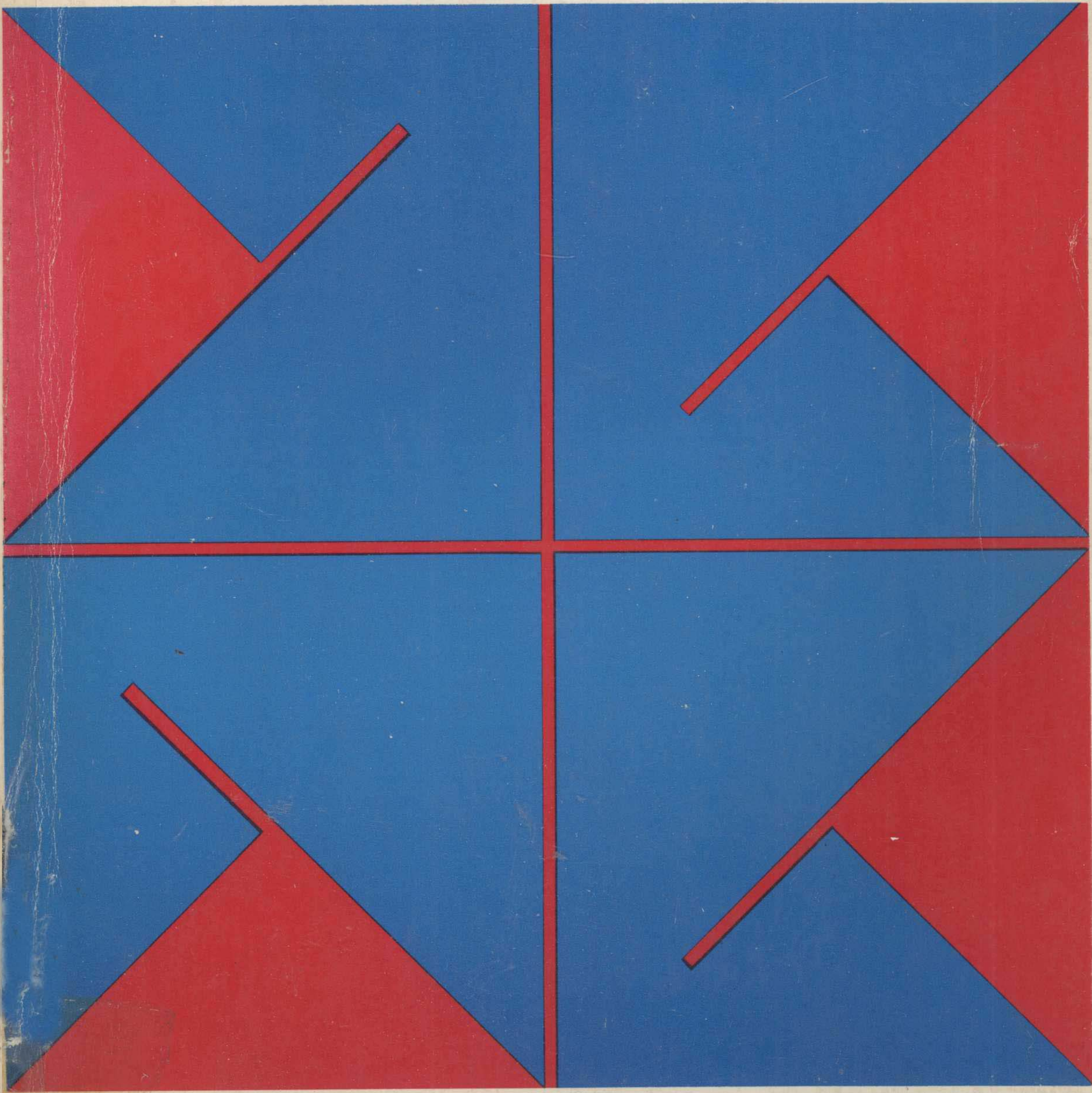
Robert G. Finkenaur

COBOL for Students: A Programming Primer

Foreword by Gerald M. Weinberg



Little, Brown Computer Systems Series

Robert G. Finkenaur
Northeastern University

COBOL for Students: A Programming Primer

Little, Brown and Company
Boston *Toronto*

Library of Congress Cataloging in Publication Data

Finkenaur, Robert G
COBOL for students.

