30

Assembly Language Programming for PDP-11 and LSI-11 Computers

an introduction to computer organization

Assembly Language Programming for PDP-11 and LSI-11 Computers

an introduction to computer organization

Edouard J. Desautels
University of Wisconsin / Madison

wcb
Wm. C. Brown Company Publishers
Dubuque, Iowa

wcb group
Wm. C. Brown *Chairman of the Board*
Mark C. Falb *Executive Vice-President*

wcb
Wm. C. Brown Company Publishers, College Division

Lawrence E. Cremer *President*
David Wm. Smith *Vice-President, Marketing*
E. F. Jogerst *Vice-President, Cost Analyst*
David A. Corona *Assistant Vice-President, Production Development and Design*
James L. Romig *Executive Editor*
Marcia H. Stout *Marketing Manager*
Janis M. Machala *Director of Marketing Research*
William A. Moss *Production Editorial Manager*
Marilyn A. Phelps *Manager of Design*
Mary M. Heller *Visual Research Manager*

Library of Congress Catalog Card Number: 81–70686

ISBN: 0–697–08164–8

Second Printing, 1983

Printed in the United States of America

2-08164-02

This book is dedicated to my wife, Jeannine, and my children, Francine, Nicole, and Philip, for their help and encouragement.

contents

preface

The goal of this book is to help readers who have some experience using computers to develop a deeper understanding of how they compute. The very popular PDP-11 computer (and its look-alike, the LSI-11) are used to illustrate the concepts involved.

This is a very substantial book. Don't let it overwhelm you. Among other virtues, it is essentially self-contained. This means you do not have to keep looking through the PDP-11 MACRO-11 reference manual (138 pages plus appendixes), or the PDP-11 Processor Handbook (468 pages plus appendixes), or the PDP-11 Peripherals Handbook (435 pages), or the PDP-11 Terminals and Communications Handbook (344 pages), to understand the topic at hand. One of the major advantages of having all this material integrated in one volume is that it can all be extensively cross-referenced in the index. The index is one of the most important parts of this book. The index is presented in four parts. The first two parts provide a quick reference to the PDP-11 instructions and the MACRO-11 directives. The third part consists of special character references, since these often have a critical role in the writing of computer programs. The fourth part consists of a standard but very comprehensive index. The instructions and directives are repeated in the index for the benefit of the readers may not yet appreciate these distinctions.

The material is sequenced so that the key topics can be studied even in the shortest typical quarter. Each instructor or reader can then select from among the many other topics for semester courses, honors courses, and independent (self-study) courses. Some examples of course organization are as follows.

1. Short course covering only fundamentals:
   chapters 1–9; 10 from beginning through Hard-Copy Terminals; 11; 12 (macros only); selected topics from other chapters, as desired.
2. Course for those familiar with some other machine language:
   chapters 1 (first two sections); 3–19.
3. Course emphasizing software concepts:
   chapters 1–9; 10 and 11 (selected topics); 12; 14 and 15 (selected topics); 18–19.
4. Course emphasizing hardware concepts:
   chapters 1–11; 12 and 13 (selected topics); 14; 15 (selected topics); 16–19.

Once the fundamentals have been covered, the instructor or reader can pick and choose among many essentially independent topics. Thus, in chapter 12 assembly-time conditional directives are treated independently of macros, and vice versa. Chapter 14 first discusses high speed input and output; the later sections on spooling, performance, analog data, processor traps, etc., can be included or not, as desired. Chapter 15, "Selected Topics" has a similar structure. Any of the topics may be included in a course or not, as desired.

The programming examples in the text are supplemented by programming examples in the solutions to the over 280 end-of-chapter exercises; the selected solutions, at the end of the book, cover questions that range from simple review exercises to thought-provoking problems.

We hope readers will find the bibliographic essay in the last chapter more interesting and useful than a conventional annotated bibliography.

The goal of this text is not to make readers experts at PDP-11 programming, but to lay the foundation that is essential for any one who wishes to pursue the study of computer organization, computer architecture, the design of compilers for high level languages, and many other facets of computing. This foundation will be very helpful for those who have personal computers of any kind and want to find out what makes them tick.

We assume that readers have programmed some computer in some high level language. This experience should include having used two-dimensional arrays and procedures or subroutines. Readers are therefore assumed to be familiar with the program development cycle: problem definition, data definition, algorithm selection, implementation, documentation, optimization, testing, and enhancement.

Readers must have access to a DEC PDP-11 or an LSI-11 or some system which uses either CPU (any model will do) and which provides some version of the MACRO-11 assembler. Some of the systems which could be used, besides those sold directly by DEC, include the Heath/Zenith H11 and the Terak 8510. We are convinced that merely reading about using machine language and assembly language is not a substitute for writing and running programs at this language level.

The exercises at the end of most chapters are important. Readers should take the time to at least think through an approach to solving the more demanding problems if time does not permit writing out and possibly programming the solutions. Some of the exercises introduce new ideas, new terms, new tools, and useful algorithms.

Material is presented in a carefully thought-out sequence. Readers are presented with relevant factual information and background, but not so much as to overwhelm them. Learning the details of a computer's instruction set can be as deadly as reading a dictionary if the presentation does not take human nature into account. We have partitioned topics into manageable units, interspersed with other topics, and we take advantage of opportunities to use simpler concepts before proceeding to more advanced ones. The presentation is also enlivened by newspaper and magazine articles on computer-based system malfunctions whose cause is very likely to be related directly to the topic at hand. Some of the articles discuss broader issues such as the legal challenges facing the marriage of computing and telecommunications services.

It is important to note that we first examine machine language before we approach assembly language. We recognize that the assembler is a very useful tool, and make extensive use of it in the text. But in the beginning, the assembler and the assembly process obscure what is going on at the machine level just as much as do high level languages. After a brief but illuminating venture into absolute coding in machine language, we can then appreciate and exploit symbolic programming as a means to gaining more insight into computing.

We do not advocate using machine language or assembly language as a general purpose problem-solving tool. These low level languages are like the ancient Latin and Greek that people were supposed to study as a means of gaining insight into the structure of contemporary English. For instance, the text often uses octal memory dumps, and we expect readers to be able to interpret these. These are just a means to an end—to understand what the computer was doing, stripped of all layers of software. Anyone who concludes that octal memory dumps are an efficient tool for debugging programs written in a high level language has missed the point.

Learning about computing can be enjoyable and addictive, particularly if you have sufficiently ready access to a computer so you can try all the "what if" situations that should be popping through your mind as you proceed.

Notes to the Instructor

Our practice has been to assign computer exercises beginning with the first week of class and to have one assignment immediately follow another continuously throughout the semester. Depending on the class size, the computing resources available, the magnitude of the assigned work, and the assistance provided in grading the programs in a timely fashion, from four to eight programming assignments are made in a 13–16 week semester.

Typically, the first assignment involves finding the laboratory, verifying that one can log in, that one can create and modify a small file using the on-line editor. This course has been conducted in batch-processing mode (using a self-service PDP-11); it has also been run with RT-11, RSTS, and UNIX. In those cases it is advisable to also have a bare PDP-11 for one-at-a-time hands-on use, if the multi-user system cannot be taken off the air.

A typical second assignment has students design and implement a small machine-language program, coded in absolute octal. The students use the instruction subset of chapter 2 and installation-specific instructions on loading and running this program. If a hands-on machine is available, then it can be keyed in at the console and run. For many students this may be the only chance they will ever have to completely control a computer, and this experience gives them a great deal of confidence.

The next assignment has students use a few of the PDP-11 addressing modes while writing their first MACRO-11 program. They are generally given a file with instructions on assembling, linking, and executing it. Then they are told to replace the innards with their own code and repeat the process. This is a painless way to get them introduced to seeing and

using tools such as a contingency post-mortem dump, a memory dump, and a register dump, with the setup required for their use (e.g., .MCALLs, .GLOBLs, invocations, etc.).

The next assignments can involve number representation transformations—say, in implementing a very simple two-function decimal calculator. Students may be asked to implement it using ASCII string-to-binary conversion or programmed BCD arithmetic. The more ambitious students might be asked to write the machine-language loading program (in assembly language) and to propose and implement extensions to it. Similarly, all of the other tools used in the class could themselves be the subject of assignments: students take pride in constructing their own dump programs. Many students have a mental block regarding the transformations from bits in a memory to octal digits on paper. Having them write a dump routine, perhaps adding ASCII interpretations to it, is very salutary. It further develops self-confidence and a sense of knowing what is going on, knowing that you could build up a usable system even if you had nothing but a bare machine. If hands-on access is possible, it is very instructive for students to estimate (by hand) what their program execution time should be, then devise an experiment to actually measure this time.

For the assignment just described, students would need to have access to the appropriate PDP-11 or LSI-11 Processor Handbook, or at least to the relevant instruction timing information. This textbook is otherwise sufficiently self-contained that it should not be necessary for students to buy either the Processor Handbook or the MACRO-11 Reference Manual. However, copies should be made available as references. Many of the problems given at the end of each chapter could be used as the basis for programming assignments.

Instructors sometimes regard teaching a course such as this as a way to give students an opportunity to work on very large projects. We think it is entirely inappropriate to assign large projects to be programmed in assembly language. That is the kind of experience students should be getting when using high level languages. The point of writing programs in machine and assembly language in this course should be to develop understanding of some fundamental ideas: indexing, indirection, manipulation of a stack, mastering the overflow bit, fielding an interrupt, etc. These objectives are more likely to be realized by assignment of smaller, more narrowly focussed problems, and more of them.

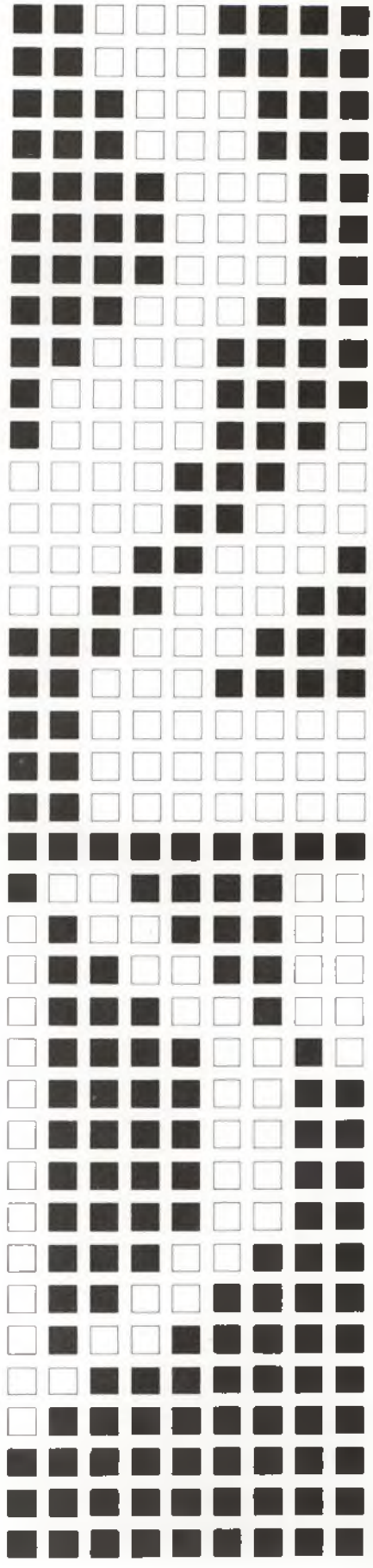
Careful readers will appreciate that by the time they reach the end of the text they will know how to design and implement each of the basic tools they have been using from the beginning. This should help dispel much of the mystery that shrouds computing.

I would like to thank the following reviewers for their constructive suggestions: William Bregar, Oregon State University; Linda Eshleman, Western Maryland College; Gordon Fish, Bucks County Community College; Bryan Hansche, Arizona State University; Alex Nichols, Cleveland State Community College; Michael Schneider, University of Minnesota; Abraham Silbershatz, University of Texas at Austin; and Larry Symes, University of Regina.

Edouard J. Desautels

1

the computing context



The goal of this book is to help you reach a better understanding of computing. This involves learning something about both computer hardware and computer software, since they must coexist and cooperate. Many approaches can be taken to learn more about computing. We happen to believe that taking the assembly language point of view leads to the fastest and firmest understanding of computing for someone already familiar with programming computers in some high level language. In turn, a firm grasp of assembly language programming must be built on understanding machine-language programming. We will work with machine language first, then with the lowest level symbolic language computers use, assembly language. We will see numerous programming examples written for a contemporary computer.

The experience of many students and teachers demonstrates that it is essential to actually write and execute low level programs to truly understand and appreciate the concepts involved. To this end, many exercises are suggested, and many actual computer runs are displayed and discussed.

Much of our contact with computers today probably has the computer so cleverly disguised and integrated in the application that most people are not even aware that computers are involved. For instance, an automated bank teller station and a microwave oven both use computers, but the ordinary user is probably not aware of that fact.

For many people the initial exposure to computers as computers involves using them to solve problems, usually by writing computer programs in some high level language such as Basic, Cobol, Fortran, Pascal, or PL/1. By design, these high level languages hide the low level details involved in actually getting the computing done. Learning about machine-language and assembly language programming, the low level languages used with computers, can help one begin to appreciate the marvelous intricacy of a computing system, its potential, and its limitations.

Introduction

We emphasize again our hope that you will actually try many of the things discussed in this book with the computer you have access to. Unless you work with a computer, you will learn about as much about computing as you would learn about cooking without going into a kitchen and trying things. However, the actual hardware configurations may vary greatly (make and model of computer; size of its main memory; kind and amount of secondary memory, such as magnetic disk, magnetic tape, paper tape, punched card reader, interactive video terminals, keyboard-printer terminals, etc.). Consequently, the support software may also vary greatly (the operating system software, utility programs, special programs designed to help one learn about computing, etc.). We therefore thought it was worth describing the various configurations and support systems that are in use.

Software

A particular model of computer can support widely different kinds of software packages. The usual way of categorizing software is depicted in figure 1.1. The abbreviation CPU (central processing unit) symbolizes

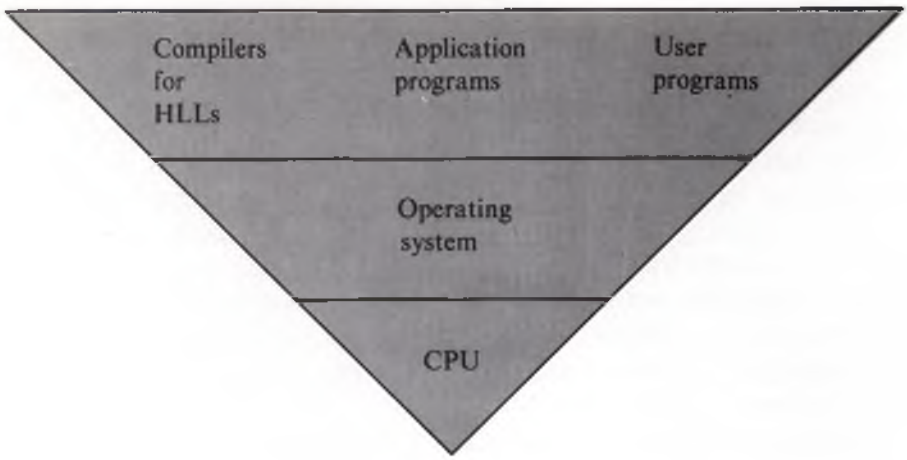


Figure 1.1 Hardware and software relationship.

the computer hardware, and HLL (high level language) represents languages such as Basic, Fortran, Pascal, etc. The software transforms the hardware into a usable operating system. It provides a way to create programs, file them away, retrieve them, and execute them. Similarly, collections of data can be prepared, filed away, then accessed by the programs that need them. The various services provided by an operating system include but are not limited to the following:

1. An editor to let you prepare and modify program and data files.
2. Compilers to translate programs written in high level languages into machine-language programs for a specific computer.
3. A filing system to support the storing and retrieval of files (a file may contain data or a program).
4. A librarian to manage files of subroutines.
5. Assemblers and loaders to allow the use of assembly and machine-language programs.

Some of the kinds of operating systems you may encounter are discussed next, in terms of the richness of the hardware they support.

Computers

A *computer* is a machine which processes information. This brief definition will be successively refined in an informal fashion until the details hidden by the words "machine," "processing," and "information" become very clear.

Using this definition, one could say that the telephone system or a photocopier are computers because they both process information—the first by transmitting voices and other audio information, the second by copying images. We will soon tighten our notion of "processing" to include more than mere transmission of information. In the meantime, it is worth noting that the telephone system and the better photocopiers could not operate without computers.

Naked Computers

A naked computer, sometimes called a bare machine, is one that has almost no accessories (peripheral devices) attached to it. Consequently, it has almost no usable software. Such is the case if you buy a single-board computer (SBC), which fits entirely on a single circuit board (as

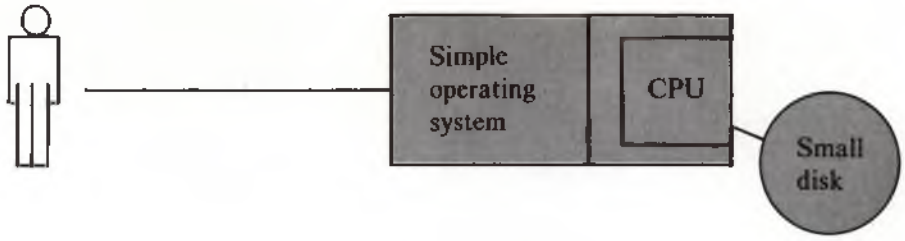


Figure 1.2 One-user system.

does an LSI-11, the smallest member of the PDP-11 family), and has no devices connected to it. It is not unusual for a student (or employee) to be asked by a professor in the sciences (or an employer in industry) to "use this mini or micro we just bought" to support some experiment (or industrial application). Such a bare machine is very hard to use to develop programs, so it will be even harder to use to learn about computing. But it can be done if you are determined.

The next step up is a computer with an adequate number of peripheral devices so you can easily develop software for it. A computer with a floppy disk (diskette) drive and either a video display terminal or a hard-copy keyboard-printer terminal provides a very nice system for learning. Some of these are marketed as "personal" computers. We will call them single-user systems. The most common support software for a single-user PDP-11 is the RT-11 operating system available from Digital Equipment Corporation (DEC), which manufactures the PDP-11.

Multi-User Systems

Slightly larger PDP-11s can use either the multi-user version of DEC's RT-11 or RSX11-M or the Western Electric (Bell Laboratories) UNIX operating system. Both of these support multiple users in "time-sharing" mode. The largest PDP-11s use either UNIX or DEC's RSTS operating system, which supports time-sharing. There are other operating systems used with PDP-11s, but they are not quite as "general purpose" as the ones we have named.

It is possible to adapt the RT-11 software so that it can support a punched card reader to provide a "batch processing" service. Although this is a single-user system, it can support a fairly high volume of user jobs even on a small computer.

Host and Target Computers

There is a particularly good way to take advantage of the best features of a single-user system and a multi-user system if you can't provide each student or employee with his or her own private, fully configured single-user system. This involves using the multi-user system for program preparation (which tends to take time because of the typing involved) and using the single-user system for execution of the previously prepared program, when it is advantageous or necessary to do so. The transfer of programs can be accomplished either by some removable storage device common to both systems (e.g., floppy disk) or by wire when both systems are linked by a physical communications line. When two systems are used to get work accomplished in this way, the multi-user system is often called the host system and the other one is called the target system. Of course, the target computer can be used by only one person at a time.

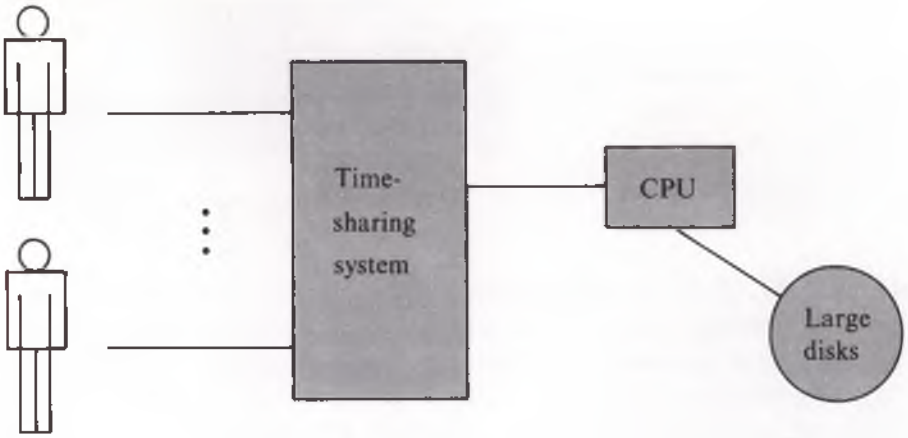


Figure 1.3 Multi-user system.

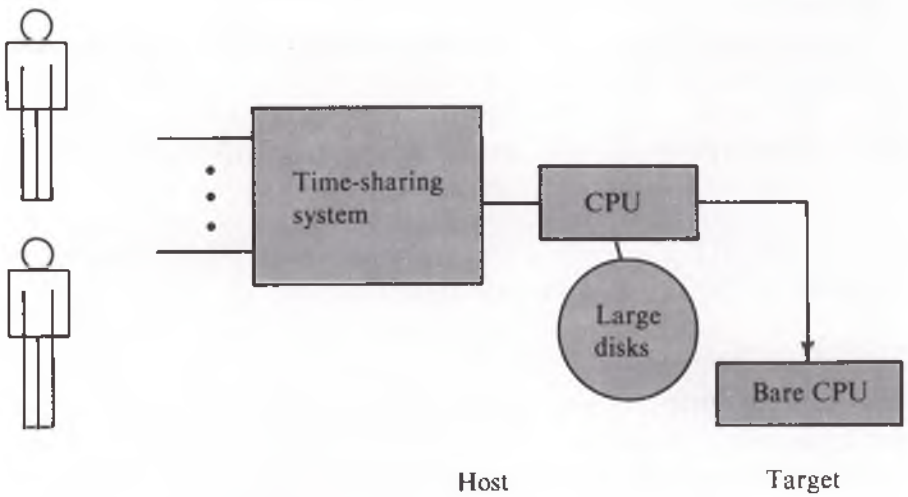


Figure 1.4 Host and target computers.

We will suggest different approaches to designing, debugging, or using programs, depending on what kind of configuration is involved. You should not take a narrow view of computing and assume that everyone else uses the same kind of system you do. Develop some sensitivity to the context in which the computing takes place. A technique which may be appropriate for use on a small system may not be advisable for use on a larger system, or vice versa.

Computing Machines

We are all familiar with calculators of one kind or another. They are the simplest kinds of computing machines. Calculators are similar to computers in many ways, and so it should be instructive to look more closely at a very simple calculator. Then we can begin adding features to it so that it becomes more like a computer. Let us begin by examining a simple four-function calculator, as shown in figure 1.5.

The purpose of the various keys is clear, as each one is labeled. It is helpful to group the keys according to their purposes:

data input (0,1, . . ., 9)
function (+,−,*,÷,=)
control (C,CE)

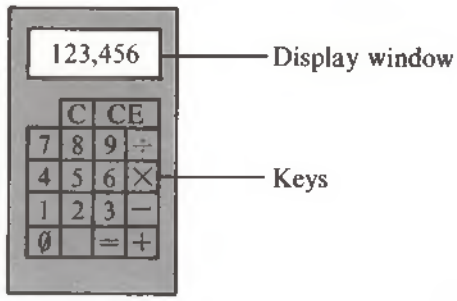


Figure 1.5 Simple four function calculator.

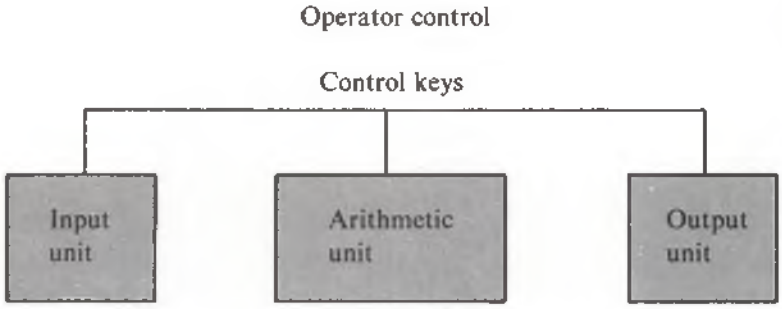


Figure 1.6 Abstract model of a calculator.

It is normal practice to omit the = key when counting the functions a calculator supports. The display window provides a way of verifying data upon entry and of examining results of a computation. We call this use *output*. The part of the calculator which performs the mathematical operations, the *arithmetic unit*, is hidden from view. Overall control of the calculator is provided by its human operator. The CE (clear entry) allows one to immediately erase the number being keyed in if an error is made. The C (clear) key also erases the current result (if any) being held within the arithmetic unit. As one uses the calculator, the intermediate result shown in the display window is also held within the arithmetic unit in a special place called the accumulator register, or simply the accumulator.

It is useful to picture the calculator in a more abstract way, as shown in figure 1.6. This "model" of a calculator turns out to be very close to that used in describing computers.

We can explain what is happening as the calculator is used by writing down a history or "trace" of the sequence of keystrokes, the visible effect on the display window, and the expected effect on the accumulator. Thus, in adding the three numbers 123, 634, and 221, we would have the trace, as shown in figure 1.7. Since it is tedious to write down each digit on a separate line, we will shorten our traces by writing all the digits of the same number on a single line, as shown in figure 1.8.

How could it be possible to write out in advance all the keystrokes necessary to solve a particular problem, record these keystrokes in some way, then modify the calculator so that it could be driven by these prerecorded keystrokes? Without getting into any complicated electronics, one can see that the 17 keys of the keyboard are, in effect, spring-loaded switches which are normally in the open (up) position. In the normal use of the calculator, only one of the 17 keys should be depressed at any one time. It is easy to imagine a pair of wires coming from each key, leading

Figure 1.7 Trace of all keystrokes.

Keystroke	Display	Accumulator
1	1	0
2	12	0
3	123	0
+	123	123
6	6	123
3	63	123
4	634	123
+	757	757
2	2	757
2	22	757
1	221	757
=	978	978

Figure 1.8 Tracing a calculation.

Keystrokes	Display	Accumulator
123	123	0
+	123	123
634	634	123
+	757	757
221	221	757
=	978	978

to a bank of 17 sensors. Each sensor can detect the presence or absence of a hole in a paper tape in a row capable of accommodating up to 17 holes (we can think of this as a 17-channel paper tape). As the tape moves, one 17-channel row passes the sensors at a time. We are, in effect, simulating the keystroking activity.

The punched tapes to solve particular problems can be described as program tapes. From the point of view of anyone with some experience in computer programming using high level languages, preparing these program tapes would be regarded as very low level programming. In fact, we can say it is machine-language programming. The program tape for the previous addition problem is shown in figure 1.9.

Each row of the program tape which has one hole punched in it (designated by an X) can be regarded as an instruction to the calculator. As the tape is advanced to the next row, a new instruction becomes available. The strange modification we have made to the calculator to allow it to be programmed is not that far removed from some computing machines in use in the 1950s. These kinds of programs are external to the computing machine or calculator, and each instruction is brought in, one at a time. Player pianos used the same kind of idea, with as many channels on the piano roll (instead of a tape) as there were keys and pedals on the piano. We should now revise our old model for calculators to reflect the addition of the automatic control mechanism provided by the addition of the sensors and the program tape. This gives the calculator a control unit which reduces the burden on the human operator to merely mounting the correct program tape, entering the necessary data at the appropriate time, and copying and reentering intermediate results as required. The revised model is shown in figure 1.10.

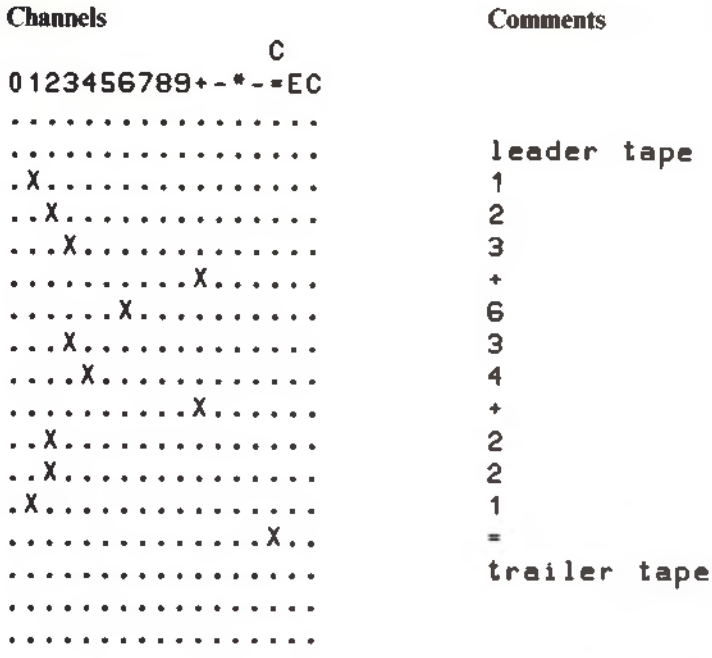


Figure 1.9 Program tape.

