MINICOMPUTERS: STRUCTURE & PROGRAMMING

T.G. Lewis & J.W. Doerr

HAYDEN

MINICOMPUTERS

Structure & Programming

T. G. LEWIS

Associate Professor, Computer Science
University of Southwestern Louisiana

J. W. DOERR

Project Supervisor
Moore Systems, Inc.

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

To our own little "minis":

Paige and Stacey for beauty
Todd and Eric for truth.

Library of Congress Cataloging in Publication Data

Lewis, Theodore Gyle, date
Minicomputers: structure and programming.

Bibliography: p.
Includes index.
1. Miniature computers. I. Doerr, J. W., joint author. II. Title.
QA76.5.L494   001.6'42   76-21228
ISBN 0-8104-5642-7

2 3 4 5 6 7 8 9 PRINTING

77 78 79 80 81 82 83 84 YEAR

The authors wish to express their appreciation to the people who contributed to this work. To John Hamblen and Billie Gillett we owe support and encouragement. To the students at the University of Missouri—Rolla, and the University of Southwestern Louisiana we owe thanks for enduring courses while developing this material. In particular we deeply appreciate the leadership/friendship and imagination of Terry Walker. Also, our thanks to the flawless typing of Lynette LaCaze and the penmanship code-breaking and typing of Becky Terrell.

CONTENTS

INTRODUCTION

Even though minicomputers are widely used in industry they are slow to gain acceptance in computer science education. However, because of its increasingly great impact on industry and its potential as a teaching device, the minicomputer should be considered a part of the computer science curriculum. This book is appropriate for a minicomputer laboratory course at the lower division level emphasizing hands-on use, fundamental computer concepts in conjunction with computer lore, and problems peculiar to small machines. Also, an upper division course is possible which emphasizes demonstration of advanced programming techniques.

The Silent Explosion

Mini Population Explosion

The estimated number of minicomputers as of January, 1973 was 55,000 worldwide [1]. This represents a sizeable part of computers in general. The impressive fact, however, is that an additional 55,000 minis were purchased in 1973. Thus, the indications are that minis are mushrooming in numbers.

The mini is found in a variety of applications, but over half are used in process control, engineering-problem-solving, and data communications applications [1]. Educational use, however, amounts to only 7% of mini applications. By sheer numbers alone educators are lagging in minicomputer usage, and by inference, in minicomputer training.

This lack of recognition of the need for minicomputer exposure in the university may partially account for the fact that minicomputer software is inadequate (one-third of the respondents to Modern Data's Survey complained about various aspects of software). Traditionally, minicomputer manufacturers have been hardware oriented and have neglected software development.

Superprogrammer Enters

The hardware explosion in minis must be followed by a software explosion if these machines are to be useful. The level of sophistication needed to implement a minicomputer software system does not decrease merely because the machines are smaller and have less powerful instruction sets. On the contrary, the abilities of a programmer are perhaps challenged by minis. The short word length, limited core, and use in time constrained applications contribute to the challenge. How are we, as educators helping our students meet this challenge?

A second significant contribution of the mini explosion comes from advancing technology. From semiconductor memory, floppy disks, to stack processors and microprogrammed processors, the mini leads the way in many respects. It may be that ideas and concepts spawned in computer science classes are beginning to impact the real world. With minis in the classroom we are now in a position to demonstrate these concepts in a way not possible using a large central machine.

As an example of the software engineering gap, examine the stack processing concepts currently being advocated by most computer science departments. In a large machine dominated world, a student has little opportunity to program a stack machine. But in the mini world the stack operators are frequently found in hardware. Indeed, many new minis are designed around a stack/interrupt driven processor.

The result of stack implementation is that theory (e.g., parsing, recursion) is reflected in practice. The authors' personal experience has shown a combination of theory and practice yields a high return.

Computer Heritage

Babbage and the Poet's Daughter

How many computer science students appreciate the history, folklore, and excitement that is part of the computer scene? How many students realize that the stored program concept is part of the foundation of computer science? Do they appreciate the work of Von Neumann, Babbage, and Lady Lovelace? Do they get a feeling of perspective, excitement, and involvement from tracing their own development with that of computer greats?

