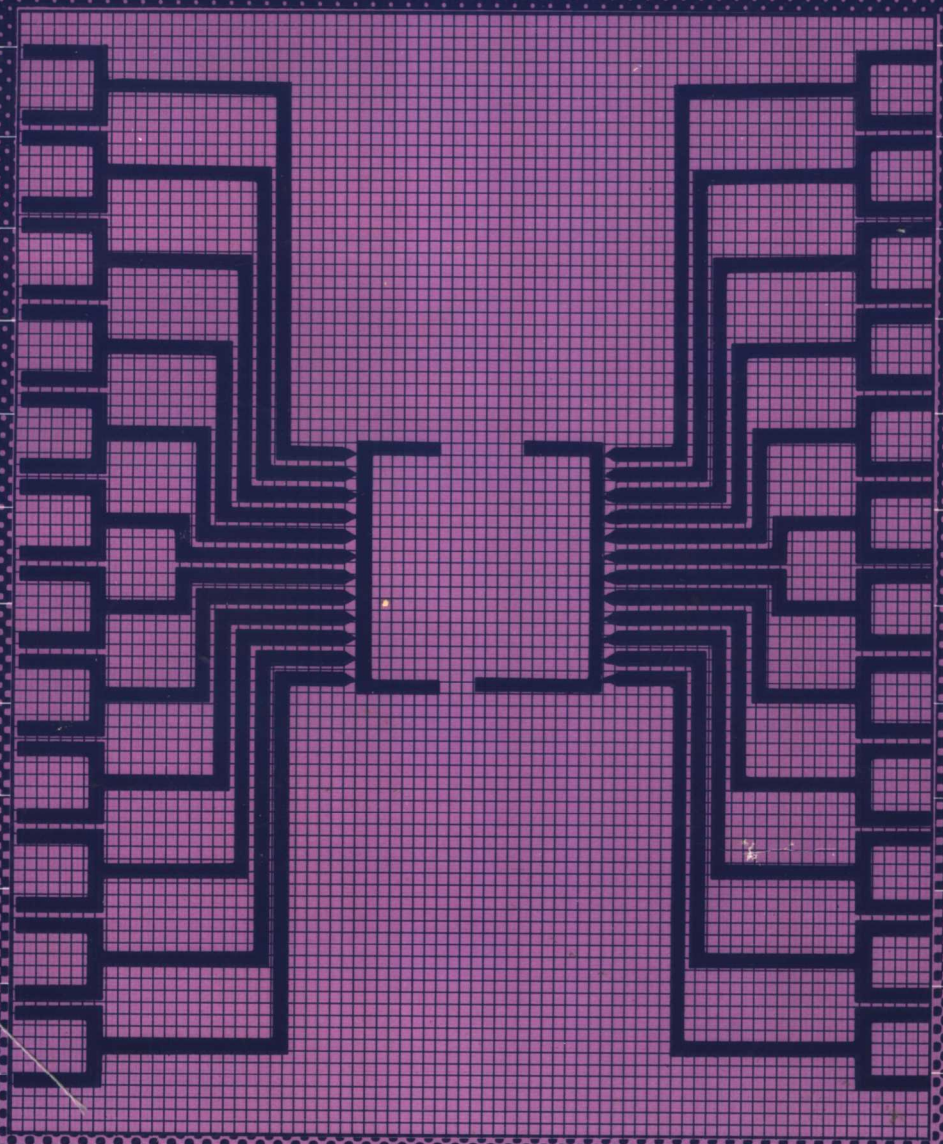
COMPUTER ORGANIZATION AND PROGRAMMING
With an Emphasis on the Personal Computer



COMPUTER ORGANIZATION AND PROGRAMMING

WITH AN EMPHASIS ON THE PERSONAL COMPUTER

FOURTH EDITION

C. William Gear

Department of Computer Science
University of Illinois at Urbana-Champaign

McGRAW-HILL BOOK COMPANY

New York St. Louis San Francisco *Auckland* *Bogotá*
Hamburg Johannesburg London Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo *Toronto*

This book was set in Times Roman by Beacon Graphics Corporation.
The editors were Eric M. Munson, Kaye Pace, and Sheila H. Gillams;
the production supervisor was Diane Renda.
New drawings were done by Danmark & Michaels, Inc.
Halliday Lithograph Corporation was printer and binder.