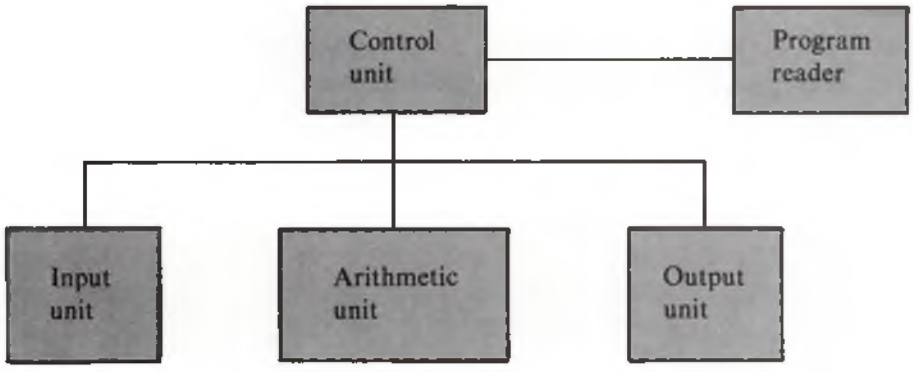


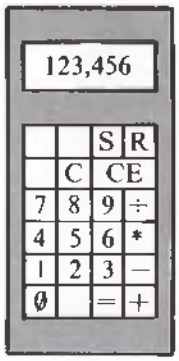
Figure 1.10 Abstract model of an externally programmed calculator.

Programs written for this calculator will be of little interest after they have been used once. This is primarily due to the fact that the data items are deeply imbedded in the program itself, and to solve the same kind of problem with new data items requires rewriting the entire program and preparing a whole new program tape. Obviously this is not an attractive proposition, and it severely limits the practicality of this kind of programming. Clearly, we need to find a better way of computing, some way of making the calculator more "general purpose" so that general-purpose program tapes can be prepared for it. To this end, consider adding two new keys to the calculator:

S (store at location n) and
R (recall from location n).

As we add these two keys, we also introduce a storage device capable of holding ten different numbers in ten locations, referred to by some calculator manufacturers as memories. We will speak of them as memory location 0, memory location 1, . . . , memory location 9. When the C key is depressed, each of the ten memory locations is set to hold the number 0. Whenever the store key, S, is depressed, the calculator expects one of

Figure 1.11 Four-function calculator with memory.



the digit keys to be pressed next, to indicate in which of the ten memory locations the number currently being displayed should be stored. Storing a number in a memory location destroys the number previously stored at that location. Once a number has been stored in a memory location, it can be used over and over again simply by pushing the R key and then followed by the single digit key which specifies the correct memory location. Recalling a number from a memory location does not change the number at the memory location. It merely makes a copy of it, placing the copy in the display. Our new calculator is shown in figure 1.11.

Let us look at the problem of calculating the surface area of a box of height H, length L, and width W. The formula is $2HW+2LW+2HL$. Suppose the dimensions of a box are 20 by 30 by 5 (H, L, W). Using the calculator without the storage unit, to get 2HW, we would have to key in 2 * 20 * 5 =. After recording the result, 200, by hand, we would get 2LW by keying in C 2 * 30 * 5 =. We could then take care of 2HW + 2LW by keying in + 200 =, and recording the result, 500. Finally, C 2 * 20 * 30 = provides the 2HL, to which we can add the partial result 500. Notice that we cannot prepare a program tape to solve this problem, because intermediate results had to be recorded by hand.

If we use a calculator with memory, the keystroke sequence could be:

```
20 S 0  (store the height H = 20 in location 0)
30 S 1  (store the length L = 30 in location 1)
5 S 2   (store the width W = 5 in location 2)
R 0     (get H)
* R 1   (get L and compute H * L)
S 7     (store H * L in location 7)
R 1     (get L)
* R 2   (get W and compute L * W)
S 8     (store L * W in location 8)
R 0     (get H)
* R 2   (get W and compute H * W)
+ R 8 (add L * W to H * W)
+ R 7 (add H * L * 2 giving final result)
```

Simply by having a program tape with two extra channels to accommodate the two new keys R and S, the entire calculation could be preprogrammed. There is no need to stop to record or enter intermediate results. An even greater advance is possible. A program tape could be prepared which deliberately left out the first three S operations. Then whoever needs to obtain the surface area of a box could store the dimensions in locations 0, 1, and 2 and use the program tape. We now have freed the program from having the data imbedded in it.

If we had many surface area calculations to perform, we could consider modifying the control unit so that instead of merely ignoring an unpunched row of the program tape, it would cause the tape-reading mechanism to pause, allowing the user to key in a number and then signal the tape reading to resume by pushing a GO button located either on the

keyboard or on the tape reader. Then a program which needed three numbers to perform a calculation could begin with the sequence Pause, S 0, Pause, S 1, Pause, S 2, etc. if the program expected to find its three operands in memory locations 0, 1, and 2, respectively. Furthermore, by splicing the end of the program tape to the beginning of the program tape, we would literally have an infinite loop. The program would keep on repeating itself so long as it was fed new sets of three operands. If we expect to use such a loop, the program should take care to either begin or end with a C command.

Bits

The program for adding three numbers is represented on the 17-channel tape we saw in figure 1.9. Each row of the tape could be represented by a 17-digit number. We have done this with figure 1.12. Each "X" (hole) position shown in figure 1.9 has been replaced with the digit 1, and each "." has been replaced by the digit 0.

When the digits of a set of numbers can only have the two values 0 and 1, these numbers are called binary numbers, and the digits within them are called *bits* (a contraction of "binary digit"). Thus the rows of our tape, which represent instructions to the calculator, can also be thought of as 17-bit numbers.

A 17-channel paper tape is very unwieldy. Could we get by with a narrower tape? That is, could we manage with fewer channels; or, equivalently, could we use fewer bits per row? We might expect to because each row has at most one bit set (a bit is said to be "set" or "on" if its value is 1; otherwise we say it is "reset" or "off").

Channels	Comments
C	
0123456789+-*÷=EC	
00000000000000000	
00000000000000000	leader tape
01000000000000000	1
00100000000000000	2
00010000000000000	3
00000000001000000	+
00000100000000000	6
00010000000000000	3
00001000000000000	4
00000000001000000	+
00100000000000000	2
00100000000000000	2
01000000000000000	1
00000000000000100	=
00000000000000000	trailer tape
00000000000000000	
00000000000000000	

Figure 1.12 Numeric equivalent of program tape.

Binary Codes

Consider what can be done with a 1-channel tape. A row can have either a single 0 or a single 1. Rows of a 1-channel tape could be interpreted as meaning:

0 represents "push no keys"
1 represents "push the only key"

A 2-channel tape would allow each row to specify one of four possible interpretations:

00 represents "push no keys";
01 represents "push key 1 of 3 keys";
10 represents "push key 2 of 3 keys";
11 represents "push key 3 of 3 keys."

Similarly, a 3-channel tape would allow support of a calculator with as many as seven keys.

Each extra channel doubles the number of distinct interpretations or meanings which can be associated with a row of bits. In the general case, if we had an n-channel tape (or an n-bit row), we could accommodate 2^n (two raised to the power n, or 2 * 2 * 2 * . . .* 2 with n 2s) mutually exclusive cases. We can examine a brief table showing selected powers of 2 (figure 1.13). We notice that a 4-channel tape would suffice for a 15-key calculator, and that the narrowest tape we could use for a 17-key calculator would have to have 5 channels. Observe that a 17-channel tape could handle a 131,071-key keyboard!

If we agree to use a 5-channel tape, we have to assign a 5-bit code to correspond to each key. The assignment of codes can be more or less arbitrary, so long as we use them consistently. We have assigned the codes as shown in figure 1.14.

Our new 5-channel program tape for adding the three numbers would now be the one shown in figure 1.15. Of course, the five sensors that "read" the holes on the tape can no longer be connected directly to the keys on the keyboard. They must feed the bit values they sense through a "black box" called a decoder. When the decoder receives a set of five bits from the five sensors, it sends out the appropriate signal to the keys, using at most one of the seventeen sets of lines to the keyboard. In effect,

Figure 1.13 Some powers of 2.

n	2^n
1	2
2	4
3	8
4	16
5	32
6	64
. . .	. . .
10	1,024
. . .	. . .
17	131,072
. . .	. . .
20	1,048,576

Key	Code
1	00001
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000
9	01001
0	01010
+	01011
-	01100
*	01101
÷	01110
=	01111
CE	10000
C	10001
no key	00000

Figure 1.14 A binary code table.

00000	leader
00001	1
00010	2
00011	3
01011	+
00110	6
00011	3
00100	4
01011	+
00010	2
00010	2
00001	1
01111	=
00000	trailer
00000	

Figure 1.15 Five-channel program tape.

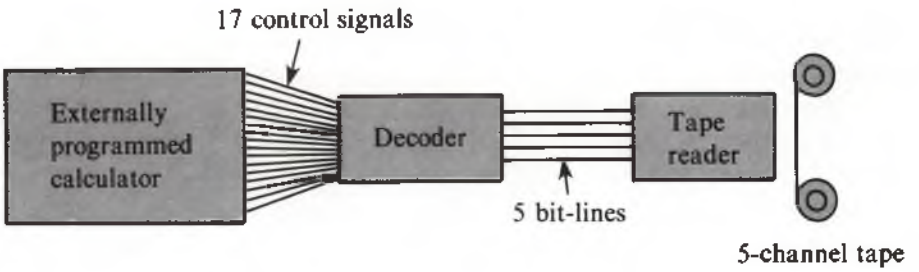


Figure 1.16 Calculator with decoder.

the decoder does a table-lookup using the table in figure 1.14. The act of going from a compact binary code to the selection of a particular item associated with that code is called *decoding*. The original association of a particular item with a code word is called *encoding*.

Teletype TTY

Binary codes are used in many applications, not just those involving computers. One important use is in telecommunications. A famous device in that field is the ASR-33 Teletype terminal, often called the TTY, manufactured by the Teletype Corporation, pictured in figure 1.17.

Over 500,000 of these have been built. The automatic send and receive ability of the ASR derives from its use of punched paper tape. A person can prepare a message on the paper tape with the TTY "off-line"—that is, not connected to the communications network. Then the paper tape can be read by the TTY in "on-line" mode, at the appropriate time, thereby reading the binary codes from the paper tape and broadcasting them to all other TTYs connected to the network which happen to be in on-line mode. A receiving TTY can elect to record the message it receives by using its paper tape. Such a communications network is used by many

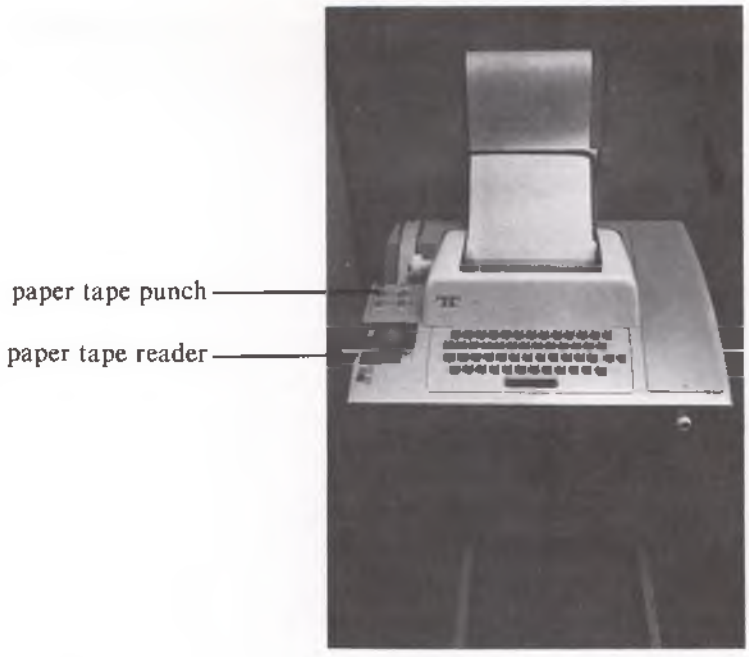


Figure 1.17 The ASR-33 Teletype.

national weather services to report and record hourly weather observations. The TTY is the precursor of the computer terminal, teletypewriter, the CRT (cathode ray tube) or VDT (video display terminal), and other keyboard and display devices we will discuss later. One thing they all have in common is their use of binary codes. Paper tape is being replaced by magnetic tape and magnetic disks, but the same binary codes are used on the magnetic media.

ASCII

The binary code used on the ASR-33 Teletype has been adopted as the American Standard Code for Information Interchange, ASCII (pronounced "ask-kee"). The International Standards Organization has also adopted it, and it is sometimes referred to as the ISO code. The ISO code can differ from ASCII in that some printing (i.e., graphic as opposed to nongraphic) characters vary from country to country. Thus, in countries that need it, the British monetary pound sign (£) is available as an ASCII graphic character, and other characters can be used in its place in other countries.

Figure 1.18 provides the ASCII codes for all 128 possible 7-bit combinations. The 33 nonprinting codes are included. Note that the use of standard codes on computers is not required by law. There are other binary codes in use in computing as well as in telecommunications.

Figure 1.19 shows how the 7 bits of an ASCII character are punched on an 8-channel paper tape. The eighth (leftmost) channel, b_7, happens to be unused here, so it is unpunched. It can be used to accommodate a check bit. If it is used, this check bit is usually called a parity bit (see exercise 1.9). Because different check-bit schemes can be used, it is possible to have two different bit patterns for the same ASCII character. For instance, see figure 1.20. We will have more to say about binary numbers and codes in the following chapters.

b_6	0	0	0	0	1	1	1	1
b_5	0	0	1	1	0	0	1	1
b_4	0	1	0	1	0	1	0	1
b_3-b_0								
0000	nul	dle		0	@	P		p
0001	soh	dc1	!	1	A	Q	a	q
0010	stx	dc2	"	2	B	R	b	r
0011	etx	dc3	#	3	C	S	c	s
0100	eot	dc4	$	4	D	T	d	t
0101	enq	nak	%	5	E	U	e	u
0110	ack	syn	&	6	F	V	f	v
0111	bel	etb	'	7	G	W	g	w
1000	bs	can	(	8	H	X	h	x
1001	ht	em	)	9	I	Y	i	y
1010	lf	sub	*	:	J	Z	j	z
1011	vt	esc	+	;	K	†[	k	†{
1100	ff	fs	,	<	L	\	l	|
1101	cr	gs	-	=	M	]	m	}
1110	so	rs	.	>	N	^	n	
1111	si	us	/	?	O	_	o	del

Control characters‡ Graphic characters

Restricted ASCII character set

Figure 1.18 ASCII codes.

†Undefined in the ISO Standard. Symbols vary among different national versions: this is the ASCII version.

‡The meaning of control characters for data transmission

Layout characters:

`bs` backspace
`ht` horizontal tabulation
`lf` line feed
`vt` vertical tabulation
`ff` form feed
`cr` carriage return

Ignore characters:

`nul` null characters
`can` cancel
`sub` substitute
`del` delete

Separator characters:

`fs` file separator
`gs` group separator
`rs` record separator
`us` unit separator

Escape characters:

`so` shift-out
`si` shift-in
`esc` escape

Medium control characters:

`bel` ring bell
`dc1-dc4` device control
`em` end of medium

Communication control characters:

`soh` start of heading
`stx` start of text
`etx` end of text
`eot` end of transmission
`enq` enquiry
`ack` acknowledgment
`nak` negative acknowledgment
`dle` data link escape
`syn` synchronous idle
`etb` end of transmission block

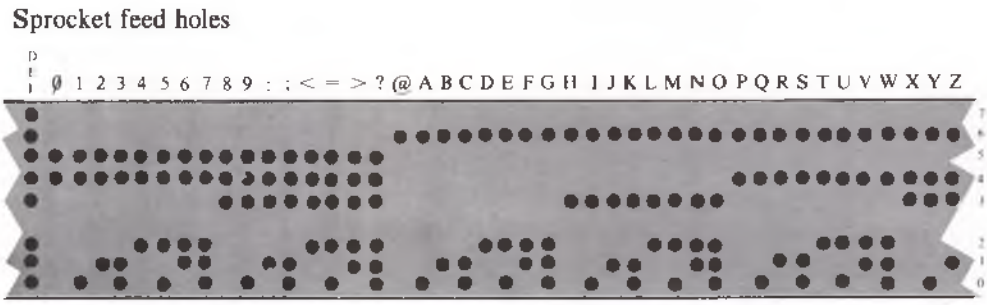
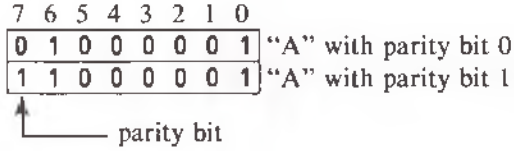


Figure 1.19 ASCII codes on punched paper tape.

Figure 1.20



Programmable Calculators

Programmable calculators are now quite common. Of course, none of these programmable calculators uses for their programs the primitive paper tape mechanism we described earlier. Programmable calculators store their programs in an electronic storage unit within the calculator. This storage unit, often called program memory, stores each program instruction as a number, and each such instruction is stored in a different memory location. Under normal circumstances, as the calculator's control unit reads out an instruction from its program memory, it will expect to find the next instruction it needs in the next consecutive memory location.

At this point we could enter into a detailed discussion of a particular programmable calculator. However, since that is a sizable undertaking in its own right, we will reach our primary objective of understanding computers more rapidly by now focusing our attention on a simple hypothetical computer. We will then be ready to begin work with a simple, but real, computer.

Analog and Hybrid Computers

This book is concerned solely with electronic digital computers. There are other kinds of computers, not necessarily digital, not necessarily electronic. Mechanical and hydraulic analog computers have been built. However, since almost all computers in use today are the electronic digital computer type, the single word "computer" has come to be identified with that type exclusively.

An analog computer uses a physical representation of the quantities it will manipulate. Physical quantities can be thought of as taking on continuous, smoothly changing attributes, such as temperature, length, loudness, etc. The quantities which an electronic digital computer manipulates are discrete; they change only by discrete increments. Furthermore, in an electronic digital computer, quantities bear no particular relationship to physical attributes. The qualifier "digital" emphasizes the characteristic of handling discrete numeric information (as on the ubiquitous digital readout clock, which competes with the traditional analog readout clock). The wooden or plastic slide rule is a simple example of

Figure 1.21 Analog versus digital. © 1973 by Sidney Harris-American Scientist Magazine.

an analog computer. It uses physical distances to represent the variables involved in multiplication or division. An electronic analog computer uses electronic circuits to represent relationships between electrical quantities expressed as voltages (V), currents (I), and resistances (R). A simple electrical analog computer to perform division can be built based on Ohm's Law: V = I * R. If you have a battery providing a constant voltage V, and a potentiometer which allows you to vary the resistance R at will, you can connect these to an ammeter to display the resulting amperes as the current I. Since I = V/R, this will perform division of the constant V by R.

To solve a problem with an analog computer requires finding a set of mathematical equations which represent a model for the problem, and implementing an electronic circuit whose behavior reflects that predicted by the mathematical model. The circuit is usually realized by using a "patch board" to interconnect various parts of the analog computer.

A hybrid computer is one which combines the features of electronic digital computers with those of analog computers. This makes the hybrid more useful, but also more expensive.

Analog computers and hybrid computers can solve certain classes of problems in real time. That is, once they are set up to solve a specific problem (which may take a lot of time), a change in an input variable elicits an immediate response from the computer. However, analog computers and hybrid computers do not lend themselves to handling exact

arithmetic, which is a necessity in business data processing. Nor can they produce the arithmetic precision required for most scientific and engineering uses. They are not as general purpose as the electronic digital computer, so they are not so widely used, and we will not be discussing them further.

We might just mention an interesting historical footnote. In the 1940s and early 1950s the word "computer" was used to refer to a person whose profession was the full-time operation of calculators. It was not unusual in those days for a large insurance company or a government agency to have hundreds of these human computers employed in large rooms, each working at a calculator. The word "computer" has now come to mean the computing hardware we see about us. The occupation previously called "computer" has evolved into many kinds of occupations, such as computer operator, data entry clerk, computer programmer, etc.

Summary

We have discussed computers in general terms, characterizing the richness of the hardware configuration (bare machine versus more fully configured computer systems). We have also pointed out some differences in the kinds of software that computers can be endowed with (single-user systems versus a variety of multi-user operating systems). What differences these distinctions can make will occupy us as we proceed.

The mechanics of computing with a simple calculator were introduced. We saw how calculator operations can themselves be represented as numbers (e.g., 17-bit or 5-bit commands). The process of automating the calculator led us to the desirability of storing and retrieving numeric operands under program control. This made it possible to make the calculator's programs data-independent, and thus more useful for general purpose work than would otherwise be the case.

The use of *binary numbers* to encode commands and data was introduced. We had a brief look at the very popular ASCII character set. We will see more of it later. Finally, the distinction between our kind of computers (electronic digital computers) and other kinds was touched upon.

Exercises

1.1 Using the trace for the surface area program as a model, rewrite the program to use fewer memory locations.

1.2 Write a program for the externally programmed calculator with memory so that it accepts a polynomial of the form $a+bx+cx^2$ and allows you to evaluate it for selected values of x (hint: store the values of a, b, and c).

1.3 Most typewriters have keyboards which support the printing of approximately 90 symbols. How many bits are required to encode 90 different symbols?

1.4 The Baudot code used for international telegraphic service (TELEX) uses 5 bits. How does it manage to encode a character set of some 60 symbols? The technique used is a very simple one you can recreate after a little thought. It is a very powerful technique well worth remembering.

1.5 Teletypes use a 7-bit code for domestic telegraph or TWX service. How large a character set can such a code support?

1.6 The computer card we now call the "IBM" card originated in the late 1800s. It was designed by Herman Hollerith to record only numeric information to aid in processing census data. It provides on each card 80 columns of 12 rows. The 12 rows are identified with the symbols +−0123456789, so in each column, at most one hole is punched to represent a sign or a decimal digit. In our terms, the punched card uses a 12-bit code, the Hollerith code. If you can examine a punched card, describe how non-numeric characters are encoded.

1.7 A piano has 88 keys and 3 pedals. What is the shortest binary code length that can be used to encode musical compositions for the piano? Why is this so?

1.8 Amateur radio operators use a code called the Morse code. It can be thought of as a binary code, but it is quite different from any we have seen. See the description of the Morse code in exercise 10.4 and contrast it with other binary codes.

1.9 The *parity* of a group of bits is said to be odd if the group has an odd number of 1-bits; otherwise it is said to be even. Forcing a check bit to be an odd parity bit means setting the check bit so that the group it is part of comes to have odd parity. With this in mind:

(a) Indicate the parity of (i) 1100; (ii) 100010; (iii) 011001.

(b) Attach a check bit to the left of each of the above groups, forcing them to have odd parity.

(c) Repeat (b), forcing even parity checks.

(d) Write your initials in ASCII

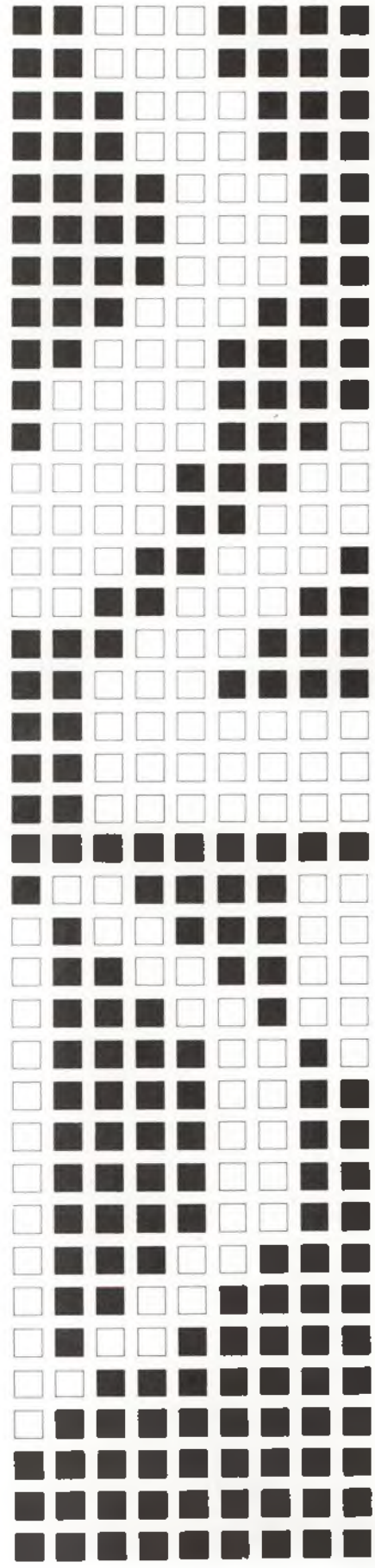
  (i) with a 0 check bit.

  (ii) with an odd parity check bit.

  (iii) with an even parity check bit.

2

a simple hypothetical computer



A machine-language program is a series of instructions and data encoded as numbers, designed to interact with a particular computer to solve a specific problem or class of problems. Each of these numbers must be stored in the computer's memory in order to execute the program. When the computer's control unit needs an instruction, the control unit will fetch a copy of the desired instruction from memory and place it in a special location which is used to hold the current active instruction, the *instruction register* (IR). A "register" is similar to a memory location in that they both store numeric information. Some registers are used for a special function, as is the case with the instruction register.

Since programs can be stored in any part of the memory, the control unit needs to know where the next instruction is located in memory. This memory location, or memory address, is kept in the control unit's *program counter* (PC). This pair of registers, the *PC*, and the *IR*, plays a central role in the operation of the entire computer. We can paraphrase the role of the IR as reminding the control unit about "what is to be done now," while the PC points to a memory location which contains the instruction which will provide "what is to be done next." The model for our simple hypothetical computer is shown in figure 2.1.

This computer uses an accumulator register, AC, just as our calculator did, and it has a somewhat more sophisticated arithmetic unit called the arithmetic-logic unit, ALU. It is quite similar to the model we had for our externally programmed calculator, but the differences are important, and they illustrate the fundamental property common to all modern digital computers. The program *must be stored* in the computer's memory in order for the computer to operate efficiently. From now on we will be working with "stored program computers."

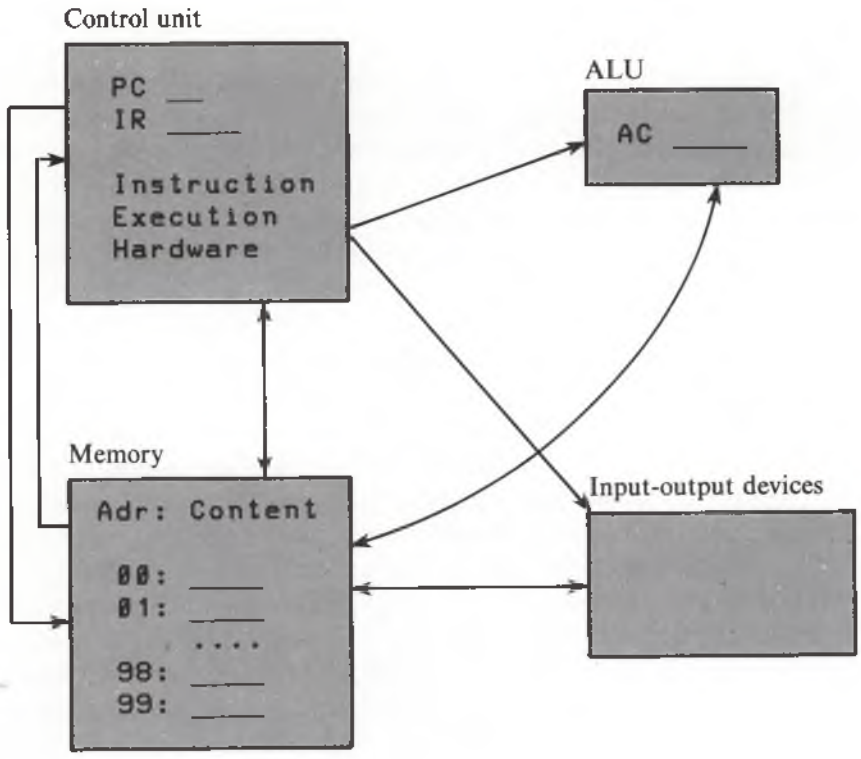


Figure 2.1 A hypothetical computer.