Unfortunately, the answers to the questions above are no, in the case of most early courses. We educators are failing to convey a sense of historical perspective to our novice programmers. As an example, here is a story that adds to the excitement of learning about stored programs, Charles Babbage, and the notion of a looping algorithm. We use this story to prepare students for a programming assignment in their minicomputer laboratory.

When Babbage was a young boy he attempted to prove the existence of the devil by drawing a circle on the floor with his own blood. He expected the devil to appear while reciting the Lord's Prayer backwards. This same eccentric genius had a great idea: the difference engine. This was a mechanical calculator that produced tables by adding differences. For example, the table of squares, below, is computed by addition only.

n	n^2	first difference	second difference
1	1		
2	4	3	
3	9	5	2
4	16	7	2

Thus, 9 is obtained by adding 2 to 3 to get 5. The 5 is added to 4 (2^2) to get 9 (3^2). This process is repeated to get 4^2, etc.

Babbage's machine was very accurate and led to the discovery of errors in established astronomical tables. Babbage once exclaimed upon discovery of an error: "I wish to God that these had been calculated by steam."

The point is that a perspective is given. A student begins to appreciate the struggle men had in developing computers. As a participating laboratory student, he can repeat the experiment of Babbage's difference engine himself. Perhaps his programming errors are similar to the bug in Lady Lovelace's programs. Perhaps some students will answer "yes" to the previously posed questions of excitement, discovery, and involvement.

Contemporary Prophets and the Touching Syndrome

Many computer science departments are implementing minicomputer laboratory courses [2, 3, 4, 5]. The computerware ranges from simulators (terminal based via timesharing) to real minicomputer systems. The software varies from similitude devices and fully multiprogrammed systems to simplicity itself [3, 4].

The authors conjecture that an educational software laboratory that stimulates student interest, motivates in-depth study, and provides minicomputer training per se in addition to a foundation in general computer science, will employ the following:

1. hands-on laboratory
2. open shop, 24 hrs/day, 7 days/week
3. historical perspective
4. use of advanced technology in hardware and software techniques
5. minimization of the "magical" aspects of computers

The hands-on laboratory has a psychological effect noted by others [3, 4]. This touching syndrome is known to waylay a few students who become addicted and have difficulty staying in school. The majority, however, are sensibly motivated.

An open shop laboratory is essential. The best lab is one run by students themselves. At the University of Missouri-Rolla, for example, the software laboratory was almost entirely installed, administrated, maintained, and policed by student volunteers.

The touching symbiosis not only leads to student involvement, but also animates historical points. Again, this book intends to emphasize perspective as an important ingredient in a minicomputer laboratory course. Who are the innovators of modern computing? Who are the Babbages and Von Neumanns? These people are the ones that contemporary students can relate to.

Again, I use the stack mechanism that exists on many minis as an example. Dijkstra published the first paper on stacks and recursion. He is usually given as a historical figure when relating stacks to the student. I tell my students Dijkstra's story about getting a wedding license and writing "programmer" as his vocation. The city officials knew of no such profession. Yet this occurred within the lifetime of most students.

Another motivation for using minicomputers in a software laboratory stems from the rapid incorporation of new technology. An example is the current availability of microprogrammable minis. These machines open an entirely new door to students and teachers. Are students being equipped to use microprocessors by the traditional computer science formula? Can they get close to hard core computing by growing up with the maxi in the computer center?

The minicomputer lab is definitely intended to rid students of their feelings of uncertainty when it comes to separating the function of hardware and software. Magic and other "Merlin-the-Programmer" effects are rapidly dissipated by touching the equipment. Development of basic concepts must be done in a specific manner, however, before one succeeds in dismissing witchcraft.

In short, students should be led through a progression of more and more sophisticated software. As a suggestion start with a barebones machine, graduate to a bootstrap system, a loader, a simple assembler, an editor, a two-pass assembler, a linking loader, a software engineered package or operating system, and finally, an entire system of file/editor/operating system. Each step represents a layer of software that separates man from machine. The layers are applied successively, however, so that they may be better understood.