Includes index.
1. COBOL (Computer program language) I. Title.
QA76.73.C25F54 001.6'424 76-55362
ISBN 0-316-283207

Illustrations by Robert F. Rosenberger. Photographs by the author except where otherwise noted.

Library of Congress Catalog Card No. 76-55362

ISBN 0-316-283207

9 8 7 6 5

HAM
Published simultaneously in Canada
by Little, Brown & Company (Canada) Limited

Printed in the United States of America

Acknowledgment

The following acknowledgment is reprinted at the request of the American National Standards Institute, Inc., from their publication, ANSI X3.23-1974, *American National Standard Programming Language COBOL;* May 10, 1974:

> COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.
>
> No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the Committee, in connection herewith.
>
> The authors and copyright holders of the copyrighted material used herein
>
> > FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC ® I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell
>
> have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Foreword

In the first six months following the announcement of this series, I personally was offered no fewer than eleven outlines or manuscripts for "Introduction to COBOL." Out of this short dozen, only one had anything to distinguish it from the five dozen COBOL texts already published. That one was Bob Finkenaur's *COBOL for Students.*

Why does *COBOL for Students* represent a true contribution to programming education? Isn't the field adequately covered by the three score texts already in use? Although there are some rather good texts among them, the weaknesses of these existing texts provide the best explanation of the strengths of *COBOL for Students.*

A leading weakness of introductory texts in all languages is their lack of attention to the tenets of good programming style. As a result, some fairly undisciplined programming habits are being taught to students. Kernighan and Plauger, in their *Elements of Programming Style* (McGraw-Hill, 1974), literally shredded scores of examples from FORTRAN and PL/I texts, and could find but two authors with whom they were reasonably satisfied. They have promised to do the same with COBOL texts, and there the shredding job should prove even easier. Finkenaur's text and its examples, however, will stand up well to critical examination from most real programmers. Students are admonished repeatedly that their programs must be written in a style that makes them easily understood and contributes to the ease of their later, inevitable modification. The text's programs demonstrate by example how this is accomplished. The disciplined programming style espoused by this text will leave students well equipped to graduate into the realm of structured programming and the other advanced topics covered in the field's more rigorous literature. It is, for example, completely consistent in spirit with the stylistic approach of the more advanced COBOL text in our series, *High Level COBOL Programming,* making *COBOL For Students* a perfectly suitable introduction for students who might later use that book.

The most serious failing of most introductory texts, however, is the author's "inability to put [himself] in the readers' shoes and write for the average beginner. Too many books seem to have been written with the top student in mind, by authors who either don't realize that most readers are starting at zero, or who seem to assume that the reader is as smart as the writer.*" Programming is a subject whose full depth has never yet been explored. In writing an introduction to such a deep subject, the author must have an exquisite sense of when to stop exploring and when to start explaining. He must resist the temptation to write a knowing side remark that will impress his colleagues on the tenure board, but will befuddle hundreds of students. He must know when a simplified discussion teaches better than the full gory details, and when saying nothing is better than saying anything at all. Finkenaur, for example, in describing each COBOL program component, wisely chooses to describe it only in its most commonly used form and avoids obscuring its central purpose and usefulness with long lists of seldom used options and seldom violated restrictions.

The level of *COBOL for Students* is right for its audience—beginning students. Some of the programs won't achieve high marks for sophistication, yet the truly sophisticated reader will appreciate that their simplicity is the conscious result of a dedication to effective communication.

It's not enough, unfortunately, to be *dedicated* to effective communication. If that were all there was to it, we'd have sixty wonderful versions of "Introduction to COBOL"—fifty-nine, at least. To write an effective text, one must be able to *write*. Many of the introductory COBOL texts would make excellent casebooks for a freshman English class—providing an abundance of material to be rewritten into effective language. How can one teach programming style when one is unable to write English with any style at all? *COBOL for Students* convinces any skeptic that good writing and good programming are not separate subjects.

Another common failing of introductory texts is in the examples. Publishers commonly boast of the *number* of examples in a text, as if to substitute for the missing *quality*. The experienced teacher who studies Finkenaur's examples will see that each one has a purpose that is specific, significant, and unique. Taken together, the examples cover the subject—but don't bury it.