Figure 2.2 Instruction subset 1 for hypothetical computer.

Instruction Name	Code	Function	Comment
LAC	01	(m)→AC	Load the AC from memory
STO	02	(AC)→m	Store the AC in memory
ADD	03	(AC)+(m)→AC	Add memory to the AC
SUB	04	(AC)-(m)→AC	Subtract memory from AC
MPY	05	(AC)*(m)→AC	Multiply AC using memory
HLT	00	Halt	Stop the control unit

The Instruction-Fetch-Execute Cycle

The basic pattern of operation involving the PC and IR is important enough to warrant using a special phrase to describe it: the *instruction-fetch-execute cycle.* It consists of the following series of steps:

1. Use the PC to locate the next instruction in memory.
2. Copy the new instruction into the IR.
3. Update the PC.
4. Decode and execute the instruction in the IR.
5. Repeat from step 1.

The specific capabilities of a computer are determined by the complexity of its control unit and its arithmetic-logic unit (ALU). In the case of our hypothetical computer, a subset of the capabilities is summarized in figure 2.2. The notation (m) → AC means that the two-digit address represented by "m" is used to locate the number at memory location m, fetch a copy of it, and copy it into the AC. Whenever (. . .) appears on the left-hand side of a →, we expect either to fetch a copy of a number from memory, using the number . . . given as its address, or to fetch a copy of the number from the register specified by . . . . Similarly (AC) → m is read as: fetch a copy of the content of the AC and copy it into memory at location m.

The computer's accumulator register, AC, is similar to the one in the calculator. When directed by the control unit to do so, the ALU will take the number currently held in the AC and add, subtract, or multiply it by some other number, as found in the indicated memory location, and place the result in the AC. Unlike the calculator's AC, the computer's AC can only obtain its second operand *from memory.* Therefore, somehow the control unit must be provided with instructions which specify not only the arithmetic function to be performed, but also the memory address of the number which is to be combined with the number already in the AC.

Each instruction is four digits long and has two parts: a two-digit operation code followed by a two-digit memory address (symbolized by "m" in figure 2.2). The left two digits are called the opcode field (short for operation code), and the right two digits are called the address field.

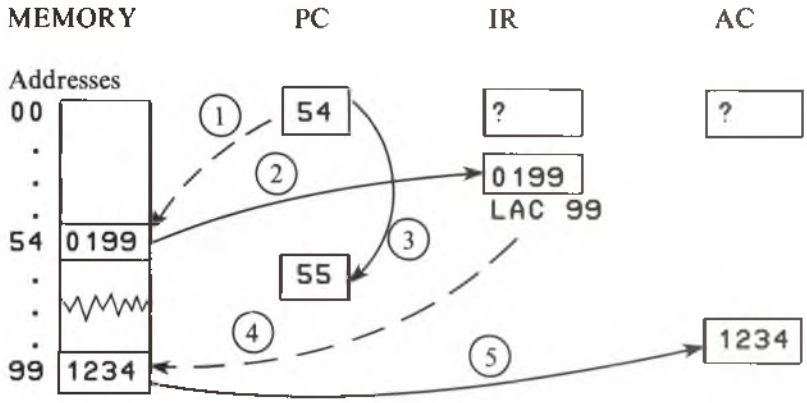


Figure 2.3 Executing an LAC instruction.

With a two-digit address being used, we can number the memory locations 00, 01, 02, . . . , 97, 98, 99. We will assume that each of these one hundred memory locations is capable of storing a signed four-digit number. This means, then, that any memory location can store either a signed four-digit data item or a four-digit instruction.

How can the computer tell if an item in memory is a data item or an instruction? The computer has no way of telling them apart! It is only when a number is fetched from memory that its role becomes clear to the computer. If the number is directed to the IR, then it will be used as an instruction. If the number is not sent to the IR, then it will not be used as an instruction. The computer has no other way of distinguishing between the "data item" 0300 and the encoded "instruction" 0300, which means *ADD using the number at memory address 00.*

We are almost ready to construct a machine-language program, but a careful reading of the instruction set we have to work with (figure 2.2) is in order. Let us consider the following cases.

a. The number 0199 has just been copied into the IR, as illustrated in figure 2.3, steps 1 and 2. Each step in the execution of the LAC instruction is numbered. The events are depicted by dashed lines to represent use of an address to locate an item in the memory and by solid lines to indicate moving an item into or from memory. What happens next?

According to figure 2.3, we have a code 01, which stands for an LAC, and a memory address 99. So the content of memory at address 99, symbolized by (99), will be copied into the AC (steps 4 and 5). The content of location 99 will not be disturbed. If the number 0199 had come from memory location 54, then immediately after fetching (54) and copying it into the IR, the control unit would update the PC to location 55 (step 3) to be ready to fetch the next instruction from the next consecutive memory location. Updating the PC in this case means adding 1.

b. The number 0234 has just been copied into the IR (see figure 2.4). After updating the PC (incrementing it by 1), the control unit interprets code 02 to mean "store the content of the AC" in the indicated memory location, which has the address 34. The (AC) is not changed. Code 02, the STO operation (figure 2.2), is the only one in this computer that can change the content of a memory location.

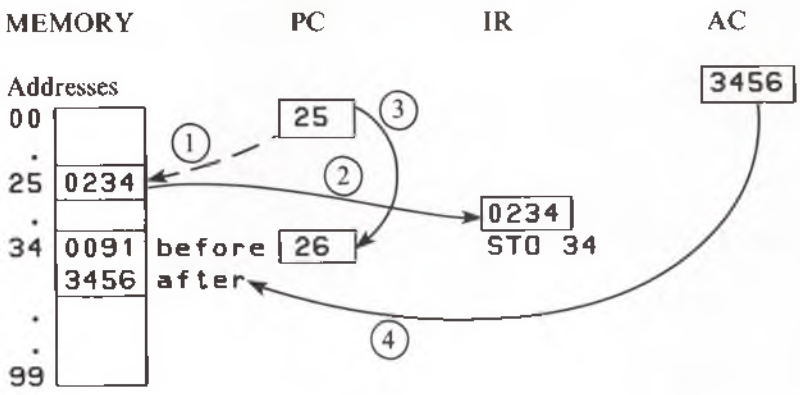


Figure 2.4 Executing an STO instruction.

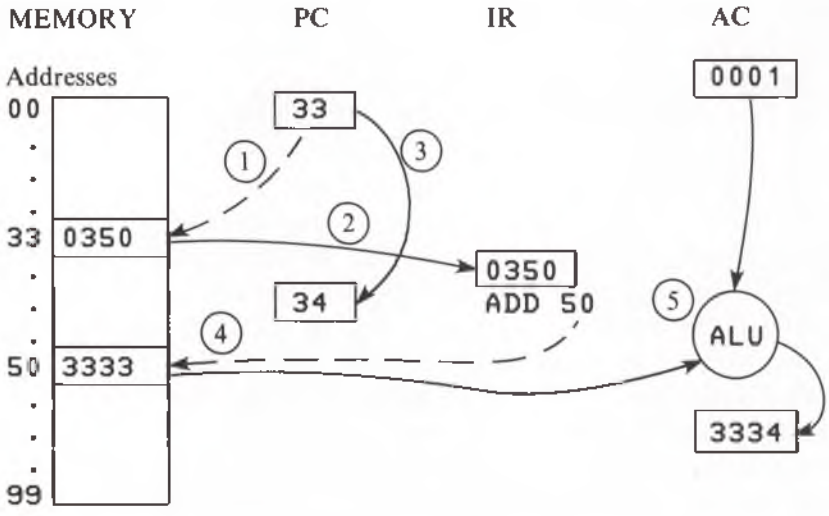


Figure 2.5 Executing an ADD instruction.

c. If an ADD instruction, say 0350, is copied into the IR, then after updating the PC (again, by +1), the control unit will interpret the 03 as an ADD function, fetching a copy of the number found in location 50, add that number to the number found in the AC, and place the result of the addition in the AC. The previous content of the AC will, of course, be destroyed, while the number in memory will remain undisturbed. The execution of this ADD instruction is shown in figure 2.5.

Instruction codes 04 and 05 for subtraction and multiplication follow a similar pattern. Keep in mind that in a subtraction, the operand in memory is subtracted from the operand in the AC, not vice versa. As always, the new result is placed in the AC. In the case of a multiplication, we must restrict ourselves to using numbers whose product will fit in the AC; otherwise we will get the wrong result.

The purpose of the HLT (halt) instruction, code 00, is clear: stop the computer. That is, the control unit is directed to refrain from fetching the next instruction until it is directed to do so (usually through manual intervention on the computer control panel). The two other digits of a HLT instruction have no significance other than to ensure that the HLT instruction will be four digits long.

Address	Content	Comment
00	0123	Item 1
01	0634	Item 2
02	0221	Item 3
...		
10	0100	(00)→AC
11	0301	(AC)+(01)→AC
12	0302	(AC)+(02)→AC
13	0000	Halt
...		

Figure 2.6 A small machine-language program.

The First Machine-Language Program

We have now set the stage for writing machine-language programs. Consider the simple problem of adding the three numbers 123, 634, and 221. If we were using a calculator, we would key each number followed by a + and a final =. With a computer, we have no option but to first place these numbers in memory. So our first problem involves finding three unused memory locations. Since all 100 memory locations are currently free, we can pick any three locations. We could pick three at random, but let us arbitrarily pick three consecutive locations, say 00, 01, and 02.

So far we have the following situation in memory: locations 00–02 will be used for the three operands. If we were using an externally programmed calculator, we would now begin punching holes in successive rows of the paper tape to represent the instructions we want; the paper tape would become the program tape. With a stored program computer, we have to store the program somewhere in the computer's memory, so we have another decision to be made *before* we can write the program: where shall the program be placed? Clearly, we can place it anywhere in the 100 memory locations, provided that it does not conflict with other uses of the memory locations. Since the only cells in use are 00, 01, and 02, let us arbitrarily choose to begin placing the program's instructions at location 10. We now choose instructions to load the first number into the AC (01 00), to add the second number using the AC for the running total (03 01), another instruction to add the third and last number into the AC (03 02) and a final instruction to halt the computer (00 00), since we have finished adding the three numbers. After we manage to place these instructions and the three operands in memory, our use of memory is shown as the program in figure 2.6.

In order to have the computer execute (perform) the program in figure 2.6, we have to place the numbers 123, 634, 221, and also the instructions 0100,0301,0302, and 0000 in the two sections of memory beginning at 00 and 10, respectively. Note that the addresses next to the data items, and those next to the instructions are not stored in memory. These addresses are used while the other things are being stored. It is important to avoid confusing *the address of an instruction* (say address 10 for instruction 0100) with the address used *in* an instruction (the address 00 in the instruction 0100).

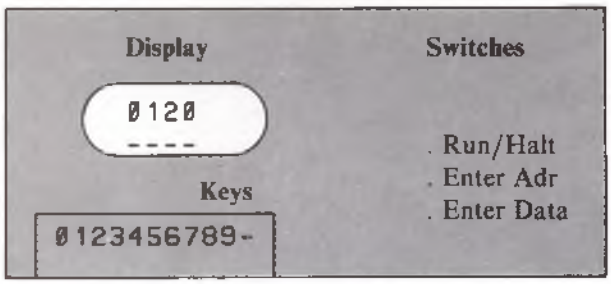


Figure 2.7 Simple control panel.

How can we get this program and its data into the memory? Every computer has some kind of an operator control panel or control console. Some of these control panels have switches and keys through which one can halt the computer, key in a starting address, then key in a series of numbers to be stored in consecutive memory locations, beginning with the first number going to the starting addressed location, and the following numbers going into consecutive memory locations. These numbers are either data items or instructions; it is of no consequence to either the control panel or the memory.

When all the data items and instructions have been keyed into memory, it is possible to display the content of individual memory locations on the control panel display. In this way we can verify that the program was properly keyed in. Finally, to start program execution, we have to force the program's starting address into the PC. This is accomplished by using the control panel to key in the starting address, then pushing a "RUN" switch. This forces the address just keyed in to be copied into the PC, and the instruction-fetch-execute cycle is then initiated. A simplified computer control panel is shown in figure 2.7.

Figure 2.8 Console emulation trace.

```
@00/1111 0123
@01/0000 0634
@02/9999 0221
@10/1234 0100
@11/0000 0301
@12/4150 0302
@13/6140 0000
@10G
```

The process of placing a program in memory is called program loading. Some computer control panels have no displays, switches, or buttons to permit the kind of program loading which we just described. In those cases, such computers usually have within their control units some special feature which makes it possible to use a designated teletypewriter, called the control console terminal, as if it were a computer control panel with the capabilities we just described. The teletypewriter is then said to function as a control panel emulator. When used in this special mode of operation, various keys of the teletypewriter's keyboard are temporarily given special meanings. The typing required to enter our program using this technique is shown in figure 2.8.

The characters which are underlined were typed by the computer. The "@" is a prompt to which we respond by typing an address. If we then type a "/", the computer responds by typing what is then the current content of memory at that address. If we type a number, it will be stored in memory, replacing the number just displayed. The last line that was typed is address "10" followed by a "G"; this is a "GO execute" signal, the preceding address to be used as the start address. At that point the teletypewriter resumes being an ordinary terminal, while the program just typed in has its execution initiated.

IPL and Boot

Many computers have a front panel switch labeled "IPL," for initial program load, or labeled "Boot," for bootstrap. If the computer is stopped—that is, its instruction fetch-and-execute cycle has been halted—then pushing an IPL or Boot switch forces a predetermined address into the PC, and the instruction-fetch-execute cycle is resumed. The IPL/Boot address forced into the PC is the first address of a permanently recorded machine-language program. This program, when it is executed, issues an "input" instruction which copies another program into memory, from some predefined location on a peripheral storage device (such as a disk). The IPL/Boot program then transfers control to the program it has just read in. Such a sequence of events eventually leads to user programs which are in machine-readable form, on some device the computer can control, being read into memory and gaining control of the CPU so they may be executed.

As a general rule, in a computer system being used for routine data processing or general purpose computing, the IPL/Boot switch is usually the only switch on the computer's front panel that the computer's operator uses. We, of course, want to know how we could run even without an IPL/Boot switch.

Program Execution

After having loaded the program in figure 2.6 into memory, at the specified locations, we manually force the address of the first instruction, location 10, into the PC, using the console control panel or the control panel emulator. The execution of the program is thereby initiated, and we can follow its progress by hand simulation of the instruction-fetch-execute cycles, recording significant events in the trace history shown in figure 2.9.

```
1. With (PC)=10, fetch (10)
2. (10) = 0100 → IR
3. Set PC = 11
4. Execute code 01, using
address 00
      (00) = 0123 → AC

1. With (PC)=11, fetch (11)
2. (11) = 0301 → IR
3. Set PC = 12
4. Execute code 03, using
address 01
        (AC)+(01)→AC
        0123+ 634=757→ AC

1. With (PC)=12, fetch (12)
2. (12) = 0302 → IR
3. Set PC = 13
4. Execute code 03, using address 02
        (AC)+(02)→AC
        757 + 221= 978→AC

1. With (PC)=13, fetch (13)
2. (13) = 0000 → IR
3. Set PC = 14
4. Execute code 00, no address needed
        Halt
```

Program Address	Content
00	0123
01	0634
02	0221
...	
10	0100
11	0301
12	0302
13	0000

Figure 2.9 Instruction-fetch-execute cycles.

Figure 2.10 Trace table.

PC	IR	OP	AC	MEM-ADR	OPERAND	Comment
10	?	00	?	?	?	Computer stopped Push RUN switch
11	0100	01	0123	00	0123	LAC
12	0301	03	0757	01	0634	ADD
13	0302	03	0978	02	0221	ADD
14	0000	00	-	-	-	Halt

Since such a trace history is very lengthy, we can abbreviate it using a trace table which shows the state of every significant item, immediately *after* execution of the indicated instruction, as depicted in figure 2.10.

The "?" represents items which are either unknown or irrelevant. The "-" means the previous value for this item has not changed. The column headed "OP" shows the opcode extracted from the instruction just executed, and if it involved using memory, the columns headed "MEM-ADR" and "OPERAND" show the memory address involved and the value left in memory after the instruction was completed.

If our control panel provides no way to let us display the content of the AC, then we can't look at our result, since it is in the AC when the program halts. In such a case, the program can be changed to store the AC value in memory, say at location 03 (since it is not being used), then we can display location 03 when the program halts, and see our result.

Any fair-minded person will conclude that adding three numbers with a computer is much harder than with a calculator. That is correct. Calculators are much better for many kinds of problems. As we proceed, we will see problems where the computer excels.

Program Loops

The instruction subset we have used permits us to write nothing but "straight-line" programs, so called because each instruction is used once and only once as the program flow proceeds from the first instruction on to the next instruction, and so on, until the last instruction is reached. We need a way of reusing instructions. With an externally programmed calculator, we could splice the end of the program tape to the beginning of the tape, to get a program loop. With a stored program computer, we can't splice the memory too easily. We need a new kind of instruction, one which can force transfer of control from one part of the program to some other part of the program. Since "control" of the computer is governed by the number in the PC, we need instructions which can force new values into the PC, thus switching control to some sequence of instructions other than the usual "next consecutive" instruction.

A "TRA m" instruction forces the two-digit address "m" into the PC. This is analogous to the "GO TO" statement found in some high level languages. The TRA instruction *always* forces the two-digit address from its address field into the PC, no matter what else has been happening; for this reason it is often called an unconditional transfer.

Instruction Name	Code	Function	Comment
TRA	06	m→PC	Unconditional transfer
TZE	07	if (AC)=0, then m→PC	Transfer if zero
TMI	08	if (AC)<0, then m→PC	Transfer if minus
TPL	09	if (AC)>0, then m→PC	Transfer if plus

Figure 2.11 Instruction subset 2 for hypothetical computer.

There will be times when certain instruction sequences should or should not be executed, in the spirit of the "IF" statement found in many high level languages. This can be handled by conditional transfer instructions, which force their two-digit address fields into the PC only if the number in the AC meets their stated condition. Thus a "TZE 75" will cause transfer of control to pass to the instruction sequence beginning at location 75 if and only if the content of the AC is zero when the TZE is executed. If not, then, as usual, the instruction following the TZE will be fetched, so no transfer of control will occur. The new transfer instructions are shown in figure 2.11.

Be very careful in studying these instructions. Compare their actions with those we saw earlier in figure 2.2. The difference between "m→" and "(m)→" is critical: "m→" means copy the two-digit number into the specified target; "(m)→" means use the two-digit number m as an address to fetch a copy of some item at location m in memory, and copy that item from memory into the specified target. Note that " →m" implies a store into memory at location m, and not something as silly as "destroy the number m, replacing it by . . . ."

Let us examine a few machine-language programs using this expanded instruction set. The first example is as follows:

Address	Content	Comment
50	0650	TRA 50

This is both the shortest and the longest program you can write for this computer! It is very short because it has only one instruction. It is very long because it will run forever (until someone uses the control panel to stop the computer). As soon as the instruction is fetched from location 50, the PC is updated automatically so that it will hold the address of the next instruction; since the old (PC) was 50, the new (PC) becomes 51. But then execution of opcode 06 forces the address 50 into the PC, and the control unit will then continue fetching and executing the same instruction ad infinitum.

Let us consider a more constructive example. A person is to be paid at a given rate for up to forty hours of work per week and at double that rate for hours worked in excess of forty per week. Let us store the hours worked (rounded out to whole hours) in location 50 and the rate (in pennies) in location 51. Let the program calculate and store the regular pay in 60, overtime pay in 61, and total pay in 62. We can start writing the first instruction at location 70.

Address	Content	Comment
50	——	Hours
51	——	Rate
. . .		
60	——	Regular pay
61	——	Overtime pay
62	——	Total pay
. . .		

We can sketch the program in the following pseudocode:
If hours $\leq$ 40, 0 $\rightarrow$ overtime

hours * rate $\rightarrow$ regular pay
regular pay $\rightarrow$ total pay
Halt

If hours $>$ 40, (hours $-$ 40) * 2 * rate $\rightarrow$ overtime

40 * rate $\rightarrow$ regular pay
overtime pay + regular pay $\rightarrow$ total pay
Halt

It is very likely that we will need the constant 40, so let us store the number 40 in location 69. If we need other constants as we write the program, we can store them in locations 68, 67, etc, so long as we don't bump into location 62! The entire program is shown in figure 2.12.

Figure 2.12 Pay program.

Address	Content	Comments	
70	0150	LAC 50	Get hours
71	0469	SUB 69	hours−40
72	09 ?	TPL ?	hours>40 ?
73	0168	LAC 68	Get 0
74	0261	STO 61	0 → overtime
75	0150	LAC 50	Get hours
76	0551	MPY 51	* rate
77	0260	STO 60	→ reg pay
78	0262	STO 62	→ tot pay
79	0000	HLT	stop
80	0551	MPY 51	Extra hours * rate
81	0567	MPY 67	* 2
82	0261	STO 61	→ ovtm pay
83	0169	LAC 69	get 40
84	0551	MPY 51	* rate
85	0260	STO 60	→reg pay
86	0361	ADD 61	add ovtm pay
87	0262	STO 62	→ tot pay
89	0000	HLT	stop
. . .			
	Constants		
69	0040		
68	0000		
67	0002		

While writing the program, when we reached the point where the TPL instruction was used, we did not know which location the target of the TPL would occupy, so we filled in its address field with a "?". When we later began writing the instructions the TPL refers to, we could immediately fill in the missing address in place of the "?". So location 72 can now be completed: 0980.

When the program is loaded, the instructions will be copied into locations 70–89, the constants will be placed in locations 67–69, and the data for this particular execution of the program (hours and rate) will be placed in locations 50 and 51. There is no need to put anything in locations 60–62. These locations will be filled in by the program when it is executed.

Self-Modifying Programs

The program to add three numbers used one LAC and two ADD instructions. If we had to add n numbers, we would need a program with one LAC and $n - 1$ ADDs. It seems that the program must be as big as the set of numbers to be added is long. Clearly, it is not an attractive proposition to have to write a 1-million-instruction program if we want to add 1 million numbers. How can we harness the computer's power to significantly reduce the number of instructions we have to write and store? We are not suggesting that you can add n numbers without *executing* $n - 1$ ADD instructions. However, there is a technique which can enormously reduce the number of instructions which you have to write and store, thus making you far more productive. Furthermore, since the program can be much shorter by using less memory space, this technique means that the size of blocks of data which can reside in memory can be correspondingly much larger.

Consider the program to add three numbers. The instructions flowing through the IR are:

```
0100  LAC  00
0301  ADD  01
0302  ADD  02
0000  HLT
```

These add the contents of locations 00, 01, and 02. If we had ten numbers to add, the following instruction flow through the IR would add the ten numbers in locations 00–09. The successive contents of the IR are shown below:

```
   IR  Comment

0100  LAC  00
0301  ADD  01
0302  ADD  02
0303  ADD  03
0304  ADD  04
0305  ADD  05
0306  ADD  06
0307  ADD  07
0308  ADD  08
0309  ADD  09
0000  HLT
```

The ADD instruction is repeated nine times, with each successive occurrence differing only in the address field, and that difference is always 1. If we could arrange for the program to have but a single ADD instruction, the "number" 0301, then after fetching and executing 0301, the program could fetch the "number" 0301, add 1 to it, put it back and loop back to use the "instruction" 0302 which was just generated. We can keep repeating this process until all the data items have been accounted for. We just have to worry about a few more details:

1. stopping the program after the last data item has been processed;
2. avoiding destruction of the running total kept in the AC.

Consider the program in figure 2.13. The execution of the program in figure 2.13 can be sketched as follows:

```
    (00)→AC          Get first data item
87: (AC)+(01)→AC     Add second data item
    (87)+1→87        Modify ADD instruction
    (98)−1→98        Count down: now 8

87: (AC)+(02)→AC     Add third data item
    (87)+1→87        Modify ADD instruction
    (98)−1→98        Count down: now 7

    . . .
    . . .
    . . .
87: (AC)+(09)→AC     ADD tenth (and last) item
    (87)+1→87        Modify ADD instruction
    (98)−1→98        Count down: now 0, so stop
```

Notice that when the program begins, the count is 9, and the address *in* the ADD instruction at location 87 is 01. Each time we come back to execute that instruction, the count is lower by 1, and the address field is higher by 1. So the sum of the two remains constant at 10. When the program ends, the count is at 0, and the address field is 10. But since we don't execute the ADD instruction after its last modification, that means the last ADD we executed was an ADD 09, which accounts for the last data item quite nicely. We now have a program which fits in sixteen locations which can process a block of data of *any* size which we can fit in memory. We need only set the count in location 98 appropriately (set count = number of data items − 1). Without this self-modification technique, a program to add n items takes n instructions. Our sixteen-location program has thirteen instructions, of which three will be executed only once; the other ten will be repeated n times. So the benefit of a short program in terms of memory use can be very costly in terms of execution speed. Our short program takes about ten times as long as the old straightforward program did!

Notice that execution of the program a second time for a new set of data items could be allowed, without reloading the program, provided that the ADD instruction in location 87 was reinitialized to 0301, since it had the value 0310 when the program halted after adding ten items.

```
        Address     Content     Comments
84      0100        LAC 00      Get first item
85      0299        STO 99      Initialize sum
86      0199        LAC 99      Get sum
        ----

87      0301        ADD 01      **changes**
        ----

88      0299        STO 99      Save sum
89      0187        LAC 87      Modify instruction
90      0397        ADD 97      by adding 1 to it
91      0287        STO 87      and put it back

92      0198        LAC 98      Count down
93      0497        SUB 97      by 1
94      0298        STO 98      and put it back

95      0986        TPL 86      Loop back if not done
96      0000        HLT         Otherwise stop

97      0001                    Constant 1
98      0009                    Count
99      ----                    Sum
```

Figure 2.13 A self-modifying program.

This technique whereby the program modifies itself—called program modification, instruction modification, or address modification—illustrates the power of the stored-program computer which derives from treating instructions as if they were data. In the next chapters we will examine far better ways of getting the effect of program self-modification without actually having to change the stored program! In fact, we will generally consider any program that changes itself as being somehow suspect.

The point of studying this technique used with the early computers is that we will thus be in a position to better understand and appreciate the "better ways" by which modern computers eliminate the need to use self-modifying programming techniques.

Lest there be any misunderstanding, self-modifying programs should be avoided. After we examine some new capabilities, we will see that there is no need to use such a primitive technique.

Consequences of Stored Programs

The credit for the idea of placing a computer's program in the computer's memory goes to John von Neuman. Why is it such an important idea? First, it makes it possible for the program to process itself just as if it were ordinary data, as we saw when we discussed self-modifying programs. Self-modification is no longer necessary or desirable; however, because programs can be processed as data means that programs can be used to "manufacture" other programs, as is the case with a compiler. It also allows programs to be stored in libraries, just as if they were data files.

The second consequence of von Neuman's simple idea is that programs can be executed much more rapidly. Consider having to use a computer that does not use stored programs. For instance, suppose this computer reads each instruction it needs from a paper tape. A slow tape reader provides ten characters per second. For the sake of simplicity, let us take each instruction to be equivalent to one paper tape character. If all other factors are disregarded, this computer could execute no more than ten instructions per second because of the tape speed limitation. A very fast paper tape reader might read a thousand characters per second. A computer using this fast tape reader to fetch its instructions would then read up to a thousand instructions per second. It could achieve this rate only if it never branched or transferred control to a nonconsecutive instruction. If ever the program required a backward transfer-of-control, the tape reader would have to be stopped, the tape motion reversed, and the appropriate tape frame reached. This is an inherent characteristic of devices like paper tape readers. They support only *sequential access* in a reasonably efficient fashion.

The memory a program is stored in when you want to execute it on a stored-program computer has a *random access* characteristic. A better name for it would have been uniform access. Each memory location in a random access memory can be accessed in the same amount of time. There is no greater penalty for transferring control to a "far away" instruction, forward or backward, than there is for accessing a "nearby" instruction, unlike the case with sequential access devices. A modern computer memory can support fetching as many as one million instructions per second; and some can go much faster.

A few years ago almost all computer memories were built with ferromagnetic cores. Memory designers now use RAM (random access memory) integrated circuit chips to build memories. When we discuss input-output devices later on, we will see another kind of random access device in which access times are not uniform, in spite of the name!