COMPUTER ORGANIZATION AND PROGRAMMING
With an Emphasis on the Personal Computer

1 2 3 4 5 6 7 8 9 0 HALHAL 8 9 8 7 6 5

ISBN 0-07-023049-8

Library of Congress Cataloging in Publication Data

Gear, C. William (Charles William), date
Computer organization and programming.

(McGraw-Hill series in computer organization and architecture)
1. Electronic digital computers—Programming.
2. Computer architecture. 3. Microcomputers—Programming.
I. Title. II. Series.
QA76.6G38 1985 001.64'2 84-20180
ISBN 0-07-023049-8

PREFACE

This book is intended for a course in programming at the machine level and is aimed at students who have taken one or two earlier courses in a procedure-oriented language. The material covered is essentially that in Association for Computing Machinery Curriculum '78 courses SC3 and CS4, although it does not cover as much logical design as CS4. The intent of the book is to let the student see the principles behind various machine-language features and understand the alternatives available to the computer designer. I do not view this as a course for training assembly-language programmers; indeed, few such programmers are needed any more. However, there are many reasons for knowing something about the machine at this level: a high-level language programmer can produce better code when there is an understanding of the processes occurring at the machine level; occasionally, some inner part of a high-level language program must be written at the machine level, either to gain efficiency or to gain access to devices at the bit level; computer designers need to understand machine-level programming and how it is used in the implementation of high-level language constructs; and future microprogrammers certainly need to understand the concepts of machine-level programming!

The objective of this book continues to be presentation of the basic concepts of machine-level architecture and programming, so that all ideas are discussed in a non-machine-specific setting. The objective I state to students taking this course is that at the end they should be able to pick up the assembler and principles-of-operation manuals for a computer and be able to find out how to program it in assembly language. The machine- or system-specific information in this book is not intended to be a complete reference for the computer or its system, but to introduce the student to the computer's use. The student should have copies of the reference manuals for the particular hardware and operating system in use. Unfortunately, those manuals are mostly unreadable, so one purpose of this text is to show the student the underlying principles to make it possible for her or him to use the reference manuals.

I place heavy emphasis not on learning details of a particular machine but on understanding concepts. However, I have found that it is necessary to teach students a

particular assembly language at first and neither leave them in the air with vague generalities nor confuse them with the details of more than one computer. Therefore, I introduce each new topic (such as index registers) by first discussing the general principles and giving some motivation for their use. Then I illustrate with the particular computer being used for homework assignments. I do not discuss other computers until late in the course when the basic ideas are fairly well understood and confusion is less of a problem.

This fourth edition has been rewritten to reflect the fact that the majority of instructional use is now interactive, and reorganized considerably to present the material in a more top-down manner. The introduction now includes a brief overview of typical operating systems, some specifics on actual operating systems, and a quick look at editors. Most students will have had some experience with the use of systems in an earlier class, but if not, this material is intended to make it possible for them to start using a computer as soon as possible. Chapter 2 starts with a quick look at machine-level programming of the simplest type and then presents an initial look at assembly so that the student can start to try out simple programs. It also looks at debugging systems, since these can be used to watch the execution of simple machine-language code interactively. I strongly recommend that this method be used in the early stages of instruction, as it bypasses all problems of input and output. In fact, with some absolute debuggers it even allows avoidance of the assembler, as the debuggers permit direct assembly of code with absolute addresses, and this is sufficient for the initial experiments with machine code.

In this version of the fourth edition, specific examples are taken from the INTEL 8080 and 8086/88 hardware with the CP/M and IBM PC DOS operating systems, respectively. There are enough similarities between the two architectures and systems that the student can see the ideas in a real setting without the confusion of too many different implementations. At the same time, the two architectures and systems include many of the ideas of modern computer and system design.

I supplement the book with additional material on the computer being used. I have found that some locally prepared short summaries of the key information, coupled with manufacturers' documentation, offer a good compromise, as one of the skills to be learned is the use of typical computer reference documentation.