How does an author arrive at such an ideal set of examples—both for the text and the problem sets? In practice, there can be but one way: through attempting to teach, failing to teach, developing examples to remedy the failure, and testing the examples in actual teaching. Behind each of Finkenaur's examples, there is a former student somewhere who didn't understand something—and raised a hand about it. For a book developed in this humble and interactive manner, what title could be more appropriate than *COBOL for Students*?

The teacher who adopts *COBOL for Students* will find its use full of small pleasures. For one thing, the programs will work in most installations, for they have been tested by machines and by many pairs of eyes. They avoid system-dependent tricks in favor of straightforward style that will work well in almost any sensible installation, and which will prepare a student properly for later courses in programming. And, for those students who will take no other programming course, *COBOL for Students* will not give the common impression that programming is a game for magicians or frustrated Tom Swift's born seventy years too late.

* *Note:* The quoted material is from Stephen B. Gray's article "34 Books on BASIC" in the March–April 1975 issue of *Creative Computing* magazine.

Because of the excellent examples and sensible approach, *COBOL for Students* will encourage students to raise questions of significance, not bit-picking puzzles. What a pleasure for a teacher not to have to apologize for nonsense in a text, and to be able to concentrate on imparting the *wisdom* that an excellent text can suggest, but only an excellent teacher can provide. The problems, for instance, suggest many directions that can be treated more expansively, according to the personal taste, style, and educational philosophy of the instructor.

Some teachers may be momentarily puzzled that *COBOL for Students* does not "cover" all of COBOL—indeed, only about half of the language "features" are presented. Few good instructors (but, unfortunately, the majority of textbook writers) think that complete "coverage" is an important educational objective for an introductory course. Most teachers will be jubilant to discover a text that doesn't submerge students in a bath of advanced terms, technical jargon, and grammatical details—before they've even learned to dog-paddle. No reasonable person believes that a raw student becomes a COBOL programmer in three months or three credits.

COBOL for Students is the proper *size* for a first course. It doesn't intimidate the student with its thickness; it doesn't flood the brain with next semester's material; yet it never gives the impression that you close the subject when you close its covers. It encourages good students to take the next course, but doesn't charge the one-term student for a two-term text—either financially or intellectually.

I hope, however, that someday I will be writing a Foreword to a second Finkenaur text—perhaps, in the Hollywood tradition, entitled *COBOL for Students II*. The material is there, evolving with the full interaction of Bob's students. What we await now is a response from the readers and teachers, telling us that they agree with our assessment of this student-oriented approach—or, if they disagree, giving suggestions for future editions and successors.

Gerald M. Weinberg
Swiss National Holiday, 1976

Preface

This book was written by a person who is a teacher first and a programmer second. It reflects an accumulation of eight years' experience in teaching computer programming to students of every conceivable background and interest. My pupils have represented the full range of the academic spectrum: elective students and reluctant students; bright students and slow students; easy-going liberal artists and regimented military academy cadets; men and women; every known American minority; young full-time day students and adult part-time evening students; and every imaginable major: computer science, engineering, math, business, history, music, astronomy, geology, nursing, physical education, psychology, library science, English, actuarial science, and military science.

Thanks to my students, I have never been allowed to forget that programming does *not* come easily to everyone. What is responsible for the difficulty many students experience in mastering the art of programming a computer?

To be sure, there is the question of "knack." Each of us is stronger in certain talents than in others. You probably know someone who is a good athlete, and someone else who is completely devoid of hand-to-eye coordination and couldn't be taught to play any sport well no matter how many lessons were taken. Yet the latter might be a respectable musician, while the former may not have any musical talents at all, and no amount of music lessons could ever change that. Likewise, there are some who find themselves lacking in the "knack" of organizing their thoughts in the manner necessary to master computer programming with ease, and nothing short of their own superhuman effort will ever make programmers of them.

Another source of difficulty is the manner in which the material is presented. All too often, the teacher of computer programming or the author of programming textbooks is a programmer first and a teacher second (if that). He (or she) is someone to whom the subject *did* come easily and is therefore lacking in the ability to understand why it shouldn't be easy for all. His presentation of the material tends to be overly detailed and rigorous, too rapid, lacking in sufficient examples, and "pitched" to a small group of students who've already had programming experience (though perhaps in another language) and who in the classroom are recognized by their heads nodding in agreement with the instructor, their "Oh, how *right* you are!" smiles, and the types of questions they ask.