Speed in Perspective

A person can walk some 5 miles per hour. Using an automobile can raise your speed to some 50 mph, more or less; this is an improvement by a factor of 10. If you could take an airplane, say at 500 mph, you could gain another factor of 10 in speed, for a cumulative speedup of a factor of 100. If you use a supersonic aircraft, you might gain another factor of 4. We see this below:

Walk	Auto	Airplane	Supersonic
5	50	500	2,000
Gain			
*10	*10	*4	
Cumulative gain			
*10	*100	*400	

If you are computing with a simple calculator, your calculating speed is limited by your speed of keying. Perhaps you can key in 2 operations per second, disregarding the data keying. If you could use an externally programmed calculator which reads instructions from a tape, say at the rate

of 100 instructions per second, the calculation can proceed much more rapidly (once again we ignore the time for data input).

The first electronic computers (e.g., ENIAC) could run at approximately 1,000 instructions per second. The computers we will be examining in detail (PDP-11s and LSI-11s) can run at speeds between 500,000 to 3,000,000 instructions per second. Some computers (e.g., Cray I) can perform up to 100 million computations per second. We can summarize the performance gains in computing here (IPS stands for instructions per second):

	Calculator	Early Computers	Ordinary Computers	High Speed Computers
IPS	2	1,000	1,000,000	100,000,000
Gain		*500	*1000	*100
Cumulative gain		*500	*500,000	*50,000,000

What does this mean? Suppose you calculated nonstop for 8 hours per day, 5 days a week, for 50 weeks (one working year). If you were using the calculator at the rate of 2 operations per second for 2,000 hours in that year, you would have performed 14,400,000 operations. In principle, an ordinary computer might get the same work done in as little as 14.4 seconds (2,000 hours divided by a speedup of 500,000). With a high-speed computer, it might take as little as 0.144 seconds.

Of course, such ridiculously short times do not take into account many other aspects which cannot be neglected when computers are used, such as the data preparation time, the program preparation time, the program execution overhead time (i.e., the time spent doing things that do not directly advance the computation, like modifying addresses). Nonetheless, an ordinary computer allows you to do in a few days or weeks what would otherwise have taken years. In fact, many of the things we now do routinely because of computer availability were simply inconceivable in the 1940's.

Summary

We have examined the concept of the stored-program computer, which gives rise to the need for special control unit registers such as the program counter (PC) and the instruction register (IR). The execution of a machine-language program can be described in terms of the instruction-fetch-execute cycle that is repeated for each instruction to be executed.

The mechanics of loading a machine-language program into memory were discussed, as well as the purpose of an initial program load switch, IPL.

An instruction set adequate for performing arithmetic operations only permits you to write straight-line programs. That makes computers impractical for handling either data-dependent computations or large sets of data. This motivates the introduction of conditional and unconditional transfer instructions. Using these, we can execute parts of a program

selectively or repeatedly. Thus, a very small program can process large amounts of memory-resident data, due to the ability of a computer to modify its own programs.

The practicality of stored program computers is enhanced by having the executing program and its data stored in a random access memory (core or RAM). The chapter concludes with a discussion of the stunning performance improvements in computing.

Exercises

2.1 Rewrite the pay program, figure 2.12, so that it fits into one block of contiguous memory, beginning at location zero.

2.2 Improve the pay program so that it uses fewer instructions.

2.3 What is the largest pay rate that can be used with the pay program?

2.4 What do you think would happen if the pay program were incorrectly loaded so that the first instruction (at location 70) became 0170?

2.5 A Teletype paper tape reader can read 10 paper tape frames (a frame is a row of 8 bits) per second. A fast paper tape reader might read 500 cps (characters per second; the words "frame" and "character" are used interchangeably). If each 8-bit frame encodes one instruction for an externally programmed calculator, and the execution of each instruction is instantaneous, how many MIPS (millions of instructions per second) can an externally programmed calculator execute using a 500-cps tape reader?

2.6 The type of random access memory used in most computers can be accessed in times ranging from approximately 1 μsec (μsec = microsecond = 1/1,000,000 second) to 100 nsec (nsec = nanosecond = 1/1,000,000,000 second). Suppose the hypothetical computer had a 1-μsec memory access time. That is, a request to fetch or store an item in memory takes 1 μsec. Further suppose that instruction execution time is instantaneous (this is not an unreasonable first approximation). How many MIPS can this computer execute? Be careful; instruction-fetch and operand-store-or-fetch must be accounted for.

2.7 Why does the control unit update the PC by one?

2.8 Write a program for the hypothetical computer which zeroes (clears) ten consecutive locations beginning at location 90.

2.9 How does the control unit distinguish between a number representing an instruction and one that represents data? Explain this clearly.

2.10 How does the ALU distinguish between a number representing an instruction and one which represents data? Explain this clearly.

2.11 Why is it necessary for data to be placed in low memory and instructions in high memory?

2.12 Rewrite the program in figure 2.6 so that all of it (instructions and data) fit into locations 20–26.

2.13 Rewrite the program in figure 2.6 so that all of it (instructions and data) fit into locations 50–56.

2.14 Compare the content of memory for the programs in exercises 2.12 and 2.13. That is, compute (50)–(20), (51)–(21), etc. What has changed in memory? Account for each change.

2.15 Write the typing history generated in loading the program from exercise 2.12, using the technique depicted in figure 2.8.

2.16 What is the difference between "m →d" and "(m)→d"?

2.17 Write a program that will create a table of twenty words beginning at location 50. The program should store each word's address in that word.

2.18 Rewrite the program in figure 2.13 so that it tests for completion *before* the key addition.

2.19 Why would you want a so-called "leading decision" (testing for completion before a computation in a loop rather than after)? This is the name for the technique used in the previous problem.

2.20 Suppose the TPL instruction began to malfunction (i.e., the control unit hardware interpreting the TPL began to fail). Create an instruction sequence using the other instructions that could be used in place of the TPL.

2.21 Write a program to add ten numbers, then rewrite it so that it uses self-modification. If each instruction takes 1 μsec to execute, and each operand or instruction fetch/store takes 1 μsec, how long does each program take? How much faster (the ratio of the execution times) is the first program compared to the second?

2.22 How much longer does it take to add n numbers with a self-modifying program than with a non-self-modifying program? Express your result as a function of n.

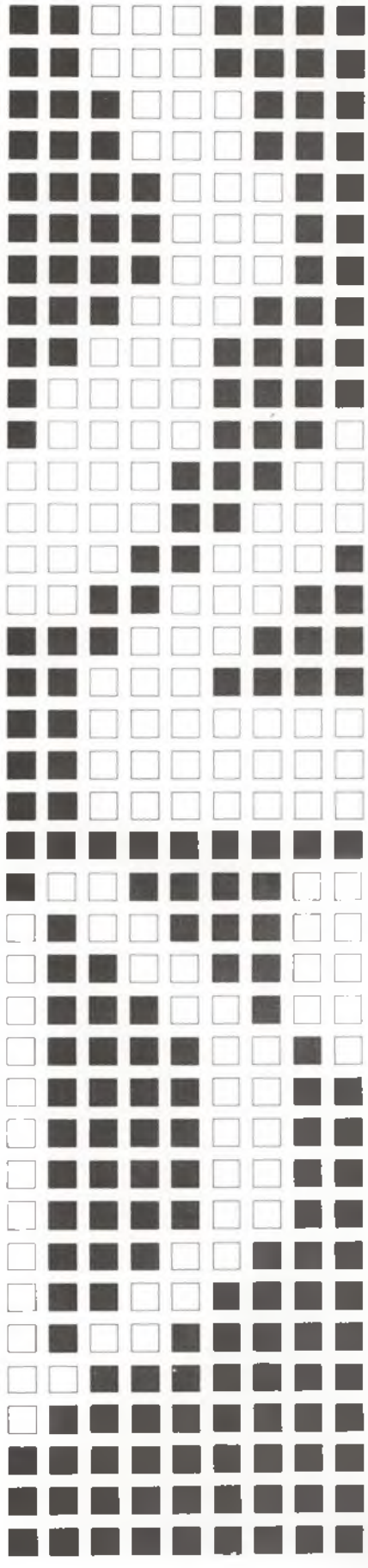
2.23 If self-modifying programs are slower than their non-self-modifying counterparts, why should you ever want to use that technique?

2.24 Compute the ratio of the program lengths (excluding data) for the two programs of exercise 2.21. How does this ratio compare to the ratio of their execution times?

2.25 Many of the sample programs use a HLT instruction and also use a constant with the value zero. Why can't you use the HLT instruction for both purposes?

3

machine-language programming for a real computer



The "real computer" we will concentrate on is the PDP-11 (Programmed Data Processor) manufactured by the Digital Equipment Corporation (DEC). Since the PDP-11 includes many models going back to the first PDP-11 manufactured in 1970, those features common to all of them will get the most attention. Some of the interesting differences between the models will also be discussed.

The PDP-11 family includes the following models:

LSI-11, LSI-11/2, LSI-11/23
PDP-11/03 (actually an LSI-11 in a box)
PDP-11/05, PDP-11/10, PDP-11/20 (the oldest model)
PDP-11/24 (the newest model), PDP-11/34
PDP-11/40, PDP-11/44, PDP-11/45, PDP-11/60
PDP-11/70

Many of the characteristics of the PDP-11 are found in the DEC VAX-11/780 and the VAX-11/580. Other companies also sell computing systems in which the computer is either an LSI-11 or a PDP-11. For instance, the Heathkit (or Zenith) H11 and the Terak Corporation 8510/a are built using LSI-11s.

Names can be misleading. The other families of computers manufactured by DEC—the PDP-8, PDP-10, and PDP-20 being the best known—have not much more in common with the PDP-11 family than they do with computers manufactured by IBM or UNIVAC. However, once you become familiar with the details of any one computer, it is much easier to understand a second or a third model of computer. So let us get back to the PDP-11 family.

The performance range and cost of PDP-11s include single-user tabletop personal computers, such as an H11, on up to multi-user systems capable of supporting 50 or more simultaneous users. The cost ranges from the low thousands to some $300,000. Well over 100,000 PDP-11 systems have been manufactured, and the features of a PDP-11 are an excellent base for starting an in-depth study of computing.

A Machine-Language Program

Let us examine a short machine-language program for the PDP-11, the program in figure 3.1. The bits are written in groups of three simply to make it easier for us to read them. The program illustrates many important properties of the PDP-11.

1. Memory addresses are 16-bits wide.
2. Memory addresses appear to all end with a 0 bit.
3. Memory locations are 16-bits wide.
4. Some instructions (MOV, ADD) take up three memory locations, using 48 bits.
5. Some instructions (HALT) use a single memory location.
6. The PDP-11 does everything in binary.

The last observation, that the PDP-11 does everything in binary, is quite important; it is true of *all* digital computers. So it is worth some effort to learn more about binary numbers.

Address	Content	Comment
0 000 001 000 000 000	0 001 011 111 011 111	MOV a,c
0 000 001 000 000 010	0 000 001 000 001 110	
0 000 001 000 000 100	0 000 001 000 010 010	
0 000 001 000 000 110	0 110 011 111 011 111	ADD b,c
0 000 001 000 001 000	0 000 001 000 010 000	
0 000 001 000 001 010	0 000 001 000 010 010	
0 000 001 000 001 100	0 000 000 000 000 000	HALT
		Locations
0 000 001 000 001 110	0 000 000 000 000 001	a
0 000 001 000 010 000	0 000 000 000 001 000	b
0 000 001 000 010 010	0 000 000 000 000 000	c

Figure 3.1 A machine-language program for the PDP-11.

More About Binary

In our first encounter with binary, we used a set of binary digits to represent a particular keyboard action: if bit i of a 17-bit row is set, then push key i of a 17-key keyboard. Our next encounter with binary involved encoding and decoding; a 5-bit code can be used to represent any one of 32 distinct objects, actions, or whatever.

We now consider bits as not necessarily representing actions or objects, but as having values and representing numbers. So we can speak of the *binary number system*, just as we speak of the decimal number system.

In the decimal number system, only ten distinct symbols are used to represent numbers: these are the decimal digits 0, 1, . . . , 8, 9. The relative position of a digit in a multidigit number gives that digit a much greater or smaller contribution to the value of the number in which it appears. Consider the number 234. It is really shorthand for
2 hundreds, plus 3 tens, plus 4 ones, or

$$2 * 100 + 3 * 10 + 4 * 1, \text{ or}$$
$$2 * 10^2 + 3 * 10^1 + 4 * 10^0$$

In a similar fashion, if we restrict the set of digits to merely the two bits 0 and 1, the binary number 11 001 can be regarded as shorthand for

$$1 * 10{,}000 + 1 * 1{,}000 + 0 * 100 + 0 * 10 + 1 * 1, \text{ or}$$
$$1 * 10^4 + 1 * 10^3 + 0 * 10^2 + 0 * 10^1 + 1 * 10^0$$

Note that in the last line the exponents are being expressed in base ten (decimal). That is, $1 * 10^4$ has the components 1 (binary) * 10 (binary) to the power 4 (decimal). If we wish to look at the binary number 11 001's equivalent in the decimal system, then we simply have to evaluate

$$1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

(since 10 binary is 2 decimal), and we get

$$16 + 8 + 0 + 0 + 1$$

which is 25.

For the immediate future, our major uses of binary will involve counting things (such as memory locations) and adding. Addition can be assisted by using the binary addition table in figure 3.2. If we have to add a pair of binary numbers, say 101 and 110, we proceed from right to left, getting the sum of column 1 using the add table, then we get the sum for column 2, again using the add table, and finally we look up the sum for the third and last column. We then add up the sums we have been writing down, again by looking at the add table. This is exactly how we do decimal addition. The only thing that has changed is the addition table! Figure 3.3 shows a binary addition done in great detail.

Figure 3.2 Binary addition table.

+	0	1
0	0	1
1	1	10

```
          Column Number
          3   2   1
          ---------

          1   0   1

          1   1   0
          ---------
a:            1     <- from 1+0, adding column 1
b:        1         <- from 0+1, adding column 2
c:    1 0           <- from 1+1, adding column 3
```

Figure 3.3 Performing a binary addition.

Now we add the numbers from lines a, b, and c:

```
          Column Number
          3   2   1
          ---------

          0   0   1  <- from line a
          0   1   0  <- from line b
      1   0   0   0  <- from line c
          ---------  Perform addition, getting
d:            1     <- from 1+0+0, adding new column 1
e:        1         <- from 0+1+0, adding new column 2
f:    1 0           <- from 0+0+10, adding new column 3
```

We now simply rewrite these last three lines in a single line getting 1 0 1 1

The result of adding binary 101 and 110 is 1011, which is equivalent to adding decimal 5 and 6, giving a result of decimal 11. When adding in binary, a 1 + 1 situation always leads to a "carry" into the next position on the left. Figure 3.4 shows this happening sooner. After some practice, you begin to see how simple working with binary really is.

Figure 3.4 Another binary addition.

```
                Column Number
                3   2   1
               ---------
                1   0   1

                0   1   1
               ---------
a:              1   0   ← from 1+1, adding column 1
b:              1       ← from 0+1, adding column 2
c:      1               ← from 1+0, adding column 3
```

Now we add the numbers from lines a, b, and c:

```
                Column Number
                3   2   1
               ---------
                0   1   0   ← from line a
                0   1   0   ← from line b
                1   0   0   ← from line c
               ---------    Perform addition, getting
d:                  0       ← from 0+0+0, adding new column 1
e:      1   0               ← from 1+1+0, adding new column 2
f:      1                   ← from 0+0+1, adding new column 3
```

We repeat the process one more time, adding the numbers from lines d, e, and f:

```
                Column Number
                3   2   1
               ---------
                0   0   0   ← from line d
                1   0   0   ← from line e
                1   0   0   ← from line f
               ---------    Perform addition, getting
                    0       ← from 0+0+0, adding new column 1
                0           ← from 0+0+0, adding new column 2
        1   0               ← from 0+1+1, adding new column 3
```

We now simply rewrite these last three lines in a single line getting 1 0 0 0

Counting is simple. Figure 3.5 shows the progression, as we count by ones, from 0, to 1, to 10 (two in binary), to 11 (three in binary), and so on.

The binary number 10000000000, or 10,000,000,000, or 2^{10} has the symbol "K" associated with it. Its value of 1,024 is close to the metric system's use of K for 1,000, but not identical. Later on when we refer to, say, 4K of memory, the exact number of locations is four times 1,024.

Octal Numbers

Having mastered binary counting and binary addition (we will look at other features of binary arithmetic later), we now hurry to state that, although all digital computers use binary, most of the time *people* will use not binary, but a shorthand for binary. On the PDP-11 we will group bits together in sets of 3 (always proceeding from right to left) and replace each set of three bits with the corresponding *octal* digit, as shown in figure 3.6.

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
...	
10000000000	1024
10000000001	1025
...	

Figure 3.5 Counting in binary.

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Figure 3.6 Binary-octal conversion.

Address	Content	Comment
001000	013733	MOV a,c
001002	001016	
001004	001022	
001006	063720	ADD b,c
001010	001020	
001012	001022	
001014	000000	HALT
001016	000001	Location a
001020	000010	Location b
001022	000000	Location c

Figure 3.7 Octal conversion of figure 3.1.

If there are too few bits to form a group of 3 bits, one or two binary zeroes are assumed to be available on the left. We can now rewrite the PDP-11 machine-language program in figure 3.1 in a much more compact and readable form, as figure 3.7, by replacing all the triplets of binary digits by single octal digits.

The fifth address shown in figure 3.7, and those following it, may seem to be in error. After all, if we are using consecutive even-numbered memory locations, shouldn't location 1008 follow 1006, instead of the 1010? It would be if we were using decimal numbers, but in the octal number system, 001006 + 2 is 001010, not 001008.

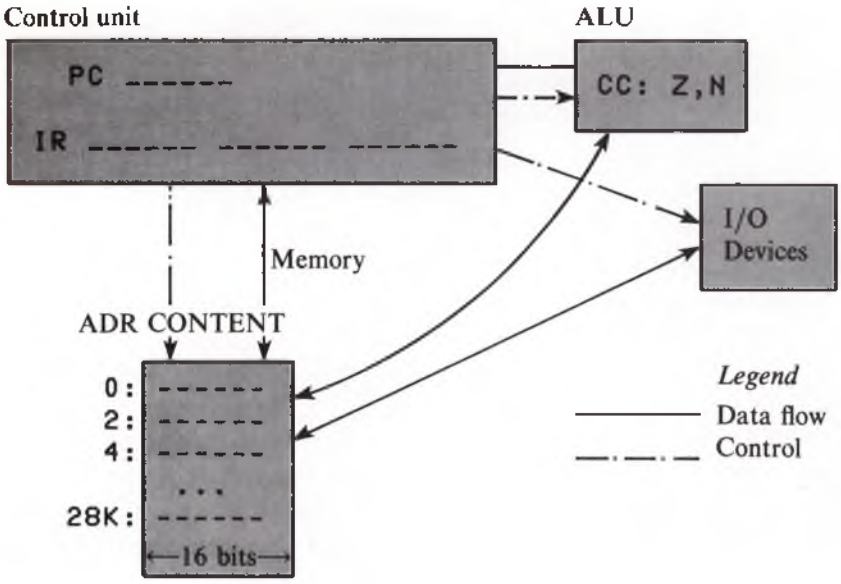


Figure 3.8 A model for the PDP-11.

A Model for the PDP-11

In order to make sense of the previous machine-language program, and to be able to write other programs, we need to see how the PDP-11 is organized; we need a model. The first model we will use follows. As we learn more, the model will be refined to show more detail.

The major components of a PDP-11 are:

1. A control unit
2. An arithmetic-logic processing unit (ALU)
3. Memory
4. Input-output devices (I/O)

Figure 3.8 illustrates how these components are interconnected by showing the control lines that emanate from the control unit. (The status lines used by the control unit to find out what state each component is in at any time are assumed to be included with the control lines.) The solid lines represent the flow of data, which can be instructions, addresses, or actual numeric operands, depending on how they are used. Each component will be discussed in turn. How they all work together will occupy us for many more chapters.

Control Unit

The *control unit* consists of a program counter register (PC), which can hold a 16-bit address, and an instruction register (IR) capable of holding an instruction, along with the hardware necessary to carry out (execute) each instruction. The PDP-11 instructions can be 16, 32, or 48 bits long; that is, they can occupy one, two, or three consecutive words of 16 bits each. We saw some 16- and some 48-bit instructions in the PDP-11 machine-language program in figure 3.1. The IR must be long enough to accommodate any PDP-11 instruction, so the IR can be thought of as having three 16-bit-wide sections.

A PDP-11 memory is made up of *even-numbered* consecutive locations with addresses starting at 0. Each location is capable of storing 16 bits. From now on we will usually refer to such 16-bit-wide locations with even addresses as *words*.

With a 16-bit address, you could expect to address 2^{16} locations, or 64K (K=1024) locations. However, since we are here dealing with even-numbered locations, each of which can hold a 16-bit value, we can express only 32K such addresses in a 16-bit address field. As we shall see later, the last 4K of these addresses are reserved for a special purpose, so under normal circumstances, a program will not use more than 28K words (28KW) of memory. In fact some PDP-11s have much less memory, perhaps as little as 4KW, but even so, addresses in the PDP-11 instructions are always 16 bits wide. An attempt to use the address of a nonexisting memory location will be detected by the control unit. How that event is reported to you depends upon the software that you are using, and particularly on how much of that software is still in good working condition when the control unit detects the error.

The arithmetic logic unit (ALU) consists of the circuitry (hardware) necessary to perform arithmetic, comparisons, and other operations. As data flow through the ALU, the bits called the Z (for zero) and N (for negative) are set or reset to indicate that the latest result produced by the ALU was zero (Z=1) or nonzero (Z=0), and negative (N=1) or nonnegative (N=0).The content of a word is considered negative if its left-most bit is 1; otherwise it is considered nonnegative (positive).The Z and N bits are part of a group of 4 bits known as the condition code (CC).The ALU manages the CC. There is only one CC for the whole computer, not one CC for each word! We will discuss the other two CC bits (C and V) later.

Since the input and output devices (I/O) can't be understood until we know more about the PDP-11, we will defer detailed consideration of the I/O unit until later. Suffice it to say that the control unit can cause bits to be transferred to or from the ALU or memory, from or to various I/O units.

Once again, the CPU (central processing unit) refers to the combination of control unit and ALU as one item, or to the trio of control unit and ALU and memory as one unit. As a general rule, the I/O devices are not lumped in with the CPU. In fact the common name given to I/O devices, "peripheral devices," emphasizes their distinction from the CPU.

Using the control panel, or a panel emulator, or some software developed specifically for loading machine language programs, a program can be loaded into memory, a starting address can be forced into the PC, and program execution can be initiated.

The trace of program execution for our example is shown in figure 3.9. If you have an opportunity to have exclusive use of a PDP-11 for a few minutes, you can key in a program such as this, and then cause it to be executed one instruction at a time. This manner of program execution is called *single-stepping*. The control unit pauses after executing an instruction, so you can examine what its effects were. Then you can cause the next instruction to be fetched and executed, and so on. It is obviously extremely time consuming, and would normally be used only as a last resort. It is, of course, very educational for us. The trace we have in figure 3.10 is analogous to the kind of information single-stepping could provide.

```
PC     IR
1000                              Fetch 1st word of
       013737                     3-word MOV
1002                              Fetch 2nd word of MOV
       013737 001016
1004                              Fetch (001016), getting 000001
                                  Fetch 3rd word of MOV
       013737 001016 001022       Store 000001 at 001022
                                  Set CC Z=0, N=0
1006   ...execution of MOV instruction in 1000-1004 complete
                                  Beginning of a 3-word ADD
       063737                     Fetch 1st word of ADD
1010   063737 001020              Fetch 2nd word of ADD
1012
       063737 001020              Fetch (001020), getting 000010
1014                              Fetch 3rd word of ADD
       063737 001020 001022       Fetch (001022), getting 000001
                                  Add it to 000010
                                  Store result 000011 at 001022
                                  Set CC Z=0, N=0
1014   ...execution of ADD in 1006-1012 complete
       000000                     Fetch 1 word HALT
1016                              Halt, leaving Z=0 and N=0
```

Figure 3.9 Trace of a three-instruction program.

Instruction Name	Mnemonic	Code	Length	Action
Move	MOV	013737,s,d	3	(s)→d
Add	ADD	063737,s,d	3	(s)+(d)→d
Subtract	SUB	163737,s,d	3	(d)−(s)→d
Compare	CMP	023737,s,d	3	(s)−(d) sets CC
Skip if plus	SKP	100002	1	if N=0, PC+4→PC
Skip if equal	SKZ	001402	1	if Z=1, PC+4→PC
Skip if minus	SKM	100402	1	if N=1, PC+4→PC
Jump	JMP	000137,d	2	d→PC
No operation	NOP	000240	1	continue
Halt	HALT	000000	1	stop

Figure 3.10 Instruction subset for PDP-11.

When we were working with the hypothetical computer that had an accumulator (AC) register, the latest result was always held in the AC, so we tested the AC when we needed to know something about the latest result. In the PDP-11 there is no AC register, but since the state of the latest result produced by the ALU is reflected in the CC, we can get a similar effect by testing the CC.

On the PDP-11, instructions are fetched into the IR, one word at a time. Following each fetch of a word into the IR, the control unit automatically updates the PC by adding two to it, so that the PC is pointing either to the next word that is part of the current instruction or to the word immediately following the current instruction. Only fetches from

memory into the IR cause the PC to be updated in this way. Fetching an operand from memory, as when the number 000001 was fetched as the first operand of the MOV (figure 3.9), has no effect on the PC.

Apart from the change to stepping by two instead of one, we see how the instruction-fetch-execute cycle for the PDP-11 and our earlier hypothetical computer are quite similar. Following execution of a HALT instruction, both are left with the PC pointing to the next location following the HALT. In figure 3.9, if the HALT stops the CPU, the PC is left holding the value 1016, since the HALT was fetched from location 1014.

PDP-11 Instruction Subset

The PDP-11 has a sufficiently large instruction set (summarized in Appendix 1) that it is advisable to start with a more manageable subset. We can learn and do surprisingly many things with the small subset of PDP-11 instructions given in figure 3.10.

The symbols "s" and "d" stand for the source operand address and destination operand address respectively, and because in this subset we are dealing only with 16-bit words as operands, the s and d addresses *must* be even addresses. The column entitled "Length" specifies the length of an instruction measured in words.

The ADD and SUB instructions are memory-to-memory operations. That is, they assume both operands are in memory and they cause them to be fetched and processed by the ALU, which then returns the result to memory while it updates the CC properly. The result is always returned to memory at the location that provided the second (the "d") destination operand address. The execution of an ADD instruction is shown in the previous trace (figure 3.9); the ADD occupies words 1006, 1010, and 1012.

A MOV is similar to the old LAC (load AC) instruction. It copies the source operand into the destination location *and* it updates the CC.

The setting of the N (negative) bit in the CC can be thought of as merely copying the leftmost bit of whatever number is being sent to the destination location, in the case of a MOV, ADD, or SUB. In the preceding example, executing the MOV copies the content of location 1016 into location 1022. The number copied is 000 001. Its leftmost bit is 0, so this MOV leaves N=0. If location 1016 had contained the number 100 111, the leftmost bit would have been copied into N, leaving N=1.

Comparing two numbers can be performed by subtracting one from the other using a SUB instruction and then using the appropriate skip instruction. Unfortunately, the SUB changes the destination operand. This can be avoided by using the CMP (compare) instruction. It subtracts the destination operand from the source operand (exactly the *opposite* of the order in which the SUB treats its two operands!) but it does not store the result. It does however update the CC to reflect the state of the result. In the case of a CMP, the leftmost bit of the result of the implied subtraction is copied into the N bit.

The set of conditional branch instructions recognized by the PDP-11 is extensive. Here we are looking at a special subset which we call "skip" instructions. The "skip if condition x" instruction causes the PC to be incremented by 4 if the specified condition is satisfied. Incrementing the PC by 4 results in skipping over two words. By combining a "skip if condition x" with a jump (JMP), we can get effects similar to those of

the old TMI, TPL, etc. Note that a "skip if condition x" can only be followed by an instruction of length 2. Skip and jump instructions *do not* affect the CC bits. They examine the CC bits, but they never modify the CC.

The NOP (no operation) instruction is unusual in having almost no effect. It does manage to cause the PC to increment in the normal way. We sometimes use an NOP to "patch" a machine-language program. A patch would nullify an instruction (by overwriting it with one, two, or three NOPs); it is used to avoid the nuisance of having to rewrite the machine-language program in order to remove an instruction from it.

The HALT instruction is treated differently on different models of the PDP-11. On a single-user system, the HALT causes the control unit to stop the fetch-execute cycle. On a multi-user system, the HALT is classified as a "reserved" instruction; instead of stopping the computation, it causes your program to lose control to the support software.

