# System / 360–370 Assembler Language (OS)

Kevin McQuillen

# System/360-370 Assembler Language (OS)

Kevin McQuillen

Edited by Mike Murach

**Mike Murach & Associates, Inc.**

4222 West Alamos, Suite 101
Fresno, California 93711

Library of Congress Catalog Card Number:
74-29645

Thanks to IBM for permission to reprint or
adapt the materials listed.

ISBN 0-911625-02-X

20 19 18 17 16 15 14 13 12 11

# System/360-370 Assembler Language (OS)

# Preface for Instructors

*System/360–370 Assembler Language (OS)* has been designed for use in both introductory and advanced courses. Specifically, this book teaches System/360–370 assembler language for the OS and OS/VS operating system. In chapters 1 through 5, I believe this book provides the most effective introduction to assembler-language programming that is currently available. In chapters 6 through 16, I believe this book provides the most effective advanced material on assembler-language programming that is on the market today.

Because most students taking this course will have taken a previous course in data processing or programming, or will have had equivalent experience in the field, it is assumed that the readers have certain skills. In particular, the readers should be able to do the following:

1 Describe the components of a typical card, tape, or direct-access system.

2 Explain what is meant by the term *continuous form*.

3 Decode the data in an uninterpreted punched card.

4 Describe how data is keypunched into a standard punched card.

5 Explain what is meant by *loading a program*.

6 Describe blocked tape records.

Since this is a very limited set of requirements, it is possible to use this book for a first course in computing by giving the class a computer and keypunching demonstration plus an introductory lecture that provides the background material indicated above. Similarly, if the courses previously taken by some students haven't covered all the material required by these prerequisites, the omitted material can be covered in a single lecture or demonstration. If all six prerequisites should have been met in a previous course, you may want to give a pretest on the first day of class to make sure that they actually have been met; you can then review accordingly.

This is the second book published by our company, and it was developed in a way that I think adds some much needed professionalism to the process of preparing instructional materials. To begin with, more than three months were spent in analyzing the possible subjects to be included in this book, selecting the actual content, and organizing that content based on a theory of instruction. Since few writers have the luxury of this much time for planning, one of the major shortcomings of most texts is the content selection and organization. Second, this book was written by a first-rate industry specialist, Kevin McQuillen, on a full time basis. In contrast, many books in the field are written on nights and weekends as secondary projects to some full time job. Finally, Kevin's manuscript was extensively rewritten and edited in an attempt to bridge the gap between professional and novice. I think the resulting product indicates that this method of preparing instructional materials is an effective one.

## BOOK FEATURES

### Content Selection

One important feature of this book is the breadth and usefulness of its content. This is true because the content was selected based on an analysis of the tasks done by a professional assembler-language programmer. For each task required of him, there is explanatory material in this book. As a result, I think you will find that this book is the most complete assembler-language book currently available. If you check the table of contents, you will find material on diagnostics, debugging, tape and disk concepts, operating systems, job-control language, and keypunching. Although all of these subjects are related to essential programming tasks, it is common for one or more of them to be omitted from assembler-language texts.

On the other hand, subjects that aren't related to the tasks of a professional programmer have been omitted from this text. This makes the book relevant. Although you may think that all programming books use task analysis as a guideline for selecting content, I can find numerous examples of irrelevant material in the sampling of assembler-language books that I have in my library. Multiplying binary numbers, decoding the PSW word, writing channel commands, coding privileged instructions, using physical IOCS, knowing the interrupt system—all are irrelevant to the assembler-language programmer but are included in one text or another. (Granted some of these are related to the tasks of a software specialist, but that isn't the context in which these subjects are presented.)

The only material in this book that isn't based on a strict task analysis is the topic on floating-point arithmetic and the topic on advanced macro writing. Although these topics are more relevant to the software specialist than to the assembler-language programmer, they have been included to give some important exposure. Otherwise, you can be sure that the coding and techniques illustrated in this book are also those found in industry. In contrast, a student who uses another text will all too often discover that the techniques illustrated are not those of the real world.

