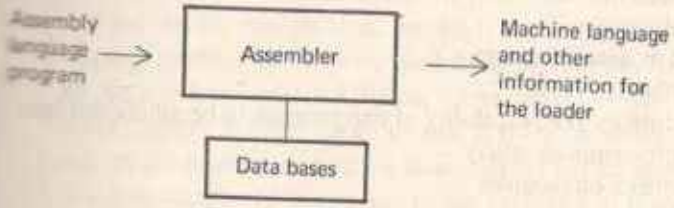


FIGURE 3.1 Function of an assembler

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.

state the various translation phases of an assembler

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

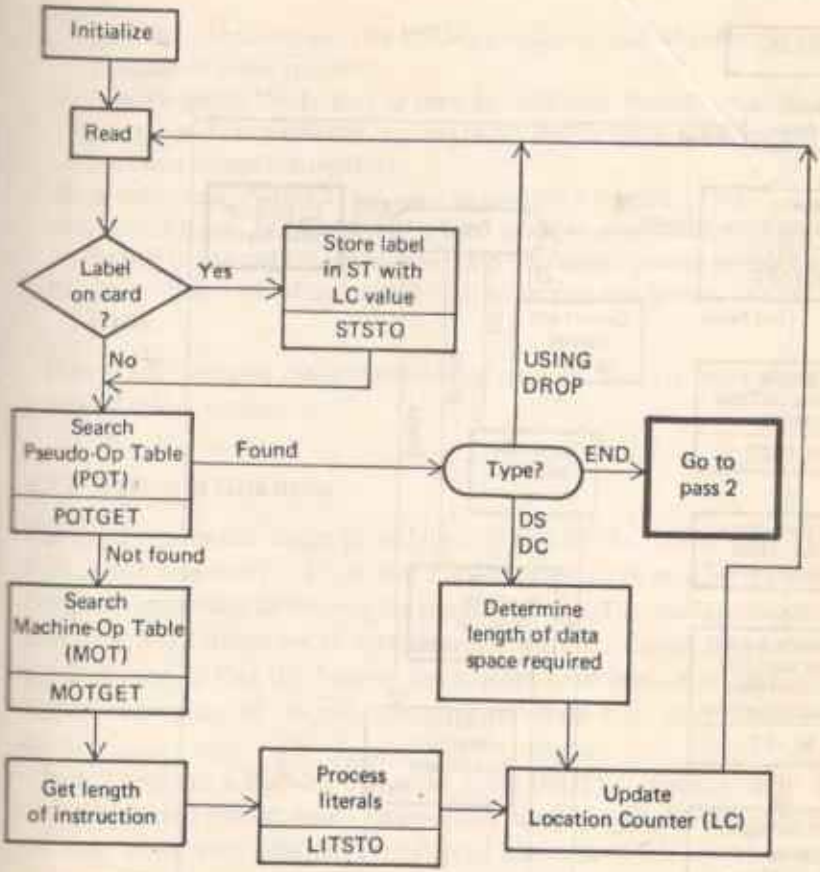


FIGURE 3.3 Pass 1 overview: define symbols

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).

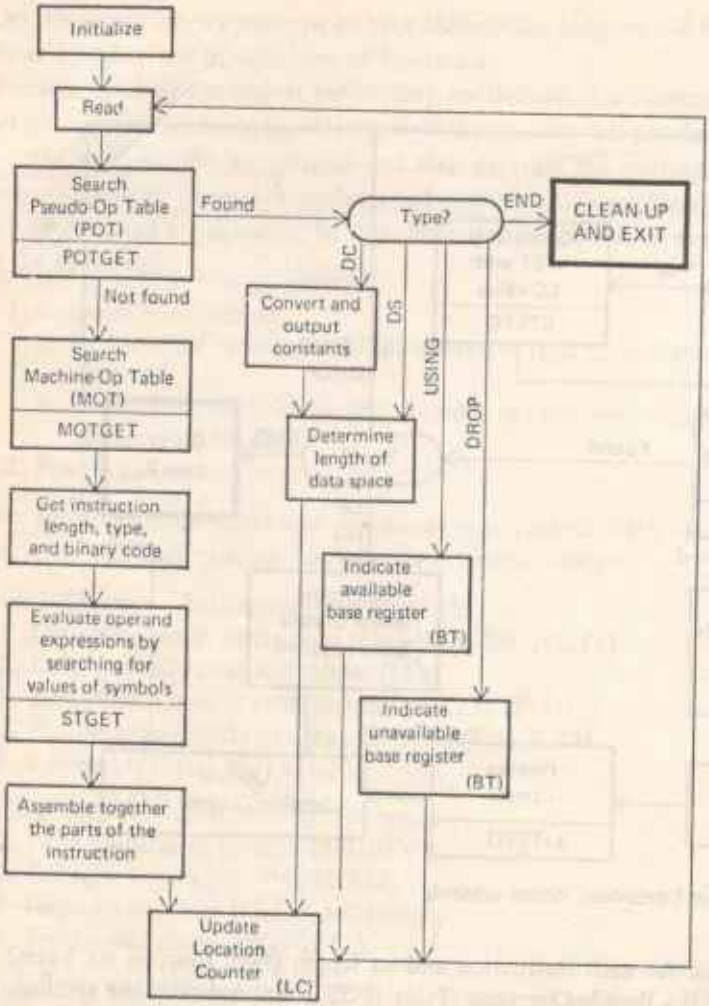


FIGURE 3.4 Pass 2 overview: evaluate fields and generate code

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

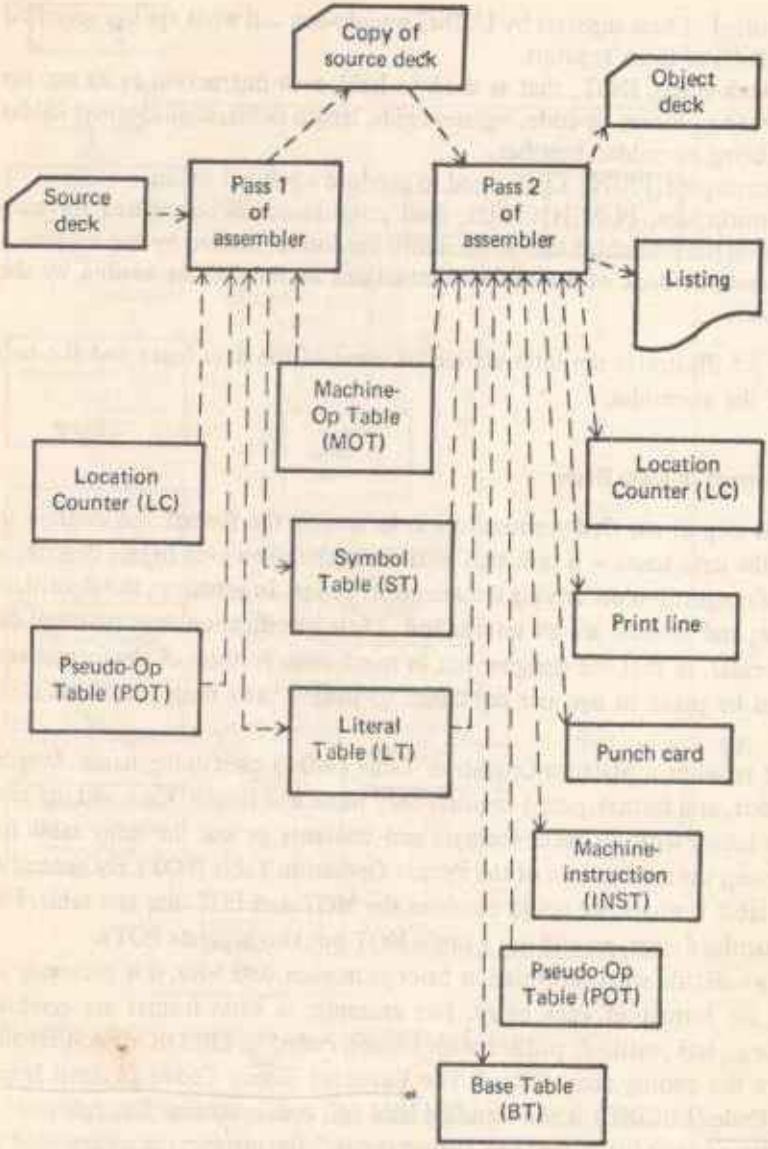


FIGURE 3.5 Use of data bases by assembler passes

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)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALbb"	5E	10	001	
"ALRb"	1E	01	000	
"ARbb"	1A	01	000	
...	...	...	...	
"MVCb"	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)
"DROPb"	P1DROP
"ENDbb"	P1END
"EQUbb"	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)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	000C	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	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).

4-bytes per entry

	Availability indicator (1-byte) (character)	Designated relative-address Contents of base register (3-bytes = 24-bit address) (hexadecimal)
1	"N"	–
2	"N"	–
⋮	⋮	
14	"N"	–
15	"Y"	00 00 00

15 entries

Code=

Availability

Y – register specified in USING pseudo-op

N – register never specified in USING pseudo-op or subsequently made unavailable by the DROP pseudo-op

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
```

BR → 8
LTORG → 16

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

using is seen in Base table

content is the value calculated

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-

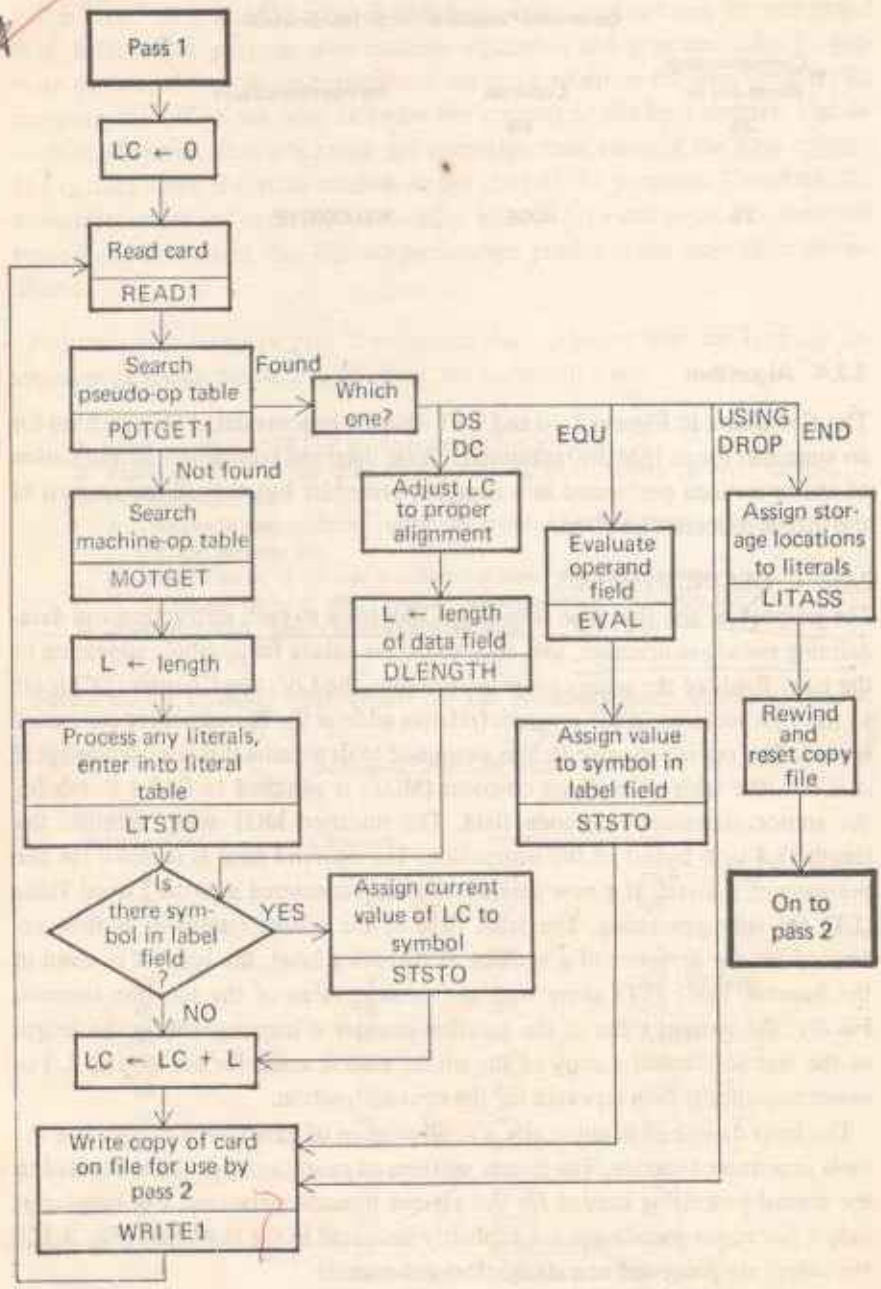


FIGURE 3.10 Detailed pass 1 flowchart

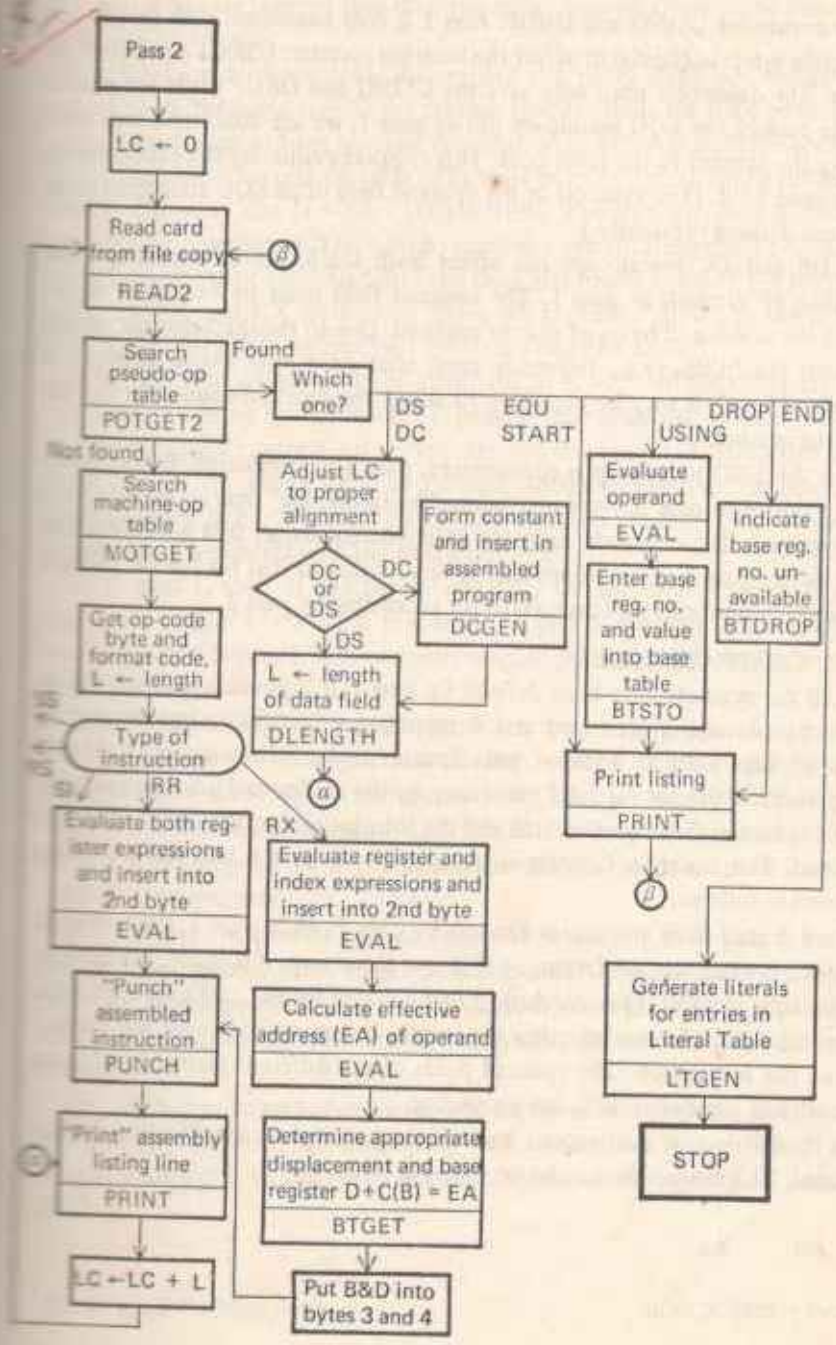


FIGURE 3.11 Detailed pass 2 flowchart

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).

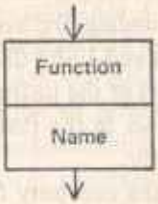
The END pseudo-op indicates the end of the source program and terminates the assembly. Various "housekeeping" tasks must now be performed. For example, code must be generated for any literals remaining in the Literal Table (LT).

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).

Explain various functions used in Pass 1 and Pass

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.

3.2.1 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

Number	Symbol	Probe 1	Probe 2	Probe 3	Probe 4
1	AL				
2	EX				
3	FN				
4	FU		IF > FU		
5	IF				IF = IF
6	IW			IF < IW	
7	LE				
8	LO	IF < LO			
9	NC				
10	OP				
11	OR				
12	RD				
13	RN				
14	TE				
15	TI				

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.

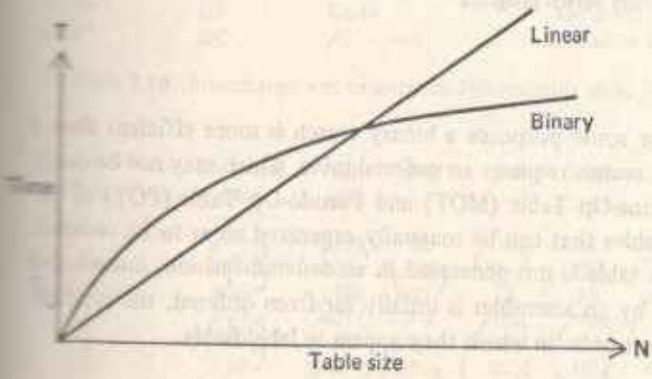


FIGURE 3.14 Search time versus N

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

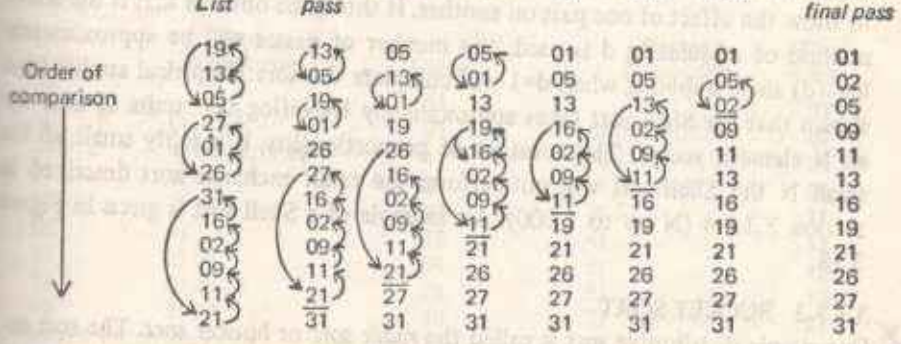


FIGURE 3.17 Illustration of interchange sort

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;

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

In 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.

	Pass 1 ($d_1 = 6$)	Pass 2 ($d_2 = 3$)	Pass 3 ($d_3 = 2$)	Pass 4 ($d_4 = 1$)
19	19	*09	*02	*01
13	13	*01	01	*02
05	*02	02	*09	*05
27	*09	*19	*05	*09
01	01	**11	11	11
26	*21	*05	**13	13
31	31	*27	**16	16
16	16	**13	*19	19
02	*05	*21	***21	21
09	*27	*31	*26	26
11	11	*16	*27	27
21	*26	26	*31	31

d_1 d_2 d_3 d_4

* = Exchange

** = Dual exchange

*** = Triple exchange

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

Original table	*First distribution*	*Merge*	*Second distribution*	*Final merge*
19		01		01
13	0)	31	0) 01,02,05,09	02
05	1) 01,31,11,21	11	1) 11,13,16,19	05
27	2) 02	21	2) 21,26,27	09
01	3) 13	02	3) 31	11
26	4)	13	4)	13
31	5) 05	05	5)	16
16	6) 26,16	26	6)	19
02	7) 27	16	7)	21
09	8)	27	8)	26
11	9) 19,09	19	9)	27
21		09		31
↑ Separate, based on last digit		↑ Separate, based on first digit		

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

Decimal	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5
19	10011 T1	*01011 T2	**00010 T3	00010 T4	*00001 T5,B5 ØK
13	01101	01101	*00001	00001 B4	*00010 T6,B6 ØK
05	00101	00101	00101 B3	00101 T7,B7 ØK	
27	11011	*01001	01001 T8	01001 T9	01001 T10,B10 ØK
01	00001	00001	*01101	*01011 B9	01011 T11,B11 ØK
26	11010	*00010 B2	*01011 B8	*01101 T12,B12 ØK	
31	11111	11111 T13	*10101 T14	*10011 T15	10000 T16,B16 ØK
16	10000	10000	10000	10000 B15	10011 T17,B17 ØK
32	00010	*11010	*10011 B14	*10101 T18,B18 ØK	
09	01001	*11011	11011 T19	11011 T20	11011 T21
11	01011	*10011	*11010	11010 B20	11010 B21
21	10101 B1	10101 B13	*11111 B19	11111 T24,B24 ØK	

T25,B25 DØNE

*11010 T22,B22 ØK
*11011 T23,B23 ØK

*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	$T_s = M+M*\log_2(M) = 55$
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 $\rho = (K-1)/N$.

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

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

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

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

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

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

For a density of p = 12/17 this yields:

$$T_P = 1 + \frac{6}{5} = 2.2$$

$$T_N = \frac{17}{5} = 3.4$$

which compare well with the example shown (see Chapter 10 references for more detailed figures).

GENERATION OF PSEUDO-RANDOM NUMBERS

Many algorithms exist for the generation of pseudo-random numbers, one of the most commonly used being the power residue method. The reader is referred to Hamming's book, Chapter 32, section 5, or Knuth (see bibliography, Chapter 10).

3.4 SUMMARY

A design of a two-pass assembler was presented. Pass 1 defined symbols, and pass 2 generated the object deck. It was emphasized that, in producing the design of an assembler or of any software, six basic steps are followed.

1. Specify the problem.
2. Specify data structures.
3. Define format of data structures.
4. Specify algorithm.
5. Look for modularity.
6. Repeat 1 through 5 on modules.

It was noted that a particularly important component of an assembler is the processing of various data bases (e.g., symbol table, machine-op table). Techniques for organizing, searching, and sorting tables were presented.

3.5 EPILOG

You may have heard the expression "programming is an art, not a science." Of course, there are various programming styles, just as each person has a unique writing or speaking style. But the basic design process, as exemplified by the assembler design, can be and usually is quite straightforward.

It is rather ironic then that many observers interpret the software designer's scientifically motivated actions as evidence of some mysterious "black art." For

4

macro language and the macro processor

The assembly language programmer often finds it necessary to repeat some blocks of code many times in the course of a program. The block may consist of code to save or exchange sets of registers, for example, or code to set up linkages or perform a series of arithmetic operations. In this situation the programmer will find a macro instruction facility useful. *Macro instructions* (often called macros) are single-line abbreviations for groups of instructions. In employing a macro, the programmer essentially defines a single "instruction" to represent a block of code. For every occurrence of this one-line macro instruction in his program, the macro processing assembler will substitute the entire block.

By defining the appropriate macro instructions, an assembly language programmer can tailor his own higher level facility in a convenient manner, at no cost in control over the structure of his program. He can achieve the conciseness and ease in coding of high level languages without losing the basic advantage of assembly language programming. Integral macro operations simplify debugging and program modification and they facilitate standardization. Many computer manufacturers use macro instructions to automate the writing of "tailored" operating systems in a process called *systems generation.*

Macro instructions are usually considered an extension of the basic assembler language, and the macro processor is viewed as an extension of the basic assembler algorithm. As a form of programming language, however, macro instruction languages differ significantly from assembly languages and compiled algebraic languages. Important analogs are to be found in some high level languages and text editing systems.

In this section we will discuss an assembler macro facility. We will treat the use of macro instructions in programming and the implementation of macros within an assembler. Certain features that are characteristic of most macro processors will be introduced and developed.

4.1 MACRO INSTRUCTIONS

In its simplest form, a macro is an abbreviation for a sequence of operations. Consider the following program:

Example 1:

```
        ⋮
        A      1, DATA    Add contents of DATA to register 1
        A      2, DATA    Add contents of DATA to register 2
        A      3, DATA    Add contents of DATA to register 3
        ⋮
        A      1, DATA    Add contents of DATA to register 1
        A      2, DATA    Add contents of DATA to register 2
        A      3, DATA    Add contents of DATA to register 3
        ⋮
DATA    DC     F'5'
        ⋮
```

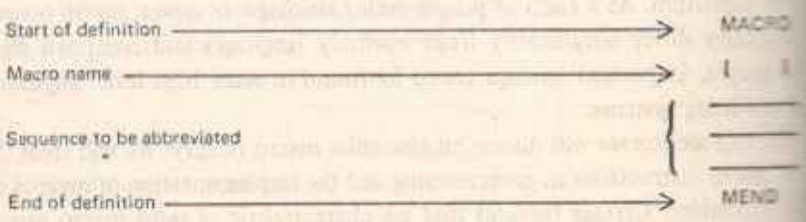
In the above program the sequence

```
        A      1, DATA
        A      2, DATA
        A      3, DATA
```

occurs twice. A macro facility permits us to attach a name to this sequence and to use this name in its place. We can invent a macro language that allows us to specify the above as a macro definition and allows us to refer to the definition later. We have chosen to use a 360-type macro language.

A macro processor effectively constitutes a separate language processor with its own language.

We attach a name to a sequence by means of a macro instruction definition, which is formed in the following manner:



A macro is an abbreviation for a sequence of operation. A macro facility permits us to attach a name to this sequence and then use this name in its place.

The MACRO pseudo-op is the first line of the definition and identifies the following line as the *macro instruction name.* Following the name line is the sequence of instructions being abbreviated—the instructions comprising the 'macro' instruction. The definition is terminated by a line with the MEND ('macro end') pseudo-op.

Once the macro has been defined, the use of the macro name as an operation mnemonic in an assembly program is equivalent to the use of the corresponding instruction sequence. Our example might be rewritten as follows, assigning the name 'INCR' to the repeated sequence.

```
        Source                          |        Expanded source
                                        |
        MACRO                           |
        INCR                            |
        A        1,DATA                 |
        A        2,DATA                 |
        A        3,DATA                 |
        MEND                            |
          .                             |                 .
          .                             |                 .
          .                             |                 .
        INCR                            |          { A        1,DATA
          .                             |          { A        2,DATA
          .                             |          { A        3,DATA
          .                             |                 .
        INCR                            |          { A        1,DATA
          .                             |          { A        2,DATA
          .                             |          { A        3,DATA
          .                             |                 .
DATA    DC       F'5'                   |   DATA   DC         F'5'
          .                             |
          .                             |
          .                             |
```

In this case the macro processor replaces each macro call with the lines

```
A        1, DATA
A        2, DATA
A        3, DATA
```

This process of replacement is called *expanding* the macro. Notice that the macro definition itself does not appear in the expanded source code. The definition is saved by the macro processor. The occurrence in the source program of the macro name, as an operation mnemonic to be expanded, is called a *macro call*

4.2 FEATURES OF A MACRO FACILITY

4.2.1 Macro Instruction Arguments

The macro facility presented thus far is capable of inserting blocks of instructions in place of macro calls. All of the calls to any given macro will be replaced by identical blocks. This macro facility lacks flexibility: there is no way for a specific macro call to modify the coding that replaces it. An important extension of this facility consists of providing for arguments, or parameters, in macro calls. Corresponding *macro dummy arguments* will appear in macro definitions. Consider the following program.

Example 2:

```
          ⋮
          A       1, DATA 1
          A       2, DATA 1
          A       3, DATA 1
          ⋮
          A       1, DATA 2
          A       2, DATA 2
          A       3, DATA 2
          ⋮
DATA 1    DC      F'5'
DATA 2    DC      F'10'
```

In this case the instruction sequences are very similar but not identical. The first sequence performs an operation using DATA1 as operand; the second, using DATA2. They can be considered to perform the same operation with a variable parameter, or argument. Such a parameter is called a *macro instruction argument*, or "dummy argument;" it is specified on the macro name line and distinguished (as a macro language symbol rather than an assembly language symbol) by the ampersand (&), which is always its first character. The preceding program could be written as:

Source			*Expanded source*		
	MACRO		Macro INCR has one argument		
	INCR	&ARG			
	A	1,&ARG			
	A	2,&ARG			
	A	3,&ARG			
	MEND				
	.				
	.				
	.				
	INCR	DATA1	Use DATA1 as operand		
	.				
	.				
	.				
	INCR	DATA2	Use DATA2 as operand		
	.				
	.				
	.				
DATA1	DC	F'5'		DATA1	DC F'5'
DATA2	DC	F'10'		DATA2	DC F'10'
	.				
	.				
	.				

It is possible to supply more than one argument in a macro call. Each argument must correspond to a definition ("dummy") argument on the macro name line of the macro definition. When a macro call is processed, the arguments supplied are substituted for the respective dummy arguments in the macro definition.

Consider the following program:

Example 3:

```
        :
LOOP1   A       1, DATA1
        A       2, DATA2
        A       3, DATA3
        :
LOOP2   A       1, DATA3
        A       2, DATA2
        A       3, DATA1
        :
DATA1   DC      F'5'
DATA2   DC      F'10'
DATA3   DC      F'15'
```

In this case the operands in the common sequence are different, as are the labels on the first cards of the sequences. This program could be written as:

```
                SOURCE                          EXPANDED SOURCE
                  :                   |
          MACRO                       |
&LAB      INCR    &ARG1,&ARG2,&ARG3   |
&LAB      A       1,&ARG1             |
          A       2,&ARG2             |
          A       3,&ARG3             |
          MEND                        |
                  :                   |                :
LOOP1     INCR    DATA1,DATA2,DATA3   | { LOOP1   A      1,DATA1
                                      |           A      2,DATA2
                                      |           A      3,DATA3
                  :                   |                :
LOOP2     INCR    DATA3,DATA2,DATA1   | { LOOP2   A      1,DATA3
                                      |           A      2,DATA2
                                      |           A      3,DATA1
                  :                   |                :
DATA1     DC      F'5'                |   DATA1   DC     F'5'
DATA2     DC      F'10'               |   DATA2   DC     F'10'
DATA3     DC      F'15'               |   DATA3   DC     F'15'
                  :                   |
```

Here we have specified four arguments, including a *label argument.* Label arguments are treated just like those written as "operands" of a macro instruction; the argument appearing in the label field on the macro name line and in calls to the macro differs from other arguments only in its location. Arguments used as labels within the macro need not appear in the label field of a call, and arguments appearing in the label field of a call may be used as operands within the macro instruction code. Label arguments are convenient but nonessential features that enhance the similarity between macro instructions and ordinary assembly language instructions. Following is another encoding of example 3.

```
                    Source                                  Expanded source

        MACRO
        INCR    &ARG1,&ARG2,&ARG3,&LAB
&LAB    A       1,&ARG1
        A       2,&ARG2
        A       3,&ARG3
        MEND
        .                                                   .
        .                                                   .
        INCR    DATA1,DATA2,DATA3,LOOP1     LOOP1   A   1,DATA1
        .                                               A   2,DATA2
        .                                               A   3,DATA3
        .                                               .
        INCR    DATA3,DATA2,DATA1,LOOP2     LOOP2   A   1,DATA3
        .                                               A   2,DATA2
        .                                               A   3,DATA1
        .                                               .
```

There are generally two ways of specifying arguments to a macro call. The first, *positional argument*, is demonstrated in the preceding examples. Arguments are matched with dummy arguments according to the order in which they appear. Thus, in a macro call 'INCR A,B,C' A, B, and C replace the first, second, and third dummy arguments. A more general argument facility, *keyword arguments*, allows reference to dummy arguments by name as well as by position. Referring to the dummy arguments appearing in the definition of INCR, we might write

'INCR &ARG1=A,&ARG3=C,&ARG2=B' or 'INCR &ARG1=A,&ARG2=&ARG3=C'.

This can be useful when a macro has arguments—labels, for example—that are not always needed. Any arguments not supplied are presumed blank by the macro processor.

4.2.2 Conditional Macro Expansion

Two important macro processor pseudo-ops, AIF and AGO, permit conditional ordering of the sequence of macro expansion. This allows conditional selection of the machine instructions that appear in expansions of a macro call.

Consider the following program:

Example 4:

```
        :
LOOP1   A       1,DATA1
        A       2,DATA2
        A       3,DATA3
        :
LOOP2   A       1,DATA3
        A       2,DATA2
        :
LOOP3   A       1,DATA1
        :
DATA1   DC      F'5'
DATA2   DC      F'10'
DATA3   DC      F'15'
        :
```

In this example, the operands, labels, and the number of instructions generated change in each sequence. This program could be written as follows:

```
        .
        .
        MACRO
&ARG0   VARY    &COUNT,&ARG1,&ARG2,&ARG3
&ARG0   A       1,&ARG1
        AIF     (&COUNT EQ 1).FINI          Test if & COUNT = 1
        A       2,&ARG2
        AIF     (&COUNT EQ 2).FINI          Test if & COUNT = 2
        A       3,&ARG3
.FINI   MEND
        .
        .
LOOP1   VARY    3,DATA1,DATA2,DATA3
        .
        .
LOOP2   VARY    2,DATA3,DATA2
        .
        .
LOOP3   VARY    1,DATA1
        .
        .
DATA1   DC      F'5'
DATA2   DC      F'10'
DATA3   DC      F'15'
```

Expanded source

```
        :
LOOP1   A       1,DATA1
        A       2,DATA2
        A       3,DATA3
        :
LOOP2   A       1,DATA3
        A       2,DATA2
        :
LOOP3   A       1,DATA1
        :
```

Labels starting with a period (.), such as .FINI, are *macro labels* and do not appear in the output of the macro processor. The statement AIF (&COUNT EQ 1).FINI directs the macro processor to skip to the statement labelled .FINI if the parameter corresponding to &COUNT is a 1; otherwise, the macro processor is to continue with the statement following the AIF pseudo-op.

AIF is a *conditional branch* pseudo-op; it performs an arithmetic test and branches only if the tested condition is true. The AGO is an *unconditional branch* pseudo-op or 'go to' statement. It specifies a label appearing on some other statement in the macro instruction definition; the macro processor continues sequential processing of instructions with the indicated statement. These statements are directives to the macro processor and do not appear in macro expansions. Just as conditional and unconditional branch instructions in a machine language program direct the order of program flow, AIF and AGO control the sequence in which the macro processor expands the statements in macro instructions. This makes it possible to tailor specific occurrences of instructions sequences to their various contexts. If machine language branches and tests are used, they will waste execution time; also, the tests and unused instructions will waste memory space. Branches and tests in the macro instruction language permit the use of highly general macros that assemble selectively, and by testing the parameters to each call, omit code not needed. This facility for selective assembly of highly general macros is one of the systems programmer's most powerful programming tools.

4.2.3 Macro Calls Within Macros

Since macro calls are "abbreviations" of instruction sequences, it seems reasonable that such "abbreviations" should be available within other macro definitions. For example,

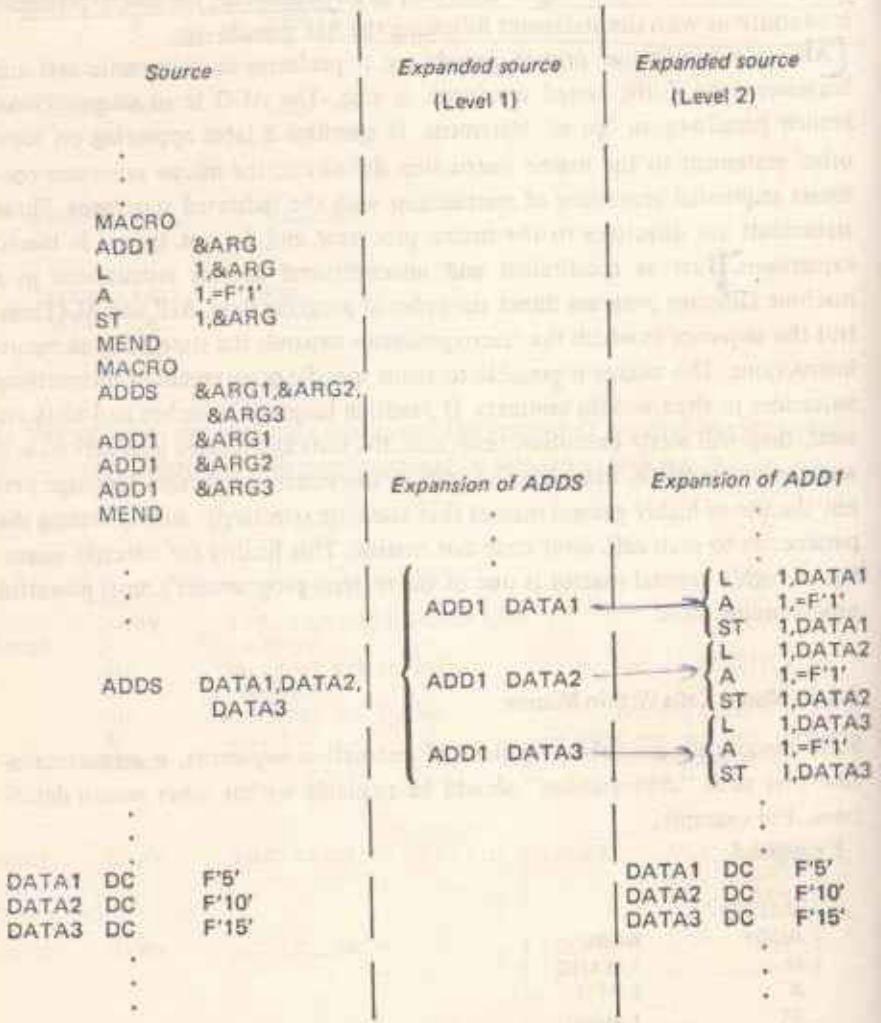
Example 5:

```
MACRO
ADD1      &ARG
L         1,&ARG
A         1,=F'1'
ST        1,&ARG
MEND
MACRO
ADDS      &ARG1,&ARG2,&ARG3
ADD1      &ARG1
ADD1      &ARG2
ADD1      &ARG3
MEND
```

Within the definition of the macro 'ADDS' are three separate calls to a pre-

viously defined macro 'ADD1'. The use of the macro 'ADD1' has shortened the length of the definition of 'ADDS' and thus has made it more easily understood.

Such use of macros results in macro expansions on multiple 'levels'. Thus:



Macro calls within macros can involve several levels. For example, the macro ADDS might be called within the definition of another macro. In fact, conditional macro facilities (such as AIF and AGO) make it possible for a macro to call itself. So long as this does not cause an infinite loop--so long as at some point the macro, having been called for the nth time, decides not to call itself

again—it makes perfectly good sense. The problems inherent to macro calls within macros will be discussed in the section on *recursion*; the preceding example is treated further in Figure 4.5.

4.2.4 Macro Instructions Defining Macros

We have viewed macros as generalized abbreviations for instruction sequences, noting that it seems reasonable to permit any valid statements in the abbreviated sequence, including macro definitions. In this manner a single macro instruction might be used to simplify the process of defining a group of similar macros. It is important to realize that the inner macro definition is not defined (i.e., callable) until after the outer macro has been called. This is because of the method by which definitions are implemented. For example, a user might wish to define a group of macros for subroutine calls with some standardized calling sequence. The following example defines a macro instruction DEFINE, which when called with a subroutine name defines a macro with the same name as the subroutine. The individual macros generated bear the names (given through the argument &SUB) of their associated subroutines.