We can think of this simple instruction set as providing us with as many accumulators (ACs) as we need, since each memory location can be used as an accumulator. Computers which support memory-to-memory operations such as the MOV and ADD avoid the bottleneck which would otherwise be present if you had a single-accumulator computer. With the hypothetical computer, adding two numbers requires writing statement sequences such as

```
LAC A
ADD B
STO B
```

With the PDP-11, using memory location B as an accumulator, we can instead write

```
ADD A,B
```

Of course we still only have one condition code register to test the result of the latest operation involving the ALU.

Sample Machine-Language Programs

We can now construct some new programs. If you plan to load them and execute them on a single-user PDP-11, it is advisable to avoid using the first 256 words—those with octal addresses from 0 to 776, just below 1000 octal. These first 256 words have a special use involving error notification, I/O, and stacks, all of which we shall discuss in due course.

Counting is an important activity. We will often want to count the number of times a program segment has been executed in order to control looping. The machine-language code to cause something to be done four times is shown in figure 3.11.

A Program-Loading Program

The actual form of the program shown in figure 3.11 lends itself to loading by another program. By observing a few simple rules of punctuation, it becomes reasonably easy (for someone else) to write a program which can take lines from a disk file or computer cards or keyboard input in the

```
Address     Content     Comment
1000:       013737      ; MOV constant, count
1002:       001026      ; adr for "constant"
1004:       001032      ; adr for "count"

1006:       000240      ; NOP (do nothing)

1010:       163737      ; SUB one,count
1012:       001030      ; adr for "one"
1014:       001032      ; adr for "count"

1016:       001402      ; SKZ to the Halt

1020:       000137      ; JMP back to 1006
1022:       001006

1024:       000000      ; Halt

1026:       000004      ; constant of 4
1030:       000001      ; one
1032:       000000      ; count

1000        ; entry point (starting address)
```

Figure 3.11 Machine-language program in machine-readable form.

form used in figure 3.11, and process that data so that the content field is loaded into memory at the specified address, and execution is initiated using the indicated entry point.

The rules of punctuation we have adopted are: an even octal address followed by a colon (:) indicates that the numeric information following the colon should be loaded into memory at the specified address. Several numeric fields may follow the colon; they must be octal numbers no larger than 177777 and each field must be separated from the next by a comma. The numeric quantities will be loaded into consecutive memory locations, with the leftmost item loaded into memory at the address specified at the beginning of the line, the next item to the right will be loaded using the previous address plus 2, and so on. Blanks will be ignored. A comment field may be introduced as the last item of a line by using a semicolon (;). A comment field has no effect on the loading process; it is completely ignored. Blank lines are also ignored. A line beginning with an even octal address, not followed by a colon, will be taken as a request to terminate loading, and the specified address will be used as the starting address (entry point) for the program just loaded, and its execution will be initiated. Any lines which follow the entry point line will be ignored, as will lines not conforming to the above specifications. Should two lines cause the same location to be loaded twice, the most recent information will override the previous content.

Figure 3.12 is the same program we had in figure 3.11 except that leading zeroes (leftmost zeroes) have been omitted at our convenience. The missing 0's will be detected and replaced by the loading program. This action is a fairly common one in many applications of computers; it is called a *zero-fill, right-adjusted* data transformation. Figure 3.13 also shows the same program, except that two- and three-word instructions are now written on a single line, as are consecutive constants. Note

Figure 3.12 Machine-language program in machine-readable form.

```
Address   Content          Comment
1000:     13737            ; MOV constant, count
1002:     1026             ; adr for "constant"
1004:     1032             ; adr for "count"

1006:     240              ; NOP (do nothing)

1010:     163737           ; SUB one,count
1012:     1030             ; adr for "one"
1014:     1032             ; adr for "count"

1016:     1402             ; SKZ to the Halt

1020:     137              ; JMP back to 1006
1022:     1006

1024:     0                ; Halt

1026:     4                ; constant of 4
1030:     1                ; one
1032:     0                ; count

1000                       ; entry point
                           (starting address)
```

Figure 3.13 Compact form for machine loading.

```
Address  Content              Comment
1000:    013737,1026,1032     ; MOV const,count

1006:    000240               ; NOP
1010:    163737,1030,1032     ; SUB one,count
1016:    001402               ; SKZ
1020:    000137,1006          ; JMP
1024:    000000               ; HALT
1026:    4,1                  ; const,one
1000       ; entry point
```

that in the program in figure 3.13, location 1032 is not provided with an initial value. Since the program itself will copy the value from location 1026 into 1032 when it is executed, there is no need to initialize location 1032 at program-loading time.

Figure 3.14 is a copy of a printout generated by the program-loading program which was used to process lines with the information from any of figure 3.11, 3.12, or 3.13. The first part of the printout is a "memory dump" in octal, showing the state of memory at the instant when the entry point line was encountered.

After the initial memory dump printout is generated by the loading program, the execution of the program which it just loaded is initiated, using the entry point provided with the octal program as the program's starting address.

```
INITIAL MEMORY DUMP FOLLOWS:
1000:  013737 001026 001032 000240 163737 001030
       001032 001402
1020:  000137 001006 000000 000004 000001 000000
       000000 000000

FINAL MEMORY DUMP FOLLOWS:
1000:  013737 001026 001032 000240 163737 001030
       001032 001402
1020:  000137 001006 000000 000004 000001 000000
       000000 000000
```

Figure 3.14 Dumps for the looping program.

The second part of the printout is another memory dump showing the state of memory when the program terminated. This second dump is traditionally called a PMD (post mortem dump). If we examine the PMD carefully, we note that word-for-word nothing has changed since the program was loaded. This is not usually the case, but it is the correct state of affairs for this particular "do nothing" program.

In examining this dump printout, consecutive words are printed 8 per line from consecutive locations beginning with the address shown on the left. Thus the line:

1000: 013737 001026 001032 000240 163737 001030 001032 001402

is read as:

Location 1000 contains 013737 (first word)
Location 1002 contains 001026 (second word)
Location 1004 contains 001032 (third word)
...
Location 1016 contains 001402 (last word)

We now have a fairly convenient mechanism for expressing PDP-11 machine-language programs, which also facilitates the loading process. So let us proceed to rewrite some of the programs we had for our earlier hypothetical computer.

The program to add three numbers can be written as in figure 3.15. After loading and executing the program, we can see from the PMD in figure 3.16 that the result is 114 (in location 1000).

This program illustrates that data may be stored "above" or "below" the program instructions, and that the loading sequence is a matter of convenience.

We can generalize this program to add larger sets of numbers in much the same manner used in the previous chapter. We do have to be careful about fetching the proper item to modify to generate a series of consecutive addresses. In the PDP-11 the desired address field is certainly not located in the first word of the instruction we wish to modify. Consider the program in figure 3.17.

Figure 3.15 Program to add three numbers.

```
1020: 013737, 1002, 1000        ; MOV a,sum
1026: 063737, 1004, 1000        ; ADD b,sum
1034: 063737, 1006, 1000        ; ADD c,sum
1042: 0                         ; HALT
1002: 100, 10, 4                ; values for a, b, c
1020    ; entry point
```

Figure 3.16 Dumps for addition program.

```
INITIAL MEMORY DUMP FOLLOWS:
1000: 000000 000100 000010 000004 000000 000000
      000000 000000
1020: 013737 001002 001000 063737 001004 001000
      063737 001006
1040: 001000 000000 000000 000000 000000 000000
      000000 000000

FINAL MEMORY DUMP FOLLOWS:
1000: 000114 000100 000010 000004 000000 000000
      000000 000000
1020: 013737 001002 001000 063737 001004 001000
      063737 001006
1040: 001000 000000 000000 000000 000000 000000
      000000 000000
```

Figure 3.17 A looping program.

```
1000: 013737, 1050, 1044  ; MOV a1,sum

1006: 063737              ; ADD a2,sum
1010: 1052
1012: 1044

1014: 063737, 1040, 1010  ; ADD two,adr

1022: 023737, 1010, 1042  ; CMP adr,limit

1030: 001402              ; SKZ

1032: 000137, 1006        ; JMP loop

1036: 000000              ; HALT

1040: 2                   ; two
1042: 1056                ; limit (last adr+2)

1050: 1, 10, 5 ; data for a1, a2, ...
```

The data are expected to be loaded into locations 1050, 1052, . . . , and the limit is determined by storing the address of the last data item, plus two, in location 1042. The PMD for the program shown in figure 3.17 gives the result 000016 in location 1044, and we see that the only part of the program's instructions that changed was word 1010, which started out as the address 1052 and ended with the value 1056, as we expected. If you cannot account for *every* changed item in a PMD, then

you do not understand what is going on. Make it a habit to look for and to account for changed items, and you will find yourself learning much more rapidly and with more confidence.

We will shortly be examining features of the PDP-11 which make the primitive address modification technique we just used quite obsolete. The point of using this primitive technique was to illustrate the power of such a simple idea, and to demonstrate that if by some accident your programs happen to modify themselves, the modified programs will be executed, no matter how much or how little sense they make! As a general rule, we will want to avoid writing programs which modify themselves.

Instead of terminating programs with a HALT, which on a single-user system would prevent the CPU from executing the instructions which could generate the PMD printout, we could arrange for the loading program to leave in memory the PMD service program—say, beginning at location 2000. If you use the two-word instruction 4537,2000 in place of the HALT instruction, you will obtain a printout of the final state of your memory. This instruction is actually a subroutine-call instruction, and we will be discussing its mechanics in great detail later.

Common Errors

It is easy to forget that consecutive words have addresses which differ by two on the PDP-11. Quite often people mistakenly use an increment of one instead of two in stepping through consecutive words. If you try this, as soon as the PDP-11's control unit attempts to execute a word-operation (such as a MOV, ADD, SUB) using an odd operand address, the control unit will object and attempt to have this form of addressing error reported by the support software you are using, if you are using any. Suppose the program in figure 3.13 had an error in it; say, the content of word 1004 was 1033 instead of 1032. Then when the control unit is completing execution of the MOV instruction in locations 1000–1002–1004, in trying to use the number 1033 it finds in location 1004, as the destination address for an operation on a 16-bit word (as is implied by a MOV instruction), it will object because 1033 is an odd address. The control unit will turn control over to the support software, which in turn will attempt to report the problem to you by displaying the message:

```
* * * * * * ? M-TRAP TO 4, PC= 001006 * * * * * *

REGISTERS
R0-R5,SP,PC,PS: 021134 000000 000000 000000 000000 000000
                000776 001006 174004
```

The "M-TRAP TO 4" means that the control unit detected an attempt to either use an odd address where it requires an even address, or that there was no memory at that address (nonexisting memory).The "PC=" specifies the address found in the PC when the problem was detected. This is 2 greater than the location with the "bad" address. So by looking at location (PC)−2, you should see the cause of the problem. The low four bits of the last word printed above correspond to the condition code bits. The PS (processor status) word contains within it the CC. The CC bits we have mentioned are the N and Z bits; the other two, which we will discuss later, are called V and C. In the PS word, they appear in the

4 CC bits

Figure 3.18 Processor status word (PS).

order N, Z, V, and C, as the rightmost bits. An "M-TRAP TO 10" message is generated when the control unit attempts to execute an illegal or a reserved instruction (e.g. a HALT is considered a reserved instruction on a multi-user system). The other items printed will become helpful as we probe more deeply into the PDP-11.

What if, due to a typographical error while typing the program in figure 3.17, the 1000 expected to be loaded into location 1034 went in as 0000? Then when 1032 eventually appears in the PC, the instruction JMP 0 will be executed, and whatever happens to be in location 0 will be used as an instruction!

What if we forgot to provide a constant at load time? Suppose the line with "1040: 2" was missing (figure 3.17). With the loading program we are using, memory from locations 1000 to 1776 is initialized to zero just before the loader loads your program. So in this case location 1040 would have the default value 0. This means that the address being modified would never change, so the program would loop forever! If you are using a single-user system, you would soon notice that the program is taking a long time. If you are using a multi-user system, more likely than not the operating system will enforce a time limit and cause the program to terminate when the default time limit is exhausted. If your system does not impose a time limit, then the program will continue running until *you* terminate it.

Suppose that due to some error the initial value of location 1010 was 10. Your program would then proceed to add the items in locations 10, 12, . . . , 1054! What if you made a mistake while writing the program, and had it modify the destination address in location 1012 instead of the source address in location 1010? The program would then add the content of 1052, which is 10, to each of locations 1044, 1046, . . . and would continue doing so until it generated an address exceeding the limits of your memory. This type of addressing error, caused by a "run-away program," leads to a nonexisting memory location reference, which is detected by the control unit.

There are many other ways in which things can go wrong. Most of them can easily be avoided by careful checking *before* you run your programs. The more difficult errors to find are the logical errors, since the hardware or support software can't provide much help in reading your mind, especially when you are working at the machine-language level. As with writing programs in a high level language, the more thought you put into designing a program, the less will be the effort required to make it run correctly.

Instruction Formats

The hypothetical computer we examined has an ADD instruction with only one explicitly specified operand address. The symbolic form of this ADD could be written as:

$$\text{ADD s ; (s)} + \text{(AC)} \rightarrow \text{AC}$$

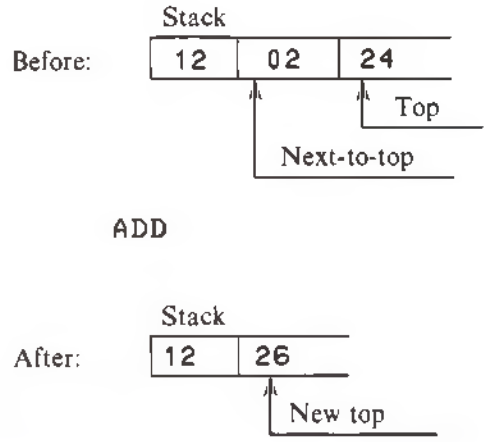


Figure 3.19 Stack, before and after ADD.

Contrast that with the PDP-11's ADD, which always has two explicitly specified operands. The symbolic form for the PDP-11 ADD instruction is:

ADD s,d ; (s) + (d) → d

There are computers that support arithmetic instructions with no explicitly specified operands; the general form of an ADD instruction in that case is simply:

ADD ; operands and result location are implicit

Any ADD, of course, needs two operands and a place to put its result. For a no-explicitly-specified-operand ADD, the operand values are expected to be found in a special set of registers called a stack, and the result of the ADD is expected to be placed in one of those special registers. The interpretation of this 0-operand ADD is:

(value at top stack register) +
(value at next-to-top stack register) →
replaces two operands just used.

This is illustrated in figure 3.19. You may be familiar with the use of a stack on the Hewlett-Packard series of calculators. We will be discussing the implementation of and use of stacks on the PDP-11 shortly.

You could build a computer having arithmetic instructions which had all of their operand addresses explicitly specified. Since an ADD needs three operands, its general form would be:

ADD s1,s2,d ; (s1) + (s2) → d

There have been computers built (e.g., IBM 650) in which yet another address field was used in most instructions. If we include this field, a three-operand ADD then becomes

ADD s1,s2,d,nia

The "nia" field represents the "next instruction address." Such a computer does not need a PC. Instead, each instruction explicitly specifies where its "next instruction" is to come from. It is as if each instruction had a "PC field" built in. This strange arrangement is a very clever way of compensating for the delays introduced if you have to use a mechanical rotating storage device such as a magnetic disk or drum as your CPU's main memory. The time to fetch an item or an instruction from such a rotating memory would vary according to the item's location relative to the location of the instruction currently being executed. Such a memory is clearly not a RAM memory.

We can summarize this discussion as follows. Computers differ in the way they represent instructions. Since all computers support arithmetic instructions such as ADD, we can categorize computers according to the instruction format they use for their arithmetic instructions. The principal formats in current use are:

Type	Typical Computer
0-operand instruction	Burroughs 6700
1-operand instruction	UNIVAC 1100/80
2-operand instruction	PDP-11
3-operand instruction	IBM 1620

What difference does it make that a computer uses one format rather than another? Presumably the fewer operands you must identify in each instruction, the shorter these instructions can be. With a two-operand ADD on the PDP-11, we need three words to store a PDP-11 memory-to-memory ADD instruction. If we could find some way of eliminating one or both of the operand addresses, we could reduce our memory requirements for the storage of arithmetic instructions by as much as a factor of three; or, alternatively, we could write significantly longer programs using the same amount of memory. The instruction format can have a significant practical impact.

We shall soon see ways which allow you to consider the PDP-11 as also having the characteristics of a 0-operand and a 1-operand computer. The PDP-11 has some very sophisticated capabilities.

PDP-11 Instruction Summary

We have been using PDP-11 instructions that are either data oriented or control oriented. We classify them in figure 3.20. We have also dealt with 0-, 1-, and 2-operand instructions. The classification of the PDP-11 instructions we have seen, by operand count, is given in figure 3.21. As we examine the other PDP-11 instructions, we shall refine the distinctions among the data-oriented instructions, and introduce more sophisticated instructions in all categories. The format classes themselves will be refined, as we shall see in chapter 9.

Data oriented	Control oriented
MOV	JMP
ADD	SKP
SUB	SKZ
CMP	SKM
	NOP
	HALT

Figure 3.20 PDP-11 instruction subset classification.

Number of operands		
0	**1**	**2**
SKP	JMP	MOV
SKZ		ADD
SKM		SUB
NOP		CMP
HALT		

Figure 3.21 Classification of instructions by operand.

Summary

There are many members in the PDP-11 family of computers. Being computers, they use binary representations for their instructions and for the data. Binary is awkward for people to deal with, so PDP-11 software translates it into octal numbers. Octal is somewhat easier for us to deal with, and it does not obscure the computer's design details, as would be the case if we did everything in decimal.

The basic features of PDP-11 organization follow. Its basic unit of information is the 16-bit word. Consecutive words have even addresses, and all memory addresses are 16 bits long. The PDP-11's ALU does not revolve around a single accumulator. Every memory word can be used as an accumulator. A special register, the program status (PS) register, reflects the status of the latest product of the ALU, using condition code (CC) bits. The two CC bits we have dealt with are the Z (zero) and N (negative) condition status bits.

Loading PDP-11 machine-language programs can be facilitated by using a loading program. The machine-language programs are presented to the loading program in a form very much like that seen in a computer memory printout, or memory dump. A PMD is a dump of memory which shows the state of a program's memory immediately after its execution was terminated. It is important to note that the PC value reported in a PMD is, as expected, 2 larger than the address associated with the instruction (or instruction word) which triggers an error detected by the control unit.

Computers can be classified according to the number of addresses found in their arithmetic instructions. At this point the PDP-11 appears to be a two-address computer. We shall soon see it take on other appearances.

Exercises

3.1 What is the largest address you can write for the PDP-11? Give it in (a) binary; (b) octal; (c) decimal; (d) K units.

3.2 Convert the following binary numbers into decimal numbers: (a) 11001; (b) 10101; (c) 10001; (d) 11111111

3.3 Perform the following binary additions:

(a)	(b)	(c)
11001	10101	11001
10101	10001	10001

3.4 What bit pattern do all even binary numbers share?

3.5 Conversion of decimal numbers into binary numbers can be performed with the following algorithm. Given a number *n*, find the largest power of 2 (pot) that is smaller than *n*. Then find the difference: $n -$ pot $\rightarrow n'$. Record the number pot in binary; it will necessarily only have one 1 in it. Then repeat the procedure beginning with n', which leads to pot', find $n' -$ pot' $\rightarrow n''$, etc. Stop when $n''''\ldots''''$ is zero. (Is this guaranteed to happen?) Then add pot, pot', . . . to obtain the binary representation. For example, given n=5:

$5 - 2^4$ does not fit
$5 - 2^3$ does not fit
$5 - 2^2$ fits!
$5 - 4 \rightarrow$ n'=1, with pot=$2^2 = 4 = 100$

Repeat on n'=1, find pot'=$2^0 = 1$
Then add pot =100
+ pot' =001

101

Result 101 binary = 5 decimal

Using this algorithm, find the binary equivalents of the following decimal numbers:

(a) 129 ; (b) 55; (c) 100; (d) 512

3.6 Part of the octal addition table is:

+	0	1	2	3	4	5	6	7	
0	0	1	2	3	4	5	6	7	
1	1	1	2	3	4	5	6	7	10
2	2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12	
4	. . .								
. . .									
7	. . .								

(a) Fill in the missing parts; turn in a complete table. (b) Using the addition algorithm described in the text, find the sums of the following octal numbers:

(i)	(ii)	(iii)	(iv)
123	770	076	222
321	123	753	666

3.7 Explain why K represents the number 1024 in the world of computers while it usually means exactly 1000 in the metric world.

3.8 If the SKP began malfunctioning on your computer, how would you use the other instructions to make up for this lack? Give specific instruction substitutions.

3.9 Is the ADD instruction really essential? Show how—if possible, for any program—all ADD instructions could be replaced by other existing instructions.

3.10 Why is it necessary to have both a SUB and a CMP instruction? If you had to sacrifice one, which would it be? Show exactly what would be used in its place.

3.11 What is the largest octal number representable in a PDP-11 word, if we do not consider the leftmost bit as a sign bit? What is its decimal equivalent? What is this number in K units?

3.12 Based on how the loading program we have been using works, what would be in memory locations 1000–1016 after the following had been loaded, in the indicated sequence:

1002: 1, 2, 3
1000: 4, 5
1010: 6, 7, 10
1000: 11, 12
1012: 13

3.13 How does a PDP-11 react to your using the address 000777 for a word operation?

3.14 You can always expect memory locations to contain zeroes until you change them. True or False? Please explain your answer.

3.15 What is a PMD? What is it used for?

3.16 Write a PDP-11 octal machine-language program that adds four consecutive data items stored in memory locations 1030–1036 and show its load format.

3.17 If your program had an odd address in the source address field of a SUB instruction beginning at location 1020, which PC address would you expect to find in a PMD?

3.18 How can a JMP instruction have an illegal destination address? Describe two ways in which this can happen.

3.19 How could any one of SKZ, SKM, or SKP ever lead directly to an M-TRAP, since they can't conceivably specify illegal addresses?

3.20 How many memory references are required to fetch and execute a MOV s,d? Account for each reference.

3.21 If you are running on a multi-user system, does your operating system impose an execution time limit on you? If so, what is its default setting, and how can you change it? If not, how do you terminate a program you suspect may be in an infinite loop?

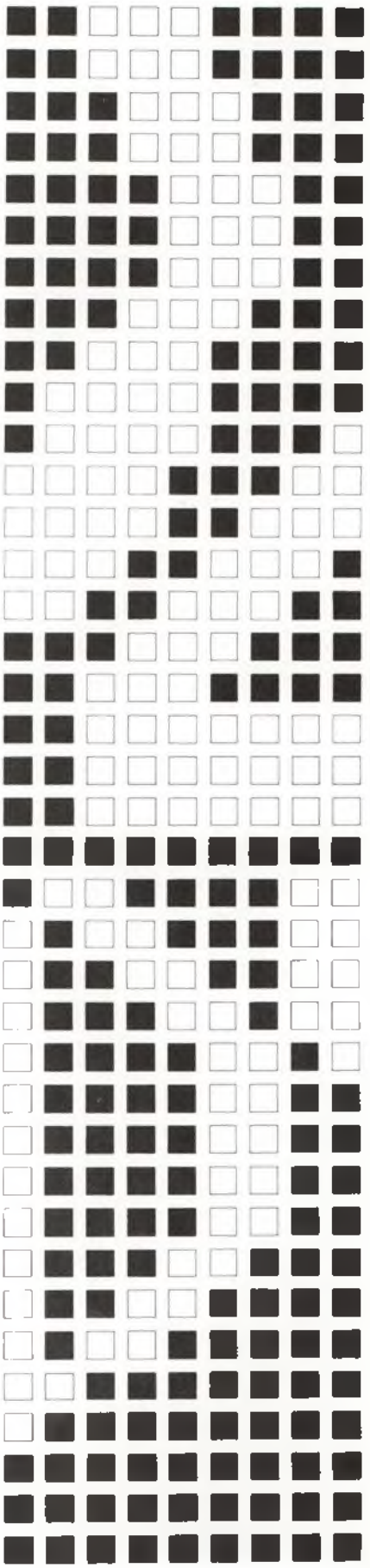
3.22 Rewrite the pay program, figure 2.12, for the PDP-11, assuming the pay rate is 5. Why would such a program be inefficient for larger pay rates?

3.23 Compare the program you wrote in the preceding problem with the program in figure 2.13. Assume that one HC "word" is equivalent to two PDP-11words. Ignoring the data, are the programs of comparable length? How much difference does the 1-address nature of the HC make in contrast to the 2-address nature of the PDP-11 instruction set you are working with?

3.24 Those who designed the PDP-11 software chose to represent all the binary information using octal notation. On some other computers a different choice was made. For instance, on most IBM computers, and on many microcomputers, *hexadecimal* notation is used. This is a base-16 notation. The digits 0–9 remain 0–9, and the digit symbols associated with the values 10, 11, 12, 13, 14 and 15 are A, B, C, D, E, and F. Thus, octal 123456 is equivalent to binary 1 010 011 100 101 110; or binary 1010 0111 0010 1110 is equivalent to hex (short for hexadecimal) A72E. Rewrite the first machine-language program of this chapter in hexadecimal.

4

a better way: assembly language



Even as late as in the early 1960s, a text of this kind would simply have continued introducing the remainder of the computer's instruction set and other hardware features, perhaps also discussing some algorithms, since there were few places to look for good algorithms back then. We will, of course, examine the remaining instructions and their uses, but first we will introduce a powerful software tool that will make the study of and use of machine language much less painful.

Programmers in the 1950s discovered that many of the low level details and decisions involved in writing machine-language programs could themselves be handled by a special computer program (now called an *assembler*) which would accept as input a symbolic representation of the desired machine-language program. This symbolic representation is called an *assembly language program,* and its machine-readable representation is called a *source module.* The assembler program translates its input, the source module, and produces as output the desired machine-language program in a form referred to as the *object module.* The object module, in turn, can be processed by a loading program that leads to execution of your machine-language equivalent program. This flow is depicted in figure 4.1.

You will notice the similarity between this process and that which you used in connection with high level languages: the FORTRAN or COBOL compilers or other compilers played the role of the assembler. For the PDP-11, the assembler is given the name MACRO-11, short for macro-assembler. The concept of macros will be discussed later. The handwritten symbolic program can be made into a machine-readable source module by any number of approaches. You can prepare it on punched cards (using a keypunch), on punched paper tape (using a teleprinter with a tape punch), or with a CRT or other on-line terminal (using an interactive editor that stores the source module on a disk or tape).

Let us look now at the flow of information in taking a program from its representation as a source module into a running machine-language program in memory, as shown in figure 4.2.

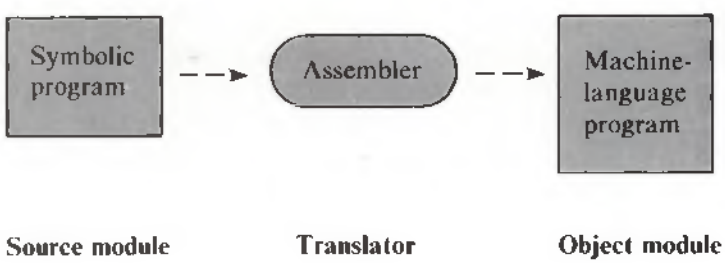


Figure 4.1 Assembling a source program.

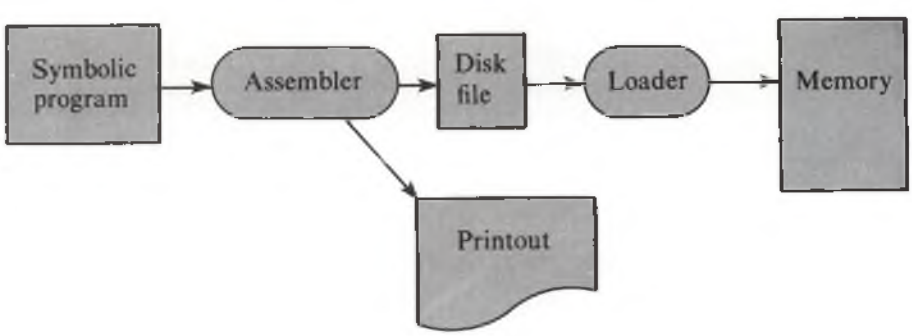


Figure 4.2 Assembling, loading, executing.

This may seem like a complicated process but it has a number of important benefits. By having the assembler write its output on a disk file (wherever a disk file is used, a magnetic tape file could be used), larger programs can be processed by the assembler than would otherwise be possible; for if the assembler placed its machine-language output directly into computer memory, that output could sooner or later begin to overlap the assembler itself! It also follows that you can save the object module on the disk and rerun it later without assembling it anew.

The simple loader we depict here takes the assembler's output, as it is found in a disk file, and makes it into an executable program, then places it into memory and transfers control to this copy in memory.

Even greater advantages fall out if we break down the processing following the assembler's work into a few more steps. More later about these steps (called linking and relocating).