### Modular Organization

A second feature of this book is its organization, sometimes referred to as "modular" organization. After reading the first five chapters of the text, the student can continue with any of its other parts, or modules. In particular, the book is organized as follows:

| Part | Chapters | Part Title | Prerequisite Parts | Design |
|---|---|---|---|---|
| 1 | 1–2 | Required Background | – | Sequential |
| 2 | 3–5 | Assembler Language: The Core Content | 1 | Sequential |
| 3 | 6–11 | Advanced Assembler-Language Subjects | 1,2 | Random |
| 4 | 12–15 | Tape and Direct-Access Programming | 1,2 | Sequential |
| 5 | 16 | The Operating System | 1,2 | |

This means that the chapters in part 4 should be studied in sequence, but the chapters in part 3 can be studied in any order. Within the few limitations indicated in this table, it is you who will determine the sequence of instruction and the material to be emphasized. Similarly, the student is free to jump to a topic that interests him without fear of missing related background material. In short, the course can be teacher-directed or student-directed, but it will not be textbook-directed.

In addition to the teaching flexibility that modular organization gives, there is an important educational reason for organizing a book in this way. Briefly stated, this organization forces the author to present the essence, or "core content," of the subject in just a few chapters early in the book. This in turn means that the student is shown all the important relationships between the elements of the subject early in the course. Because one of the major problems of learning is the failure to see the relationships between the parts, the emphasis on core content makes learning more efficient.

I might add that although many books are advertised as modular, few actually are. To be truly modular, the essence, totality, or core content of the subject (call it what you will) must be presented early in the course and all subsequent modules must need only this core content as prerequisite material. When a book is designed in this way, its format proceeds from a theory of education rather than from marketing considerations.

## Educational Methodology

I think the primary feature of this book is that it works—you can actually learn how to program in assembler language from it. I know because I learned DOS assembler language from Kevin's raw manuscript without any outside help of any kind. Since that time, we've refined the manuscript considerably and have made it more effective didactically.

As I see it, there are two main reasons for this book's effectiveness. One of these reasons is its modular approach. In general, there are two basic approaches to teaching assembler language—the parts-to-the-whole method and the modular method. The first teaches the separate elements of the language until a great deal of detail has been covered and only at that time are a few of these elements put together in a complete program. Using this approach, it isn't uncommon for the first complete program to be presented in the second half of the book, and sometimes the first complete program is presented very late in the book. For instance, one of the leading texts presents its first complete program on page 285, while another presents its first complete program in chapter 14 of a 15-chapter book.

The problem with this parts-to-the-whole method of teaching is twofold. First, a student doesn't have the perspective to appreciate the relationships between the parts until he is familiar with a complete program. As a result, he learns the parts through memory rather than through some underlying structure or concept. Second, from a point of view of classroom teaching, this method is impractical. Normally, you must wait at least several weeks before a student has learned enough of the parts in order to be able to write a complete program. In the meantime, motivation dwindles, and what should be an exciting problem-solving class becomes a frustrating struggle to learn the massive amount of detail associated with the language. Also, if the assignment of computer laboratory time is a fixed number of hours per week from the start of the course, the instructor usually has to create supplementary material so students can run programs, or segments of programs, in the early weeks of the course.

As indicated earlier, this book uses the second (modular) approach to the teaching of assembler language. After some background material is provided in chapters 1 and 2 (some of which may be review), topic 1 of chapter 3 presents a complete program including card input, printer output, data movement, editing, arithmetic, and logic. As soon as this basic program is understood, the student can begin to write significant programs of his own. Before chapter 3 is completed, though, two refinements of this first program and two additional programs are presented so a total of five complete programs are shown in chapter 3. By this time, a full subset of the language has been presented, and the student is prepared to do independent work in a computer lab.

Following the subset presented in chapter 3, chapter 4 offers a definitive presentation on correcting diagnostics, preparing test data, and debugging programs, including the analysis of core dumps. Since these skills are essential to assembler-language programming and to successful lab work, it is amazing that they are often treated so lightly in other texts. To complete the core content, chapter 5 presents a collection of elements and techniques that expands the subset and makes the book truly modular.