There is nothing this book can do to improve your "knack" for computer programming. It is my belief, however, that the book represents a big step forward in improving the manner in which the material is presented.

First, the book is written based on the assumption that you have never even *seen* a computer before. (I may assume that you already know how to *spell* "computer," but not much more. You will never hear me say, "It is therefore intuitively obvious that . . ." or any other such expression. My students have taught me well the locations of potential points of confusion, and you will see me pausing at each of those troublesome spots to help you through.

My approach to teaching you COBOL programming is what I call a *functional approach.* Having given you the bare skeletal bones of a COBOL program, I'll suggest, one at a time, some additional functions you might want the computer to perform, and having established the motivation for each new function, I'll teach you about only those additional COBOL elements necessary to achieve that additional function. By the time I'm finished, you'll have command of a full set of COBOL programming elements and be able to have the computer produce the fanciest pages of printed results, perform a wide range of calculations, and store, update, and retrieve "tons" of information (or data) on magnetic tape.

Along the road to that end, however, *you* must play *your* part in the learning process. You *must* work the dozen or so Exercises that are presented at the end of each chapter. You *must* study the Example Program that is provided at the end of each chapter starting with Chapter 7. And most important of all, you must work the Student's Program located immediately after each Example Program.

For, like learning to read, learning computer programming is very much a building block process. In reading you must learn your alphabet before you can read words, you must be able to read words before you can read stories. Exactly the same principle applies to the study of computer programming. If one of the blocks is missing, none of the rest will follow. You must assure yourself at the end of each chapter that you have mastered the complete block which that chapter was supposed to provide before you even glance at the next chapter. Only by honestly testing yourself at the end of each chapter with the Exercises, the Example Program, and the Student's Program will you have that assurance.

Before closing, I would like to express appreciation to some people who provided necessary stepping stones along the way to the completion of this book: First comes Colonel Gilbert Kirby, a Professor on the faculty of the U. S. Military Academy at West Point, whose direct order to me (a member of his department at the time) in 1970 to "learn COBOL and get a course going in it!" was responsible for my having learned the language in the first place. Next comes my boss at Northeastern University, Professor Wilfred Rule, who, by inviting me to serve as co-author of his (now our) successful text, *FORTRAN IV Programming,* provided some invaluable experience and inspiration. Both served as catalysts out of which sprang the idea to begin this book. Next are my many students whose repeated prodding ("How's the book going?") did more to *keep* the book going than they can ever imagine. Finally come my wife, Carol, and my teen age son, Bob, who suffered with me through the drudgery of proofreading the final copy.

Robert G. Finkenaur Boston, 1977

Contents

1 An Introduction to Digital Computers

1.1 Computers and Their Languages

In this day and age, there is little need to tell students what a computer is or what it does. The word *computer* has become as familiar a part of their household vocabulary as *refrigerator*. Daily we receive through the mail credit card bills, checking account statements, grade reports, etc., all bearing the distinctive "handwriting" of a computer. We know that thousands of others are receiving similar bills and statements, and from that comes our almost intuitive while basically correct understanding of what a computer does—it performs at lightning speed overwhelming numbers of repetitive calculations that at one time would have required countless bookkeepers many hours to perform.

Hand in hand with the word *computer* comes the term *computer programmer*. Most of us have a fairly valid impression of what a computer programmer does. He (or just as often, she) is the person who puts the computer through its paces. He tells it what calculations to make and what to do with the answers to those calculations.

Most people who read this book are doing so because they have decided they would like to become, even if only to a limited degree, computer programmers. In order to program a computer, you must be able to "speak" to it in a language it can understand. There are many languages available for use with the computer, but two are predominant: COBOL and FORTRAN. The latter (whose name is derived from the words "FORmula TRANslation") is best suited for use in science and engineering applications.

This text deals with COBOL, the "COmmon Business Oriented Language," a language designed for use in business applications. COBOL is in fact *the* most widely used language in the computer world today. This is so, because the vast majority of computer users are business firms using the computer to handle basically (though massive) bookkeeping tasks, and COBOL is the best language available for that purpose. A glance through the data processing want ads in any metropolitan newspaper will attest to the popularity of the language and the high level of demand for people who can write programs in it.