```
Definition of macro DEFINE:
        MACRO
        DEFINE  &SUB            Macro name: DEFINE
        Definition of macro &SUB:
                MACRO
                &SUB    &Y      Dummy macro name
                CNOP    0,4     Align boundary
                BAL     1,*+8   Set register 1 to parameter list pointer
                DC      A(&Y)   Parameter list pointer
                L       15,=V(&SUB)  Address of subroutine
                BALR    14,15   Transfer control to subroutine
                MEND
        MEND
```

The user might then call this macro with the statement

```
DEFINE      COS
```

defining a new macro named COS: the statement expands into a new macro definition. The user might subsequently call the COS macro as follows:

```
COS         AR
```

and the macro processor will generate the calling sequence:

```
BAL     1,*+8
DC      A(AR)          address of AR
L       15,=V(COS)     V denotes address of external symbol
BALR    14,15
```

Macro definitions within macros are sometimes called "macro definitions within macro definitions." (See Section 4.3.2 for further treatment of this example.)

4.3 IMPLEMENTATION

Thus far we have described a macro instruction scheme for extending the basic assembly language. In the remainder of this chapter we will outline a method for implementing such a scheme.

STATEMENT OF PROBLEM

There are four basic tasks that any macro instruction processor must perform.

1. *Recognize macro definitions* A macro instruction processor must recognize macro definitions identified by the MACRO and MEND pseudo-ops. This task can be complicated when macro definitions appear within macros. When MACROs and MENDs are *nested*, as in the example of the previous section, the macro processor must recognize the nesting and correctly match the last or outer MEND with the first MACRO. All of the intervening text, including nested MACROs and MENDs, defines a single macro instruction.

2. *Save the definitions* The processor must store the macro instruction definitions, which it will need for expanding macro calls.

3. *Recognize calls* The processor must recognize macro calls that appear as operation mnemonics. This suggests that macro names be handled as a type of op-code.

4. *Expand calls and substitute arguments* The processor must substitute for dummy or macro definition arguments the corresponding arguments from a macro call; the resulting symbolic (in this case, assembly language) text is then substituted for the macro call. This text, of course, may contain additional macro definitions or calls.

In summary: the macro processor must recognize and process macro definitions and macro calls.

With regard to arguments, the designer of the macro processor must make several decisions. He must determine where dummy arguments may appear in a macro definition – for example, whether they may appear as op-codes. He must

also define the syntax of permissible arguments. Although we do not treat these matters in detail, we present some typical (and reasonable) answers. Dummy arguments may appear *anywhere* in a macro definition – for example, they may appear as op-codes. We want to be able to concatenate macro arguments with fixed strings. For example, we might want to write A&A, which we take to mean A concatenated with the argument replacing &A. This is easy to arrange. It is harder to have arguments inside a string. For this we permit the argument to be enclosed in single quotes. Thus, we take A&AB to mean A concatenated with the argument replacing AB, and A'&A'B to mean the concatenation of A, the argument replacing &A, and *B*. Any string enclosed in parentheses (e.g., (.A&B)) may be used as an argument.

In order for AIF and AGO to work, some arithmetic expressions will have to be evaluated. A symbol defining capability, similar to the EQU facility by which variables may be assigned and reassigned values, is often useful.

4.3.1 Implementation of a Restricted Facility: A Two-Pass Algorithm

We begin by making some simplifying assumptions. We will assume that our macro processor is functionally independent of the assembler and that the output text from the macro processor will be fed into the assembler. Initially we will not permit macro calls or definitions within macro definitions. Because these features introduce complications, we postpone discussion of them until after we have presented the basic macro processor algorithm.

A macro processor, like an assembler, scans and processes lines of a text. In assembly language, lines are interrelated by addressing: a line can refer to another by its address or name, which must be available ("known") to the assembler. Moreover, the address assigned to each line depends upon preceding lines, upon their addresses, and possibly upon their contents as well. If we consider macro definitions to constitute integral entities, we may say that the lines of our macro language are not so closely interrelated. Macro definitions refer to nothing outside themselves, and macro calls refer only to macro definitions. (Remember our restrictions. Remember also that a macro call substitutes *text*, not values, for parameters. Suppose, for example, that a macro call INCR Y is preceded by a statement defining Y–e.g., Y EQU 10. The macro processor substitutes 'Y', not '10' for the macro definition argument–indeed, it never processes the EQU statement at all, except as a complete line of text.)

Our macro processor algorithm will make two systematic scans, or passes, over the input text, searching first for macro definitions and then for macro calls. Just as our assembler cannot process a reference to a symbol before its definition, so the macro processor cannot expand a macro call before having found

and saved the corresponding macro definition. Thus we need the two passes over the input text, one to handle definitions and one to handle calls. The first pass, examining every operation code, will save all macro definitions in a Macro Definition Table (MDT) and save a copy of the input text, minus macro definitions, on secondary storage (e.g., magnetic tape) for use in the second pass. The first pass will also prepare a Macro Name Table (MNT) along with the MDT. The second pass will then examine every operation mnemonic and replace each macro name with the appropriate text from the macro definitions. (A separate MNT is not essential, but it facilitates searching for macro names.)

SPECIFICATION OF DATA BASES

The following data bases are used by the two passes of the macro processor.

Pass 1 data bases:

1. The input macro source deck
2. The output macro source deck copy for use by pass 2
3. The Macro Definition Table (MDT), used to store the body of the macro definitions
4. The Macro Name Table (MNT), used to store the names of defined macros
5. The Macro Definition Table Counter (MDTC), used to indicate the next available entry in the MDT
6. The Macro Name Table Counter (MNTC), used to indicate the next available entry in the MNT
7. The Argument List Array (ALA), used to substitute index markers for dummy arguments before storing a macro definition

Pass 2 data bases:

1. The copy of the input macro source deck
2. The output expanded source deck to be used as input to the assembler
3. The Macro Definition Table (MDT), created by pass 1
4. The Macro Name Table (MNT), created by pass 1
5. The Macro Definition Table Pointer (MDTP), used to indicate the next line of text to be used during macro expansion
6. The Argument List Array (ALA), used to substitute macro call arguments for the index markers in the stored macro definition

SPECIFICATION OF DATA BASE FORMAT

The only data bases with nontrivial format are the Macro Definition Table (MDT), the Macro Name Table (MNT), and the Argument List Array (ALA). The others we will not present in detail.

ARGUMENT LIST ARRAY: The Argument List Array (ALA) is used during both pass 1 and pass 2 but for somewhat reverse functions. During pass 1, in order to simplify later argument replacement during macro expansion, dummy

what data bases are used by the two passes of macro processor.

arguments in the macro definition are replaced with positional indicators when the definition is stored. The ith dummy argument on the macro name card is represented in the body of the macro by the index marker symbol # where #i is a symbol reserved for the use of the macro processor (i.e., not available to the programmers). These symbols are used in conjunction with the argument list prepared before expansion of a macro call. The symbolic dummy arguments are retained on the macro name card to enable the macro processor to handle argument replacement by name rather than by position.

As an example, consider the macro INCR used in Example 3. The stored macro definition would be:

what is the use of ALA in design of a Microprocessor explain with ex

Macro Definition Table MDT

```
        :
&LAB    INCR    &ARG1,&ARG2,&ARG3
#0      A       1, #1
        A       2, #2
        A       3, #3
        MEND
        :
```

During pass 2 it is necessary to substitute macro call arguments for the index markers stored in the macro definition. Thus upon encountering the call

```
LOOP    INCR    DATA1,DATA2,DATA3
```

the macro call expander would prepare an argument list array:

Argument List Array

8 bytes per entry

Index	*Argument*
0	"LOOP1*bbb*"
1	"DATA1*bbb*"
2	"DATA2*bbb*"
3	"DATA3*bbb*"

(*b* denotes the blank character)

The list would be used only while expanding this particular call. Suppose that a succeeding call were:

```
INCR    &ARG1=DATA3,&ARG2=DATA2,&ARG3=DATA1
```

The macro processor would find that '&ARG1', '&ARG2', and '&ARG3' occupy argument positions 1, 2, and 3 on the macro name card. The resulting argument list array would be:

Argument List Array

Index	*Argument*
0	*"bbbbbbbb"* (all blank)
1	"DATA3*bbb*"
2	"DATA2*bbb*"
3	"DATA1*bbb*"

MACRO DEFINITION TABLE: The Macro Definition Table (MDT) is a table of text lines; if input is from 80-column cards, the MDT can be a table with 80-byte strings as entries. Every line of each macro definition, except the MACRO line, is stored in the MDT. (The MACRO line is useless during macro expansion.) The MEND is kept to indicate the end of the definition; and the macro name line is retained to facilitate keyword argument replacement. Thus, for example, the INCR macro discussed might be stored as follows:

Macro Definition Table

80 bytes per entry

Index	*Card*		
⋮		⋮	
15	&LAB	INCR	&ARG1,&ARG2,&ARG3
16	#0	A	1, #1
17		A	2, #2
18		A	3, #3
19		MEND	
⋮		⋮	

MACRO NAME TABLE: The Macro Name Table (MNT) serves a function very similar to that of the assembler's Machine-Op Table (MOT) and Pseudo-Op Table (POT). Each MNT entry consists of a character string (the macro name) and a pointer (index) to the entry in the MDT that corresponds to the beginning of the macro definition. The MNT entry for the INCR macro discussed might be:

Index	8 bytes *Name*	4 bytes *MDT index*
⋮	⋮	⋮
3	"INCR*bbbb*"	15
⋮	⋮	⋮

ALGORITHM

Figures 4.1 and 4.2 are flowcharts of the macro definition and expansion algorithms. Each of the algorithms makes a line-by-line scan over its input. The 'READ' boxes refer to the fetching of successive input lines from secondary storage into a workspace.

PASS 1–MACRO DEFINITION: The algorithm for pass 1 (Fig. 4.1) tests each input line. If it is a MACRO pseudo-op, the entire macro definition that follows is saved in the next available locations in the Macro Definition Table (MDT). The first line of the definition is the macro name line. The name is entered into the Macro Name Table (MNT), along with a pointer to the first location of the MDT entry of the definition. When the END pseudo-op is encountered, all of the macro definitions have been processed so control transfers to pass 2 in order to process macro calls.

PASS 2–MACRO CALLS AND EXPANSION: The algorithm for pass 2 (Fig. 4.2) tests the operation mnemonic of each input line to see if it is a name in the MNT. When a call is found, the call processor sets a pointer, the Macro Definition Table Pointer (MDTP), to the corresponding macro definition stored in the MDT. The initial value of the MDTP is obtained from the "MDT index" field of the MNT entry. The macro expander prepares the Argument List Array (ALA) consisting of a table of dummy argument indices and corresponding arguments to the call. This list is simply a succession of symbols ordered to match the dummy arguments on the name card (the first is the label argument, which is considered to have an index of zero). Arguments not represented in a call are considered blank, and superfluous arguments are ignored. In the case of argument reference by position, this scheme is completely straightforward. For references by name, the macro processor locates the dummy argument on the macro name line (which is available at the beginning of the definition in the MDT) in order to determine the proper index.

Reading proceeds from the MDT; as each successive line is read, the values from the argument list are substituted for dummy argument indices in the macro definition. Reading of the MEND line in the MDT terminates expansion of the macro, and scanning continues from the input file. When the END pseudo-op is encountered, the expanded source deck is transferred to the assembler for further processing.

4.3.2 A Single-Pass Algorithm

Suppose we wanted to provide for macro definitions within macros. The basic problem here is that the inner macro is defined only after the outer one has been called; in order to provide for any use of the inner macro, we would have to re-

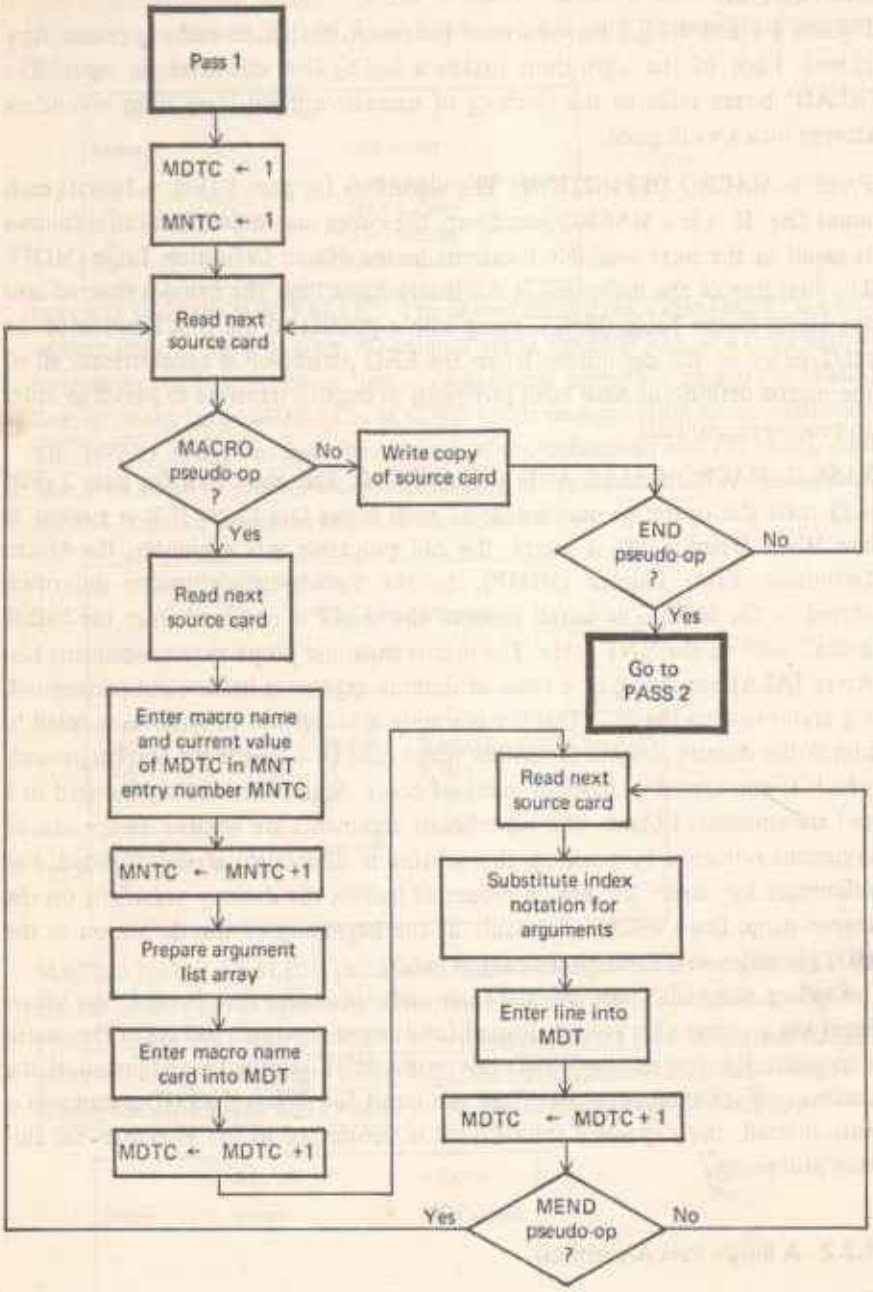


FIGURE 4.1 Pass 1–processing macro definitions

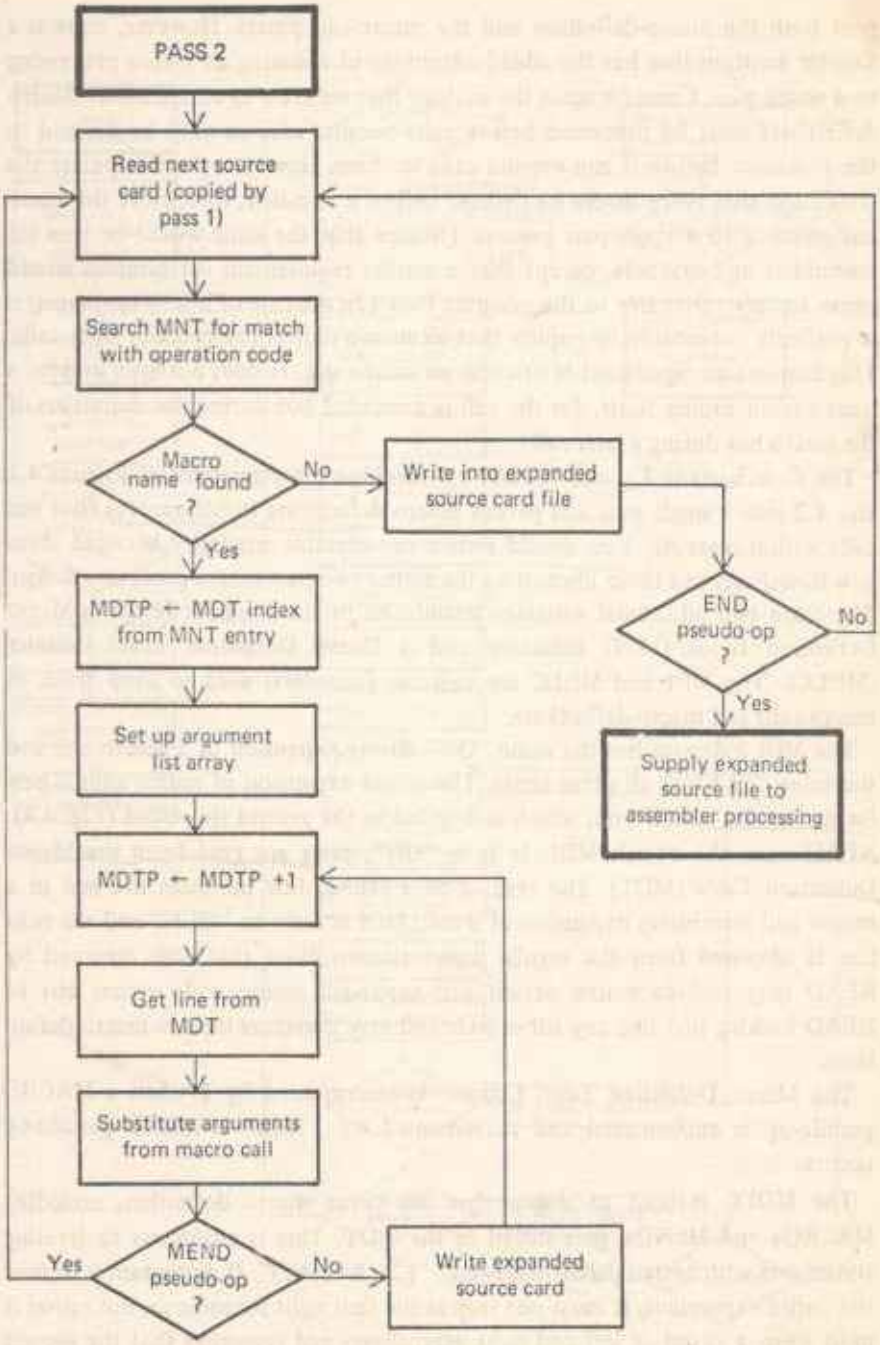


FIGURE 4.2 Pass 2–processing macro calls and expansion

peat both the macro-definition and the macro-call passes. However, there is a simpler solution that has the added advantage of reducing all macro processing to a single pass. Consider again the analogy that we drew to an assembler. Macro definitions must be processed before calls because macros must be defined to the processor before it can expand calls to them. However, if we stipulate the restriction that every macro be defined before it is called, we remove the essential obstacle to a single-pass process. (Notice that the same would be true for assemblers and symbols, except that a similar requirement for symbols would prove unduly restrictive to the program flow.) In the case of macro languages, it is perfectly reasonable to require that all macro definitions precede their calls. This imposes no significant restriction on macro use. It does not even prevent a macro from calling itself, for the call is expanded not during the definition of the macro but during a later call.

The flowcharts in Figures 4.3 and 4.4 combine the algorithms of Figures 4.1 and 4.2 into a single pass and permit macro definitions within macros (but not calls within macros). You should notice considerable similarity between these new flowcharts and those illustrating the earlier two-pass macro processor design. There are two additional variables introduced in the one-pass design: a Macro Definition Input (MDI) indicator and a Macro Definition Level Counter (MDLC). The MDI and MDLC are switches (counters) used to keep track of macro calls and macro definitions.

The MDI indicator has the value "ON" during expansion of a macro call and the value "OFF" at all other times. The actual expansion of macro calls is performed in the READ box, which is detailed in the second flowchart (Fig. 4.4). READ tests the switch MDI. If it is "ON", lines are read from the Macro Definition Table (MDT). The reading of a MEND line indicates the end of a macro and terminates expansion of a call; MDI is reset to "OFF" and the next line is obtained from the regular input stream. Note that lines returned by READ may include macro definitions; expanded macro code comes out of READ looking just like any other code and may therefore include macro definitions.

The Macro Definition Level Counter is incremented by 1 when a MACRO pseudo-op is encountered and decremented by 1 when a MEND pseudo-op occurs.

The MDLC is used to insure that the entire macro definition, including MACROs and MENDs, gets stored in the MDT. This is analogous to treating statements with nested parentheses—e.g., "(2*(A+B)+C)". If a program is to read the entire expression, it must not stop at the first right parenthesis, but rather it must keep a count of left and right parentheses and recognize that the second parenthesis is the true end of the expression. The MDLC acts as a counter for the difference between the numbers of MACROs and MENDs encountered; it

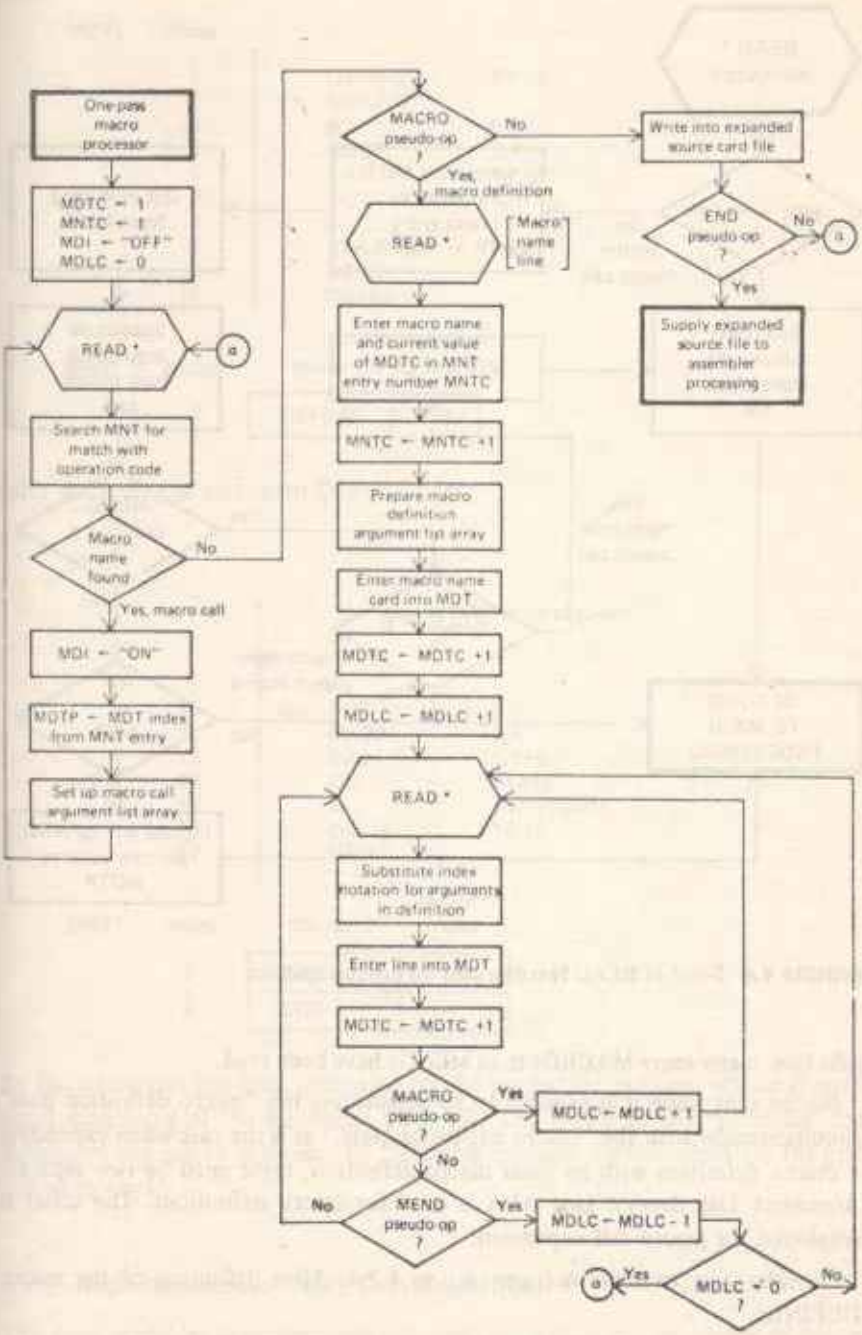


FIGURE 4.3 Simple one-pass macro processor

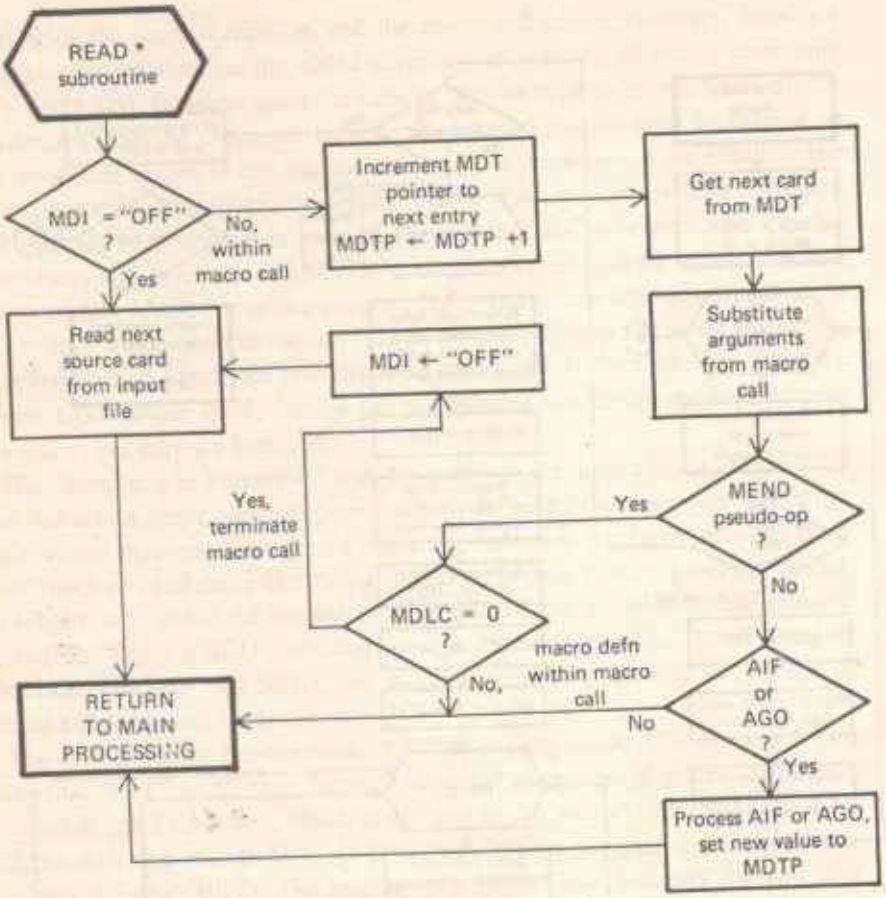


FIGURE 4.4 Detail of READ function used for macro expansion

tells how many more MACROs than MENDs have been read.

Notice that since it is possible to be performing the "macro definition pass" simultaneously with the "macro expansion pass," as is the case when expanding a macro definition with an inner macro definition, there must be two separate Argument List Arrays. One ALA is used for macro definitions. The other is employed for macro call expansion.

Consider now example 6 from section 4.2.4. After definition of the macro DEFINE:

MDT:

Index		
1	DEFINE	&SUB
2	MACRO	
3	#1	&Y
4	CNOP	0,4
5	BAL	1,*+8
6	DC	A(&Y)
7	L	15,=V(#1)
8	BALR	14,15
9	MEND	
10	MEND	

MNT:

Index	*Name*	*MDT Index*
1	DEFINE	1

after processing of statement DEFINE COS:

MDT:

	Lines 1-10 same as above	
.	.	
.	.	
.	.	
11	COS	&Y
12	CNOP	0,4
13	BAL	1,*+8
14	DC	A(#1)
15	L	15,=V(COS)
16	BALR	14,15
17	MEND	

MNT:

Index	*Name*	*MDT Index*
1	DEFINE	1
2	COS	11

So far no output has been generated by the macro: output lines will be created upon expansion of the statement COS AR. The reader should "simulate" the actions of the macro processor by following through the flowchart for the preceding example.

4.3.3 Implementation of Macro Calls Within Macros

The basic problem in implementing macro calls within macros is that of *recur-*

sion. If a macro call is encountered during the expansion of a macro, the macro processor will have to expand the included macro call and then finish expanding the enclosing macro. The second call might be expanded by a second macro processor, which would look up the macro definition in the MDT and return the expanded code to the first macro processor. Having many macro processors is neither efficient nor general; however, if a single macro processor is to handle such nested macro calls, it must in some way save its status when it encounters nested calls.

Example 5 illustrates this problem. It defines two macros, ADDS and ADD1. The macro definitions in the MDT are shown in Figure 4.5.

Macro Definition Table MDT

Index	*Contents*	
1	ADD1	&ARG
2	L	1, #1
3	A	1,=F'1'
4	ST	1, #1
5	MEND	
6	ADDS	&ARG1,&ARG2,&ARG3
7	ADD1	#1
8	ADD1	#2
9	ADD1	#3
10	MEND	

FIGURE 4.5 MDT after macro definition of example 5

Consider the action of the macro processor of Figure 4.3 and 4.4 when it encounters the macro call