Once a student has satisfactorily completed part 2—in particular, chapters 3 and 4—the most demanding part of the instructor's job is finished. With an understanding of the structure of the language and all the related skills for coding, testing, and debugging a program, learning other assembler-language elements and techniques becomes part of a pattern. If a student can see the need for an element or technique and can see how that element or technique relates to the whole task of programming, mastering the material is a manageable task.

The second reason for the book's didactic effectiveness is its illustrative material. Although many authors rely heavily on verbal description, we know from experience that a programming language cannot be learned without extensive illustrative material. In fact, the illustrative material is far more important than the descriptive material. For this reason, this book has a carefully planned sequence of program listings. In all, there are 26 complete program listings, about 45 self-contained segments of coding, several diagnostic and debugging listings, and numerous supporting examples. As soon as I saw the illustrative material that Kevin had selected for this book, I knew the product would be effective.

## Apparatus by Topic

Because learning depends on what the student does, not upon what he sees or hears, each topic is followed by terminology lists, behavioral objectives, and, whenever relevant, problems and their solutions. The terminology lists are listings of the new words presented in the text. The intent is for the student to scan the list to check his comprehension of the terminology. If he understands the words, he can proceed. If he feels that he doesn't have a clear understanding of a term, he can reread applicable sections or note the term so he can question its meaning in class. In any case, the intent of the list is for use as a quick review; a student shouldn't be expected to write definitions of the words.

Following the terminology lists are behavioral objectives that describe the activities a student should be able to do upon completion of the topic. The intent of these objectives is to give the student a clear picture of what his learning goals should be. Since this book deals with programming, the primary objectives have to do with solving various types of programming problems using assembler language. In addition, there are objectives that deal with related skills such as reading core dumps, describing a type of file organization, choosing blocking factors for disk files, and preparing OS job-control cards. Although some students will ignore the objectives, others will be more efficient learners because of them.

Although some instructors seem to feel that preparing and using objectives is busywork, I would like to see them in all textbooks. At the least, listing objectives would force the author to focus more clearly on what he is trying to accomplish. Without objectives, I think an author all too often concerns himself with writing a definitive work rather than concentrating on the goals of education.

At any rate, I believe objectives can contribute significantly to the success of a course. If students are convinced that the objective lists describe *all* activities that they will be expected to perform, their learning will become much more directed. In each class I've taught, I have found students who wouldn't rest until they felt they could satisfy all of the course objectives. Because some students find it hard to believe an instructor is telling them everything they will be required to do, it is important for you to refer to the objectives as you review a topic in class. If the objectives are made prominent in all classroom activity, I am convinced that teaching has a greater likelihood of success.

Since it is unlikely that two people will agree on a list of objectives for a course, you will probably want to modify the objectives to suit your class. If, for example, you are using the text for a computer science course, you may want to add objectives that emphasize software development. The objective lists, then, are only a starting point. However, if only the objectives given in the text are fulfilled, I would say that you have taught a highly successful course.

When the objectives deal with problem solving, they are followed by problems and their solutions. These problems are intended to give practice in the skills described by the objectives. As much as possible, these problems have been designed to show how the elements and techniques described in the text are used in a different context. There are no multiple-choice, true/false, matching, or fill-in questions, because those types of activities have nothing to do with the important objectives of a programming course.

So there is immediacy to the problem-solving activity, solutions are presented immediately after the problems. This has the advantage of letting a student know that he is right when he is right, and just as important, of letting him know right away when he's wrong. For those students who wouldn't otherwise know how to get started in solving a problem, the solutions are an essential part of the learning process. Although compiling and testing programs on a computer system has the same effect as doing the problems and checking the solutions, studying the solutions of a professional can correct many false notions and bad habits before problems are actually tried on a computer system. Because of the expense of computer time, this is a practical consideration.