MACRO-11 Source Statements

Let us now focus on DEC's MACRO-11 assembler. What kind of input does it accept, what kind of outputs does it produce, and how can we use it intelligently? MACRO-11 processes as input source modules which consist of lines that have the following general form:

```
Label:      Operation      Operand list      ; Comment
```

When a source module is being prepared, it is convenient to use the "horizontal tab" code (usually associated with the key labeled "tab") to separate these fields. However, each field of a line does not have to fall into a specific column but the order of the fields must be observed. All of the fields are optional, and a field is frequently omitted. The comment field has no affect on the assembly process; any text to the right of a semicolon will merely be echoed in the assembly printout. You can have statements which contain nothing but comments, and these could start with the ";" in the leftmost character position.

The operation field (also called the opcode field) may contain either an instruction mnemonic such as a MOV, ADD, SUB, etc. or a new type of item used to direct the assembler in its work. This item is called a *directive;* it is sometimes called a pseudo-op.

Figure 4.3 is a list of some of the directives recognized by MACRO-11. We see that these directives all begin with a "."; this is true for all MACRO-11 directives. None of the instruction mnemonics begins with a period; the two kinds of items can thus be distinguished. Appendix 2 summarizes all the directives we use.

Figure 4.3 Key MACRO-11 directives.

```
[name:]     .BLKW     count
[name:]     .WORD     constant
            .END      [entry-point]
```

The symbolism [. . .] means that the item enclosed by the pair of square brackets is permitted, but it may be omitted. Let us consider these three directives in turn. The .END is very special, since it must appear in each assembly language program. MACRO-11 processes the lines of a source module until it reaches a line with a .END. If it runs out of lines before it finds the .END line, it will issue a complaint by way of a diagnostic message. The entire repertoire of MACRO-11 diagnostic messages is given in Appendix 3. If your source module has lines beyond a .END, MACRO-11 will ignore these extra lines.

The .END should not be confused with the HALT instruction. The former is a signal to MACRO-11 that the last line of the text of your assembly language program has been reached and that MACRO-11 should finish its work of assembling your program. The .END does not stop execution of your program. The effect of a line containing a HALT is not substantially different from the effect of a line containing an ADD or other instruction mnemonic, so far as MACRO-11 is concerned. It is only when and if the machine code (00000) corresponding to the HALT instruction enters the IR, well after MACRO-11 has done its work, that the HALT will be executed.

Since PDP-11 support software provides for independent compilation or assembly of subroutines and main programs, some convention has to be adopted to inform the support software which module is the main program (i.e., which module should be the first to be executed when a complete program is constructed from a set of modules). If the main program happens to be written in assembly language, you inform the support software that this source module is the main program by filling in the .END entry-point field with the name you will attach to the instruction you wish to have used as your starting point. So a module ending with

```
.END   ABC
```

would cause that module to be treated as the main program, and some line in the same module should begin with "ABC:". The instruction on that line would be the first to be executed after the program was assembled and loaded. A module ending with

```
.END
```

would be taken to be subservient to some other module, so it would be written as a subroutine. We will consider subroutines later.

A .WORD directive allows you to write an octal constant, causing it to be part of your program. So if you needed the constant 12 octal, you could write

```
.WORD 12
```

However, in order to use that number, you have to make up a symbolic name for it and attach this name as a *label*, as in

```
TWLV:   .WORD   12
```

When a name appears as the leftmost item of a line, followed by a colon, we say that the name is being *defined*. Any use of the name TWLV will amount to using the memory address that MACRO-11 will assign to the memory word containing the constant 12. Each name you make up (user-defined names) must be defined by you, as a label, somewhere in your program, and you can provide only one such definition. If TWLV: were to appear twice in the same program, MACRO-11 would be confused as to which definition applied, so it would complain about the multiply defined symbol TWLV.

Finally, the .BLKW directive allows you to reserve one or more memory words, as many as you indicate in the "count" field (a positive octal number). Clearly, you will want to attach a name to these words so that other parts of your program can refer to them. For instance,

```
SUM:    .BLKW   1
```

causes MACRO-11 to reserve one word of memory, associate with its memory address the name SUM, and cause all other uses of the name SUM to be replaced by the address of this word. If you reserved several words, as in

```
BUNCH:    .BLKW   3
```

MACRO-11 will reserve 3 consecutive words, and any use of the name BUNCH will refer to the first word (the word with the lowest address). You can refer to the second and third words by using the compound names BUNCH + 2 and BUNCH + 4, respectively. You can refer to the word preceding BUNCH by using the compound name BUNCH−2, and so on. It is important to note that .BLKW *reserves* memory locations; it *does not* initialize them. If you fetch a value from such a location, without first having assigned it a value, what you will fetch is unpredictable. Therefore, by definition, such a program would be in error.

Complete Source Programs

We can put all these ideas to work by reconsidering the earlier machine-language program which added two numbers. Suppose the two numbers are 1 and 10 octal. We can compose our program on a sheet having columns with the indicated headings. We can make up symbolic names—say, A and B—for the two constants and place the .WORD directives at the beginning of the program.

Then if we want to reserve a word for the result, and wish to give it the name C, we use a .BLKW. The instructions to perform the arithmetic are a MOV and an ADD. We can terminate the program with a HALT instruction and follow that with an .END to inform MACRO-11 that the text of the symbolic program is now complete, as depicted in figure 4.4.

When an opcode has two or more operands, the operands in the operand list are separated by commas, as you see in the MOV A,C statement. This program will assemble properly, but since it does not specify in any way that it is the main program, the loader won't know which location is to be used as the program's starting address when we request that our program be loaded and executed. We can correct that problem by placing a label on the first instruction we wish to have executed (the MOV A,C)

```
Label   Opcode   Operands   Comment
A:      .WORD    1          ; two data items in octal
B:      .WORD    10
C:      .BLKW    1          ; location of result

        MOV      A,C
        ADD      B,C
        HALT                ; stop execution here

        .END
```

Figure 4.4 MACRO-11 source module.

```
Label   Opcode   Operands   Comment
A:      .WORD    1          ; two data items in octal
B:      .WORD    10
C:      .BLKW    1          ; location of result

GO:     MOV      A,C
        ADD      B,C
        HALT                ; stop execution here

        .END     GO
```

Figure 4.5 MACRO-11 source module, main program.

and by writing that name as the operand for the .END. Now if we assemble the revised program, as shown in figure 4.5, it can be loaded and executed. All the names or labels we made up (A, B, C, GO) are more or less arbitrary. The rules for making up symbolic names acceptable to MACRO-11 are similar to those used in many high level languages:

1. Only letters, digits, dollar signs, and periods may be used.
2. The leftmost character must not be numeric.
3. Names may exceed 6 characters in length, but only the first 6 will matter.

Should you attempt to define or use a name with any other character in it (such as a space, a tab, punctuation, etc.), MACRO-11 will complain. Since the PDP-11 support software uses the $ and . in its names, you should avoid using these characters in the names you define. You can even make up and use names such as MOV, ADD, etc. MACRO-11 will accept these, but in order to avoid confusing mere mortals, such names should not be used for anything except instruction mnemonics in the operation field. We could reassign new names in the last program and also change the order of the lines as shown in figure 4.6.

This program will produce the same result as the previous version, although now, instead of the result being in the second word of the program (we start counting at the zeroeth word), it is found in the seventh word (the one labeled SUM).

You can use the lowercase letters a–z as well as the uppercase letters A–Z freely in your source modules. MACRO-11 will convert all labels and names into uppercase letters so the names abc, ABC, and Abc would be considered the same, and an add is the same as an ADD. Unless otherwise specified, MACRO-11 is insensitive to the use of uppercase or

Figure 4.6 MACRO-11 source module, main program.

```
Label    Opcode   Operands    Comment
FIRST:   MOV      A1,SUM
         ADD      A2,SUM
         HALT                 ; stop execution here

SUM:     .BLKW    1           ; location of result
A1:      .WORD    1           ; two data items in octal
A2:      .WORD    10

         .END     FIRST
```

lowercase letters. It will not distinguish between MOV, Mov or mOv, nor will it distinguish between Here, here, or hErE. It does, however, make sense for you to be consistent in using uppercase and lowercase spellings, for the sake of other people, who may wonder what you are doing.

Reading an Assembly Listing

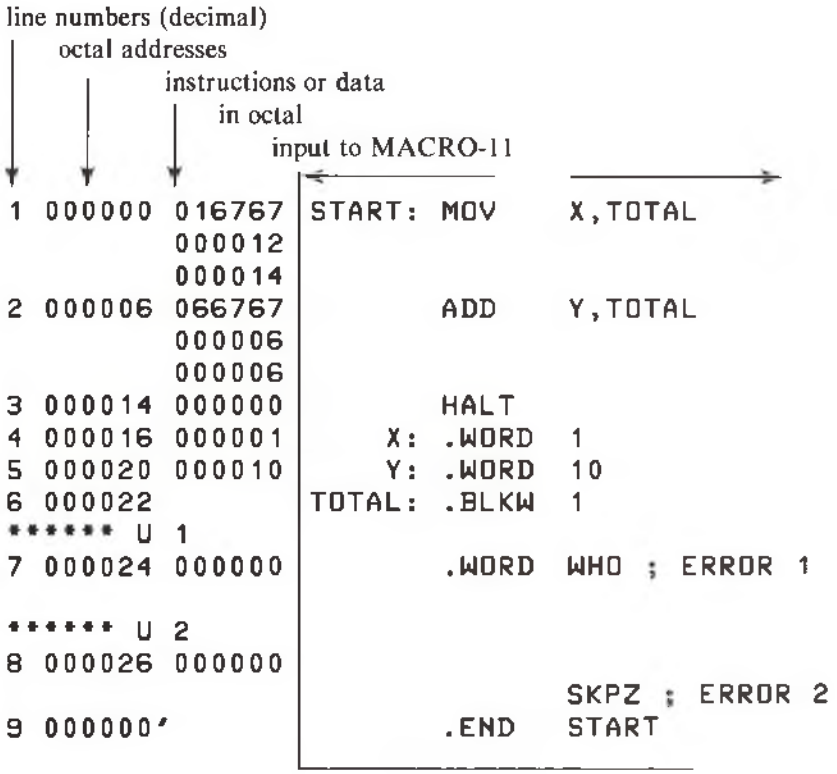
The most important output of the assembler is the machine-readable machine-language object module, so far as the computer is concerned. The assembler has another output, a by-product of performing its work. This output is the assembly listing, or printout. Consider the source module shown in figure 4.7 (the hand-coded machine-language equivalent for the first five lines appeared in figure 3.7).

The MACRO-11 printout is reproduced as figure 4.8, with several annotations. It is important to study this printout carefully; much of our work from now on will involve understanding MACRO-11 printouts. We notice that the text of the source module is reproduced in the printout; it always appears to the right of the information generated by MACRO-11. Examining the printout from left to right, we see on the far left a column of numbers. These are consecutive decimal line numbers assigned by MACRO-11 as it reads lines of our source module. These numbers have no effect on either assembling or executing our programs.

The next column is very significant. As MACRO-11 processes a new line of the source module, it anticipates having to allocate one or more memory locations to store the instruction or the data items it will encounter on that line. The address field shows the address MACRO-11 is assigning. This field is also called the "location counter" field. Notice that the first such address is 000000 and that the next one is 000006. Since the MOV X,TOTAL needs three words, MACRO-11 assigns locations 0, 2, and 4 for this instruction. When MACRO-11 reads the second line, it is ready to allocate location 6 (the next available word) to the first word for the ADD. Since the ADD also takes up three words (locations 6, 10, and 12), when we get to line 3, the address field reads 000014. MACRO-11 will always begin allocating memory locations starting with location 0. We need not fear that this will conflict with other uses of memory locations 0 through 776 (we were warned to avoid these memory locations earlier), because another program (the linker) will ensure that our program is moved up (relocated) in memory, to avoid using real memory locations 0–776 for those PDP-11s in which the use of low memory should be avoided.

```
START:      MOV       X,TOTAL
            ADD       Y,TOTAL
            HALT
X:          .WORD     1
Y:          .WORD     10
TOTAL:      .BLKW     1
            .WORD     WHO       ; ERROR 1
            SKPZ                ; ERROR 2
            .END      START
```

Figure 4.7 Source module.




```
Symbol table
SKPZ = ****** START 000000r TOTAL 000022r
WHO  = ****** X     000016r Y     000022r

ERRORS DETECTED 2
```

Figure 4.8 MACRO-11 listing.

The remaining part of the printout shows the octal representation for the machine-language instructions and data that MACRO-11 is placing in the object module.

The careful reader will see that the second and third words generated by MACRO-11, which should be the addresses assigned by MACRO-11 for the symbols X and TOTAL, are apparently wrong! Shouldn't they be 000016 and 000020? A closer examination of the printout reveals that even the first word, 016767, does not correspond to the MOV instruction code 013737 we have been using. We clearly have some explaining to do. We will see shortly that MACRO-11 takes every advantage of the sophistication of the PDP-11 instruction set, and this puzzle will soon be resolved to everyone's satisfaction.

We deliberately introduced an extraneous statement (line 7) in the source module, to see how MACRO-11 reacts to seeing an undefined symbol (the name WHO). A "flag" character is printed *above* the offending line, and a count of all errors found by MACRO-11 appears at the bottom of the printout.

Note that MACRO-11 does not recognize the SKPZ; nor, for that matter, any of the SKPz mnemonics. This will be of no great loss as we will be using a far more flexible set of branch instructions shortly (BR, BPL, etc.), which will eliminate any need for the old SKP type instructions.

After the .END line we see a "symbol table." Each name we made up appears in this table, along with the address of the memory location assigned to it. If the address is followed by the letter r, the symbol (and the address corresponding to it) is said to be relocatable (more about relocation later).

After Assembling, Then What Do You Do?

MACRO-11 produces an object module in which are stored the machine-language instructions and additional information about where they should ultimately be placed in memory, as well as information about any other object modules that may be needed. In order for your source module to be activated as an executing machine-language program, one or more further stages of processing are required following the assembly process. The specific manner in which you proceed depends upon the operating system you are using. The following sections describe the key considerations.

Operating Systems Considerations

If you are working directly in machine language, with no support software, you could in principle skip this section. However, if your goal is to understand computing, it is important for you to know how a particular set of support software, or a given operating system, can impose various constraints, and why this is so.

Even small PDP-11's can have the benefit of an operating system. The most common "small system" operating system for PDP-11's is RT-11. Its use in a single-user setting is described below.

There is also a multi-user version of RT-11 available for larger PDP-11's. It can handle several users. The larger computer multi-user systems would use either RSX-11M, RSTS, or UNIX. Some effects of using these multi-user systems are discussed below.

Using RT-11

The object modules produced by MACRO-11 are expected to be processed by another program, called the *linker*, which gathers together all the object modules necessary to produce an executable program. Then it reallocates memory to them, beginning either at octal address 0 or at octal 1000, and it makes any adjustments necessary so that addresses within the program instructions remain correct. It then copies its output, called a SAV (for save) module on the disk. The SAV module, in turn, can be executed by using the RUN command. The RT-11 commands necessary to assemble, link, and execute a source module whose file name is "PROG" are shown below:

```
RUN MACRO          Remarks
*PROG,PROG=PROG    * is a prompt
*^Z                from MACRO
...                ^Z for end-of-file
                   response
                   ...
RUN LINKER
*PROG=PROG,LIB     * is a prompt
*^Z                from LINKER
...
RUN PROG
```

The syntactic conventions used need some explanation. The RUN-MACRO command is followed by a line which provides several names. The name of the source module is given on the right-hand side of the equals sign. The names provided on the left-hand side of the equals sign will be used as the names of the object module and printout file produced by the assembler. These last two names will automatically be qualified by RT-11; it will append the suffixes .OBJ and .LST to them.

The name PROG has been used over and over again. Each time it is used RT-11 appends a different suffix to it, depending upon the context. RT-11 conventions allow you to use different names if you wish. Suppose our source module was called SM. Then we could have MACRO-11 name the object module it produces OM, and have it assign the name PR to the print file it also produces. Finally we could give the linker the name OM for its input, to be combined with any other modules to be found in file LIB, and have the linker output called EX. This is illustrated below.

```
RUN MACRO
*OM,PR=SM
*^Z
...
RUN LINKER
*EX=OM,LIB
*^Z
...
RUN EX
```

The flow of information is depicted in figure 4.9. Afterward you can request that RT-11 send the printout file (PROG.LST in the first example and PR.LST in the second example) to either a CRT or a printer.

If our source module had a .END statement such as

```
.END  GO
```

the assembler would encode the address corresponding to GO in the object module it produces. The linker, in turn, would take note of this address, leaving that information available with its output, so that when you come to load and execute your program, the RUN processor will find the address corresponding to GO and use it as your program's entry point.

We will be taking another look at the inner workings of the assembler and linker later, after we learn some more about the PDP-11 hardware.

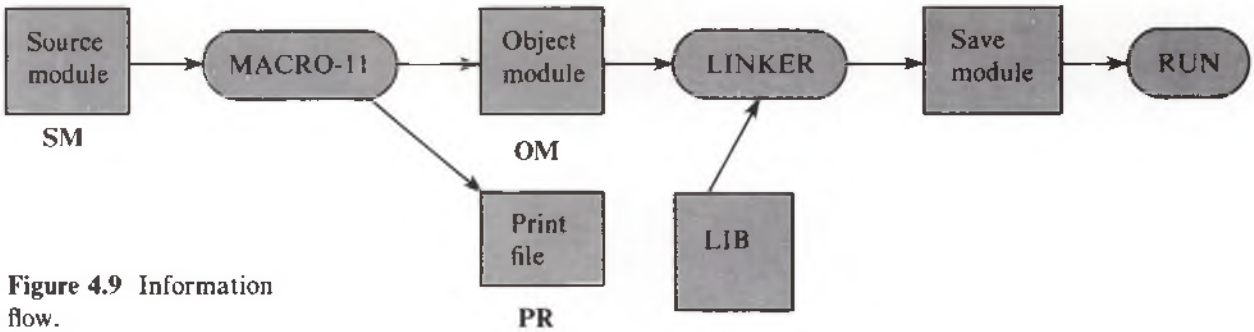


Figure 4.9 Information flow.

Running on a Multi-User System

The caution to avoid using memory locations 0–776 octal applies when using bare PDP-11's, LSI-11's, and the most common operating system for single-user PDP-11's—namely, the RT-11 operating system. Its linker will automatically relocate your programs to avoid using locations 0–776.

On a multi-user PDP-11 there is no way you could use memory locations 0–776 even if you wanted to (unless you have the privileges of a systems programmer who is a super user). You may think you are using locations 0–776, but it is an illusion. We have to distinguish now between real memory (often called physical memory), and the other kind, which is called virtual memory (VM). The programs you write and run under a multi-user operating system are allocated virtual memory space. The addresses within your program are 16-bit virtual addresses. You can only address a block of memory not exceeding 32KW. The real memory—say, for a PDP-11/34—may be as large as 128KW. The real memory needs and uses 18-bit addresses. You can't easily write an 18-bit address using a 16-bit word.

When your program is loaded in a multi-user system, the operating system initializes a special piece of hardware called the memory management unit (MMU) so that it knows where in real memory your program has been loaded. The MMU makes it possible for many users to share a large physical memory without interfering with each other. The MMU performs what is called dynamic relocation (as opposed to the static relocation done by the linker). Each time your program uses a 16-bit address (the only address size known to your program), the MMU translates that 16-bit virtual address into a "real" 18-bit address, each and every time you use that 16-bit VM address.

The net effect is that, when using a multi-user operating system, the linker does not have to relocate your program by 1000, so it doesn't. Your program, beginning at VM address 0, might be placed in real memory beginning at a location 570000. Each of the many concurrent users of the computer can have and use VM location 0, since the MMU will see to it that each user's virtual memory is placed in nonoverlapping blocks of real memory. Figure 4.10 illustrates these differences.

We will continue in the sequel to refer to relocation by 1000, as is the case on single-user RT-11 systems. Keep in mind that with a multi-user operating system (and the MMU), the linker will relocate by 0, not by 1000, and you are working in a virtual memory system.

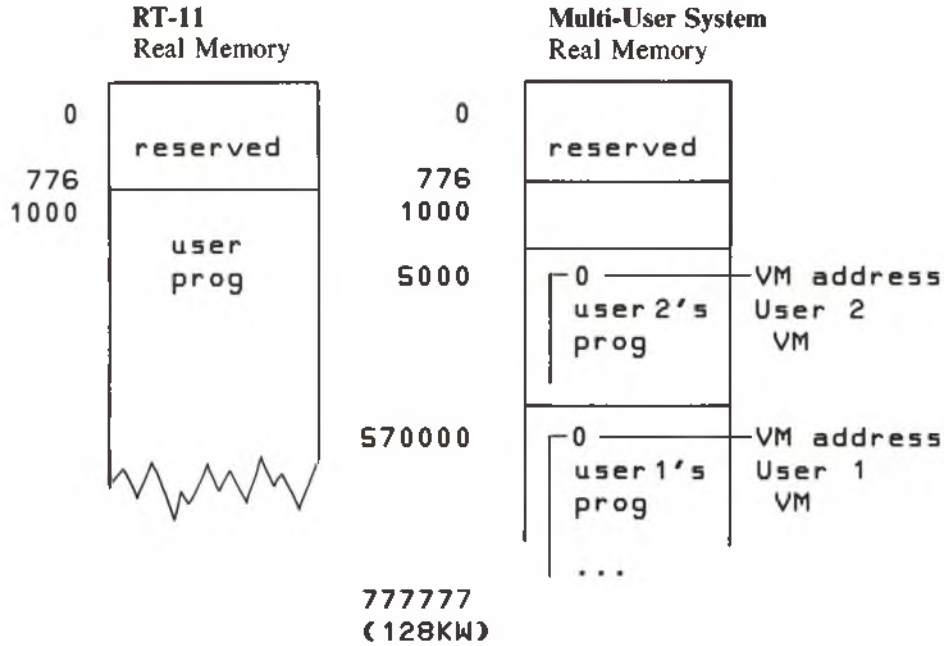


Figure 4.10 Virtual memories in a real memory.

Machine Language Revisited

Instead of using a special program to read a file containing a hand-coded machine-language program, as we did in chapter 3—one which then loaded and initiated execution of that machine-language program—we could now prepare a symbolic program in machine-readable form, such as

```
START:  .WORD   13737   ; MOV
        .WORD   1026
        ...
        .WORD   1
        .WORD   0
        .END    START
```

This is the machine-language program in figure 3.12. We can assemble it; it will require very little effort on the part of MACRO-11. After the assembler's output has been processed by the linker, if we are using the single-user version of RT-11, it will have been relocated by 1000 octal, so that it finds its way into the memory we had originally intended for it. If we request it, the module prepared by the linker can be executed. Since we are no longer using a special program to read, load, and execute, we can't expect special services such as an initial memory dump, nor will we get a PMD. There was a need earlier for these special services. We will soon see how similar services can be requested by a symbolic program.

System Services

What can we do to have our assembly language programs produce output, or obtain input until we learn something about how the CPU performs I/O? When working in assembly language, the standard approach is to use predefined I/O service requests. Each operating system has its own

set of I/O and other system services. We will introduce a few simple ones. With a little help, you should be able to see how the ones introduced here can be replaced or rephrased in terms of the ones your operating system provides.

.MCALL

The mechanism for using an operating-system defined service involves a new idea, called a "macro," and a new directive, ".MCALL." A *macro* is a set of assembly-language statements which is given a name. Whenever you would have needed to write this set of assembly language statements (called the macro's definition), you can simply write the macro's name instead. Doing this is called using or *invoking* the macro. Simple macros are invoked merely by using the macro's name as if it were an instruction mnemonic. The assembler replaces each such invocation with the corresponding set of assembly language statements and assembles these, eliminating the original invocation.

Terminating a program with a HALT is not considered elegant. You should terminate a program by having it return control to the operating system you are using. The macro .EXIT is used for this purpose. In order to use .EXIT, or any other system-defined macro, you first have to write the .MCALL directive, as in

```
.MCALL   .EXIT
```

This statement must appear once in your program, and it must appear before you invoke .EXIT. The .MCALL does not cause any memory to be allocated or initialized at the point the .MCALL appears. It merely makes it possible for the assembler to find the definitions it will need if and when those macros are invoked. It is good practice to place any .MCALL statements at or near the beginning of your source modules. One .MCALL statement can request several definitions, simply by having their names listed in the operand field.

When the assembler processes an .MCALL directive, the assembler will search a system macro library (usually kept on a disk), looking for those macros named on the .MCALL. It will copy the definitions it needs. Using all of this is much simpler than it may appear. Instead of writing "I," you can now write "II," as shown in figure 4.11.

Figure 4.11 Use of .MCALL.

```
START:  MOV     X,Y
        HALT
X:      .WORD   123
Y:      .BLKW   1
        .END    START
```

I: Using HALT

```
        .MCALL  .EXIT
START:  MOV X,Y
         .EXIT
X:      .WORD   123
Y:      .BLKW   1
          .END  START
```

II: Using a Macro

More intricate macros expect you to supply *arguments*. These are names or numbers that are meaningful to you, to the assembler, and to the particular macro you are invoking. For instance, suppose you wanted to obtain a memory printout similar to the initial memory dump or final memory dump that the machine-language loader produced for us earlier. It would be a simple matter to have a macro named DUMPER, whose normal invocation would expect two arguments to be provided. Thus

```
.MCALL   DUMPER
  ...
DUMPER   A,B
  ...
DUMPER   A,A+12
  ...
```

shows two execution time dump printout requests. The first of these would print out (in octal) the memory content of locations A, A + 2, ... up to address B. Similarly, the second request would print (A), (A + 2), ... (A + 12).

Other services could be provided. Some of these are discussed in this chapter's problem set. System services can also be provided without using macros. Exercise 4.4 shows how a system-defined subroutine can be used. You will have to determine which if any system macros or system subroutines are available for your use, either by consulting your instructor, or by examining the various operating system system-services manuals and/or the guide to locally provided system services.

If you are working on a bare machine, none of these services is available. You will have to provide your own. How you could do this should be clear by the time you finish reading the next ten chapters.

Summary

The idea of a very low level symbolic language, called assembly language, has been introduced. Its purpose is to make it easier to write and to modify machine-language-like programs. Almost every line of an assembly language program corresponds to a single machine-language instruction. Those lines that do not represent instructions are commands to the assembler, called directives. The most frequently used directives are .WORD, which lets us write and refer to octal constants, as needed, and .BLKW, which lets us reserve memory locations. It is important to distinguish between merely reserving space, with a .BLKW, and reserving space and initializing it, as with a .WORD.

The .END directive has neither function. It simply informs the assembler that the last line of the source module has been reached. The .END can also be used to specify a program's entry point when that program is to be executed as the main program.

The assembler relieves us of having to assign or use numeric memory addresses. We can now use symbolic names instead of numeric addresses. Each name we make up and use we must define once and only once, by having it appear as the leftmost item of a source statement, immediately followed by a colon (:).

The assembly-language program which we write is processed by a program called an assembler; the PDP-11's assembler is called MACRO-11. The assembler produces two output files. One of these is the machine-language equivalent of the source module, which will subsequently be loaded and executed, if we so request.The other output file contains the assembly listing. Among other uses, it shows us what memory addresses the assembler has allocated for each instruction, data item, or space reservation request. It shows us which addresses correspond to each symbolic name, and if we made any errors that the assembler could detect.

Operating procedures for assembling, linking, and executing MACRO-11 programs using RT-11 are given. Approximately the same patterns of commands would be used with almost any operating system.

When many users share one computer memory without interfering with each other, we can think of each user's memory as a virtual memory. These virtual memories are associated with nonoverlapping blocks of real (physical) memory by the intervention of the memory management unit (MMU) hardware. In a single-user system it is necessary to avoid using the lower part of memory (0–776 octal). In a system with an MMU, all of the virtual memory can be used, beginning at 0.

Performing data input and data output as well as other operations can be accommodated by use of the preprogrammed services supported by most operating systems. Predefined system services can easily be used by having the .MCALL directive fetch sets of assembly language statements, called macro definitions. At each point we wish to use a system service, we merely use the name associated with the desired macro definition. This use, called an invocation, results in the assembler replacing the one-line invocation by the corresponding set of assembly-language statements, provided by the macro definition. Writing our own macro definitions will be discussed later.

Exercises

4.1 Familiarize yourself with the procedures for creating a MACRO-11 source module in a machine-readable form. Since these procedures differ from one system to another, we can't describe them here. Prepare the program of figure 4.8, assemble it, and obtain the listing generated by the assembler. See to what extent, if any, your version of MACRO-11 differs from the one used in the text.