```
ADDS        DATA1, DATA2, DATA3
```

Our algorithm will prepare a macro call argument list array and set a pointer, MDTP, to line 6 of the MDT; switch MDI is set to "ON". The READ function increments MDTP, gets its next line from the MDT (line 7) and substitutes the argument, yielding the line

```
ADD1        DATA1
```

Now our algorithm is in trouble: it has encountered another macro call. ADD1 is a macro name, so the processor will prepare a new argument list array, set MDTP to line 1 of the MDT, and set MDI to "ON", its current value. The ADD1 macro will expand correctly; at the end MDTP will point to MDT line 5. MDI will be re-

what is the use of recursive macro calls?

set to "OFF", and reading will proceed from the input stream.

This of course is wrong: there are unexpanded lines left in the ADDS macro. Three errors have occurred. The switch MDI has been turned off, causing the processor to read from the input stream instead of from the MDT; the value of MDTP before it was reset to process the ADD1 macro has been lost; and the argument list for the call to ADDS has been lost.

To handle macro calls within macros, the macro processor must be able to work recursively—that is, to process one macro before it is finished with another, then to continue with the previous, or 'outer,' one. Recursive procedures usually operate by means of a *stack*, a storage scheme that allocates a separate storage area for the variables associated with each call to the procedure. Because a separate storage space or 'stack frame' is associated with each recursive call, the status of unfinished computations is preserved. Conceptually, a stack is an unbounded array that is treated in a last-in, first-out manner: the last element stored is the first one removed. A Stack Pointer (SP) indicates the position or *frame* at the 'top' of the stack. (*Pointer* is a term that refers to a data item which is an address or index. Intuitively, a pointer points to information stored in core.)

which is otherwise lost if recursion is not used which can the result in unexpanded lines in the macro calls

In our example the pointer MDTP and the argument list ALA associated with each call belong in the stack. The switch MDI must also be saved. The Stack Pointer (SP), which indicates the beginning of the current stack frame, can serve in place of MDI; in each succeeding frame, however, we must save the previous value of SP to prevent it from being lost. (Compare this with our use of the Macro Definition Level Counter (MDLC). 'Recursive' handling of macro definitions was easy because the inner definitions do not have any associated data—see the exercises at the end of this chapter). Ignoring the actual method of data representation, we consider the stack to be an array of pointers and character strings; S(SP) refers to the beginning of the current (top) stack frame, the position indexed by SP.

The organization of each stack frame is depicted in Figure 4.6. S(SP) contains the *previous* value of SP; for the first frame, where SP=1,S(SP)=-1. The condition SP=-1, like MDI=0 in the nonrecursive algorithm, tells the macro processor that it is not within a macro call expansion. S(SP+1) contains the current value of MDTP, the MDT pointer; S(SP+2) . . . S(SP+N+1) contain the N (0th through (N-1)th) character strings of the current argument list.

Figures 4.7 and 4.8 depict the macro processor algorithm that handles macro calls within macro definitions as well as macro definitions within macro definitions. Simple "housekeeping" steps, such as incrementing the Macro Name Table Counter (MNTC) and the Macro Definition Table Counter (MDTC) after inserting a new entry, have been omitted to simplify the flowcharts. The reader

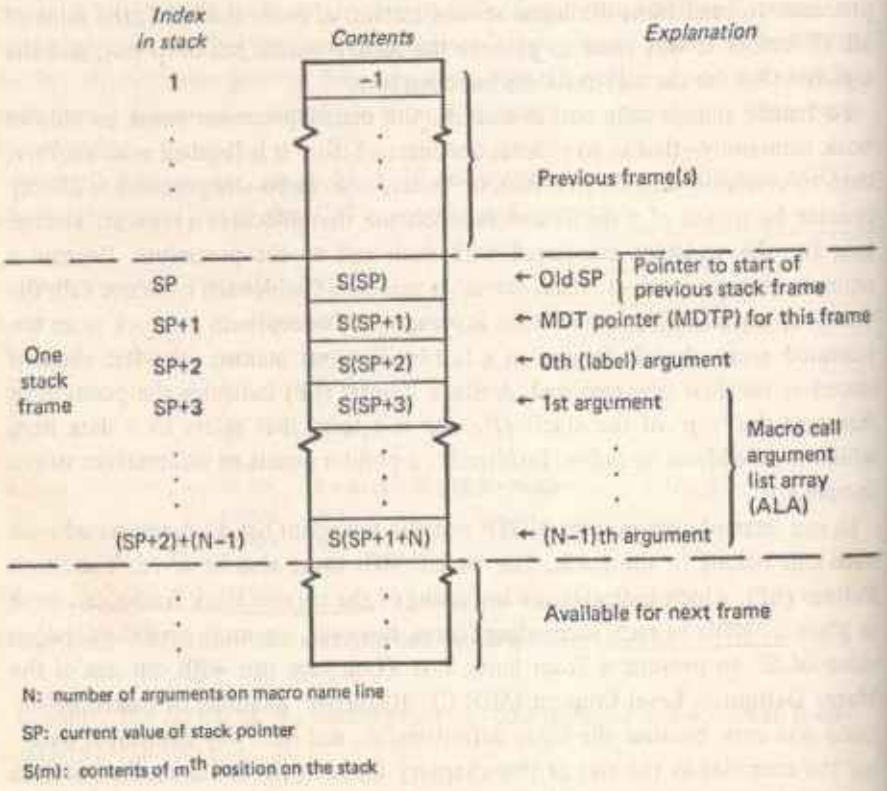


FIGURE 4.6 Stack organization

should confirm the fact that the algorithm described in these new flowcharts is indeed very similar to the previous algorithm (Figs. 4.3 and 4.4). Notice that the number of entries in a stack frame depends upon the number of elements in the argument list (i.e., the number of dummy arguments in the macro definition).

It is desirable for the reader to hand-simulate the operation of the macro processor by using the algorithm to expand a macro source program. Figure 4.9 illustrates the state of the stack and expanded source at various key points in the processing of Example 5. The corresponding Macro Definition Table (MDT) for this example was presented in Figure 4.5. It is strongly recommended that the reader go through all the steps of the macro expansion and compare results with Figures 4.9a and 4.9b.

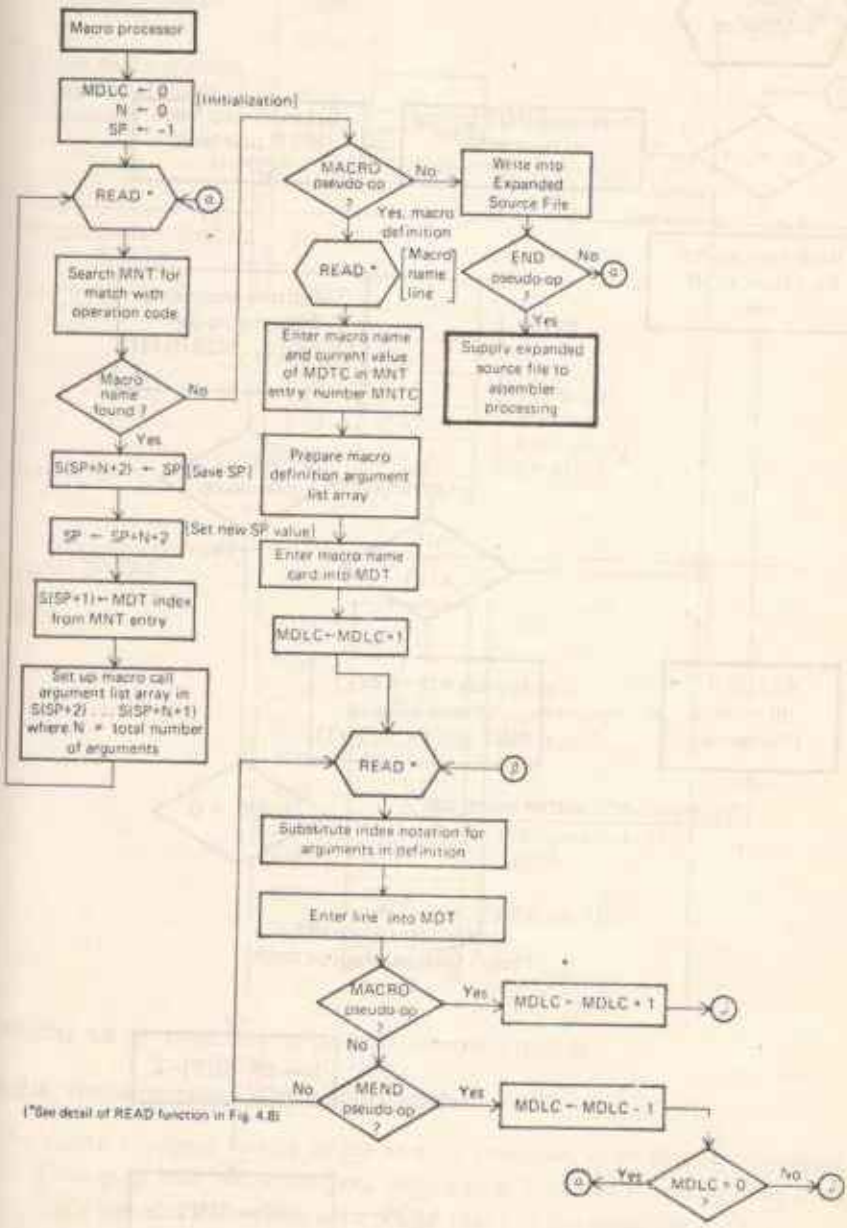


FIGURE 4.7 One-pass macro processor capable of handling macro calls within macro definitions

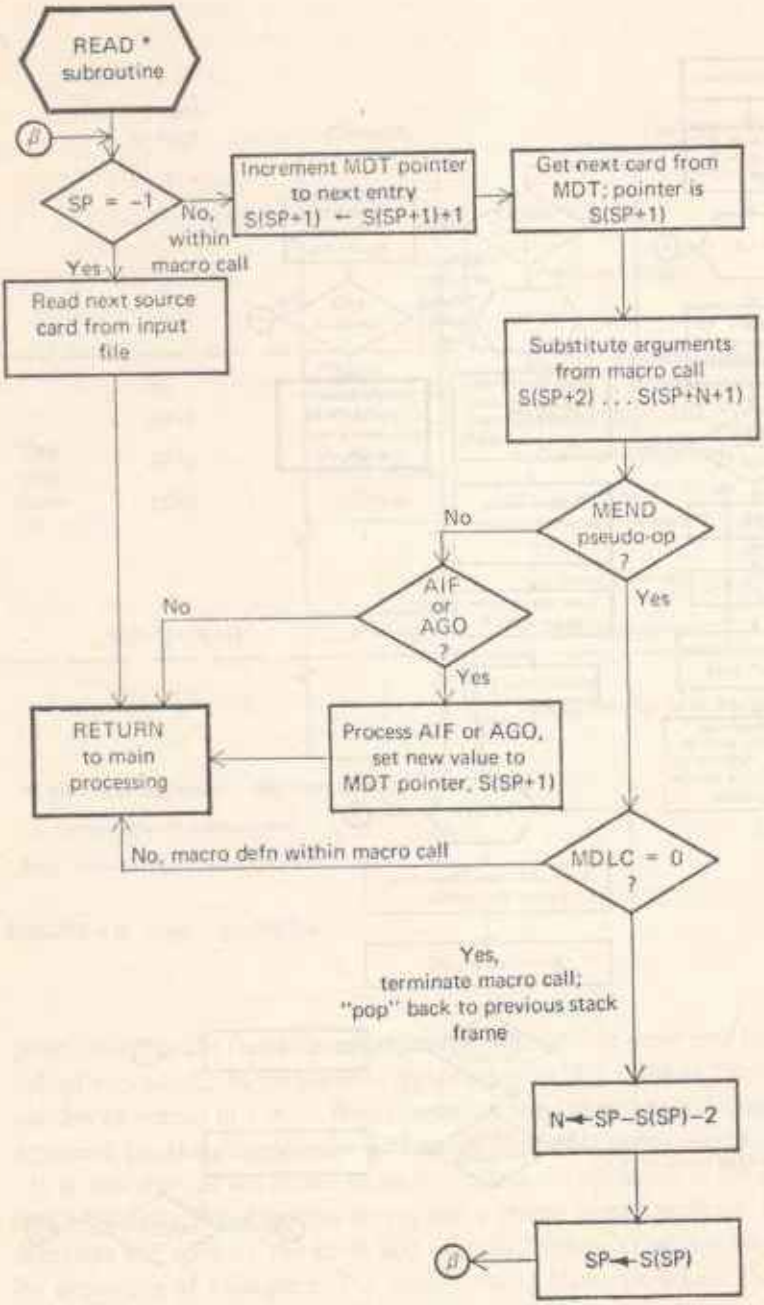


FIGURE 4.8 Detail of READ function for recursive macro expansion

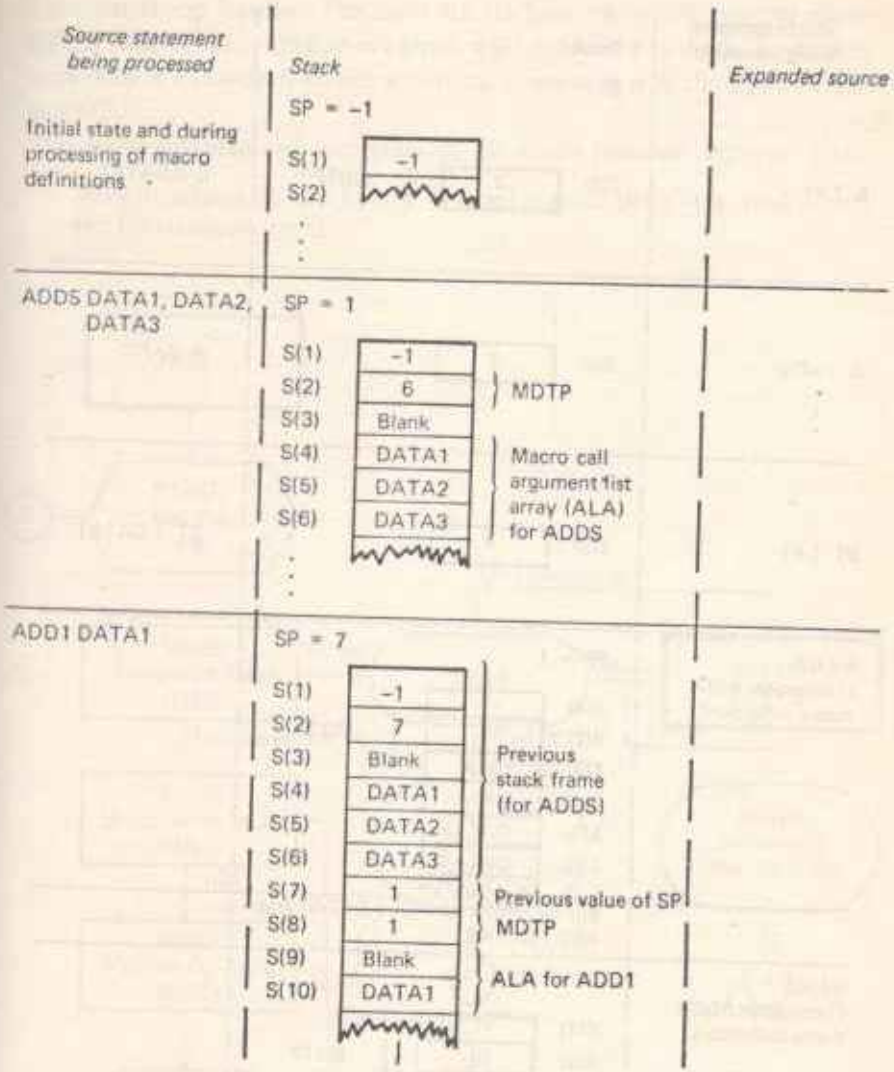


FIGURE 4.9 (a) Chronology of processing example 5 (part 1)

4.3.4 Implementation Within an Assembler

The macro processor can be added as a pre-processor to an assembler, making a complete pass over the input text before pass 1 of the assembler. The macro processor can also be implemented within pass 1 of the assembler.

The implementation of the macro processor within pass 1 eliminates the overhead of intermediate files, and we can improve this integration of macro pro-

Explain the implementation of a macro processor within an assembler. What are major advantages and disadvantages in the implementation?

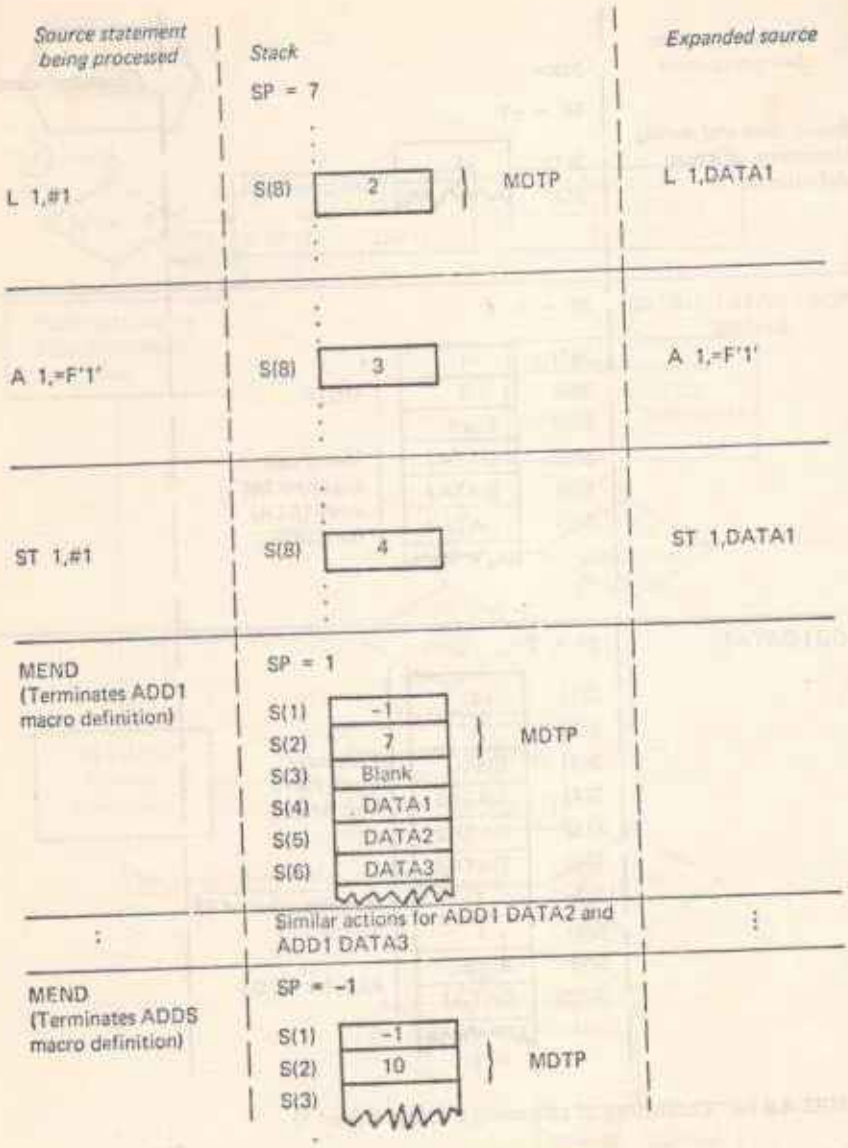


FIGURE 4.9 (b) Chronology of processing example 5 (part 2)

cessor and assembler by combining similar functions. For example, the Macro Name Table (MNT) can be combined with the assembler's op-code table (MOT or POT); a flag in each entry indicates whether or not it is a macro name. MACRO pseudo-ops to the macro processor can be detected by the assembler's

regular pseudo-op handler. The input READ function, which expands macro calls and receives the original source input, will be the same as that of Figure 4.8. Figure 4.10 is a flowchart of this algorithm; compare it with the flowcharts of Chapter 3.

The major advantages of incorporating the macro processor into pass 1 are:

1. Many functions do not have to be implemented twice (e.g., read a card, test for statement type).

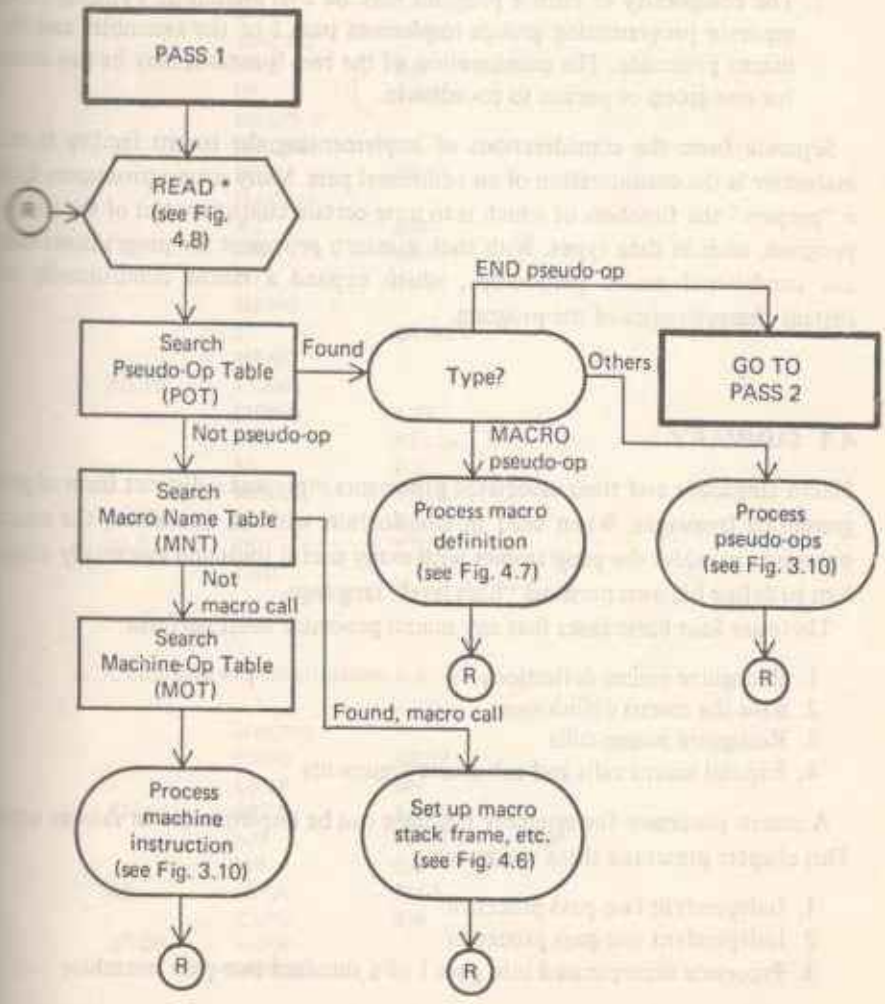


FIGURE 4.10 A macro processor combined with assembler pass 1

2. There is less overhead during processing: functions are combined and it is not necessary to create intermediate files as output from the macro processor and input to the assembler.
3. More flexibility is available to the programmer in that he may use all the features of the assembler (e.g., EQU statements) in conjunction with macros.

The major disadvantages are:

1. The combined pass 1 of the assembler and the macro processor may be too large a program to fit into core of some machines.
2. The complexity of such a program may be overwhelming. Typically, two separate programming groups implement pass 1 of the assembler and the macro processor. The combination of the two functions may be too much for one group or person to coordinate.

Separate from the considerations of implementing the macro facility in the assembler is the consideration of an additional pass. Many macro processors have a "prepass" the function of which is to note certain characteristics of the source program, such as data types. With such a macro processor the programmer may use conditional macro pseudo-ops, which expand a macro conditionally on certain characteristics of the program.

4.4 SUMMARY

Macro languages and their associated processors represent a distinct form of programming languages. When used in conjunction with an assembler, the macro processor provides the programmer with many useful tools and essentially allows him to define his own personal "high level" language.

There are four basic tasks that any macro processor must perform:

1. Recognize macro definitions
2. Save the macro definitions
3. Recognize macro calls
4. Expand macro calls and substitute arguments

A macro processor for assembly language can be implemented in various ways. This chapter presented three variations:

1. Independent two-pass processor
2. Independent one-pass processor
3. Processor incorporated into pass 1 of a standard two-pass assembler

5

loaders

The purpose of this chapter is to discuss various loader schemes and to present the design of a direct-linking loader.

As illustrated in the previous chapter, the user's *source program* decks are usually converted to *object program* decks (machine language) by assemblers and compilers. The *loader* is a program which accepts the object program decks, prepares these programs for execution by the computer, and initiates the execution (see Fig. 5.1).

In particular, the loader must 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)*

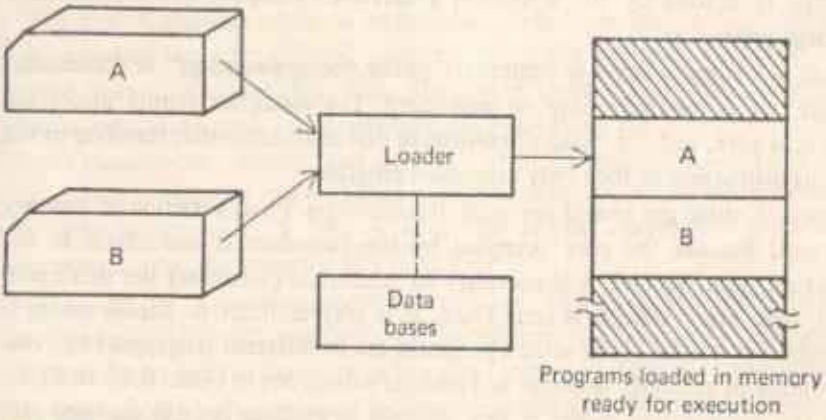


FIGURE 5.1 General loading scheme

Most of the examples used in this section are based upon the IBM System/370 assembler and loader. Several of the alternative loader schemes discussed are based upon computers with a fixed-word, direct-address instruction format, such as the IBM 7094, IBM 1130, UNIVAC 1108 and GE 635.

For simplicity of presentation, we assume card deck inputs – these of course may actually be card images on tape or any secondary storage.

5.1 LOADER SCHEMES

In this section we discuss various schemes for accomplishing the four functions of a loader. It is desirable to introduce the term *segment*, which is a unit of information that is treated as an entity, be it a program or data. Usually a segment corresponds to a single source or object deck. It is possible to produce multiple program or data segments in a single source deck by means of the assembly CSECT (Control Section) pseudo-op, the FORTRAN COMMON statement, or the PL/I EXTERNAL STATIC data attribute.