What about students who don't actually do the problems but look to the solutions? The experience is still valuable. Although the best way to learn is to actually do the problems and then compare the answers with the solutions provided, it may not always be the most efficient way of learning—particularly, for the brightest students or for those with experience in another language. In the interest of expediency, then, a student may read a problem, conceive a solution, and compare this conception with the actual solution. The important thing is that assembler language be viewed in the context of its application. Looking at the problems in this way, they can be seen simply as a means of presenting additional assembler-language applications.

### Lab Problems

Appendix F presents a progression of programming problems. Since these problems include test data listings, they are ideal for lab problems. In addition, they can be used for classroom exercises or tests. If a student can write programs for all of the types of problems given, I feel sure he is well qualified to become an entry-level programmer in industry.

## TEACHING NOTES

Because there is no instructor's guide for this text, we have tried to make the text itself as complete—both for teacher and student—as is practical. For this reason, the following teaching notes are included in this preface.

1 If students taking this course have a strong background in data processing or programming, it is possible that they will have already mastered the material in chapter

1 (introductory concepts), chapter 12 (tape concepts), and chapter 13 (direct-access concepts). For this reason, you may want to give a pretest for these chapters based on the objectives at the ends of the topics. To a lesser extent, the students may also be familiar with the material of chapter 2 (CPU concepts) and chapter 16 (OS and JCL). Here again, a pretest can determine which students need to master which objectives.

2 If your students have keypunched source and JCL decks before, appendix A should give them all the information they need for keypunching assembler-language decks. Otherwise, a keypunching demonstration that includes the mounting of a program card on a program drum should be given.

3 Because it can make the running of lab problems more efficient, I recommend that tape or sequential-disk programming be taught immediately after the core content. Then, a card-to-tape or card-to-disk utility can be used to store the test decks in tape or sequential-disk files, and the problems for chapters 6 through 12 can be run as tape-to-printer or disk-to-printer programs rather than card-to-printer programs. In general, this presents no conceptual problems for the students, but it can greatly improve computer efficiency.

4 Although the material in this text is self-sufficient, there are several references to IBM manuals that provide additional information. As a result, one or more copies of the relevant manuals should be available to the members of the class. These manuals are listed at the end of the introduction.

5 Because of its modular organization, didactic approach, illustrative material, and apparatus, I believe some new solutions to old teaching problems are possible when this text is used. Perhaps the most significant teaching problem encountered in a programming course is the range of aptitudes of the students. For instance, some students will grasp the material by merely reading the text, some will require minor assistance in addition to reading the text, some will require extensive help in the form of lecture and discussion, and some just shouldn't be taking the course. Because this text gets into the problem-solving aspects of the subject in chapters 2 and 3, the aptitudes of your students should be apparent early in the course, in time for effective counseling. Then, the brightest students can be assigned lab problems and be allowed to work independently; others can be assigned less demanding problems and given periodic assistance; and the marginal students can be given full assistance and supervision.

## CONCLUSION

In conclusion, I'd like to say that we have tried very hard to make this text as effective as possible. Nevertheless, I know that we have much to learn about preparing instructional materials. To this end, we intend to do controlled tests of this product in order to see at what points in this text learning problems develop. Maybe then we can improve our methodology in future editions and in future products. We also welcome your comments, criticisms, suggestions, or questions.

*Mike Murach*
*Fresno, California*

The text of this book is set in 10 point Zenith with heads set in Galaxy Medium by Applied Typographic Systems of Mountain View, California.

Pat Rogondino, of Palo Alto, California, prepared the illustrations. Tom Tracy, of Oakland, California, took the photographs for the part openers and cover.

Michael Rogondino of Rogondino & Associates, Palo Alto, California, did the book design and the cover design. He also guided the book's production.