4.2 Familiarize yourself with the procedures for linking the machine-language output module of the assembler. Then see how you can cause the output of the linking process to be loaded and executed.

4.3 A contingency PMD memory dump is one that is invoked only when the hardware detects a problem with your program when your program is running (e.g., an invalid address, an illegal instruction, etc.). It is possible to request such a service (if it exists on your system) using the following MACRO-11 statements:

```
            .MCALL    .PMD ; required
START:      .PMD      A,B  ; execute first
            ...
A:
            ...
B:          ...
            .END      START
```

The .MCALL is a directive which finds definitions for the items which are listed. Such a request must precede any other reference to these items. The instructions corresponding to the .PMD invocation (at location START) should be the first ones executed in your program. Nothing visible will happen; this simply informs the system that if your program should abort, then the post-mortem dump program should attempt to dump words from locations A through B. Of course, addresses A and B must be even, and A should be lower (smaller) than B. Make it a practice to use this .PMD in *all* your programs.

4.4 A standard dump service is one that you can invoke at will. It is possible to invoke such a service (if it exists on your system) by the following statements:

```
            .MCALL    .PMD
            .GLOBL    DUMP
START:      .PMD      HERE,LAST
HERE:       ...
            JSR       %5,DUMP
            .WORD     A,B
            ...
LAST:       ...
            .END      START
```

The .GLOBL is another directive. Here it is used to inform the operating system by way of the assembler and the linker that you need to use a subroutine whose name is DUMP. Since DUMP shares instructions with .PMD, the .PMD *must* be used if you plan to use DUMP. The JSR (a subroutine-call instruction) is followed by a pair of word addresses (A and B in the example), which are the low and high addresses of the area you wish to have dumped. A well-designed dump service will abort if you call it too often or if you print too much. Try using two dump requests with the program of figure 4.6.

4.5 Another useful tool can be provided. You often want to look at just a few things in a few different spots in your program, during execution. Instead of writing JSR/.WORD as above, you could use:

```
.SNAP A,B
```

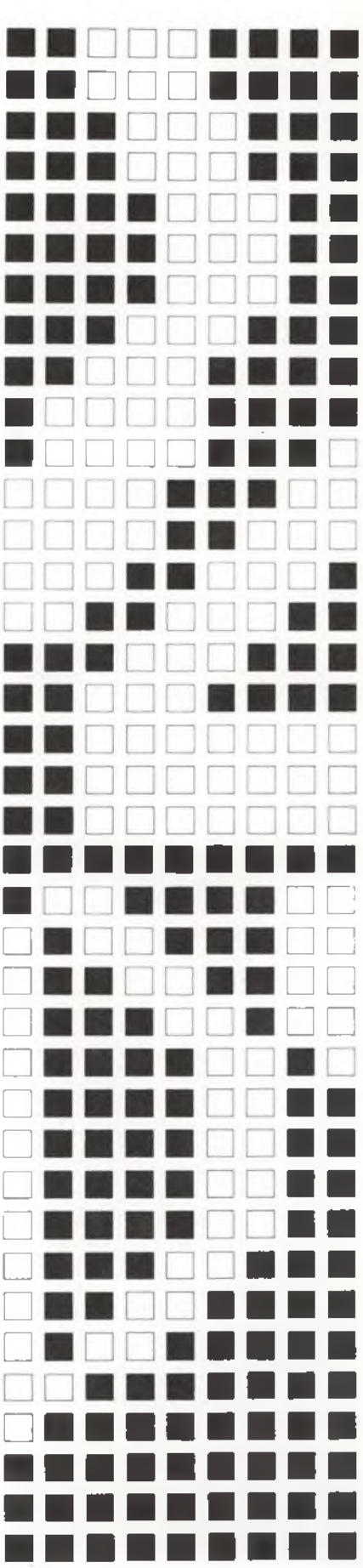
This gives you an abbreviated snapshot, perhaps only two lines of information (16 words), from location A to B (both word addresses) and the content of the general registers (to be discussed soon) along with the program counter (PC) value. This is a "snapshot." If you write a .SNAP with no arguments, only the registers will be displayed. As with DUMP, .PMD is a strict prerequisite, and the name .SNAP should appear in an .MCALL. Excessive printing using .SNAP will terminate execution.

4.6 Which of these sequences best represents the transformations your source programs undergo?

(a) Load-execute-print.
(b) Compile-load and go.
(c) Assemble-translate-run.
(d) Assemble-load-run.

5

more hardware



We alluded to the sophistication of the PDP-11 instruction set previously. Now we have to begin to justify this allusion.

Registers

The PDP-11 instruction subset we have been using supports only memory-to-memory operations. These are very useful, but they are not very efficient when complex calculations are performed. The repeated fetching and storing of intermediate results from/to memory is time consuming. There are many advantages to having an accumulator register in those circumstances. The designers of the PDP-11 took this into account when they provided the PDP-11 with not just one, but several accumulators (six, to be specific).

The PDP-11 accumulators are called "general registers." They are 16-bits wide, and they are numbered 0, 1, 2, 3, 4, and 5. The PDP-11 has two other 16-bit registers, numbered 6 and 7, but avoid using them until further notice. In order to avoid confusing registers 0, 1, . . . , 7 with memory locations 0, 1, . . . , 7, the assembler recognizes a special notation when registers are being referenced. The character % (percent), when followed by a single digit 0–7, is taken to mean that the corresponding register should be used. Without the %, the assembler would take the number as an ordinary memory address (yes, there are odd memory addresses, as we shall soon see). Consider the following examples.

Previously, when we wanted to add some numbers, we could write a program segment such as:

```
MOV A,SUM
ADD B,SUM
ADD C,SUM
```

Using register 0 as an accumulator, we could instead write:

```
MOV A,%0
ADD B,%0
ADD C,%0
```

Is this an improvement? It certainly does not look like one. Why use registers? Registers are physically closer to the ALU, and they are constructed of higher speed components so they can be accessed more rapidly (a factor of ten faster than memory access time is not unusual).The higher speed components are, of course, more expensive, so we generally have far fewer registers than we have memory locations. On the PDP-11 we gain in another significant way. Instructions that use registers are shorter (use fewer words) than the same instructions that use memory locations! As a general rule, the fewer words something involves, the faster it can be fetched from memory, so we gain a little more speed here. Perhaps of greater significance is the saving in memory space, because, after all, 28K words can be too little too often. The space savings can be dramatic.

Consider a MOV instruction. The MOV A,SUM uses three words. The MOV A,%0 only uses two words. Register-to-register operations are allowed, and we see below that a MOV %0,%1 takes only one word!

In almost every instance where you could use a memory address within an instruction, the PDP-11 will accept a register reference instead. So, for a MOV, each of the following is allowed:

```
MOV X,Y    ; MEMORY-TO-MEMORY,      3
MOV X,%1   ; MEMORY-TO-REGISTER,    2
MOV %2,Y   ; REGISTER-TO-MEMORY,    2
MOV %0,%1  ; REGISTER-TO-REGISTER,  1
```

The digit at the end of each comment shows the instruction length in words.

When using memory locations, we can either merely reserve space using a .BLKW or reserve and initialize a memory location using a .WORD directive. What are the counterpart directives for use with registers? There are none! There is no need to "reserve" or "define" registers, since the names %0–%7 are built in. There is no directive which causes a register to be initialized when the program is loaded. You have to write and execute MOV mem,reg instructions. Since there are so few registers, this is not much of a burden.

Careful use of registers can both speed up a program and reduce its memory requirements. Sloppy use of registers can make a program almost impossible to understand and debug!

In any execution of an instruction which uses the ALU (and this includes the MOV instruction) the CC will be updated to reflect the state of the result sent to the destination, whether the destination be a memory location or a register.

An Exception

If you manage to place an address in a register, say in %1, you might be tempted to try using

```
JMP %1
```

if %1 held the address of an instruction you wanted to JMP to. This particular form of addressing cannot be used with the JMP.

Renaming Registers and Other Things

It is sometimes convenient to use a name other than the built-in name for a register. MACRO-11 has a renaming operation which has the form:

new name = old name

The "old name" can be either a built-in name, such as %0, or a user-defined name. So if you wish, you can start out your source modules as follows:

```
R0  =  %0
R1  =  %1
  . . .
```

Then you can use the names R0 and R1 in place of the names %0 and %1. If you do wish to rename the registers, the renaming must occur before you use the new names. If you forget this simple rule, you may be seeing many phase-error assembly-time diagnostic flags "P". Failure to properly define symbols early enough causes many problems for the assembler, with a major exception. New names for memory locations *can* be used before they have been defined. New names for registers *cannot* be used before they have been defined. Do not be lulled into thinking that "=" in any way assigns a value to anything. It is a renaming mechanism—no more, no less—one which allows you to create synonyms. You can still use the old name even after you have created a new name for it.

If you can use the .MCALL directive, you may have access to a .REGDEF macro, which makes the definitions R0, R1, . . ., R5 available to you, as well as SP for %6 and PC for %7.

Bytes

You may be wondering why PDP-11 memory locations are numbered 0, 2, 4, etc. This wastes one bit in each address and effectively cuts the maximum addressable memory in half. There must be a good reason for doing so.

The reason has to do with changes in the impetus for using computers. Originally computer use could only be justified for the most complex problems in science and engineering, which tend to be numerically oriented. We call them number-crunching applications (e.g., nuclear weapons design). As the cost of computing has dropped, they have been put to new uses, often involving nonnumeric applications (such as assembling a program). These applications are character oriented, so the character-handling ability of computer hardware has become important.

Character Codes

Back in chapter 1 we saw how the twenty-six letters of the alphabet, the ten digits, and some punctuation could be represented using a 6-bit code. By adding one more bit, a richer character set can be used. The most widely used standard code is the 7-bit code called ASCII. With a 7-bit code, you can represent 128 items. We saw the ASCII code table in binary in figure 1.18. The entire ASCII code set is shown in octal in Appendix 4. For now it suffices to note the subset in octal, as shown in figure 5.1.

Figure 5.1 Some 7-bit ASCII codes.

Character	Code (octal)
NUL	000
...	...
space (blank)	040
...	...
0	060
1	061
2	062
...	...
9	071
...	...
A	101
B	102
...	...
Z	132
...	...
a	141
b	142
...	...
z	172
...	...
DELete	177

It is common practice to attach an extra bit to each character code to provide some measure of error detection, particularly when transmitting codes over a long distance. This extra bit, called a check bit or parity bit, is usually positioned as the leftmost bit. Even when no check bit is being used, an eighth bit is often added, so more often than not the 7-bit ASCII codes will be 8 bits wide. Computer designers recognize the importance of 7- and 8-bit codes and have provided hardware support for a unit of information larger than 1 bit but smaller than 1 word. This unit of information is the 8-bit *byte*.

Byte Manipulations

The PDP-11 makes it easy to process bytes by allowing you to address them directly. Each 16-bit word of the PDP-11 memory can hold two 8-bit bytes. Word 0 holds bytes 0 and 1, word 2 holds bytes 2 and 3, and so on. Figure 5.2 shows where bytes appear in memory.

Unlike many other computers that have a special set of instructions just to manipulate bytes, the PDP-11 allows many of its word-oriented instructions to be qualified so that, instead of processing a 16-bit operand, the qualified instruction will seek out an 8-bit byte. The qualification is accomplished in the hardware by one bit being set in such instructions. Symbolically, it is handled by appending the letter B (for byte) to those mnemonics which have byte counterparts.

Word adr	High byte	Low byte
0	1	0
2	3	2
4	5	4
	...	
2n	2n+1	2n

Figure 5.2 Byte addresses within memory words.

Consider the MOV instruction. The instruction

```
MOV X,Y
```

makes sense only if X and Y refer to 16-bit operands. The instruction

```
MOVB Z,W
```

expects to fetch an 8-bit byte from memory address Z and store it at memory location W. It does not matter whether the addresses Z and W are odd or even; the bytes at addresses Z and W will be used as a source and destination operand, respectively. The byte which shares the memory word occupied by the byte at address W will not be affected by this MOV instruction. Similarly

```
CMPB P,Q
```

will compare the 8-bit operands found at P and Q, respectively. The only data manipulation instructions we have seen that do not directly handle byte operands are ADD and SUB. There are no ADDB and SUBB instructions. Fortunately, we can easily compensate for this lack.

More Byte Instructions

All of the instructions that operate on 16-bit operands and have a byte counterpart are identified in Appendix 1 by the notation (B) following the standard mnemonic for the word operation.

Henceforth, as new PDP-11 instructions are introduced, you should consult Appendix 1 to see if the word operation has a byte counterpart. There is only one instruction in the PDP-11 which operates only on bytes; this is the SWAB instruction. A

```
SWAB X
```

will "swap" the bytes in word X (i.e., it interchanges them). As with most PDP-11 instructions, the operand address may be that of a register or of a memory location.

Single Operand Instructions

So far all the data manipulation instructions we have seen have two operands: a source and a destination. Some manipulations are used so frequently that they warrant having special instructions to support them more efficiently. In some cases the operation involves only a single operand, so the general form used for these instructions is:

```
OPC d ; use (d) as source, and result
```

Certain kinds of activities are so common they are referred to as "housekeeping." This section discusses the mundane (but essential) useful instructions which facilitate housekeeping.

CLR, CLRB

Clearing a counter, setting a register to zero, and similar operations are so frequent that they warrant having special instructions. The instructions

```
CLR word   ,   CLRB byte
```

are the first of a class of instructions which have only a single operand address. Their effects are, respectively,

0 → word, 0 → byte

For instance, a

```
CLR ABC
```

replaces

```
        MOV CON,ABC
        ...
CON:  .WORD 0
```

CLR and CLRB can have a register as the destination operand.

```
CLR %0   ; 0 → reg 0
CLRB %1  ; 0 → low 8 bits of reg 1
```

The CLRB reg will have no effect on the high 8 bits of the register; only the low 8 bits will be cleared.

INC, INCB

Counting is so important that a pair of instructions is provided:

```
INC word , INCB byte
```

Their effects are, respectively:

(word) + 1 → word , (byte) + 1 → byte

If the result does not fit in 16 bits (or 8 bits) because you are trying to add one to 177777 (or to 377), the result will be 000000 (or 000)—and if you didn't expect this, you are in trouble. More about this later.

DEC, DECB

Just as INC adds one to its word operand, DEC subtracts one from its word operand. This makes it just as easy to count down as to count up. DECB subtracts one from its byte operand.

If you happen to decrement the most negative number (100000), you suddenly get the most positive number, much to your surprise.

NEG, NEGB

Finally, we come to an easy way to change a positive number into a negative number, or vice versa.

```
NEG word , NEGB byte
```

have the following effects, respectively:

$-$ (word) $\rightarrow$ word, $-$ (byte) $\rightarrow$ byte

Not all numbers can be negated correctly on the PDP-11, so we will have to examine these special cases soon.

Byte-Oriented Assembler Directives

For each new class of instruction there are likely to be some helpful directives. When we encounter a new "data type," such as a byte, we need byte-oriented directives. MACRO-11 supports byte instructions as follows.

.BYTE

Corresponding to the .WORD const directive, there is a

.BYTE 8-bit constant (in octal)

So two consecutive directives

```
.BYTE 0
.BYTE 0
```

are (almost) equivalent to one

```
.WORD 0
```

If you need the ASCII codes (perhaps to send them to a printer which accepts only ASCII codes), you could use byte directives such as:

```
.BYTE 101 ; code for ``A''
.BYTE 102 ; code for ``B''
```

which is somewhat simpler than writing

```
.WORD 041101 ; 102 and 101
```

You can have several operands with a .BYTE (as is also the case for .WORD), simply by separating them with commas. So we could have written

```
.BYTE 101,102 ; codes for ''A'', then ''B''
```

instead of the two consecutive .BYTE directives above.

.EVEN

The PDP-11 uses a word-oriented memory, as is true of many other computers that support both word and byte operations. The control unit objects to fetching or storing words unless they are properly *aligned.* Words must have even addresses; this is called word boundary alignment. Even for those computers that don't require word boundary alignment, paying attention to boundary alignment can have a significant effect on a program's execution speed if the memory hardware is word oriented.

MACRO-11 allocates space as it is needed, a word or a byte at a time. So if you had

```
X:  .WORD 1
Y:  .BYTE 2
Z:  .WORD 3
```

then, assuming X fell on a word boundary, it might be allocated the address 000200 (before relocation); Y would then be assigned the address 000202, and Z would be assigned the address 000203 (since Y needs only one byte). Even though Z is supposed to be the address of a word, MACRO-11 will assign the next available address, even if it is that of an odd byte. Since your program probably has some word-oriented instruction such as

```
MOV Z,ABC
```

when you eventually execute it, an "M-TRAP to 4" will result because of a boundary alignment violation. The odd address assigned to Z is not compatible with the implied move-the-word-at-Z operation.

The solution? A new directive .EVEN, which has no operand and should have no label, can be used. When MACRO-11 encounters a .EVEN, if the next storage location it is about to allocate has an odd address, MACRO-11 will add one to it, forcing it to be even. This leaves a one byte "hole" in your program; but presumably you are not referencing its content so no harm is done, other than wasting a byte. If the next available address happens to be even, then the .EVEN is ignored.

You should group all your byte-oriented directives in one or two groups, ending each group with a .EVEN. In this way you will avoid having too many 1-byte holes in your programs.

It is also advisable to ensure that each module ends with an even address so that as the linker splices the next module on to the previous module, each module begins with an even address. So if your module finishes with some byte-oriented directives, follow them with a .EVEN, just before the .END.

Memory space for bytes may be reserved using

.BLKB, .ASCII, .ASCIZ

```
[label:]  .BLKB  count
```

Here the count is an octal number representing the desired number of bytes. Previously, in the .BLKW, the count was the desired number of words.

One of the pleasant aspects of using MACRO-11 is that it has memorized the 128 codes in the ASCII character set. We can evoke them in two ways:

```
[label:] .ASCII /chars not including slash/
```

or

```
[label:] .ASCIZ /chars not including slash/
```

In both cases, all the characters between the delimiters (/), including blanks, will have bytes allocated to them, with the corresponding ASCII codes placed in each byte. The second directive .ASCIZ (ASCI-Zero) will add one extra byte at the end and fill it with 8 bits of 0. For example,

```
.ASCII /A/ →   .BYTE 101
.ASCIZ /B/ →   .BYTE 102,000
.ASCIZ /0/ →   .BYTE 060,000
.ASCII /A0/ →  .BYTE 101,060
```

We see the .BYTE equivalents for the corresponding .ASCII and .ASCIZ.

The operands of .ASCII and .ASCIZ are the only ones in which lowercase characters are not automatically converted into uppercase characters by MACRO-11. The following two lines are equivalent:

```
.ASCII   /A a/          ; notice A-space-a
.BYTE    101,040,141 ; A, space, a
```

Sometimes we may wish to work with ASCII codes that have no visible print representation, such as a NUL (code 000) or a horizontal tab (HT, code 011). We can append such items following the second delimiter of a .ASCII by enclosing them in diamond braces <. . .>. So we can now say the following two statements are equivalent:

```
.ASCIZ   /ABC/
.ASCII   /ABC/<0>
```

Suppose you needed to have a "/" as part of an ASCII string. To encode

```
CALCULATE X/Y FIRST
```

you could write

```
.ASCII  /CALCULATE X/
.BYTE   057       ; ASCII code for /
.ASCII  /Y FIRST/
```

You can instead use a .ASCII with some other delimiter, as in

```
.ASCII  ?CALCULATE X/Y FIRST? ; using ?
```

Bytes and Registers

Using byte operations with registers takes some care.

```
MOV X,%0
```

is clear in its intent, provided that X corresponds to a word. It means, take the 16 bits from location X and copy them into register 0. What of a

```
MOVB Y,%0
```

Location Y can be either an even or an odd address. In either case, the 8 bits at that address will be fetched from memory. Where will they go? Since the registers are 16 bits wide, and there is no further specification we can use to say which 8 of the 16 bits of register 0 will be affected, the problem is resolved by decree of the computer designers.

The PDP-11 is built so that the rightmost 8 bits (the low order bits) of the register will be used as the destination (or the source, as the case may be), and the leftmost bits (the high order bits) will not be affected, with one significant exception.

An INCB %1, or a DECB %2, or a CMPB ABC,%3, etc., will only affect or involve the low 8 bits of registers 1, 2, and 3, respectively. However, MOVB X,%3 will copy the 8 bits from location X into the low 8 bits of register 3, *and* propagate the top (leftmost) bit of the byte it just copied through the other 8 bit positions of register 3.

Consider the effect of a MOVB K1,%1 when %1 has the value 177777 in it, and K1 contains the byte 123:

%1 Before	%1 After
177777	000123

This result is due to the fact that the byte 123 has a leading 0 bit (01 010 011). Conversely, a MOVB K2,%2 with %2 being 012012 and K2 at 303 leads to

%2 Before	$2 After
012012	177703

since 303's leading bit is a 1.

The MOVB is the only instruction operating on bytes which forces this bit propagation, which is called sign extension. It turns out to be useful, as we will see shortly. A MOVB with a memory location as the destination address will not propagate the sign; it only happens with a MOVB into a register.

In every case, every execution of a MOVB will cause the CC to be updated, based on the 8 bits that were copied, with the high order bit serving as a sign bit.

Conditional Branch Instructions

The CC reflects the state of the most recent result, in so far as it is negative or zero or some combination of these or their opposites. To facilitate writing programs in machine-language, we used "skip if x" instructions which examine the CC bits. If you are forced to write machine language programs, they are an acceptable compromise. If you can use the assembler, then you should use the complete set of conditional and unconditional branch instructions. We will introduce a subset of them now.

Offsets

MACRO-11 recognizes the mnemonic BR as an unconditional branch instruction. It expects the operand field to be a symbol which appears as the label of some instruction in the source module. However, instead of storing the address corresponding to that symbol in the second word of the BR instruction, the assembler calculates how far that location is, relative to the location immediately following the first word of the BR instruction. Provided that the distance is not over 127 words in the forward direction, or 128 words in the backward direction, the distance will be divided by two and stored in the low 8 bits of the first word of the BR instruction. Because of the constraints on the maximum distance allowed, and the division by two, the result, called an *offset*, will always fit in 8 bits. Division by two loses no essential information, since the original distance is always an even number.

The same process is involved in assembling all the Bxx instructions. MACRO-11 calculates an offset and stores it within the single-word Bxx instruction. When the object program is executed, the control unit picks out the low 8 bits (where the offset is stored) from the IR, doubles that number, adds it to the number found in the PC, and if the branch condition is satisfied, forces that result into the PC, thus effecting the desired transfer of control.

Figure 5.3 Assembly printout excerpt.

```
Location    Content     Source      Statement
000010      016701      MOV         X,%1
            000030
000014      001402      BEQ         AHA
000016      000167      JMP         NO
            000102
000022      010102      AHA:MOV     %1,%2
                          ...
```

Consider the excerpt from an assembly printout, as shown in figure 5.3. The assembler equates the symbol AHA with the address octal 22. The assembler expects that when the "BEQ AHA" at location 14 is being executed, the PC has already been updated to have the value 14 + 2, or 16. The distance from 16 to 22 octal is 4. By having the assembler store half this distance (since it will always be even, storing half loses nothing) as the offset for "BEQ AHA," giving 001400 + 2, or 001402, the control unit can reconstruct at run time the desired address by extracting the offset (here, a 2), doubling it (getting 4), adding it to the PC value (now 16), getting the desired instruction address (16 + 4 → 22 octal), which corresponds to AHA, as desired. Note the similarity between 001402 and our old SKZ.

This use of offsets is a form of PC relative addressing, which makes it easier to load a program into any available memory. If a block of PC relative instructions is moved as a block to any part of memory, the distances between instructions or data are not changed so the offsets remain correct, and the program will execute correctly.

Conditional Branch Subset

We can test the latest result produced by a data manipulation instruction by using the subset of the branch instructions shown in figure 5.4.

The symbol "do" means destination offset, and "da" is the destination address that will be regenerated at execution time. What if the distance you wish to branch to is too far? If an unconditional branch BR is involved, then you can use an unconditional JMP instead. It does not use an 8-bit offset. The JMP uses a whole second word to provide the destination address. If a conditional branch is involved, there are no conditional jump instructions you can turn to. You have to use combinations of conditional branches with the one and only unrestricted unconditional jump JMP. For instance, if you wanted to use

```
ADD     X,TOTAL
BEQ     DONE
BR      LOOP
...
```

```
BR   do ;            da→PC
BPL  do ; if N=0,  da→PC Plus
BMI  do ; if N=1,  da→PC Minus
BEQ  do ; if Z=1,  da→PC Equals 0
BNE  do ; if Z=0,  da→PC Not equal to 0
```

Figure 5.4 Some conditional branches.

and the assembler complained that "DONE" was too far for its offset to fit into the 8-bit offset field in the BEQ, you could write

```
        ADD    X,TOTAL
        BNE    SKIP
        JMP    DONE
SKIP:   BR     LOOP
        ...
```

As we stated earlier, in almost every instance when you could use a memory address within an instruction, you could substitute a register reference instead. However, from our previous discussion of offsets in branch instructions, you might conclude that branch instructions do not use ordinary memory addresses, so they may be special in other ways. For instance, you might be tempted to try a

```
BR %2
```

This is not allowed. None of the branch instructions Bxx can use anything but a plain unadorned symbol in their address fields, one which can be converted into an 8-bit offset. We will see addressing modes soon that will provide the desired flexibility when using a JMP.

Summary

The PDP-11 provides a set of six general purpose registers, which can be used freely in place of memory locations with most instructions. No new instructions are needed to take advantage of these registers. The assembler recognizes that a register reference is indicated when the % sign is used. Using registers can result in significantly shorter programs, which also have a shorter execution time.

The importance of nonnumeric applications has led to directly supporting operations on characters and character strings. Since most character codes use 6 to 8 bits per character, the unit of information consisting of 8 bits has come to be known as a byte. On the PDP-11, each byte in memory is directly addressable. Addresses for bytes are 16-bits wide. Rather than have a special set of instructions to operate on bytes, most PDP-11 data manipulation instructions have byte counterparts. The most significant exceptions to this general rule are the ADD and SUB instructions.

The assembler facilitates byte manipulations by supporting the .BYTE and .BLKB directives for byte-size constants and to reserve memory for consecutive bytes. More often than not, bytes are used in conjunction with character manipulations, and the assembler facilitates the use of the ASCII codes by supporting the .ASCII and .ASCIZ directives.

The process of allowing the use of odd addresses (for bytes located at odd locations) gives rise to the possibility of having word-oriented instructions (e.g. MOV, ADD, etc.) inadvertently reference bytes at odd locations. The PDP-11 control unit objects to doing this. The .EVEN directive is provided so that the assembler may cooperate in forcing even addresses where desired.

In line with the improvements in making programs more compact and rapid by appropriate use of registers, one-operand instructions are available for some frequently used operations. CLR, INC, DEC, and NEG are available to clear, add one, subtract one, and find the arithmetic negative of the specified word operand. These instructions are also available as byte-oriented instructions (CLRB, INCB, DECB, NEGB).

The branch instruction BR and the conditional branch instructions BPL (branch if plus), BMI (branch if minus), BEQ (branch if equal to zero), and BNE (branch if not equal to zero) were introduced. They examine but do not change the CC bits. None of them can use registers to specify their destinations. They can only use 8-bit offsets. Fortunately the assembler will compute these offsets for us. We need only provide an ordinary symbolic reference. The branch instructions have the appearance of using full-width memory addresses. In fact, they do not. This has two advantageous consequences:

1. All branch instructions are one word long.
2. All branch instructions are PC-relative.

The PDP-11 has many more branch instructions and many ways of exploiting the registers we have just introduced. We are now about to embark on an interesting exploration.

Exercises

5.1 Rewrite the following statement as a PDP-11 symbolic program (assume the operands are integers):

if (A>C and C<D) then set X= C+D or else set X= A−B.

5.2 Rewrite the pay program of chapter 2 (figure 2.12) using registers to hold the regular, overtime, and total pay figures. Assume that the pay rate is not larger than ten.

5.3 Give a plausible explanation for MACRO-11's unwillingness to accept an R=%1 renaming after use has been made of the symbol R.

5.4 The concept of parity was defined in an earlier problem. ASCII codes often have a parity bit prefixed to them, making them into 8-bit codes. Assume that the parity bit is chosen so that the 8-bit codes all have odd parity. Write down the "new" octal ASCII codes for the letters A, B, and C.

5.5 Would any 1-bit error in sending or receiving the 7-bit ASCII codes ever be undetectable? Support your answer. If you were using 8-bit ASCII with odd parity, would the situation change? Explain.