5.1.1 "Compile-and-Go" Loaders

One method of performing the loader functions is to have the assembler run in one part of memory and place the assembled machine instructions and data, as they are assembled, directly into their assigned memory locations (Fig. 5.2). When the assembly is completed, the assembler causes a transfer to the starting instruction of the program. This is a simple solution, involving no extra procedures. It is used by the WATFOR FORTRAN compiler and several other language processors.

Such a loading scheme is commonly called "compile-and-go" or "assemble-and-go." It is relatively easy to implement. The assembler simply places the code into core, and the "loader" consists of one instruction that transfers to the starting instruction of the newly assembled program.

However, there are several apparent disadvantages. First, a portion of memory is wasted because the core occupied by the assembler is unavailable to the object program. Second, it is necessary to retranslate (assemble) the user's program deck every time it is run. Third, it is very difficult to handle multiple segments, especially if the source programs are in different languages (e.g., one subroutine in assembly language and another subroutine in FORTRAN or PL/I). This last disadvantage makes it very difficult to produce orderly modular programs as discussed in the design of assemblers.

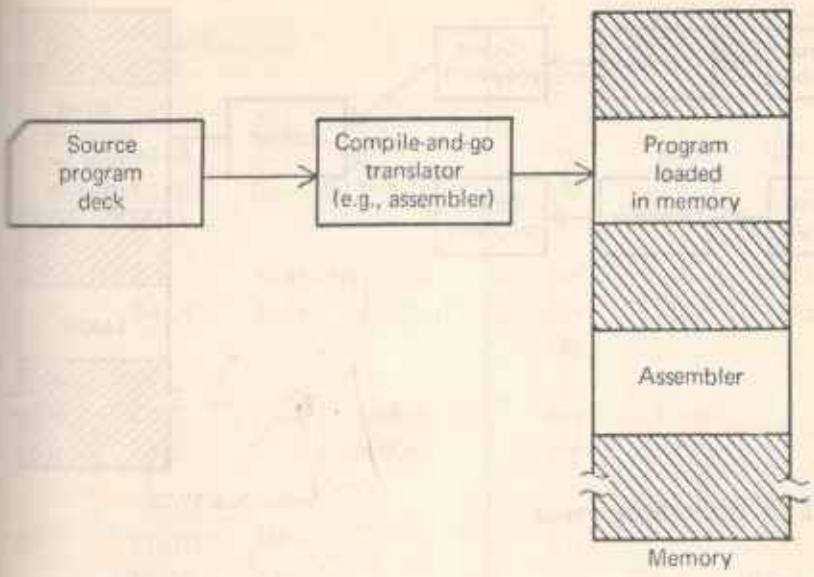


FIGURE 5.2 "Compile-and-go" loader scheme

5.1.2 General Loader Scheme

Outputting the instructions and data as they are assembled circumvents the problem of wasting core for the assembler. Such an output could be saved and loaded whenever the code was to be executed. The assembled programs could be loaded into the same area in core that the assembler occupied (since the translation will have been completed). This output form, which may be on cards containing a coded form of the instructions, is called an *object deck*.

The use of an object deck as intermediate data to avoid one disadvantage of the preceding "compile-and-go" scheme requires the addition of a new program to the system, a loader (Fig. 5.3). The *loader* accepts the assembled machine instructions, data, and other information present in the object format, and places machine instructions and data in core in an executable computer form. The loader is assumed to be smaller than the assembler, so that more memory is available to the user. A further advantage is that reassembly is no longer necessary to run the program at a later date.

Finally, if all the source program translators (assemblers and compilers) produce compatible object program deck formats and use compatible linkage conventions, it is possible to write subroutines in several different languages since the object decks to be processed by the loader will all be in the same "language" (machine language).

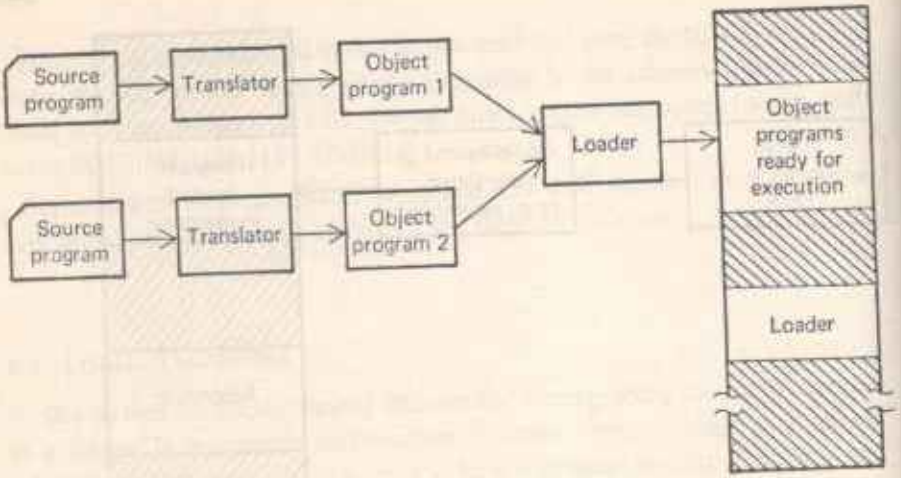


FIGURE 5.3 General loader scheme

5.1.3 Absolute Loaders

The simplest type of loader scheme, which fits the general model of Figure 5.3, is called an absolute loader. In this scheme the assembler outputs the machine language translation of the source program in almost the same form as in the "assemble-and-go" scheme, except that the data is punched on cards (object deck) instead of being placed directly in memory. The loader in turn simply accepts the machine language text and places it into core at the location prescribed by the assembler. This scheme makes more core available to the user since the assembler is not in memory at load time.

Absolute loaders are simple to implement but they do have several disadvantages. First, the programmer must specify to the assembler the address in core where the program is to be loaded. Furthermore, if there are multiple subroutines, the programmer must remember the address of each and use that absolute address explicitly in his other subroutines to perform subroutine linkage.

Figure 5.4 illustrates the operation of an absolute assembler and an absolute loader. The programmer must be careful not to assign two subroutines to the same or overlapping locations.

The MAIN program is assigned to locations 100-247 and the SQRT subroutine is assigned locations 400-477. If changes were made to MAIN that increased its length to more than 300 bytes, the end of MAIN (at 100 + 300 = 400) would overlap the start of SQRT (at 400). It would then be necessary to assign SQRT

MAIN program

```
MAIN    START   100
        BALR    12,0
        USING   MAIN+2,12
        .
        .
        .
        L       15,ASQRT  } Call
        BALR    14,15     } SQRT
        .
        .
        .
ASQRT   DC      F'400'    } Address
        END               } of SQRT
```

SQRT subroutine

```
SQRT    START   400
        USING   *,15
        .       } Compute
        .       } square root
        .
        BR      14 } Return
        END
```

SOURCE DECK INPUT
TO ABSOLUTE ASSEMBLER

MAIN program

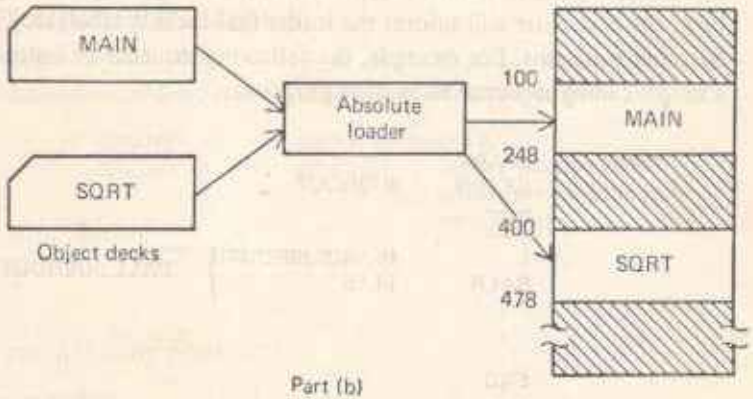
Location	*Instruction*	
100	BALR	12,0
102	⋮	
⋮		
120	L	15,142(0,12)
124	BALR	14,15
126	⋮	
⋮		
244	F'400'	
248		

SQRT subroutine

400	⋮	
⋮		
476	BCR	15,14
478	⋮	

OBJECT DECK OUTPUT
FROM ABSOLUTE ASSEMBLER

Part (a)



Part (b)

FIGURE 5.4 Absolute loader example

to a new location by changing its START pseudo-op card and reassembling it. Furthermore, it would also be necessary to modify all other subroutines that referred to the address of SQRT. In situations where dozens of subroutines are being used, this manual "shuffling" can get very complex, tedious, and wasteful of core.

The four loader functions are accomplished as follows in an absolute loading scheme:

1. Allocation – by programmer
2. Linking – by programmer
3. Relocation – by assembler
4. Loading – by loader

5.1.4 Subroutine Linkages

write short note on subroutine linkages.

In this section we briefly discuss, from a programmer's point of view, the special mechanism for calling another subroutine in an assembly language program.

The problem of subroutine linkage is this: a main program A wishes to transfer to subprogram B. The programmer, in program A, could write a transfer instruction (e.g., BAL 14,B) to subprogram B. However, the assembler does not know the value of this symbol reference and will declare it as an error (undefined symbol) unless a special mechanism has been provided.

This mechanism is typically implemented with a relocating or a direct-linking loader. The assembler pseudo-op EXTRN followed by a list of symbols indicates that these symbols are defined in other programs but referenced in the present program. Correspondingly, if a symbol is defined in one program and referenced in others, we insert it into a symbol list following the pseudo-op ENTRY. In turn, the assembler will inform the loader that these symbols may be referenced by other programs. For example, the following sequence of instructions may be a simple calling sequence to another program:

```
MAIN    START
        EXTRN   SUBROUT
        ---
        ---
        L       15,=A(SUBROUT)  } CALL SUBROUT
        BALR    14,15           }
        .
        .
        .
        END
```

The above sequence of instructions first declares SUBROUT as an external variable, that is, a variable referenced but not defined in this program. The load

instruction loads the address of that variable into register 15. The BALR instruction branches to the contents of register 15, which is the address of SUBROUT, and leaves the value of the next instruction in register 14. In most assemblies we may simply use a CALL SUBROUT macro, which is translated by the assembler into a calling sequence as shown above.

Now we may see the reason for programming conventions, since it is necessary for both the caller and called subprograms to cooperate. On a 360 we observe the convention that register 15 is used as a linkage and base register. Note that the caller in the above program has loaded register 15 with the beginning address of the subroutine being called. Thus the called subroutine does not have to load a base register. Register 14 contains the return address of the caller. The programmer must not use register 14 within a subprogram unless he saves its contents and restores them before he returns. A typical sequence for subroutines is

```
SUBROUT    START
           USING     *,15
             :
           BR        14
           END
```

No BALR instruction is necessary at the beginning since register 15 was already loaded with the address of the start of this program. The BR 14 instruction is an unconditional branch to the address contained in register 14, which is the return address of the calling program. In Appendix B we present in more detail the methods that are typically used on a 360 and discuss further the calling sequences. The following discussion introduces the basic mechanism used.

ASSEMBLER LINKAGE PSEUDO-OPS

Subroutine and Entry Naming (START and ENTRY Pseudo-ops)

```
A          START     defines subroutine A.
           ENTRY     B1,B2,B3, . . . defines locations B1, . . .,
                                      Bn as additional subroutine
                                      entry points
B1         ------
B2         ------
```

The uses of multiple entry points are:

1. Common coding.
   Example: SIN and COS involve basically the same computations and could employ different entry points of the same routine.

2. Collecting together related routines for convenience.
3. Better or convenient access to common data base.

SUBROUTINE REFERENCE (EXTRN PSEUDO-OP)

Assembler symbols are either internal or external. External means that their value is not known to the assembler but will be provided by the loader – the action of the loader will be discussed in the following section.