Before we begin delving into the details of COBOL programming, let's take an elementary look at how digital computers work. Though there are many who prefer to treat the computer as a "black box," that is, with an "I don't care *how* it does it, as long as it keeps doing it" attitude; this is seldom the best approach for any person who is serious about learning programming. Although a person unfamiliar with the inner workings of a car can still drive it, he can operate it more efficiently and realize better performance if he has some elementary understanding of what actually is happening under the hood. The same can be said of computer programmers.

Read this chapter to gain a general understanding only. It is intended to provide you with a basic feel for what goes on "under the hood" of a computer. It will prepare you to learn the art of computer programming more easily, and result in your becoming a more effective programmer.

1.2 The Digital Computer

The desk top adding machine, so familiar to everyone, is in fact almost a digital computer. Its big shortcoming, however, is that it "can't remember what to do next." It can compute the answer to any problem only as long as you its human operator are on hand to feed it each number and each operation (add, subtract, etc.) one by one and in the proper sequence. Therein lies the key difference between a digital computer and a desk top adding machine: the computer *can* remember "what to do next." It can be given a whole list of operations (or instructions) and a whole list of numbers and remember them all. The operator then simply "pushes the GO button," and the computer takes off entirely on its own, obeying the instructions it has been given beforehand, and solving the whole problem often in less time than it took you to read this sentence.

And that brings us to a second difference between the adding machine and the computer—its extreme speed. An adding machine performs its calculations through a complex set of internal, interconnected gears. The speed with which it performs those calculations is dependent upon the rotational speed of those gears. Within a computer, the components involved in calculations are all electronic circuits, so they are performed at the speed of electricity—the speed of light.

Although the battery-powered, pocket calculators of today do rival the computational speed of digital computers, they still require the presence of the human operator to feed each numerical value and each instruction as it is needed in the course of solving a problem.

It is this list of instructions I keep referring to that is called the program. The scheme which truly sets the computer apart, whereby all instructions are given to the computer and stored in its memory before it begins solving the problem, is therefore called the Stored Program Concept.

1.3 The Organization of the Basic Digital Computer

If I were about to explain to you how an automobile's engine worked, I would use the simplest, most straightforward engine as the basis of my explanation. To make the discussion less difficult to digest, I would omit

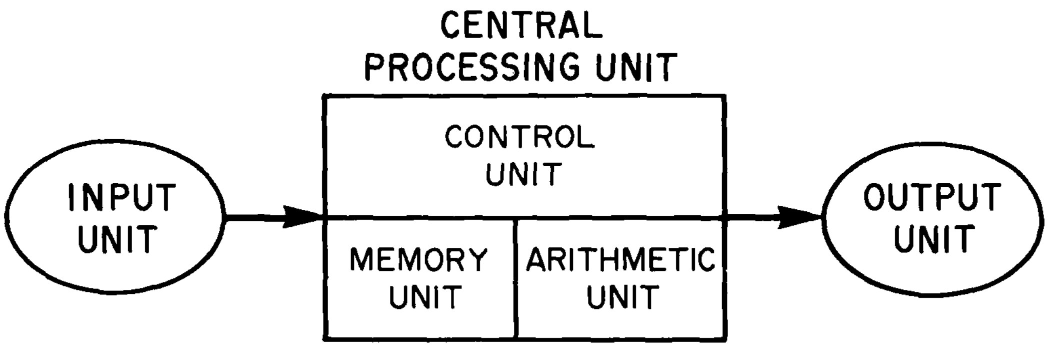


FIGURE 1.1 Organization of a Basic Digital Computer

speaking of such things as fuel injection systems, double-barreled carburetors, emission control systems, and other similarly complicated devices which are so often a part of the most up-to-date engines. Likewise, as I now approach an explanation of how computers are organized and how they function, I will choose the basic or "first generation" digital computer to be the basis of my explanation. Do not fear that you will be getting an inaccurate description. Current generation digital computers have all evolved from the basic digital computer, still retain their principles of operation, and will give the appearance to you of being organized as described here. Only after you have mastered the contents of this book will it be necessary for you to concern yourself with the features built into modern computers which were not available in the simplified version we will be discussing.