# Contents

Preface for Instructors vii
Introduction 1

PART 1 Required Background

Chapter 1 Preliminary Concepts and Terminology 7
Topic One System/360-370 7
Topic Two Writing a Program in System/360-370 Assembler Language 10
Topic Three Introduction to OS and Job-Control Language 22

Chapter 2 System/360-370 CPU Concepts 29
Topic One Data and Instructions 29
Topic Two Overlap and I/O Instructions 49

PART 2 Assembler Language: The Core Content

Chapter 3 A Subset of Assembler Language 55
Topic One An Introduction to Assembler Language 55
Topic Two Refining the Reorder-Listing Program 77
Topic Three Completing the Basic Subset 87

Chapter 4 Diagnostics and Debugging 97
Topic One Desk Checking and Diagnostics 97
Topic Two Testing and Debugging 112

Chapter 5 Expanding the Basic Subset 141
Topic One Register Operations 141
Topic Two Storage Definition Techniques 149

PART 3 Advanced Assembler Language Subjects

Chapter 6 Binary Arithmetic 161
Topic One Fixed-Point Arithmetic 162
Topic Two Floating-Point Arithmetic 165

Chapter 7 Table Handling 173
Topic One Single-Level Tables 173
Topic Two Multilevel Tables 181

Chapter 8 Editing, Bit Manipulation, and Translation 185

Chapter 9 Subroutines and Subprograms 199
Topic One Subroutines 199
Topic Two Subprograms 213

Chapter 10 Writing Macro Definitions 225
Topic One Basic Macro Writing 226
Topic Two Advanced Macro Writing 232

Chapter 11 Useful Standard Macros and Assembler Commands 247
Topic One Standard Macros 247
Topic Two Assembler Commands 254

PART 4 Tape and Direct-Access Programming

Chapter 12 Magnetic Tape Concepts 261
Topic One Tape Characteristics 261
Topic Two Programming Considerations 266

Chapter 13 Direct-Access Concepts 273
Topic One Direct-Access Devices 274
Topic Two File Organization 280
Topic Three Programming Considerations 290

Chapter 14 BAL for Sequential Files 297
Topic One Fixed-Length Records 298
Topic Two Variable-Length Records 314

Chapter 15 BAL Coding for Indexed Sequential and Direct Files 323
Topic One Indexed Sequential Access Method 323
Topic Two Direct Access Method 348

PART 5 The Operating System

Chapter 16 The Operating System and Its Job-Control Language 367
Topic One The Operating System 367
Topic Two OS Job-Control Language 380

PART 6 Appendixes

Appendix A Procedures for Keypunching Source Decks and JCL Cards 413

Appendix B BAL Instruction Summary 417

Appendix C Assembler Command Summary 422

Appendix D Common DCB Operands for QSAM 424

Appendix E System/370 Instructions 425

Appendix F Problems for Computer Lab 429

# Introduction

In the mid-1960s, computer industry experts began to predict that the use of assembler language would decline and die within ten years or so. Today, various experts continue to predict the replacement of assembler languages by various software or hardware developments. In fact, however, assembler language is the second most widely used programming language for business applications—second only to COBOL. Furthermore, the use of assembler language does not appear to have declined in the last three years.

Because of this widespread use, a professional programmer for a medium or large sized computer system is almost certain to come in contact with assembler language at some time in his career. But that's only one of the reasons for studying assembler language. In addition, the ability to write and debug assembler-language programs allows you to write routines that cannot be written in high-level languages, helps you write more efficient high-level language programs, and gives you the tools with which to debug sophisticated problems that may result from using high-level-language code. In short, a high-level-language programmer isn't in complete control of a computer system unless he knows assembler language.

In case you don't know it, there are many different assembler languages. For instance, each type of computer system has its own assembler language. In addition, there may be more than one version of assembler language for a single computer. For the IBM System/360-370, there are two main versions of assembler language. They can be referred to as the DOS and OS assembler languages. This book teaches OS assembler language.

With few exceptions, the content of this book has been selected based on frequency of use in industry. As a result, you can feel sure that this book is a true representation of what you'll find in the business world. However, because of the range of this book—it covers everything from elementary coding to advanced table handling, translating, macro writing, and disk file handling—much of the material presented here is not known or used by the average professional programmer. For this reason, I feel confident that you will have more than met the requirements of a starting programmer in industry if you are able to apply all of the material in this book to business programming problems.