```
EXTRN E1,E2,etc.
```

defines E1, E2, etc. as external symbols to be used in address constants.

Example: CALL BETA becomes:

```
        EXTRN   BETA
        :
        L       15,ABETA
        BALP    14,15
        :
ABETA DC        A(BETA)
```

Explain self relocating loader in detail

5.1.5 Relocating Loaders

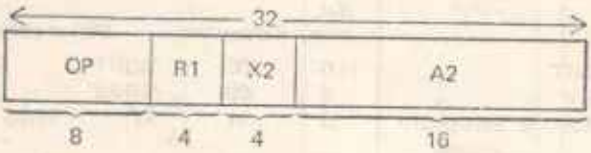
To avoid possible reassembling of all subroutines when a single subroutine is changed, and to perform the tasks of allocation and linking for the programmer, the general class of *relocating loaders* was introduced. An example of a relocating loader scheme is that of the Binary Symbolic Subroutine (BSS) loader such as was used in the IBM 7094, IBM 1130, GE 635, and UNIVAC 1108. The BSS loader allows many procedure segments, yet only one data segment (common segment). The assembler assembles each procedure segment independently and passes on to the loader the text and information as to relocation and interseg-ment references.

The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it references. In addition, there is information (relocation information) as to locations in this program that need to be changed if it is to be loaded in an arbitrary place in core, i.e.. the locations which are dependent on the core allocation.

For each source program the assembler outputs a text (machine translation of the program) prefixed by a *transfer vector* that consists of addresses containing names of the subroutines referenced by the source program. For example, if a Square Root Routine (SQRT) was referenced and was the first subroutine called, the first location in the transfer vector would contain the symbolic name SQRT.

The statement calling SQRT would be translated into a transfer instruction indicating a branch to the location of the transfer vector associated with SQRT. The assembler would also provide the loader with additional information, such as the length of the entire program and the length of the transfer vector portion. After loading the text and the transfer vector into core, the loader would load each subroutine identified in the transfer vector. It would then place a transfer instruction to the corresponding subroutine in each entry in the transfer vector. Thus, the execution of the call SQRT statement would result in a branch to the first location in the transfer vector, which would contain a transfer instruction to the location of SQRT.

The BSS loader scheme is often used on computers with a fixed-length direct-address instruction format. For example, *if* the format of the 360 RX instruction were:



where A2 was the 16-bit absolute address of the operand, this would be a *direct address* instruction format. Such a format works if there are less than 2^{16} = 65,536 bytes of storage, as was the case with most of the early computers and is still true for many of the current "minicomputers" and "midicomputers."

Since it is necessary to relocate the address portion of every instruction, computers with a direct-address instruction format have a much more severe relocation problem than the 360. In the absence of 360-type base registers, this problem is often solved by the use of "*relocation bits*." The assembler associates a bit with each instruction or address field. If this bit equals one, then the corresponding address field must be relocated; otherwise the field is not relocated. These relocation indicators, surprisingly enough, are known as relocation bits and are included in the object deck.

Figure 5.5 illustrates a simple assembly language program written for a hypothetical "direct-address" 360 that uses a BSS loader. The function of the program is not important; it supposedly calls the SQRT subroutine to get the square root of 9. If the result is not 3, it transfers to a subroutine called ERR. Since this is a direct-address computer, there is no base register field in the object code and no need for a USING pseudo-op in the source program. The EXTRN pseudo-op identifies the symbols SQRT and ERR as the names of other subroutines; since the locations of these symbols are not defined in this subroutine, they are called *external symbols*. For each external symbol the assembler generates a four-

Explain with example how linking is performed by a BSS loader and direct linking loader

→ Both a BSS loader and Direct linking loader are relocating loader, but in both cases linking is performed in different way. To explain this let us consider a assembly

byte fullword at the beginning of the program, containing the EBCDIC characters for the symbol (for simplicity, we are assuming that symbols are not more than four characters long). These extra words are called transfer vectors. Every reference to an external symbol is assigned the address of the corresponding transfer vector word. In addition, for every halfword (two bytes) in the program, the assembler produces a separate relocation bit. For example, the assembled instruction ST 14,36 is assigned relocation bits 01 since the first halfword contains the op-code, register field, and index field which should not be relocated but the second halfword contains the relative address 36, which must be relocated.

① BSS loader: Assume program length = 48 bytes
Transfer = 8 bytes

Program length = 48 bytes
Transfer vector = 8 bytes

Source program				Rel. addr.	Relocation	Object code	
MAIN	START						
	EXTRN	SQRT		0	00	'SQRT'	
	EXTRN	ERR		4	00	'ERRb'	
	ST	14,SAVE	Save return address	8	01	ST	14,36
	L	1,=F'9'	Load test value	12	01	L	1,40
	BAL	14,SQRT	Call SQRT	16	01	BAL	14,0
	C	1,=F'3'	Compare answer	20	01	C	1,44
	BNE	ERR	Transfer to ERR	24	01	BC	7,4
	L	14,SAVE	Get return address	28	01	L	14,36
	BR	14	Return to caller	32	0	BCR	15,14
				34	0	(Skipped for alignment)	
SAVE	DS	F	Temp. loc.	36	00	(Temp location)	
	END			40	00	9	
				44	00	3	

FIGURE 5.5 Assembly of program for "direct-address" 360

Assembly of program for direct address

Figure 5.6 illustrates the contents of memory after the programs have been loaded by the BSS loader. Based upon the relocation bits, the loader has relocated the address fields to correspond to the allocated address of MAIN which is 400. Using the program length information, the loader placed the subroutines SQRT and ERR at the next available locations which were 448 and 526, respectively. Finally, the transfer vector words were changed to contain

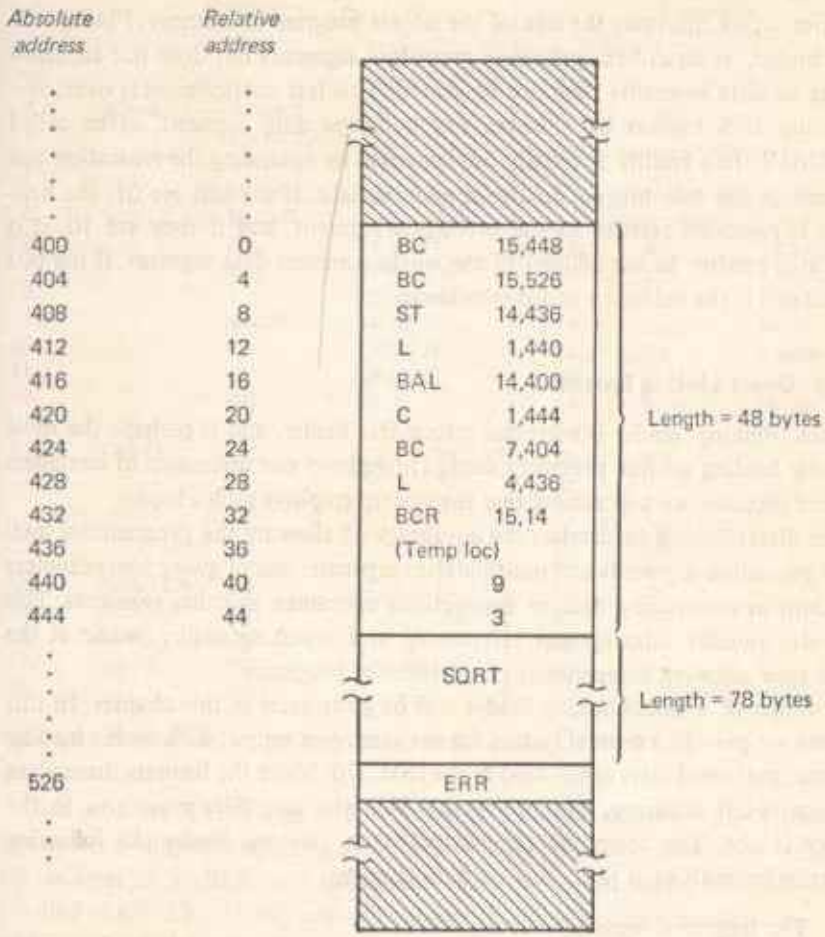


FIGURE 5.6 BSS loading of programs for "direct access" 360

branch instructions to the corresponding subroutines. Thus, the four functions of the loader (allocation, linking, relocation, and loading) were all performed automatically by the BSS loader.

It should be noted that the relocation bits are used to solve the problem of relocation; the transfer vector is used to solve the problem of linking; and the program length information is used to solve the problem of allocation.

There are several disadvantages to the BSS loader scheme. First, the transfer vector linkage is only useful for transfers, and is not well-suited for loading or storing external data (data located in another procedure segment). Second, the

transfer vector increases the size of the object program in memory. Finally, the BSS loader, as described, processes procedure segments but does not facilitate access to data segments that can be shared. This last shortcoming is overcome in many BSS loaders by allowing one common data segment, often called *COMMON*. This facility is usually implemented by extending the relocation bits scheme to use two bits per halfword address field: if the bits are 01, the halfword is relocated relative to the procedure segment, and if they are 10, it is relocated relative to the address of the single common data segment. If the bits are 00 or 11, the halfword is not relocated.

5.1.6 Direct-Linking Loaders

A direct-linking loader is a general relocatable loader, and is perhaps the most popular loading scheme presently used. Throughout our discussion of compilers in later sections, we will assume that the system employs such a loader.

The direct-linking loader has the advantage of allowing the programmer multiple procedure segments and multiple data segments and of giving him complete freedom in referencing data or instructions contained in other segments. This provides flexible intersegment referencing and accessing ability, while at the same time allowing independent translations of programs.

A design of a direct-linking loader will be given later in this chapter. In this section we present a general format for the assembler output with such a loading scheme, patterned after those used in the IBM 370. While the formats themselves are somewhat arbitrary, the information that the assembler must give to the loader is not. The assembler (translator) must give the loader the following information with each procedure or data segment:

1. The length of segment
2. A list of all the symbols in the segment that may be referenced by other segments and their relative location within the segment
3. A list of all symbols not defined in the segment but referenced in the segment
4. Information as to where address constants are located in the segment and a description of how to revise their values
5. The machine code translation of the source program and the relative addresses assigned

A simple example using a direct-linking loading scheme is presented in Figure 5.7. A source program (left-hand column) is translated by an assembler to produce the object code depicted in the right column. Mnemonic machine codes have again been used in the translation shown.

for assembly language program shown below

① Equivalent mnemonic m/c lang. translated program

② contents of ESD card, TXT card, ERLD card for linkage loader

③ object deck for direct linking loader

Card no.	Program			Rel. loc.	Translation	
1.	JOHN	START				
2.		ENTRY	RESULT			
3.		EXTRN	SUM			
4.		BALR	12,0	① 0	BALR	12,0
5.		USING	*,12			
6.		ST	14,SAVE	2	ST	14,54(0,12)
7.		L	1,POINTER	6	L	1,46(0,12)
8.		L	15,ASUM	10	L	15,58(0,12)
9.		BALR	14,15	14	BALR	14,15
10		ST	1,RESULT	16	ST	1,50(0,12)
11.		L	14,SAVE	20	L	14,54(0,12)
12.		BR	14	24	BCR	15,14
				26	—	
13.	TABLE	DC	F'1,7,9,10,3'	28	1	
				32	7	
				36	9	
				40	10	
				44	3	
14.	POINTER	DC	A(TABLE)	48	28	
15.	RESULT	DS	F	52	—	
16.	SAVE	DS	F	56	—	
17.	ASUM	DC	A(SUM)	60	?	
18.		END		64		

FIGURE 5.7 Assembly source program and its translation

Card number 14 of Figure 5.7 contains a Define Constant (DC) pseudo-operation which instructs the assembler to create a constant with the value of the address of TABLE, and causes this constant to be placed in the location labelled POINTER. At this point the assembler does not know the final absolute address of TABLE since it has no idea where the program is going to be loaded. It knows, however, that the address is the 28th byte from the beginning of this program. The assembler will put a 28 in POINTER and inform the loader that the content of location POINTER is incorrect if this program is loaded anywhere except absolute location 0. For instance, if this program were loaded in location 2000, the loader would have to change the contents of POINTER to be a 2028.

Card number 17 of Figure 5.7 is another DC pseudo-op, which instructs the assembler to create a constant with the value of the address of the subroutine SUM and cause this constant to be placed in the location labelled ASUM. Since the assembler has no idea where the procedure SUM will be loaded, it cannot generate this constant. Thus, the assembler must provide information to the loader that will cause it to put the final absolute address of SUM at the

designated location (ASUM) when the programs are loaded.

We have named the program JOHN. Hence, JOHN is a symbol that may be referenced externally or "called" by other programs. We also have stated that the symbol RESULT may be referenced by other programs. These facts must be passed on to the loader.

The design of the direct-linking loading scheme we present is similar to the standard IBM 370 scheme. The assembler produces four types of cards in the object deck: ESD, TXT, RLD, and END. External Symbol Dictionary (ESD) cards contain information about all symbols that are defined in this program but that may be referenced elsewhere, and all symbols referenced in this program but defined elsewhere. The text (TXT) cards contain the actual object code translated version of the source program. The Relocation and Linkage Directory (RLD) cards contain information about those locations in the program whose contents depend on the address at which the program is placed. For such locations the assembler must supply information enabling the loader to correct their contents. The END card indicates the end of the object deck and specifies the starting address for execution if the assembled routine is the "main" program. Figure 5.8 depicts the information that would appear for the preceding program on the ESD, TXT, RLD, and END cards.

The reference numbers in Figure 5.8 do not actually appear on the cards. They are for the benefit of the reader, and each denotes the card number of the original program that resulted in the object deck card; e.g., the first RLD card resulted from card number 14 in the original program,

As shown in Figure 5.8, three ESD cards are needed for the program JOHN. The first card contains the name of the program JOHN, which may be referenced externally. The "type" mnemonic we have used is SD, which means the symbol is a Segment Definition. The relative address of JOHN is 0, and the length is of the program that JOHN denotes, 64. On the next ESD card appears the symbol RESULT, which is a Local Definition (LD); its relative address is 52. The final ESD card specifies that the symbol SUM is an External Reference (ER). We will see in a later section how the ER symbols are actually used in conjunction with the RLD cards.

The TXT cards contain the actual assembled program. The format and use of these cards are similar to those for the absolute loader.

The RLD cards contain the following information:

1. The location of each constant that needs to be changed due to relocation
2. By what it has to be changed
3. The operation to be performed

The first RLD card of our example contains a 48, denoting the relative loca-

ESD cards

Reference no.	*Symbol*	*Type*	*Relative location*	*Length*
1	JOHN	SD	0	64
2	RESULT	LD	52	—
3	SUM	ER	—	—

TXT cards

Reference no.	*Relative location*	*Object code*	
4	0	BALR	12,0
6	2	ST	14,54(0,12)
7	6	L	1,46(0,12)
8	10	L	15,58(0,12)
9	14	BALR	14,15
10	16	ST	1,50(0,12)
11	20	L	14,54(0,12)
12	24	BCR	15,14
13	28	1	
13	32	7	
13	36	9	
13	40	10	
13	44	3	
14	48	28	
17	60	0	

RLD cards

Reference no.	*Symbol*	*Flag*	*Length*	*Relative location*
14	JOHN	+	4	48
17	SUM	+	4	60

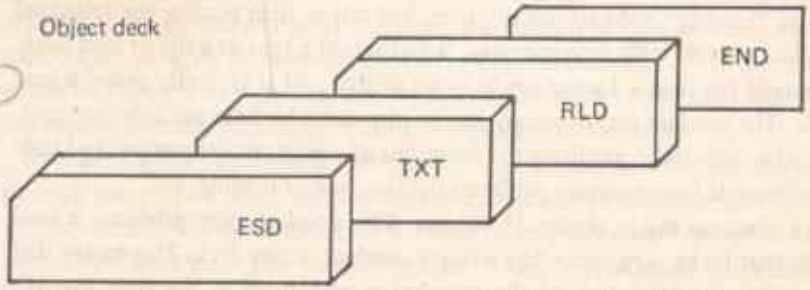


FIGURE 5.8 Example object deck for a direct-linking loader

tion of a constant that must be changed; a plus sign denoting that something must be added to the constant; and the symbol field indicating that the value of external symbol JOHN must be added to relative location 48. The relative value of JOHN is 0. When the program is loaded, the loader will determine its absolute value.

The second RLD card of our example contains a 60, denoting the relative location of a constant that must be changed. The symbol field indicates that the value of the external symbol SUM must be added to relative location 60. Although the assembler does not know the absolute address of SUM, the loader will later be able to fill in the correct value.

The process of adjusting the address constant of an internal symbol, such as TABLE, is normally called relocation; while the process of supplying the contents of an address constant for an external symbol, such as SUM, is normally referred to as linking. Significantly, the RLD card mechanism is used for both cases, which explains why they are called relocation *and* linkage directory cards. The reader may wish to compare this technique with the mechanisms used in the BSS relocating loader described earlier in this chapter.

5.1.7 Other Loader Schemes — Binders, Linking Loaders, Overlays, Dynamic Binders

There are numerous variations to the previously presented loader schemes.

One disadvantage of the direct-linking loader, as presented, is that it is necessary to allocate, relocate, link, and load all of the subroutines each time in order to execute a program. Since there may be tens and often hundreds of subroutines involved, especially when we include utility routines such as SQRT, etc., this loading process can be extremely time-consuming. Furthermore, even though the loader program may be smaller than the assembler, it does absorb a considerable amount of space. These problems can be solved by dividing the loading process into two separate programs: a binder and a module loader.

A *binder* is a program that performs the same functions as the direct-linking loader in "binding" subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck. This output file is in a format ready to be loaded and is typically called a *load module*. The *module loader* merely has to physically load the module into core. The binder essentially performs the functions of allocation, relocation, and linking; the module loader merely performs the function of loading.

There are two major classes of binders. The simplest type produces a load module that looks very much like a single absolute loader deck. This means that the specific core allocation of the program is performed at the time that the

what is meant by binder? and also explain the module loader.

advantage with core image binder or linkage editor is that a program is to be bound only once and it can be used any number of times and loaded without the need to bind once again

subroutines are bound together. Since this kind of module looks like an actual "snapshot" or "image" of a section of core, it is called a *core image module* and the corresponding binder is called a *core image builder*. A more sophisticated binder, called a *linkage editor*, can keep track of the relocation information so that the resulting load module, as an ensemble, can be further relocated and thereby loaded anywhere in core. In this case the module loader must perform additional allocation and relocation as well as loading, but it does not have to worry about the complex problems of linking.

In both cases, a program that is to be used repeatedly need only be bound once and then can be loaded whenever required. The core image builder binder is relatively simple and fast. The linkage editor binder is somewhat more complex but allows a more flexible allocation and loading scheme.

DYNAMIC LOADING

In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time. If the total amount of core required by all these subroutines exceeds the amount available, as is common with large programs or small computers, there is trouble! There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem; these are discussed in Chapter 9. In this section we will present conventional dynamic loading schemes based upon the use of a binder prior to loading.

Usually the subroutines of a program are needed at different times: for example, pass 1 and pass 2 of an assembler are mutually exclusive. By explicitly recognizing which subroutines call other subroutines it is possible to produce an *overlay structure* that identifies mutually exclusive subroutines. Figure 5.9a illustrates a program consisting of five subprograms (A,B,C,D and E) that require 100K bytes of core. The arrows indicate that subprogram A only calls B, D and E; subprogram B only calls C and E; subprogram D only calls E; and subprograms C and E do not call any other routines. Figure 5.9b highlights the interdependencies between the procedures. Note that procedures B and D are never in use at the same time; neither are C and E. If we load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure. This happens to be 70K for the example in Figure 5.9b – procedures A, B, and C. Figure 5.9c illustrates a storage assignment for each procedure consistent with the overlay structure.

In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed. We will not go into their specific details, but there are many binders capable of processing and allocating an overlay structure. The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is called the *overlay supervisor* or

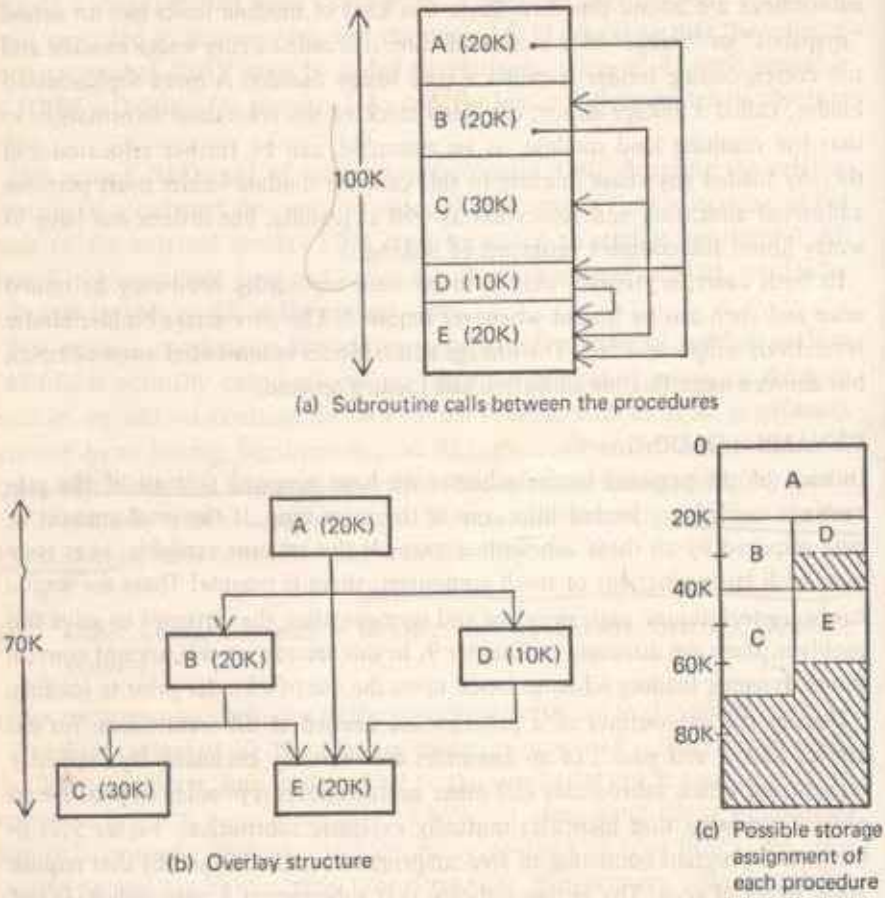


FIGURE 5.9 Other loading schemes

simply the *flipper*. This overall scheme is called *dynamic loading* or *load-on-call* (LOCAL).

DYNAMIC LINKING

A major disadvantage of all of the previous loading schemes is that if a subroutine is referenced but never executed (e.g., if the programmer had placed a call statement in his program but this statement was never executed because of a condition that branched around it), the loader would still incur the overhead of linking the subroutine.

Furthermore, all of these schemes require the programmer to explicitly name all procedures that might be called. It is not possible to write programs as follows:

```
    ⋮
READ      SUBNAME, ARGUMENT
ANSWER =  SUBNAME(ARGUMENT)
PRINT     ANSWER
    ⋮
```

where the name of the subroutine (e.g., SQRT, SINE, etc.) is an input parameter, SUBNAME, just like the other data.

A very general type of loading scheme that we will discuss in Chapter 9 is called dynamic linking. This is a mechanism by which loading and linking of external references are postponed until execution time. That is, the assembler produces text, binding, and relocation information from a source language deck. The loader loads only the main program. If the main program should execute a transfer instruction to an external address, or should reference an external variable (that is, a variable that has not been defined in this procedure segment), the loader is called. Only then is the segment containing the external reference loaded.

An advantage here is that no overhead is incurred unless the procedure to be called or referenced is actually used. A further advantage is that the system can be dynamically reconfigured. The major drawback to using this type of loading scheme is the considerable overhead and complexity incurred, due to the fact that we have postponed most of the binding process until execution time.

What are advantages & disadvantages of Dynamic linking?

5.2 DESIGN OF AN ABSOLUTE LOADER

We introduce the general topic of loader design by presenting a design of an absolute loader.

With an absolute loading scheme the programmer and the assembler perform the tasks of allocation, relocation, and linking. Therefore, it is only necessary for the loader to read cards of the object deck and move the text on the cards into the absolute locations specified by the assembler.

There are two types of information that the object deck must communicate from the assembler to the loader. First, it must convey the machine instructions that the assembler has created along with the assigned core locations. Second, it must convey the entry point of the program, which is where the loader is to transfer control when all instructions are loaded. Assuming that this information is transmitted on cards, a possible format is shown in Figure 5.10.

Note that in the card format shown the instructions are stored on the card as one core byte per column. For each of the 256 possible contents of an eight-bit

byte there is a corresponding punched card code (e.g., hexadecimal 00 is a column punched with five holes, 12-0-1-8-9, whereas a hexadecimal F1 is a column with a single punch in row 1). Thus, when a card is read, it is stored in core as 80 contiguous bytes.

Text cards (for instructions and data)

Card column	*Contents*
1	Card type = 0 (for text card identifier)
2	Count of number of bytes (1 byte per column) of information on card
3-5	Address at which data on card is to be put
6-7	Empty (could be used for validity checking)
8-72	Instructions and data to be loaded
73-80	Card sequence number

Transfer cards (to hold entry point to program)

Card column	*Contents*
1	Card type = 1 (transfer card identifier)
2	Count = 0
3-5	Address of entry point
6-72	Empty
73-80	Card sequence number

FIGURE 5.10 Card formats for an absolute loader

The algorithm for an absolute loader is quite simple. The object deck for this loader consists of a series of text cards terminated by a transfer card. Therefore, the loader should read one card at a time, moving the text to the location specified on the card, until the transfer card is reached. At this point the assembled instructions are in core, and it is only necessary to transfer to the entry point specified on the transfer card. A flowchart for this process is illustrated in Figure 5.11.

5.3 DESIGN OF A DIRECT-LINKING LOADER

In this section a design of an IBM 360-type direct-linking loader is presented. Certain obscure features (primarily related to the IBM PL/I implementation and overlay structures) have been omitted, and where alternative formats are possible, only the simplest is given.

The design steps followed will parallel those taken in the design of an assembler (Chapter 3). Note that because the direct-linking loader needs to know

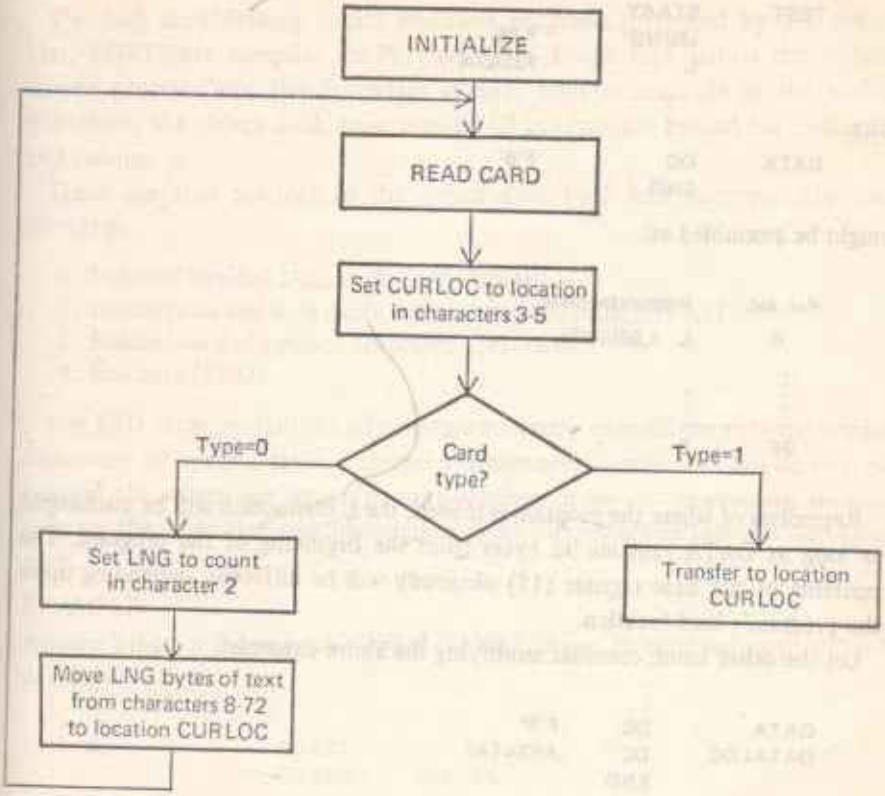


FIGURE 5.11 Absolute loader

the absolute (load time) values of some external symbols before it can perform the modifications on address constants, it requires two passes.

5.3.1 Specification of Problem

The organization of the IBM 360 facilitates the tasks to be performed by its relocating loader. On the IBM 7094, a direct access machine, it was necessary to relocate the address portion on almost all instructions. In the 360, instruction relocation is accomplished by the use of the base register, which is set by neither the assembler nor the loader. Therefore, the 360 relocating loader can treat instructions *exactly* like nonrelocatable data (full word constants, characters, etc.). However, address constants must still be relocated.

For example, the following instructions:

```
TEST     START
         USING    *,15
         L        1,DATA
           ⋮
DATA     DC       F'5'
         END
```

might be assembled as:

```
Rel. loc.   Instruction/data
   0        L  1,96(0,15).
   ⋮        ⋮
  96        5
```

Regardless of where the program is loaded, the L instruction will be unchanged as long as DATA remains 96 bytes from the beginning of the program. The contents of the base register (15) obviously will be different, depending upon the program's load location.

On the other hand, consider modifying the above example:

```
           ⋮
DATA       DC      F'5'
DATALOC    DC      A(DATA)
           END
```

DATALOC must contain the absolute address of DATA. The assembler knows only that DATA is 96 bytes from the beginning of the program, so the loader must add to this the load address of the program in finding the actual absolute address to be contained in DATALOC.

Let us clarify the scope of the address constant problem. An address constant may be (1) absolute; (2) simple relocatable; or (3) complex relocatable.

For example, the address constant A(LOC1-LOC2) will be:

1. *Absolute* if LOC1 and LOC2 are two relocatable symbols defined internal to program – the assembler can calculate the actual value, their difference.
2. *Simple relocatable* if LOC1 is a relocatable symbol within this procedure and LOC2 is an absolute number (e.g., LOC2 EQU 5). The assembler can calculate the difference between relative location of LOC1 and value of LOC2, but the loader must perform relocation by adding the program load address.
3. *Complex relocatable* if LOC1 and LOC2 are entries to some other program. The assembler can do nothing, and the value must be calculated by the loader.

The 360 direct-linking loader processes programs generated by the assembler, FORTRAN compiler, or PL/I compiler. Recall that neither the original source program nor the assembler symbol table is available to the loader. Therefore, the object deck must contain all information needed for relocation and linking.

There are four sections to the object deck (and four corresponding card formats):

1. External Symbol Dictionary cards (ESD)
2. Instructions and data cards, called "text" of program (TXT)
3. Relocation and Linkage Directory cards (RLD)
4. End card (END)

The ESD cards contain the information necessary to build the external symbol dictionary or symbol table. External symbols are symbols that can be referred beyond the subroutine level. The normal labels in the source program are used only by the assembler, and information about them is not included in the object deck.

EXAMPLE

Assume program B has a table called NAMES; it can be accessed by program A as follows.

```
A          START
           EXTRN    NAMES
           :
           L        1,ADDRNAME get address of NAME table
           :
ADDRNAME   DC       A(NAMES)
           END
-------------------------------
B          START
           ENTRY    NAMES
           :
NAMES      DC       ----
           END
```

There are three types of external symbols, as illustrated in the above:

1. *Segment Definition* (SD) – name on START or CSECT card.
2. *Local Definition* (LD) – specified on ENTRY card. There must be a label in same program with same name.
3. *External Reference* (ER) – specified on EXTRN card. There must be a

corresponding ENTRY, START, or CSECT card in another program with same name.

Each SD and ER symbol is assigned a unique number (e.g., 1,2,3, . . .) by the assembler. This number is called the symbol's *identifier*, or *ID*, and is used in conjunction with the RLD cards.

The TXT cards contain blocks of data and the relative address at which the data is to be placed. Once the loader has decided where to load the program, it merely adds the *Program Load Address* (PLA) to the relative address and moves the data into the resulting location. The data on the TXT card may be instructions, nonrelocated data, or initial values of address constants.

EXAMPLE