The Academic Connection

This material is intended for a one-semester beginning course in Computer Science. A particular philosophy is behind the book: computing is a laboratory science. Engineering, physics, chemistry and biology are recognized laboratory courses. Now, Computer Science is clearly becoming a science in its own right with its own lore and terminology. Therefore, this material is intended to complement a hands-on laboratory containing a minicomputer as minimum "laboratory equipment."

The myth that computer science can be taught using a central large-scale computer is dying. There are two reasons for this:

(1) large-scale systems are too complex and sophisticated for purely instructional purposes requiring detailed and specialized knowledge, and
(2) the central facility cannot be modified as often happens when students experiment.

A large scale central computer does offer economy of scale and the level of sophistication most likely to be met in the real world. It does not, however, provide a very enlightening device for beginners. The distinction between software and hardware as well as the level of software sophistication result in a feeling of magic being practiced by instructors and machines. There is beauty in simplicity when teaching fundamental computer science.

Secondly, students must be able to make mistakes and crash the system in order to gain wisdown through experience.

It would be unimaginable to allow a select few to interrupt an institutional-wide computer in order to test an I/O routine. Yet this must be allowed somewhere in a computer science curriculum. In particular, I/O, interrupts, and console operation are possible by hands-on computing. These features are not provided by a secure central machine.

Why minicomputers? Once we have established the need to dedicate a machine and use it as a laboratory device, why should we select minicomputers over other machines?

The obvious explanation is cost. An excellent software laboratory can be equipped with minicomputer and peripherals for $20-50K (as of this writing). In addition, the author estimates the half-life of this estimate to be three years. This means that the same equipment should be available at half the cost in three years, one-fourth in six years, and so on until some plateau is reached.

The not-so-obvious reason is due to technology. Minicomputers incorporate the latest technological advances almost immediately. This contributes to their decreasing cost, but more importantly, technology influences computer design. Therefore, students familiar with a late-model mini are exposed to recent advances. This will be evident in this book which emphasizes late-model features such as stack processing, zero-register machines, and unified approaches to I/O.

Finally, a limitation of minicomputers actually turns out to be an advantage in education. This is addressing. A minicomputer typically has a short word (16 bits) and as such the instruction length is limited. Short words force the minicomputer designer to use an abundance of addressing modes: indexed, indirect, auto-increment, page zero, relative, and combinations of these. Thus the educator has a programmers' paradise full of addressing modes and students benefit from being exposed to many ways of addressing.

A criticism of all this might be: "In a few years a particular minicomputer recently purchased will be obsolete and these advantages will be lost." This is true if educators are not careful in selection of equipment. Either a trend setting or a flexible mini that can be reorganized through microprogramming and faster memory should be selected. In either case the lifetime/capital investment ratio of minicomputers is very attractive.

This book is about fundamental computer science. How-do-you-do-it is emphasized at the risk of too much boring detail. The authors have attempted to unify many concepts and state them as laws-of-computing or rules. Unfortunately computer science is too young to possess a unified theory of programming and computing. Therefore a pragmatic thread weaves through the book.

The course must be accompanied by a laboratory. The particular form of this lab will depend greatly upon the way a computer center/computer science department is organized. The following suggestions are given.

The level of sophistication should be kept low. Computer centers are tempted to keep building more and more whistles and bells into the software. Computer Science Departments are usually trying to increase the sophistication of existing software. The laboratory must be protected against empires whether they are software, hardware, or organizational. Student run labs work well: they are apolitical and do not trample on vested interests.

The software and hardware must streamline the user's work when the user is ready to go forward. In other words the text editor is useless during the first two to three weeks, but priceless at the end of the course. A system that produces punched paper tape at every step is invaluable in dramatizing the assembly process, but becomes painful near the end of the lab course.