As seen in Figure 1.1, the computer is organized into three major units: the input unit, the central processing unit (or "CPU" as the folks down at the computer center call it), and the output unit.

The input unit is the device through which we put information into the computer. The output unit is the device through which the computer puts out for human use the answer to the problem it has solved for us. You may hear the term *peripheral devices*, or simply *peripherals*, being used to lump together the input and output devices as well as any other devices that are connected to but not included in the CPU.

The CPU is further subdivided into three important units: the control unit, the arithmetic unit, and the memory unit.

The arithmetic unit (pronounced a-rith-me'-tic rather than a-rith'-me-tic in computer circles) is the portion of the CPU where all arithmetic operations—adding, subtracting, multiplying, dividing—are performed. If it occurs to you to wonder why the arithmetic unit cannot perform higher-order mathematics such as raising numbers to various powers, or taking square roots or integrating, the answer is that it can, but not directly. Such higher-order operations are accomplished by a clever combination of the four basic operations listed.

The arithmetic unit also provides the computer with the ability to make certain logic decisions such as whether one number is larger than or equal to another, or whether a single number is negative, positive, or zero.

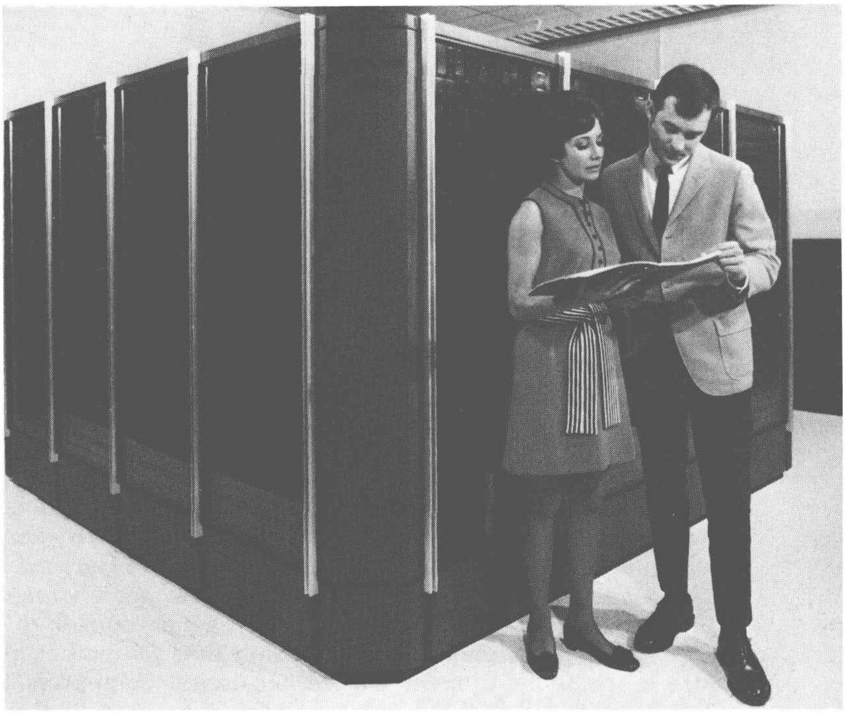


FIGURE 1.2 The Central Processing Unit

Courtesy of Control Data Corp.

You will find such decision-making ability extremely valuable later when you are faced with the necessity to write a program for the computer in which selected instructions are to be omitted under certain numerical circumstances. (Take an example: Under normal circumstances the amount of a check is subtracted from the balance in your checking account. In the hopefully rare instance where the amount of a check is larger than the balance in your account, it should *not* be subtracted from the balance, but rather "bounced" by the bank.) We count as the digital computer's three most valuable talents its Stored Program Concept, its great speed, and this ability provided by the arithmetic unit to make basic logic decisions.

The memory unit is where the computer stores all information it needs to solve the problem it is working on. We will discuss this unit in greater detail in Section 1.6.

The control unit is the boss of the whole operation. Following the instructions it received and stored prior to the operator's "pushing the GO button," it controls the movement of information from the input unit to memory, from memory to the arithmetic unit, from the arithmetic unit back to memory, and from memory to the output unit. Whenever it moves information (numbers) to the arithmetic unit, it tells the arithmetic unit which operation to perform on them (add them or subtract them or whatever), and