It is important to assign a number of programs as homework during the course. I have found it helpful to provide the students with a simple, working assembly-language program and to ask them to make minor changes to it as their first assignment. This allows them to gain some experience while the beginning ideas are covered in class. By the time some of Chapter 2 has been covered, a second assignment that requires more independent work can be made. Material in Chapters 3 and 4 can be covered at almost any time, and many instructors prefer to get into procedures in Chapter 5 as soon as possible. The amount of material covered in Chapter 3 should be determined by the interests and backgrounds of the students — it is not essential to cover more than the initial overview of twos complement. Chapter 4 can be left as a reading assignment; it is included for the sake of completeness. Chapters 6, 7, and 8 are the last that I view as essential to cover in a course of this type. Chapters 9 through 12 are optional; each can be omitted without affecting later discussions. Chapter 9 can be used for

engineering-oriented classes. Chapter 10 can be given as a reading assignment. Chapter 11 can be used to give a better insight into the assembler or to illustrate a large programming problem. Chapter 12 can be used as a source for programming problems as well as for some low-level data structure discussions.

There are a variety of questions at the end of the chapters, including some programming problems. Many of these can be modified to provide additional questions or homework.

C. William Gear

CONTENTS

CHAPTER 1

INTRODUCTION

Computer programs can be written with no knowledge of the underlying *architecture* (that is, the hardware organization) of the computer system that will be controlled by the program. Indeed, modern programming practice stresses the desirability of making programs *machine-independent*. For most purposes, the programmer can view a program as a step-by-step description of a computational *process* without any regard for the computer to be used. Any suitably precise language can be used to describe the process, and a computer can be viewed as a device that is capable of understanding that description by following the steps in the specified order in the same way a human reader might. The only restriction imposed by the computer that the programmer need be aware of is one of *representation*: the program and data have to be represented in a form acceptable to the computer system to be used.

At the the other extreme, computer architecture can be designed with little attention to programming languages. As long as the architecture is capable of representing common data and of performing the fundamental operations (communication, data manipulation, such as subtraction, and decision making by comparison, such as "Is X greater than Y?") we know that the computer can be programmed to perform any computation. However, what is important in the architectural design is the ease and speed with which common programs can be processed, and what is important in programs for some large, difficult problems is the way in which they make the best use of the computer structure.

In this book we will study the representation of programs and data in computers and the organization of computers. Our goal is to understand how programs should be written so that they can be efficiently processed, and how computers can be organized to aid in attaining that goal. The emphasis will be on current computer architectures and programming styles, so we will examine several popular examples of computers. In this

study we will stress the view that the program is a process that is to be executed on a computer. The data associated with the program and the current point of execution of the progam is the *state* of the process. As the computer "reads" through the program, following the instructions specified in each step, the data values change as the computer moves from step to step, and thus the state of the process changes. The computer organization only affects the way in which the process is represented internally; the average programmer is free to think of the process as represented in Pascal or any other language, but internally it is represented in the form understood by the architecture, called machine language. To understand the interplay between architecture and programs, we must study *machine-level languages*, which include actual machine languages and simple representations of them, called *assembly languages*, that are more comprehensible to people.

It is often fruitful to glance over the historical development of a field to gain an understanding of why things are the way they are, even in a field such as computer science, in which most developments have occurred since 1945. The digital computer was developed in answer to the need for quicker and more efficient ways of handling large numerical computations. Those computations arose in practical situations, as in the solution of engineering problems such as the design of a bridge or plane and the choice of heating-cooling strategies to minimize fuel consumption, as well as in the course of theoretical endeavors, such as the attempt to locate prime numbers. (Locating prime numbers is not a task of great economic value, but related problems are of intellectual interest to mathematicians, and excess time on early computers was used in such calculations.) Tasks such as these *could* have been doen without the help of computer technology, but in many cases their value would have been negated by the cost in human time and stress, not to mention the high probability of error in human calculation.

However, some fairly large calculations had to be done before the development of the automatic digital computer, and these were handled in the following way: They were split into many sections — each relatively independent of the others — which were then computed separately. Each section was broken down into a set of instructions and the data to which it was to be applied. The various sections were then handed on to a number of assistants who performed the indicated computations using hand calculating devices. Now, the instructions and data are given to the digital computer; it performs the routine tasks previously done by these assistants. (No matter how fast computers are, it appears that society will always have problems that exceed the capacity of computers. Today, some extremely large problems are broken into relatively independent parts, and each part is solved on a separate computerlike device, although these devices may be interconnected so that data and program can be transferred between them. This is called *parallel processing*.)