References for Introduction

1. Executive Summary of Modern Data's 1973 Minicomputer Market Survey. Modern Data Services, Inc., 3 Lockland Avenue, Framingtham, Mass. 01701.

2. ACM Curriculum Committee. "Curriculum-68: Recommendation for Academic Programs in Computer Science," *Comm. ACM*, 11, 3 (March 1968).

3. Desautels, E. J. "A Small Computer in a Large University," *Proc. of ACM SIGPLAN Symposium on Pedagogic Languages with Small Computers*, ed. Bulgren and Schweppe. Lawrence, Kansas: University of Kansas, January 1972, pp. 28-32.

4. Konstam, A. H. "The Small Computer and Undergraduate Education," *Proc. of ACM SIGPLAN Symposium on Pedagogic Languages with Small Computers,* ed. Bulgren and Schweppe. Lawrence, Kansas: University of Kansas, January 1972, pp. 9-13.

5. Marsland, T. A. and Tartar, J. "A Course in Minicomputer Systems," *SIGCSE*, Vol. 5, No. 1 (February 1973), pp. 153-156.

PART I

Preliminaries

CHAPTER 1
BITS AND PROCESSES

The arithmetic capability of a minicomputer is much the same as large computers, that is binary in nature. The availability of floating point operations and special decimal arithmetic features is usually limited and considered an option. Therefore, the purpose of this chapter is to give a firm foundation in binary representation of information. Along with the methods of counting and conversion, we wish to impress upon the reader two very important concepts: the stored program concept and the notion of algorithm. These two ideas are elevated to the level of COMPUTER FUNDAMENTAL to show their relative importance. Thus, the reader should firmly grasp the following: binary numbers, stored program concept, and algorithm process concept.

1.1 Storing Data and Programs

If you were considering the design of a computer, what would you look for? How would you store information? How would numbers be entered and retrieved? These and other questions were considered by the early computer builders. These pioneers laid down foundations upon which modern digital computers are based. Two basic ideas we will discuss are:

(1) the storage of instructions which control the machine, and

(2) the storage of information (data) both using a binary system.

Both forms of storage use a binary system.

The idea of storing *data* to be manipulated dates back to the first mechanical computer (Charles Babbage in the 19th century). In addition, the early use of punched cards led to storing instructions within the operating machine as well as data.

John von Neumann is credited with reviving the *stored program* concept [1]. This is perhaps the most important single concept in computing. The sequence of instructions, or *program*, which controls the machine is contained within a computer in the same form as data. Indeed, these instructions may be mistakenly interpreted as data by another program.

> *COMPUTER FUNDAMENTAL* #1. The best way to control a computer is to store its instructions within the memory of the computer as if these instructions were data. This idea of taking instructions from memory (just as data is retrieved from memory), is called the *stored program* concept.

The stored program concept is basic to all concepts presented in this book and will be reiterated. The question posed for now is: how are data and programs stored and interpreted? What is the best method of "coding" to accomplish program/data storage?

The *binary* or base two number system has become accepted as the most economical, simple, and speedy implementation for digital computers [2]. Binary numbers represent a quantity using only zeros and ones. For instance, 10010 is a binary number representing the quantity 18. The term *bit* has been used since the late 1940's to mean "binary digit" [3]. The example, 10010 is a 5-bit number.

Since only zeros and ones are needed to write a binary number, a group of *two-state* switches are used to store the number within the computer, with one state representing 0 and the other representing 1. These two-state switches, called *flip-flops*, are considered less complicated and therefore more economical than ten-state switches needed to store a decimal digit.

Flip-flops are fast because a simple "yes" or "no" state is needed, and the reversal from "yes" to "no" is done in one simple step. Flip-flops are also faster when doing arithmetic operations as we will see later.

The big disadvantage of the binary system is interpretation by humans. We are reluctant to think in terms of binary since we are culturally biased toward decimal. Therefore, it is necessary that we become familiar with a new number system. However, because we prefer decimal we want to be able to convert from binary to decimal and from decimal to binary. Let's examine the binary number system and others commonly employed in computers and find ways to convert from these systems to decimal.