```
Relative address        Instruction
        A               START
                        EXTRN   NAMES
                        USING   *,15
                        :
40                      L       1,ALPHA
44                      BCR     15,14
46                      ----------------------------  (Skipped by assembler)
48      ALPHA           DC      F'5'
52      ALLOC           DC      A(ALPHA)
56      ADDRNAME        DC      A(NAMES)
                        :
```

The TXT card produced is:

Relative address = 40
Data portion = *58 10 F0 30 07FE XX XX 00 00 00 05 00 00 00 30 00 00 00 00*
Length of data portion = 20 bytes

The RLD cards contain the following information.

1. The location and length of each address constant that needs to be changed for relocation or linking
2. The external symbol by which the address constant should be modified (added or subtracted)
3. The operation to be performed (add or subtract)

Rather than using the actual external symbol's name on the RLD card, as implied in section 5.1.6 and Figure 5.8, the external symbol's identifier, or ID, is used. There are various reasons for this, the major one probably being that the ID is only a single byte long, compared to the eight bytes occupied by the

What is the purpose of the ID no. on the RLD cards? why it is not needed for locally defined symbols
⇒ Same ID is not needed for locally defined symbols because the local definition symbols are specified on ENTRY cards

Source card reference	Relative address		Sample program (source deck)		
1	0	PG1	START		(a) Procedure PG1
2			ENTRY	PG1ENT1,PG1ENT2	
3			EXTRN	PG2ENT1,PG2	
4	20	PG1ENT1	≡		
5	30	PG1ENT2	≡		
6	40		DC	A(PG1ENT1)	
7	44		DC	A(PG1ENT2+15)	
8	48		DC	A(PG1ENT2-PG1ENT1-3)	
9	52		DC	A(PG2)	
10	56		DC	A(PG2ENT1+PG2-PG1ENT1+4)	
11			END		
12	0	PG2	START		(b) Procedure PG2
13			ENTRY	PG2ENT1	
14			EXTRN	PG1ENT1,PG1ENT2	
15	16	PG2ENT1	≡		
16	24		DC	A(PG1ENT1)	
17	28		DC	A(PG1ENT2+15)	
18	32		DC	A(PG1ENT2-PG1ENT1-3)	
19			END		

FIGURE 5.12 Sample procedures PG1 and PG2

reassuring to note in Figure 5.15 that both instances of this address constant (location 152, 200) have the same value − 7. Since both PG1ENT2 and PG1ENT1 are symbols internal to PG1, the assembler, processing PG1, can compute the entire expression and determine the value of 7. We see in Figure 5.13 that the TXT card for location 48-51 contains the 7 and there are no associated RLD cards for this address constant. On the other hand, these symbols are external to PG2; thus, the assembler, while processing PG2, has no means of evaluating the address constant. This is illustrated in Figure 5.14. The TXT card for relative locations 32-35 contains a -3, the only part of the address constant that can be calculated by the assembler. The last two RLD cards tell the loader to add the value of ID 03, which is PG1ENT2, to locations 32-35 and then subtract the value of ID 02, which is PG1ENT1. When processed by the loader, this address constant in PG2 will indeed have the same value as the one in PG1.

Since the direct-linking loader may encounter external references in an object deck which cannot be evaluated until a later object deck is processed, this type of loader requires two passes. Their functions are very similar to those of the two passes of an assembler. The major function of pass 1 of a direct-linking loader is to allocate and assign each program a location in core and create a

ESD cards

Source card reference	Name	Type	ID	Relative address	Length
1	PG1	SD	01	0	60
2	PG1ENT1	LD	—	20	—
2	PG1ENT2	LD	—	30	—
3	PG2	ER	02	—	—
3	PG2ENT1	ER	03	—	—

TXT cards
(only the interesting ones, i.e. those involving address constants)

Source card reference	Relative address	Contents	Comments
6	40-43	20	
7	44-47	45	= 30 + 15
8	48-51	7	= 30-20-3
9	52-55	0	unknown to PG1
10	56-59	-16	= -20 + 4

RLD cards

Source card reference	ESD ID	Length (bytes)	Flag + or −	Relative address
6	01	4	+	40
7	01	4	+	44
9	02	4	+	52
10	03	4	+	56
10	02	4	+	56
10	01	4	−	56

FIGURE 5.13 Object deck program PG1

symbol table filling in the values of the external symbols. The major function of pass 2 is to load the actual program text and perform the relocation modification of any address constants needing to be altered.

The first pass allocates and assigns storage locations to all segments and stores the values of all external symbols in a symbol table. These external symbols appear as local definitions on the ESD cards of another assembled program. For every external reference symbol there must be a corresponding internal symbol in some other program. The loader inserts the absolute address of all of these external symbols in the symbol table. In the second pass, the loader places the text into the assigned locations and performs the relocation task, modifying relocatable constants. Figure 5.16 depicts the interplay between the passes of a loader in a direct-linking loading scheme.

ESD cards

Source card reference	Name	Type	ID	ADDR	Length
12	PG2	SD	01	0	36
13	PG2ENT1	LD	—	16	—
14	PG1ENT1	ER	02	—	—
14	PG1ENT2	ER	03	—	—

TXT cards
(only the interesting ones)

Source card reference	Relative address	Contents
16	24-27	0
17	28-31	15
18	32-25	-3

RLD cards

Source card reference	ESD ID	Length flag (bytes)	Flag + or −	Relative address
16	02	4	+	24
17	03	4	+	28
18	03	4	+	32
18	02	4	−	32

FIGURE 5.14 Object deck program PG2

5.3.2 Specification of Data Structures

The next step in our design procedure is to identify the data bases required by each pass of the loader.

Pass 1 data bases:

1. Input object decks.
2. A parameter, the Initial Program Load Address (IPLA) supplied by the programmer or the operating system, that specifies the address to load the first segment.
3. A Program Load Address (PLA) counter, used to keep track of each segment's assigned location.
4. A table, the Global External Symbol Table (GEST), that is used to store each external symbol and its corresponding assigned core address.
5. A copy of the input to be used later by pass 2. This may be stored on an auxiliary storage device, such as magnetic tape, disk, or drum, or the original object decks may be reread by the loader a second time for pass 2.
6. A printed listing, the *load map*, that specifies each external symbol and its assigned value.

State the various steps for loader design to identify the data bases required by each pass

Assume:

PG1 loaded at location 104
PG2 loaded at location 168

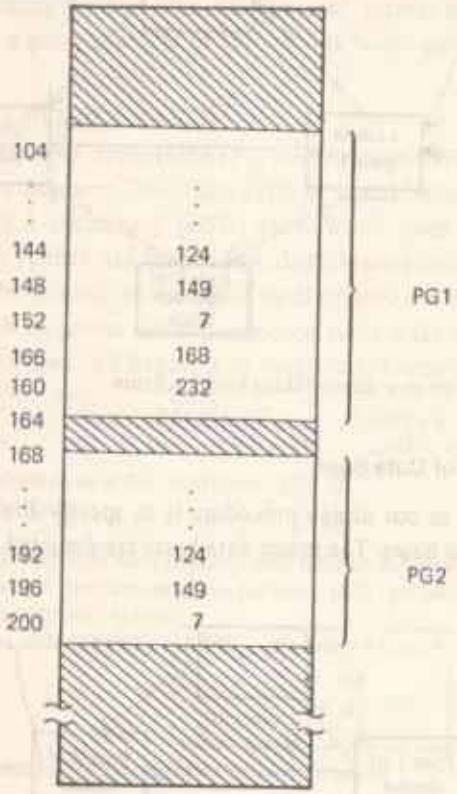


FIGURE 5.15 Main storage after loading programs PG1 and PG2

Pass 2 data bases:

1. Copy of object programs inputted to pass 1
2. The Initial Program Load Address parameter (IPLA)
3. The Program Load Address counter (PLA)
4. The Global External Symbol Table (GEST), prepared by pass 1, containing each external symbol and its corresponding absolute address value
5. An array, the Local External Symbol Array (LESA), which is used to establish a correspondence between the ESD ID numbers, used on ESD and RLD cards, and the corresponding external symbol's absolute address value

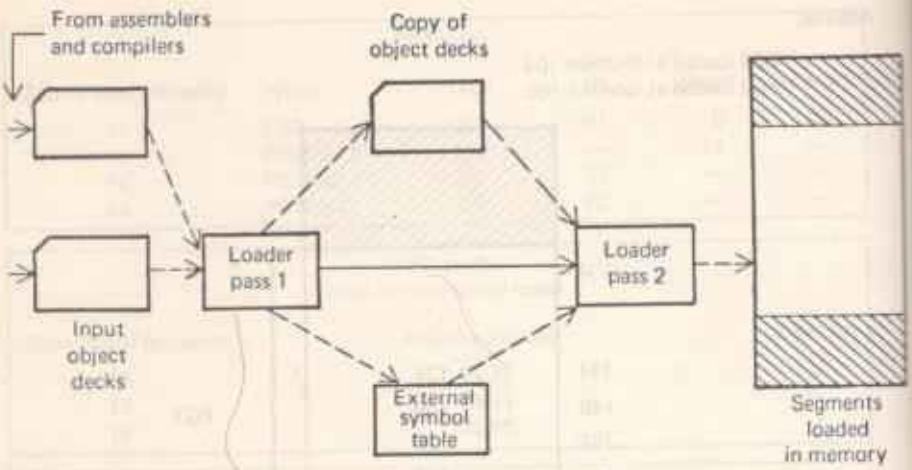


FIGURE 5.16 Two pass direct-linking loader scheme

Explain the format & contents of each of the data bases in the loader design

5.3.3 Format of Data Bases

The third step in our design procedure is to specify the format and content of each of the data bases. The major data bases are depicted in Figure 5.17.

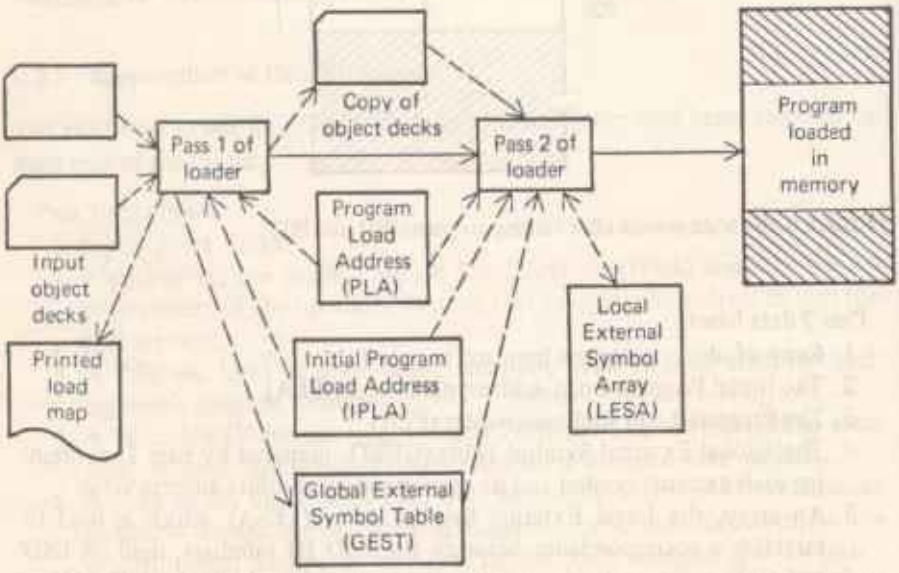


FIGURE 5.17 Use of data bases by loader passes

OBJECT DECK

The object deck has been discussed several times; Figures 5.18, 5.19, 5.20 and 5.21, depict in detail the actual card deck format that is used by various IBM 370 or 360 direct-linking loaders. The specific card format is not crucial, but these figures present a good example of the various techniques used to encode information.

GLOBAL EXTERNAL SYMBOL TABLE

The Global External Symbol Table (GEST) is used to store the external symbols defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) card. When these symbols are encountered during pass 1, they are assigned an absolute core address; this address is stored, along with the symbol, in the GEST as illustrated in Figure 5.22.

The reader may wish to review the discussion on symbol tables and searching/sorting techniques presented in Chapter 3 in conjunction with the design of an

Columns	*Contents*
1	Hexadecimal byte X'02' (card punch 12-2-9)
2-4	Characters ESD
5-14	Blank
15-16	ESD identifier (ID) for program name (SD) or external symbol (ER) (IDs unique within program) or blank for entry (LD), see box below
17-24	Name, padded with blanks
25	ESD type code (TYPE)
26-28	Relative address or blank (ADDR), see box below
29	Blank
30-32	Length of program otherwise blank (LENGTH)
33-72	Blank
73-80	Card sequence number

TYPE	*Hexadecimal code*
SD	01
LD	02
ER	03

ESD forms and conventions

	Type	*ID*	*ADDR*	*Length*
Program name (segment definition)	SD	01	Zero	Length of program
Entry (local definition)	LD	—	Relative address	—
External reference	ER	Unique number	—	—

(The unique ID numbers are usually assigned sequentially.)

FIGURE 5.18 ESD card format

Columns	*Contents*
1	Hexadecimal byte X'02'
2-4	Characters TXT
5	Blank
6-8	Relative address of first data byte (ADDR)
9-10	Blank
11-12	Byte Count (BC) = number of bytes of information in cc. 17-72
13-16	Blank
17-72	From 1 to 56 data bytes (instructions and "data" look the same)
73-80	Card sequence number

FIGURE 5.19 TXT card format

Columns	*Contents*
1	Hexadecimal byte X'02'
2-4	Characters RLD
5-18	Blank
19-20	ID corresponding to a number assigned to SD or ER on ESD card
21	Flag byte (see box below)
22-24	Relative address of first byte of address constant (ADDR)
25-72	Blank
73-80	Card sequence number

Flag byte conventions

bits	
0-3	Not used
4-5	Length (in bytes) of address constant 00 = one byte 01 = two bytes 10 = three bytes 11 = four bytes
6	0 means add ESD address to address constant 1 means subtract ESD address from address constant

FIGURE 5.20 RLD card format

Columns	*Contents*
1	Hexadecimal byte X'02'
2-4	Characters END
5	Blank
6-8	Start of execution entry (ADDR), if other than beginning of program (specified on assembly END card)
9-72	Blank
73-80	Card sequence number

FIGURE 5.21 END card format

← 12 bytes per entry →

External symbol (8-bytes) (characters)	Assigned core address (4-bytes) (decimal)
"PG1*bbbbb*"	104
"PG1ENT1*b*"	124
"PG1ENT2*b*"	134
"PG2*bbbbb*"	168
"PG2ENT1*b*"	184

Notation: This sample GEST content is based upon the example in Figures 5.12 and 5.15.

FIGURE 5.22 Global External Symbol Table (GEST) format

assembler. The GEST has the same general use and characteristics as the assembler's Symbol Table.

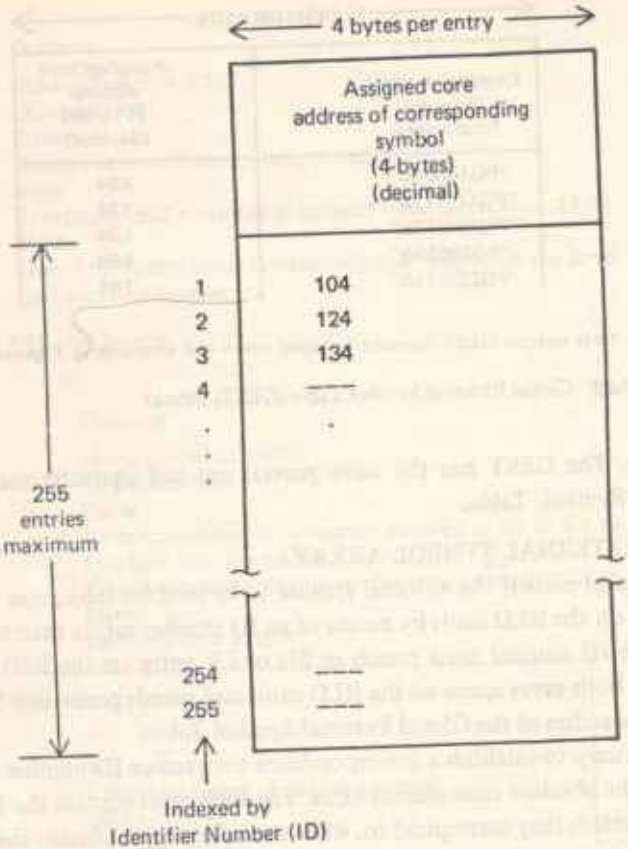
LOCAL EXTERNAL SYMBOL ARRAY

As mentioned earlier, the external symbol to be used for relocation or linking is identified on the RLD cards by means of an ID number rather than the symbol's name. The ID number must match an SD or ER entry on the ESD cards. This technique both saves space on the RLD cards and speeds processing by eliminating many searches of the Global External Symbol Table.

It is necessary to establish a correspondence between an ID number on an RLD card and the absolute core address value. The ESD cards contain the ID numbers and the symbols they correspond to, while the information relating these symbols to absolute core address values may be found in the GEST. In pass 2 of the loader the GEST and ESD information for each individual object deck is merged to produce the local external symbol array that directly relates ID number and value. In principle it is necessary to create a separate LESA for each segment, but since the LESAs are only produced one at a time, the same array can be reused for each segment. Figure 5.23 depicts the format of the Local External Symbol Array (LESA). Note that unlike the case with the GEST, it is not necessary to *search* the LESA; given an ID number, the corresponding value is written as LESA(ID) and can be immediately obtained.

5.3.4 Algorithm

The following two flowcharts (Figs. 5.24 and 5.25) describe an algorithm for a direct-linking loader for an IBM System/360-type computer While they illustrate



Notation: This sample LESA contents is based upon the PG1 segment as presented in Figures 5.12, 5.13, and 5.15.

FIGURE 5.23 Local External Symbol Array (LESA) format

most of the logical processes involved, these flowcharts are still a simplification of the operations performed in a complex loader. In particular, many "special" features, such as COMMON segments, library processing, dynamic loading and dynamic linking, are not explicitly included (many of them are discussed in the problems section and in later chapters).

Pass 1 – allocate segments and define symbols The purpose of the first pass is to assign a location to each segment, and thus to define the values of all external symbols. Since we wish to minimize the amount of core storage required for the total program, we will assign each segment the next available location after the

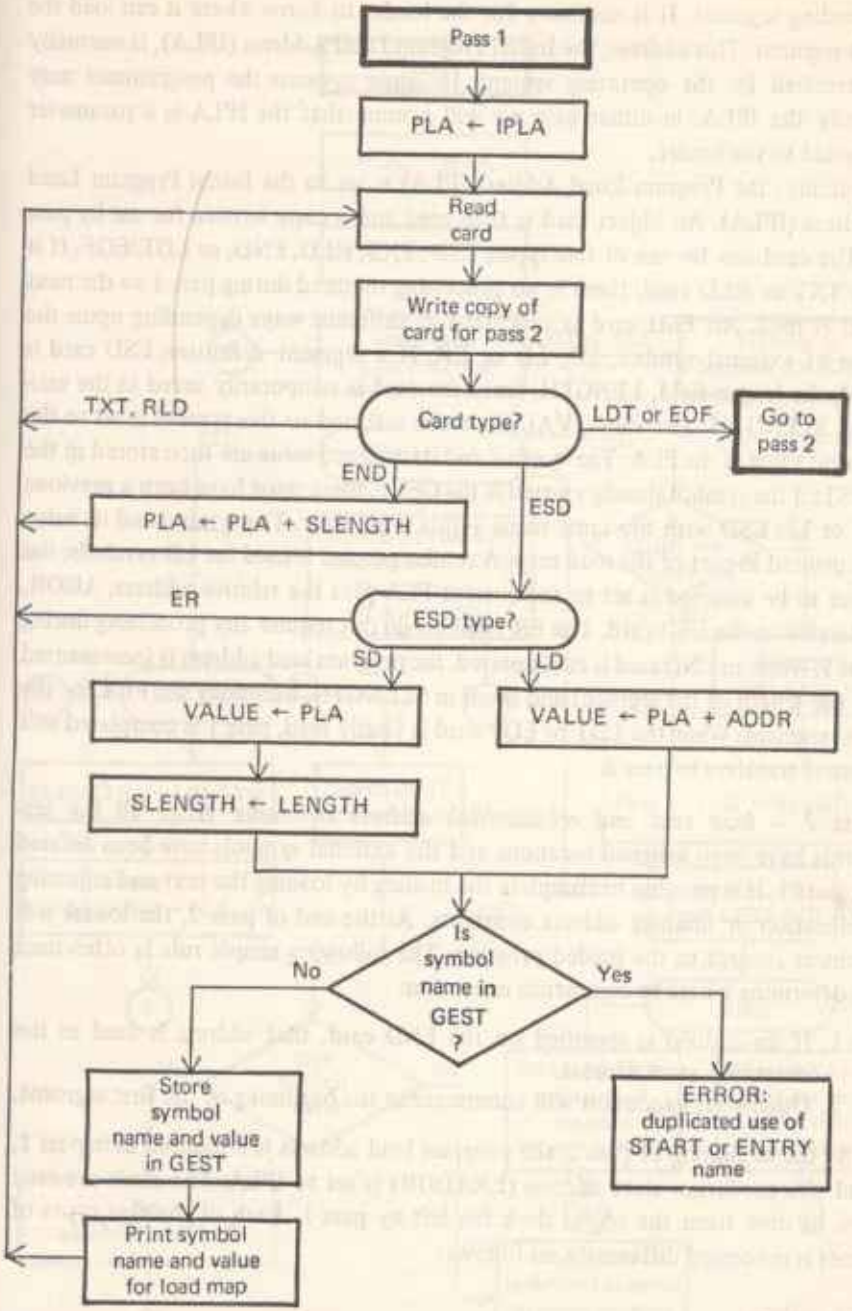


FIGURE 5.24 Detailed pass 1 flowchart

preceding segment. It is necessary for the loader to know where it can load the first segment. This address, the Initial Program Load Address (IPLA), is normally determined by the operating system. In some systems the programmer may specify the IPLA; in either case we will assume that the IPLA is a parameter supplied to the loader.

Initially, the Program Load Address (PLA) is set to the Initial Program Load Address (IPLA). An object card is then read and a copy written for use by pass 2. The card can be one of five types, ESD, TXT, RLD, END, or LDT/EOF. If it is a TXT or RLD card, there is no processing required during pass 1 so the next card is read. An ESD card is processed in different ways depending upon the type of external symbol, SD, LD or ER. If a segment definition ESD card is read, the length field, LENGTH, from the card is temporarily saved in the variable, SLENGTH. The value, VALUE, to be assigned to this symbol is set to the current value of the PLA. The symbol and its assigned value are then stored in the GEST; if the symbol already existed in the GEST, there must have been a previous SD or LD ESD with the same name – this is an error. The symbol and its value are printed as part of the load map. A similar process is used for LD symbols; the value to be assigned is set to the current PLA plus the relative address, ADDR, indicated on the ESD card. The ER symbols do not require any processing during pass 1. When an END card is encountered, the program load address is incremented by the length of the segment and saved in SLENGTH, becoming the PLA for the next segment. When the LDT or EOF card is finally read, pass 1 is completed and control transfers to pass 2.

Pass 2 – load text and relocate/link address constants After all the segments have been assigned locations and the external symbols have been defined by pass 1, it is possible to complete the loading by loading the text and adjusting (relocation or linking) address constants. At the end of pass 2, the loader will transfer control to the loaded program. The following simple rule is often used to determine where to commence execution:

1. If an address is specified on the END card, that address is used as the execution start address.
2. Otherwise, execution will commence at the beginning of the first segment.

At the beginning of pass 2 the program load address is initialized as in pass 1, and the execution start address (EXADDR) is set to IPLA. The cards are read one by one from the object deck file left by pass 1. Each of the five types of cards is processed differently, as follows:

ESD CARD

Each of the ESD card types is processed differently.

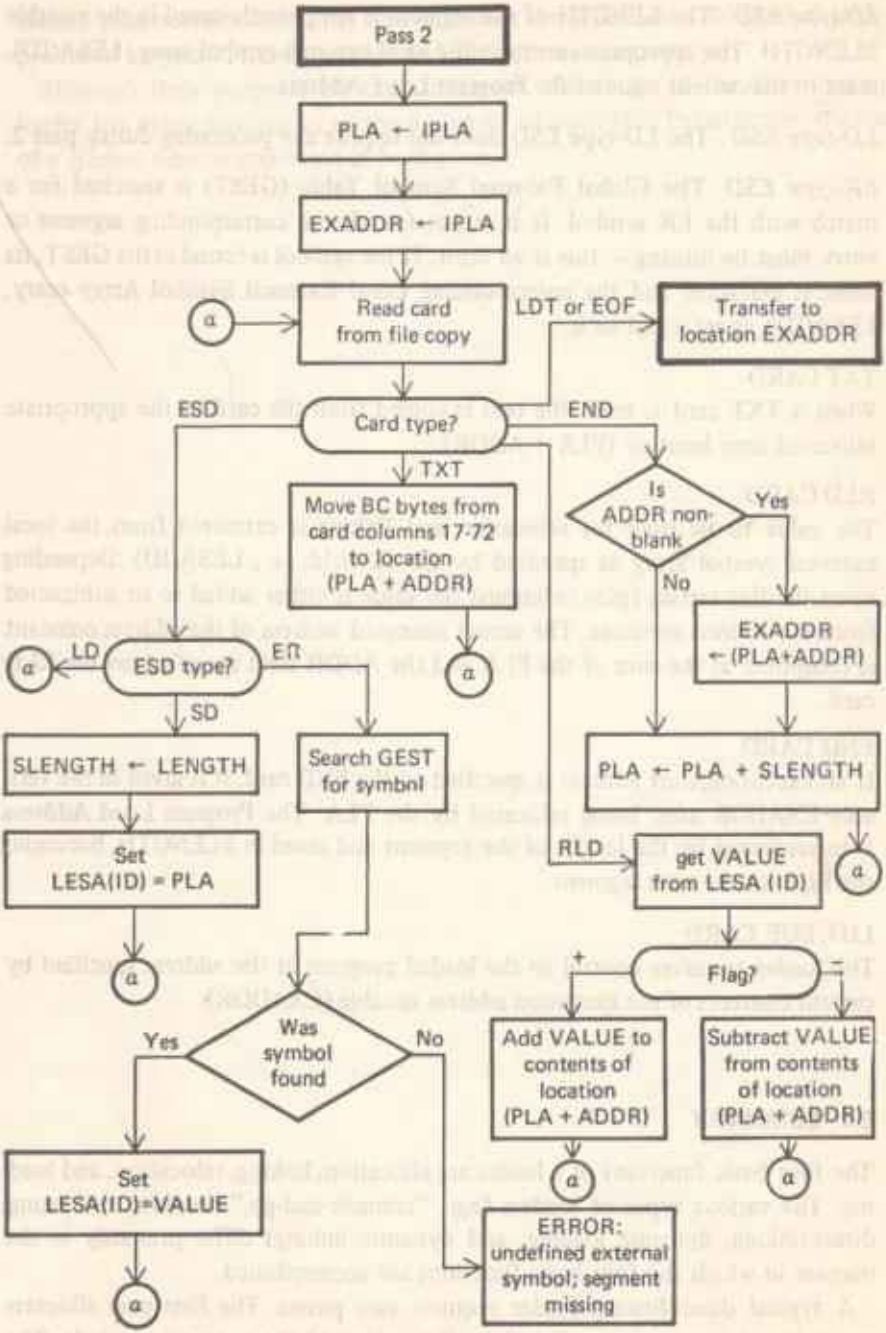


FIGURE 5.25 Detailed pass 2 flowchart

SD-type ESD The LENGTH of the segment is temporarily saved in the variable SLENGTH. The appropriate entry in the local external symbol array, LESA(ID), is set to the current value of the Program Load Address.

LD-type ESD The LD-type ESD does not require any processing during pass 2.

ER-type ESD The Global External Symbol Table (GEST) is searched for a match with the ER symbol. If it is not found, the corresponding segment or entry must be missing – this is an error. If the symbol is found in the GEST, its value is extracted and the corresponding Local External Symbol Array entry, LESA(ID), is set equal to it.

TXT CARD
When a TXT card is read, the text is copied from the card to the appropriate relocated core location (PLA + ADDR).

RLD CARD
The value to be used for relocation and linking is extracted from the local external symbol array as specified by the ID field, i.e., LESA(ID). Depending upon the flag setting (plus or minus) the value is either added to or subtracted from the address constant. The actual relocated address of the address constant is computed as the sum of the PLA and the ADDR field specified on the RLD card.

END CARD
If an execution start address is specified on the END card, it is saved in the variable EXADDR after being relocated by the PLA. The Program Load Address is incremented by the length of the segment and saved in SLENGTH, becoming the PLA for the next segment.

LDT/EOF CARD
The loader transfers control to the loaded program at the address specified by current contents of the execution address variable (EXADDR).

5.4 SUMMARY

The four basic functions of a loader are allocation, linking, relocation, and loading. The various types of loaders (e.g., "compile-and-go," absolute, relocating, direct-linking, dynamic loading, and dynamic linking) differ primarily in the manner in which the four basic functions are accomplished.

A typical direct-linking loader requires two passes. The first pass allocates space for the segments and defines the values of the external symbols. The

second pass actually loads the text and uses the global external symbol table, produced by pass 1, to relocate and link the address constants.

Although their purposes are quite different, the design of the direct-linking loader has many similarities to the design of an assembler. In particular, the use of a symbol table is important in both cases.