Before you actually begin to use this book, there are several things you ought to know about it.

1 In most cases, a student taking this course will have taken an introductory course or will have programmed in another language. As a result, this book assumes that you can do the following:
   a. Describe the components of a typical card, tape, or direct-access system.
   b. Explain what is meant by the term *continuous form.*
   c. Decode the data in an uninterpreted punched card.
   d. Describe how data is keypunched into a standard punched card.
   e. Explain what is meant by *loading a program.*
   f. Describe blocked tape records.

   Since this is a very limited set of requirements, your background is probably much stronger than required. If so, the material will be that much easier for you. In particular, chapters 1, 12, and 13 are likely to be mainly review for you, and chapters 2 and 16 are likely to be advanced versions of material you have already been introduced to.

2 This book is not designed so that you must read its sixteen chapters in sequence. Instead, its chapters are divided into five parts as indicated by this table:

| Part | Chapters | Title | Prerequisite Parts | Design |
|---|---|---|---|---|
| 1 | 1–2 | Required Background | – | Sequential |
| 2 | 3–5 | Assembler Language: The Core Content | 1 | Sequential |
| 3 | 6–11 | Advanced Assembler-Language Subjects | 1,2 | Random |
| 4 | 12–15 | Tape and Direct-Access Programming | 1,2 | Sequential |
| 5 | 16 | The Operating System | 1,2 | |

   This means that after completing the first two parts, you can continue with any other part of the book. In part 4 and 5 the chapters should be read in sequence, but part 3 is designed so the chapters can be read in any order you choose.

   The advantage of this type of organization is that you can read the parts and chapters in the sequence that interests you, not in the sequence the author thinks is best. If, for example, you are interested in writing a file-handling program for a direct-access file, you can skip to part 4 immediately after completing part 2. Similarly, a COBOL programmer who wants to know how to write an assembler-language subprogram might skip to chapter 9 immediately after part 2.

   From an educational point of view, this method of organization is effective because it gives you an honest representation of the language early in the course. By the time you complete chapter 3, you will know how to code complete assembler-language programs for card input and printer output. Then chapter 4 shows you how to prepare and debug programs, and chapter 5 presents some professional coding techniques. As a result, after completing part 2 you will have a good appreciation for the nature of assembler language as well as a good idea of your ability to master the language.

If you are studying this language on your own and want a recommended sequence of study, I recommend the following:

| | |
|---|---|
| Chapters 1, 2, 3, 4, 5 | The Core Content |
| Chapters 7, 8, 9, 11 | Selected Advanced Subjects |
| Chapters 12, 13, 14, 15 | Tape and Direct-Access Programming |
| Chapter 16 | OS and JCL for Tape Files |
| Chapter 10 | Macro Writing |
| Chapter 6 | Binary Arithmetic |

Don't feel, however, that you should rigidly adhere to this sequence. Whenever your interest in a subject is aroused, read the appropriate chapter. If, for example, a question arises about tape input and output when you are reading chapter 5, turn to part 4 next. There is no greater assurance that learning will take place than to study a subject in search of an answer.

3 At the end of each topic or chapter there are terminology lists of all the new terms. The intent of these lists is not that you be able to define the words in them, but that you feel you understand the words. After you read a topic, glance at the list and note any word whose meaning is unclear to you. Then, reread the related material. Once the words are fixed in your mind, continue.

4 Following the terminology lists for each topic or chapter there are one or more behavioral objectives. These objectives describe the activities (behavior) that you should be able to perform upon completion of a topic or chapter. The theory is that you will be a more effective learner if you know what you are expected to do and what you will be tested on. This contrasts with the traditional classroom treatment in which a student is forced to guess as to what he will be tested on.