After the initial use of computers in numerical problems, workers began to explore other areas of application. It was noted that any finite class of objects could be related to the sets of numbers that the computer could handle and could be manipulated in the same way. *Nonnumeric* data processing became important. (Even before the automatic digital computer, nonnumeric data processing had become available for some business applications through the use of punched-card machines.)

Nonnumeric processing is basically similar to numeric processing in that various logical processes, or *algorithms*, which manipulate finite sets of objects (either numbers or other data) can be prepared for the same computer used to solve numerical problems. At first, this was viewed as the numerical representation of nonnumerical data. As an example, the letters of the alphabet could be assigned numeric values 1 through 26, with a space represented by 0. (Space, or *blank*, as it is usually called in computer usage, is one of the possible characters that a computer must be able to "print" on a page.) A machine capable of manipulating 8-digit decimal numbers could then be used to manipulate four–or–less–letter words by replacing each letter by its equivalent number and adding spaces (zeros) if there were less than four letters. Thus, MOTH would be represented as 13 15 20 08 and MAT as 13 01 20 00. In this scheme, words could be arranged alphabetically by arranging their numerical representations in numerical order. MAT preceded MOTH because 13012000 is less than 13152008.

Through such techniques, business data processing, frequently involving the manipulation of large amounts of character data, was automated for computers. Nonnumeric processing has been extended to applications involving the manipulation of abstract mathematical quantities, such as groups; to the problems of natural language translation, such as Russian to English; to graphical or pictorial data processing, such as the matching of fingerprints or the examination of x-ray pictures for disease indications; to the control of complex processes, such as those involved in running oil refineries; and to the control of robots.

Programs for the first computers were written directly in machine-level languages by people who were familiar with the inner structure of the computer. As computer applications grew in number and variety, the computer became available to people with little knowledge of the details of its inner structure. A typical user now wants to specify only the steps in a computational task, not the details of the way those steps are to be executed in a particular computer, just as a person asking another person to retrieve a letter from a file, for example, does not want to also describe how the file drawer should be opened or how to alphabetize letters. Therefore, to the basic computer *hardware* (that is, the physical parts of the computer) was added the *software* of *system programs*, which handle many common operations for the user and which translate the language of the user into that of the computer. The hardware and system programs together constitute the *computer system*. The hardware provides the ability to do basic operations, the software the ability to specify the job in a convenient notation and to move from one job to another without delay. The user sees the combination and does not need to distinguish the two parts. Indeed, as technology changes, things once done by software in early systems are done by hardware in later systems.

This book discusses computer organization by describing various ways in which the basic hardware and software components of the computer are logically connected, the reasons for the different organizations, and the advantages of each. This material is then used as background for a discussion of the basic principles of machine-level programming. These principles are illustrated by a discussion of some aspects of (1) the translation of programs written in a language applicable to a task, called a *source language*, into an equivalent program expressed in machine language; (2) the scheduling of these user programs by other programs, usually referred to as

kernel, *executive*, and *supervisor* programs; and (3) providing assistance to the program in execution by system subprograms, which are typically concerned with input-output and other communication.

A computer system should provide a tool for solving problems quickly — in terms of the computer's time as well as that of the user. The hardware should be designed to operate as fast as possible within given cost limits. The software should be designed to minimize the amount of wasted computer time and yet provide as flexible a means of controlling the operation as possible. Computer functions that can be expected to be fixed throughout the life of the machine should be provided by hardware where possible. Functions that will change — by increases in capabilities without the negation of earlier properties — should be controlled by software. The objective of system design must be one of minimum cost for the whole job, including the cost of programming as well as execution on the computer. These considerations are as important for the large central computers that process a great variety of jobs as for the microcomputers that allow placement of a separate computer on everybody's desk.

The programmer who programs at the machine level may be a *system programmer*, that is, one who writes programs that will help other users access the computer system; an *application programmer* who finds it necessary to work at the machine level in sections of very large application programs where high speed is important; or a designer faced with the development of a microcomputer for a single application. This book is directed at all of these types of programmers.