In general, behavioral objectives can be divided into two classes: *knowledge objectives* and *application objectives*. A knowledge objective requires you to list, identify, describe, or explain aspects of a subject. For example, the first objective in chapter 1 is: Identify the model number and I/O device numbers of the System/360-370 that you will be writing programs for. Once you have been told what these numbers are, you should have no trouble fulfilling this objective. Although other knowledge objectives will be more involved and more difficult than this one, given the objective and source of knowledge, you should be able to perform the activity described in the objective.

Application objectives, on the other hand, require you to apply knowledge to problems. Since assembler-language programming is concerned entirely with problem solving, the primary objectives of this book are application objectives. In general, knowledge objectives are only stated when they are a prerequisite for understanding some aspect of assembler language. If only one objective were given for this entire book, it would be something like this: Given a business programming problem of any degree of difficulty, solve it using assembler language.

5 Following the behavioral objectives there are one or more problems for each application objective. These problems are intended to get you involved. There is much truth in the maxim: I hear and I forget; I see and I remember; I *do* and I understand. If there is one message coming from research in education, it is that meaningful learning depends on what the learner does—not on what he sees, hears, or reads.

Because the intent of this book is to teach assembler-language programming, the problems for the most part ask you to apply assembler language to significant programming tasks. There are no fill-in answers, no multiple choice questions, and no true/false statements because those types of activity have nothing to do with writing assembler-language programs. As much as possible, the problems are intended to stimulate the kind of thinking that would be necessary if you were actually doing the job of a programmer. Because the problems often require you to apply assembler language to situations that go beyond the applications shown in the topics themselves, I hope that at times the problems will help you to experience the joy of discovery and to receive the reward of deeper understanding.

Solutions are presented immediately after the problems. This allows you to confirm that you are right when you are right, but it also lets you learn from being wrong. By checking the solution when you finish a problem, you can discover when you are wrong and correct false notions before they become habits.

Should you actually work each problem in detail before checking the solution? This may be the surest way of learning, but it isn't necessarily the most efficient or the most practical way. As long as you determine what the problem is and conceive the essential elements of a solution before you check the given solution, learning should take place.

One important message: *don't skip the problems.* Reading, like listening, can be a very passive activity. Have you ever, for example, read an entire chapter of a book and then realized you didn't understand any of it? The problems for each topic or chapter will provide you with check points to reinforce the reading you've done, to teach you application, to keep the learning process active, and to help you progress toward deeper understanding.

6 Although this book provides all the information needed to write a wide variety of programs, it is not a complete description of all of the details or instructions of assembler language. As a result, several IBM manuals are named as reference materials in various chapters of the book. If you would like to get a complete set of these reference manuals (though it's certainly not necessary), you can order them through IBM.

For 360 systems operating under OS/MFT or OS/MVT, use these older versions:

| Order No. | Title |
|---|---|
| GX20-1703 | System/360 Reference Card |
| GA22-6821 | System/360 Principles of Operation |
| GC28-6514 | OS Assembler Language |
| GC26-3756 | OS Assembler (F) Programmer's Guide |
| GC-3794 | OS Data Management Macro Instructions |
| GC28-6646 | OS Supervisor Services and Macro Instructions |
| GC28-6631 | OS Messages and Codes |
| GC28-6703 | OS Job Control Language Reference |

For 370 systems operating under OS/VS1 or OS/VS2, there are newer versions of the same manuals:

| Order No. | Title |
|---|---|
| GX20-1850 | System/370 Reference Card |
| GA22-7000 | System/370 Principles of Operation |
| GC33-4010 | OS/VS and DOS/VS Assembler Language |
| GC33-4021 | OS/VS Assembler Programmer's Guide |
| GC26-3793 | OS/VS Data Management Macro Instructions |
| GC27-6979 | OS/VS Supervisor Services and Macro Instructions |
| GC38-1001 | VS1 System Messages |
| GC38-1003 | VS1 System Codes |
| GC38-1002 | VS2 System Messages |
| GC38-1008 | VS2 System Codes |
| GC28-0618 | OS/VS Job Control Language Reference |

At the least, you should order the reference card that applies to the system you will be using.