1.1 THE BASIC CAPABILITIES OF COMPUTERS

A digital computer is capable of basic operations of finite sets of objects such as numbers. These include the usual arithmetic operations of *addition*, *subtraction*, *multiplication*, and *division*. In addition, a computer has the important ability to compare two numbers or nonnumeric quantities in order to take one action if the comparison is successful or another if it fails. These comparisons usually include tests for *greater than or equal to*, *equal to*, *less than*, etc. In additon to operating on information, the computer must be able to read the initial data from the user, retain it for later computations, and write the answers for the user. Therefore, in addition to the operational power, it has facilities for *storing* data and for reading and writing it in a useful form. This form can be a human interface (such as keyboards, or printed output) or the sampling of data signals and control signals to devices such as those used in automatic flight control. The next few subsections will trace briefly the historical development of these facilities and the importance of each new feature.

1.1.1. The Hand-Operated Calculator

The mechanical, hand-operated calculator was one of the first computing tools. With its ability to perform the basic arithmetic operations it replaced a lot of tedious human calculation. However, human intervention was still needed, both to control the steps in the calculation and to retain intermediate results (by writing them on scratch paper). In order to use a hand calculator effectively, the steps of a problem must be carefully organized; that is, the sequence of calculations must be specified carefully. This

specification is a progam that describes a process for the hand calculator. The process can be executed by pushing the control keys specified for each step. Modern electronic, hand-held calculators provide a number of *storage registers* that eliminate the need to copy intermediate results, but the speed of a hand-operated calculator is limited by the speed with which a human operator can issue instructions in the process by manually pressing keyboard buttons.

1.1.2 The Stored-Program Computer

The step to *stored-program* computers, which bypassed the speed limitation imposed by human control of processing, occurred in two stages. The first stage was marked by the introduction of sequence-controlled calculators.[1] A fixed cycle of steps, that is, a program, was prepared ahead of time, encoded, and stored on a punched tape or a plugboard. The calculator got its sequence of operations from this storage device and repeated the cycle of steps indefinitely. The capabilities provided by such mechanisms were equivalent to those of a computer in which programs consisting of only one or two simple assignment statements were executed repetitively.

These techniques were used for repetitive operations, such as summing numbers punched in cards. With the addition of auxiliary cycles of steps that were activated under certain conditions (and with a lot of ingenuity), quite sophisticated jobs were tackled. The sequence-controlled calculator was limited by its inability to modify its cycle of steps in any significant way because the cycle was stored in a semipermanent memory such as a plugboard. Today, the more expensive hand-held calculators can store programs.

The limitation of sequence-controlled calculators was overcome by the introduction of the stored program computer,[2] in which the sequence of steps was stored in the same memory as the data instead of in a separate memory such as a plugboard. Because

[1]Although sequence-controlled calculators were designed as early as 1812, when Charles Babbage designed his "Analytical Engine" (he was unable to complete it because of technological difficulties), the first calculator capable of performing long sequences of calculations was the Harvard Mark I, designed by Howard Aiken in 1937, built by IBM, and donated to Harvard University in 1944. It was essentially mechanical. The first electronic calculator was the ENIAC (Electronic Numerical Integrator and Calculator), designed by J. P. Eckert and J. W. Mauchly of the Moore School of Electrical Engineering, University of Pennsylvania. It was completed in 1945.

[2]The first design for a stored program computer was by Eckert and Mauchly of the University of Pennsylvania. Many of the ideas resulted from the work of John von Neumann, a person who contributed greatly to both mathematics and computer science. This computer, the EDVAC, was not finished until 1951. Separate efforts of design and construction were occurring simultaneously in the United Kingdom. Alan Turing, then at the National Physical Laboratory (NPL), had proposed a design for the Automatic Computing Engine (ACE). That design was quite different from other designs and was heavily modified by others before finally being built in a prototype version called the Pilot ACE. Manchester University and Cambridge University were undertaking separate efforts. The EDSAC, designed at Cambridge in 1946 and operational in 1949, was probably the first operational stored program computer, but a prototype machine at Manchester University was operational on June 21, 1948, and was probably the first stored program computer to be operational. The latter was the basis for a subsequent design that was put into production by the Ferranti company and became the first commercial computer, beating the better-known UNIVAC I by a short period. The UNIVAC I was also designed by a team led by Eckert and Mauchly. For a brief historical summary see the opening chapters in M. V. Wilkes, *Automated Digital Computers*, J. Wiley & Sons, New York, 1956. For an interesting view of an unusual and important figure in those times, see Andrew Hodges, *Alan Turning: The Enigma*, Simon and Schuster, New York, 1983.