5.6 What is the range (i.e., the maximum distances which a reference can encompass) of a BR? What is the range of a JMP?

5.7 Encode, by hand, the message: "help me NOW!" using the octal ASCII codes in one .WORD statement.

5.8 Interpret the following series of octal words as one ASCII character string:

064124 071551 064440 020163 020040 020141 042524 052123
005056

Explain why things seem scrambled.

5.9 Under what circumstances are two consecutive .BYTE 0 statements not equivalent to one .WORD 0 statement?

5.10 If you have the statement

```
.ASCIZ /ABC/<12>
```

does the null code come before or after the 12 code?

5.11 Define each item; why are they important (be brief).

(a) PS
(b) PC
(c) Boundary alignment

5.12 One often sees the following sequence of statements being used:

```
A:    .ASCII /TESTING/
        .EVEN
B:    .BYTE 0
        .EVEN
X:    .ASCII /ERROR IN INPUT/
        .EVEN
          ...
```

What misunderstanding concerning the use of .EVEN is this indicative of?

5.13 How many bytes does each of the following generate?

```
(a) .ASCII     /6 WEEK EXAM/<12>;
(b) .ASCIZ    /HELLO/ ;
```

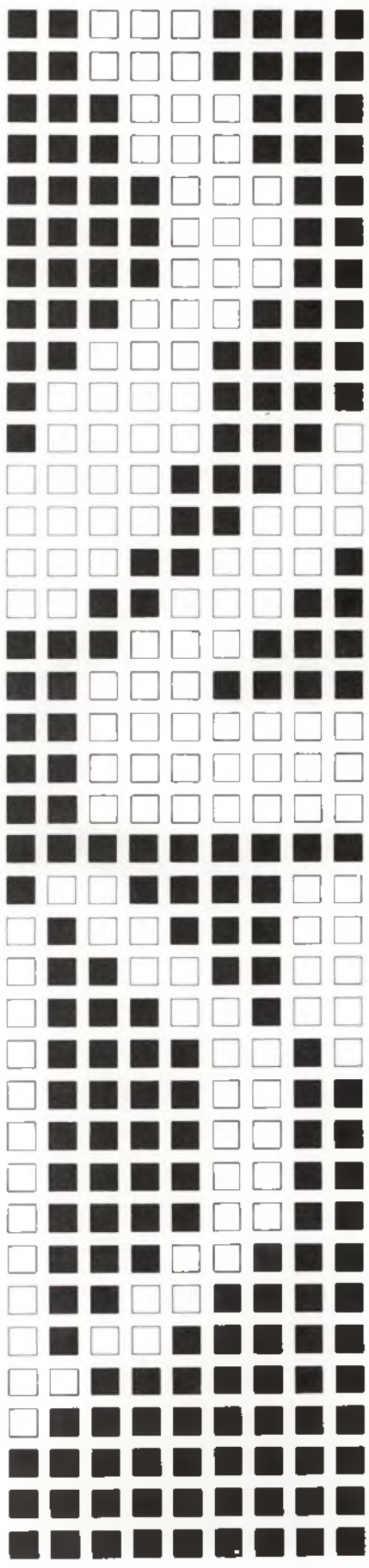
5.14 If

```
.ASCII /ABC/<12>
```

was the first statement in your program, what would appear in the first two words of an octal dump (assuming that the data were not modified)?

6

key addressing modes



How can we arrange things so that the address used by an instruction—say a CLR instruction—will be different each time the instruction is executed without the instruction being modified or changed each time? Such a capability is called *indexing*.

Indexing

The need for indexing was perceived in the early years of computing and led to the concept of index registers (not totally dissimilar to the notion of indexing tax rates). If we focus on a single operand instruction—say, a CLR— as it appears in the IR, we have:

```
CLR d
```

The intent, and effect, is: 0 → d. Suppose we could arrange to insert an extra field in the instruction, one we will represent as R, which can designate a register we will call an index register. We can now write:

```
CLR d(R) ; 0 → d + (R)
```

MACRO-11 recognizes this notation, d(R), as a request for use of indexing in conjunction with register R (which would normally be one of %0, %1, ..., %5).

The interpretation of CLR d(R) is: the first part, d, is a 16-bit field, usually designating a memory address. The second field, symbolized by (R), designates a register which, in turn, contains some 16-bit number (whose value is, of course, not known to the assembler, since it will not be set until execution time, and even then it will usually vary). At execution time the number corresponding to d and the value in the register R will be added at the request of the control unit to obtain the *effective address* (EA for short). It is the effective address which is used to locate the source or the destination operand, as the case may be. Unless otherwise specified, nothing in the instruction or index register changes. The original instruction is *never modified*.

Consider the example:

```
CLR %0      ; 0 → %0
CLRB A(%0)  ; 0 → A + 0, since (%0) is now 0
INC %0      ; 1 + (%0) → %0, giving 1
CLRB A(%0)  ; 0 byte → A + 1
```

In the above example the effective address for the first CLRB is computed by the control unit using the number corresponding to the address A and the value then found in the specified register; namely, the 0 found in %0. So the EA is A + 0, or simply A. In the second CLRB the same two items are involved (A and %0), but this time the EA is A + 1, since (%0) is now 1.

Using Indexing

We can finally rewrite the machine-language program to add a set of numbers without being forced to have the program modify itself:

```
        CLR     %0      ; use %0 for indexing
        CLR     %1      ; use %1 for total
        MOV     LIM,%2  ; use %2 to count down
LOOP:   ADD     A(%0),%1
        ADD     TWO,%0
        DEC     %2
        BNE     LOOP
        HALT
TWO:    .WORD   2
A:      .WORD 10,23,-2,5,12
; see ``.WORD and .BYTE Revisited,'' this
chapter.
LIM:    .WORD   5               ;
```

The effective addresses generated each time the ADD at LOOP is executed are: A + 0, A + 2, A + 4, A + 6, A + 10.

The PDP-11 supports indexing, but it does not have a special set of registers reserved for this purpose. All of the 8 registers are used for indexing, but we will use only registers 0–5 explicitly for indexing. As we will see, registers 6 and 7 are used for indexing in a covert way.

As we proceed, if we wish to understand how these addressing modes are supported by the hardware, we have to consider how they are seen by the control unit.

Mode and Register Fields

We express the kind of addressing we want to use in an instruction by writing the notation acceptable to the assembler. The assembler, in turn, encodes our symbolic request for a particular kind of addressing by using a 6-bit field within the encoded instruction, one 6-bit field being used for each of the instruction's operands. When the machine-language program is being executed, these 6-bit fields are interpreted by the control unit as consisting of two 3-bit fields, called the *mode field* and the *register field.*

The 3-bit mode field, or mode code, specifies the way we want the control unit to compute the effective address for the corresponding operand. The 3-bit register field indicates which register should be involved in computing this effective address. In short, the mode field, always accompanied by a register field, specifies the addressing mode we desire.

A single operand instruction is encoded as follows:

```
        opcode mode register          [extra word]
  ←          16 bits          →   ←      16 bits      →
```

The second word may or may not be present, depending on the mode in the first word. The first word will always have the form

```
opcode      mode      register
←10→        ←3→       ← 3 →     (bits)
```

In the simplest case, that of a `CLR %1,` we have the encoding in bits:

```
0 000 101 000          000    001        or 0050   0    1
      CLR               M      R             CLR   M    R
```

The destination field of the `CLR %1` refers to register 1, so we find the code 001 (1 octal) in the R field. The "mode code" associated with a direct reference to a register, the so-called register mode, uses code 000 (0 octal), which we find in the M field. The opcode field, 10 bits wide here, is filled in with the code for a CLR, namely 0500 octal.

Symbolically, each single operand instruction has the form:

OPCODE	M	R	[2nd word]
10	3	3	[16]

In a similar fashion, the double operand instructions such as MOV s,d can be represented as:

OPCODE	S M	S R	D M	D R	[SS wrd]	[DD wrd]
4	3	3	3	3	[16]	[16]

Each symbolic address s (for source) and d (for destination) is encoded in the SS and DD fields. Depending on the type of addressing which is implied, the corresponding mode field will be filled in with a code between 0 and 7. Similarly, the R fields will be filled in with a register address. The second or third words will either not be generated at all or will be filled in as the corresponding mode dictates. If the second word is not needed and the third is needed, obviously the middle word will not be wasted. The third word will simply occupy the space which would otherwise have been occupied by the second word. For example, a `MOV %1, %2` is encoded as:

```
0 001     000    001    000    010
 OP        M      R      M      R
              SS            DD
```

Only one word is required because no memory references appeared.

`MOV A,B` might be encoded as:

```
016767
000120   ← for A
000336   ← for B
```

whereas `MOV C,%2` might be encoded as:

```
016702
000444   ← for C
```

while `MOV %3,D` might be encoded as:

```
010367
000654   ← for D
```

Explaining the use of mode 6 with register 7 (the "67" which appears in the preceding SS and DD fields of the MOV instructions) will require some additional background. At this stage we are concerned only with seeing how the use of the various addressing modes affects an instruction's length.

Index Mode

The symbolic form `CLR ABC(%4)` is encoded using mode code 6:

```
0050   6   4    001050
 OP    M   R     ABC
```

For a double operand instruction such as `CMPB A(%2),B(%3)` we have:

```
12   62   63   002000      002471
OP   SS   DD   "A" adr     "B" adr
     MR   MR
```

In each case the number in the register is used to compute the effective address, but the register is not changed. The CC is updated according to the result of the whole instruction, not according to the effective address calculations.

Indexing and Offsets

The indexing we have discussed has the base address of a set of data items as part of the instruction, with desired increments, which can be thought of as offsets (full offsets, not encoded offsets, as in Bxx instructions) being provided by the specified register. Symbolically, we had:

OP d(R) ; effective adr = d + (R)

We are not limited to having only addresses in the d field! Any number or symbol reducible to a 16-bit field can be used, provided that the end result, the effective address computed at execution time, is valid. Symbolically, we can have:

OP offset(R) ; effective adr = offset + (R)

In the following three source statements, the effective address calculation which is implied is shown in the comments field.

```
CLR 0(%5)        ; EA = 0    +  (%5)
INC 2(%4)        ; EA = 2    +  (%4)
DECB -1(%3)      ; EA = -1   +  (%3)
```

If registers 5, 4, and 3 held the addresses P, Q, and S respectively, then the corresponding effective addresses would be P, Q + 2, and S-1.

Deferred Addressing

The use of an index register with a zero base address or offset turns out to be so useful that a special mode code is assigned to it, eliminating the need for you to write the zero, and saving both memory space (for the implied zero) and a memory access as well. MACRO-11 recognizes the notation:

```
OP (Reg)
```

as a use of mode 1 addressing, which is called *register deferred* addressing. It is a form of indirect addressing.

Conceptually, the following two CLRs are equivalent:

```
CLR   0(%2) ; indexing
CLR   (%2)  ; reg deferred
```

They both result in the effective address (%2) being used to locate the operand. They differ in that the first requires two words for its machine-language representation (a 16-bit zero is encoded as part of this instruction) while the second instruction needs only a single word for its machine-language representation. They both leave %2 intact. The difference between these two CLRs and

```
CLR %2
```

is *very* important:

```
CLR   %2    ; 0 → %2
CLR   (%2)  ; (%2) → EA, 0 → EA
CLR 0(%2)   ; 0 + (%2) → EA, 0 →EA
```

In using index or register deferred mode, the register is used to generate an effective address EA, and the register is *not* modified in the process. Consider these examples:

```
CLR   %2 ; 0 → %2
CLR (%3) ; EA = 3000, 0 → 3000
```

	Before	After
%1	000000	000000
%2	002000	000000 *
%3	003000	003000
memory locations		
2000:	123456	123456
3000:	007777	000000 *

The * marks the only values that were changed. Register deferred mode is provided to save space and time when indexing with a zero offset would otherwise have been used. It avoids having to set aside a whole word for the offset.

Auto-Increment Mode

In all of the addressing modes we have seen so far we had to explicitly write some instruction (e.g., `ADD TWO,%2`) in order to cause the effective address generated by an instruction to change each time. In both the following instructions:

```
CLR A(%1)
INC  (%2)
```

each time these are executed, the same memory locations—A + (%1) and (%2),respectively—will be used over and over again, unless we do something to change the values held in %1 and %2.

Stepping

There are many situations in which you wish to process a group of consecutively stored items, in consecutive order. For instance, you may wish to zero a table of several words. Using indexing, you could write the following code segment:

```
        CLR     %1
LOOP:   CLR     TABLE(%1)
        ADD     TWO,%1
        CMP     %1,LIMIT
        BMI     LOOP
        ...
TWO:    .WORD   2
LIMIT:  .WORD   40
TABLE:  .BLKW   100
```

Since addressing of consecutive words is so common, the PDP-11 has an addressing mode just for this purpose. Addressing mode code 2, called *auto-increment* addressing, is specified using the following notation:

OP (R) +

The effective address is fetched from register R, the OP is executed, and then the content of R is *modified* by the control unit, which will force 2 to be added to the value in R. So the next time you execute this instruction, it will refer to the next consecutive word (provided that you have not modified R yourself in the meantime).

We can rewrite the preceding program so that it uses auto-increment addressing:

```
        MOV     ADTAB,%1 ; put base adr in %1
L2:     CLR     (%1)+
        CMP     %1,LIM
        BMI     L2
        ...
ADTAB:  .WORD   TABLE
TABLE:  .BLKW   100
LIM:    .WORD   LIM     ; use adr as limit
```

Remember, OP (R)+ means OP will use the number in R as the effective address, then *after* performing the operation, automatically effect (R) + 2 → R, and the control unit does this for us. We no longer have to provide the constant 2 or the ADD to use it. In this example it is important that the statement:

```
LIM:      .WORD    LIM
```

immediately follow the statement:

```
TABLE:    .BLKW    100
```

since the program is using the address in %1 to control the number of iterations. You could, however, replace the LIM statement with:

```
LIM:      .WORD    TABLE+200
```

Can you explain why this should work as well?

Why So Many Variations?

Why are both indexing and auto-increment addressing supported on the PDP-11? Since auto-increment addressing does so much for you, why should you ever use indexing?

Indexing is necessary to provide direct access (sometimes called random access) to items when they are not being processed sequentially. Consider the following application.

An automated teller unit (ATM) provides the customers of a bank with twenty-four-hour-a-day banking services from multiple locations. An ATM station may provide the following services:

1. deposit to savings	DTS
2. deposit to checking	DTC
3. withdrawal from savings	WFS
4. withdrawal from checking	WFC
5. transfer to savings	TTS
6. transfer to checking	TTC
7. loan payment	LP

The customer uses a special keyboard to select the desired service. Let us assume that pushing a digit key leads to the binary equivalent of that digit being placed in a memory location called KEY. We can then process the request as follows:

```
            CMP     KEY,ONE
            BEQ     DTS
            CMP     KEY,TWO
            BEQ     DTC
            ...
            CMP     KEY,SVN
            BEQ     LP
            BR      ERROR
DTS:        ...
DTC:        ...
            ...
LP:         ...
ONE:        .WORD   1
            ...
SVN:        .WORD   7
```

The exhaustive sequence of comparisons may be suitable for checking a short list of possibilities, but it is not practical when many cases are involved. Consider this alternative:

```
            MOV     KEY,%1      ; (KEY)→ %1
            DEC     %1
            ADD     %1,%1       ; double it
            ADD     %1,%1       ; double it again
X:          JMP     JT(%1)
JT:         JMP     DTS         ; case 1
            JMP     DTC         ; case 2
            ...
            JMP     LP          ; case 7
```

The effective address for the JMP at X selects the appropriate JMP from the table at JT. So for a keying of 2, we would compute (2−1) * 2 * 2 in %1, selecting JT+4, which is JMP DTC for case 2. This is a form of a "jump table" analogous to the case statements or computed go-to's found in some high level languages. We will soon be able to eliminate all the JMPs in the table, making this useful technique even more attractive.

Recall that a JMP R is illegal. However, the other addressing modes can be used with JMP:

```
JMP     d(R)    ; indexing, d+(R) → PC
JMP     (R)     ; reg deferred, EA=(R), EA → PC
JMP     (R)+    ; auto-inc, as above, then
                  (R)+2 → R
```

Of course, no branch instruction can ever use a register as part of its destination address; only a JMP may do so.

.WORD and .BYTE Revisited

In our first encounter with .WORD we only described its use with a single octal constant. The time has come to reveal the generality of the .WORD operand field. Consider the following:

```
[label:] .WORD expr[,expr] *
```

Recall that "[. . .]" means "optional." The character "*" here means "any number of occurrences of the preceding item (even none)." The symbol "expr" stands for "expression," and this can be any number representable in 16 bits. It may be symbolic (e.g., a user-defined name). It may involve some limited arithmetic. So we can have:

```
.WORD TABLE,TABLE+2 ; 2 addresses
.WORD 2*N, -3, TABLE
```

If N was previously defined as

```
N = 10 ; octal ten
```

then MACRO-11 would evaluate 2 * N as 20 octal and store it in place of 2 * N. If TABLE was a user-defined label, then the addresses corresponding to TABLE and TABLE + 2 can be written as valid operands for .WORD.

Similar comments apply to valid operand fields for the .BYTE directive, except that each item must fit in an 8-bit byte.

The full set of rules for MACRO-11 expressions is lengthy and esoteric. Since our purpose is not to become experts in all the fine points of MACRO-11, we won't pursue expressions much further. We have almost all the important information on hand right now.

Copying

Copying things is a common occurrence. Suppose you wish to "move" a character string. You don't really move it; you duplicate it by copying it to a new area of memory. We could do the following:

```
S1:     .ASCIZ  /This is a test/
S2:     .BLKB   100
        .EVEN   ; just in case
        ...
GO:     MOV     ADRS,%1
        MOV     ADRS+2,%2
LOOP:   MOVB    (%1)+,(%2)+
        BNE     LOOP
        ...
ADRS:   .WORD   S1,S2
```

The program begins at GO, and places the addresses of the source string S1 and the destination string S2 in registers 1 and 2, respectively. Then it will perform:

```
(S1) → S2              ; byte 1
(S1 + 1) → S2 + 1 ; byte 2
...
(S1 + n) → S2 + n ; byte n
```

Each MOVB updates the CC; the last byte of S1 is the 0 generated by the .ASCIZ. When that 0 has been copied, the loop stops. How is it that the increment used by the control unit in modifying registers 1 and 2 is the number 1, not the number 2? When stepping over consecutive bytes, we would want to use an increment of 1. So when auto-increment addressing is used with byte operations, the control unit will always use a constant of 1 to modify the registers.

If we were operating on words, as in the earlier example, the increment provided by the control unit will be 2. Whether bytes or words are involved, the auto-incrementation will always occur after the register has been used. Symbolically:

```
OPW (R)+ ; use EA in R, then (R) + 2→ R
OPB (R)+ ; use EA in R, then (R) + 1→ R
```

Odds and Ends

Many little things can go wrong in writing a program. Many annoying little details must be attended to. Most assemblers try to be helpful in this regard. MACRO-11 provides the following useful aids.

Decimal Operands

You have in mind to prepare output for a line printer that handles lines of 120 characters. So you reserve space in your program by using a .BLKW 120, and when you come to test your program, it fails. The printer uses 120 decimal print positions; the assembler took it as 120 octal (80 decimal). When your program executed, the items following the .BLKB 120 were overwritten because the 80 (decimal) character buffer was too small (a buffer is an area of memory into which information is copied on its way to some other destination).

If you had written .BLKW 120. the assembler would take the "120." as a decimal number. The same holds true wherever numeric constants have been used. So we can write statements such as:

```
.BLKB 120. ; reserve 120 dec bytes
.BYTE 10.,100. ; decimal constants
.WORD 1000.,-500.,10; two dec and one
  octal const
```

.RADIX

Having come this far, if you still don't feel comfortable with octal, you could request that all your numeric constants be interpreted as decimal numbers, without even using the period as above. The directive `.RADIX 10` will force all subsequent numeric constants to be taken as decimal numbers, until the next `.RADIX` occurs. You can switch back and forth (not recommended, since it is very confusing) by also using `.RADIX 8`, which restores the default octal interpretation. You can mystify everyone by using `.RADIX 2` and `.RADIX 4` should you want to write either bits or base-4 numbers. A plain `.RADIX` assumes you mean `.RADIX 8.` The operand of a `.RADIX` is always taken as a decimal number.

Immediate Operands

It is a nuisance to have to make up names and set aside space for constants, addresses, etc. The *immediate operand* construction is another addressing mode. It is evoked by using the character # in front of the desired symbol or constant, as in:

OP #m,dest

The number or symbol "m" will be used when OP is being executed, not as the address of the desired operand (the "normal" situation), but as the actual operand itself. Thus the name "immediate operand." This construction really only makes sense when double operand instructions are being used. So we could write:

```
ADD #33,SUM      ; 33 + (SUM) → SUM
CMP ABC,#100.    ; (ABC) vs 100.
SUB #60,%1       ; (%1)-60 → %1
MOV #TABLE,%0    ; address ``TABLE'' → %0
```

We can now shorten, simplify, and speed up a previous example:

```
S1:     .ASCIZ  /This is a test/
S2:     .BLKB   100      ; 100 oct
        .EVEN
        ...
        MOV     #S1,%1 ; adr ``S1'' to %1
        MOV     #S2,%2 ; adr ``S2'' to %2
LOOP:   MOVB    (%1)+,(%2)+
        BNE     LOOP
        ...
```

You may be tempted to write expressions such as

```
ADRX:   .WORD #ABC
```

to attempt to make the address ABC available as a constant. This attempt to "force" ABC to be used as an address will be thwarted by the assembler, since it has no meaning. All the addressing modes of the PDP-11 are meaningful only when applied to addresses appearing as part of instructions. A .WORD is a mere directive, not an instruction. When an addressing mode is used within an instruction, the assembler substitutes the appropriate 3-bit mode code in the M field of the instruction. Th
are no M (mode) fields in which to "hide" mode codes when u
.WORD. Furthermore, they are not needed. A simple

```
ADRX:   .WORD ABC
```

is sufficient to have the address correspondi
location labeled ADRX.

The assembler encodes a reque
using mode code two, and it s
of the PC. The consta
mode code two

Ad

(b) `JMP  %4`

(c) `MOV  #99,%1`

(d) `BR  (%5)`

(e) `JMP  (%5)`

(f) `BEQ  A+700`

6.4 Show the machine language the assembler generates given the following text.

```
CONST = 10
      = %0
RO    MOVB  (%1),RO
      CLR   44(%1)
      ADD   #CONST, (%0)+
START: HALT
      .WORD 27.
      .END
A:
```

6.5 Write a PDP-11 symbolic program using 10 or fewer instructions which, given a character string ending with a 0 byte, creates a new string with all the characters reversed. Thus, given

```
A: .ASCII /ABCDEFGHZXC/<0>
```

the new string would be

```
CXZHGFEDCBA
```

6.7 True (T) or False (F)?

(a) Uninitialized memory locations are assumed to have the value 0.
(b) BR should execute almost twice as fast as JMP.
(c) `MOV #10,%1` uses 2 memory references during its fetch-execute cycle.
(d) Some conditional branch instructions can modify the CC.
(e) `DEC %3` and `SUB #1,%3` have the same effect.

6.8 Briefly explain what the following program segment does. Write down the final content of AL, AL+1,...,AL+4.

```
START:  MOV #LINE,%1
        MOV #AL, %2
        MOV #15, %3
L1:     CLR (%2)+
        DEC %3
        BNE L1
L2:     CMPB (%1), #12
        BEQ    FND
        MOVB (%1) +,%3
        SUB A, %3
        BMI L2
        INC AL(%3)
        BR L2
        ...
A:      .ASCII /A/<0>
AL:     .BLKW 15
LINE:   .ASCII /THIS IS A TEST SEQUENCE/<12>
        ...
```

6.9 For each of the following instructions, assume the indicated initial conditions. Indicate all changes to these items produced by executing these instructions. The initial conditions apply to each instruction. Each instruction is assumed to begin at location 1000. Use — to indicate no change.

	%1	%2	1200	1202	PC
	1200	1202	1202	1200	1000
...ations					
...(%2)					

.WORD and .BYTE Revisited

In our first encounter with .WORD we only described its use with a single octal constant. The time has come to reveal the generality of the .WORD operand field. Consider the following:

```
[label:] .WORD expr[,expr] *
```

Recall that "[. . .]" means "optional." The character "*" here means "any number of occurrences of the preceding item (even none)." The symbol "expr" stands for "expression," and this can be any number representable in 16 bits. It may be symbolic (e.g., a user-defined name). It may involve some limited arithmetic. So we can have:

```
.WORD TABLE,TABLE+2 ; 2 addresses
.WORD 2*N, -3, TABLE
```

If N was previously defined as

```
N = 10 ; octal ten
```

then MACRO-11 would evaluate 2 * N as 20 octal and store it in place of 2 * N. If TABLE was a user-defined label, then the addresses corresponding to TABLE and TABLE + 2 can be written as valid operands for .WORD.

Similar comments apply to valid operand fields for the .BYTE directive, except that each item must fit in an 8-bit byte.

The full set of rules for MACRO-11 expressions is lengthy and esoteric. Since our purpose is not to become experts in all the fine points of MACRO-11, we won't pursue expressions much further. We have almost all the important information on hand right now.

Copying

Copying things is a common occurrence. Suppose you wish to "move" a character string. You don't really move it; you duplicate it by copying it to a new area of memory. We could do the following:

```
S1:     .ASCIZ  /This is a test/
S2:     .BLKB   100
        .EVEN   ; just in case
        ...
GO:     MOV     ADRS,%1
        MOV     ADRS+2,%2
LOOP:   MOVB    (%1)+,(%2)+
        BNE     LOOP
        ...
ADRS:   .WORD   S1,S2
```

The program begins at GO, and places the addresses of the source string S1 and the destination string S2 in registers 1 and 2, respectively. Then it will perform:

```
(S1) → S2              ; byte 1
(S1 + 1) → S2 + 1 ; byte 2
          ...
(S1 + n) → S2 + n ; byte n
```

Each MOVB updates the CC; the last byte of S1 is the 0 generated by the .ASCIZ. When that 0 has been copied, the loop stops. How is it that the increment used by the control unit in modifying registers 1 and 2 is the number 1, not the number 2? When stepping over consecutive bytes, we would want to use an increment of 1. So when auto-increment addressing is used with byte operations, the control unit will always use a constant of 1 to modify the registers.

If we were operating on words, as in the earlier example, the increment provided by the control unit will be 2. Whether bytes or words are involved, the auto-incrementation will always occur after the register has been used. Symbolically:

```
OPW (R)+ ; use EA in R, then (R) + 2→ R
OPB (R)+ ; use EA in R, then (R) + 1→ R
```

Odds and Ends

Many little things can go wrong in writing a program. Many annoying little details must be attended to. Most assemblers try to be helpful in this regard. MACRO-11 provides the following useful aids.

Decimal Operands

You have in mind to prepare output for a line printer that handles lines of 120 characters. So you reserve space in your program by using a .BLKW 120, and when you come to test your program, it fails. The printer uses 120 decimal print positions; the assembler took it as 120 octal (80 decimal). When your program executed, the items following the .BLKB 120 were overwritten because the 80 (decimal) character buffer was too small (a buffer is an area of memory into which information is copied on its way to some other destination).

If you had written .BLKW 120. the assembler would take the "120." as a decimal number. The same holds true wherever numeric constants have been used. So we can write statements such as:

```
.BLKB 120. ; reserve 120 dec bytes
.BYTE 10.,100. ; decimal constants
.WORD 1000.,-500.,10; two dec and one
  octal const
```

.RADIX

Having come this far, if you still don't feel comfortable with octal, you could request that all your numeric constants be interpreted as decimal numbers, without even using the period as above. The directive `.RADIX 10` will force all subsequent numeric constants to be taken as decimal numbers, until the next `.RADIX` occurs. You can switch back and forth (not recommended, since it is very confusing) by also using `.RADIX 8`, which restores the default octal interpretation. You can mystify everyone by using `.RADIX 2` and `.RADIX 4` should you want to write either bits or base-4 numbers. A plain `.RADIX` assumes you mean `.RADIX 8.` The operand of a `.RADIX` is always taken as a decimal number.

Immediate Operands

It is a nuisance to have to make up names and set aside space for constants, addresses, etc. The *immediate operand* construction is another addressing mode. It is evoked by using the character # in front of the desired symbol or constant, as in:

```
OP #m,dest
```

The number or symbol "m" will be used when OP is being executed, not as the address of the desired operand (the "normal" situation), but as the actual operand itself. Thus the name "immediate operand." This construction really only makes sense when double operand instructions are being used. So we could write:

```
ADD #33,SUM      ; 33 + (SUM) → SUM
CMP ABC,#100.    ; (ABC) vs 100.
SUB #60,%1       ; (%1)-60 → %1
MOV #TABLE,%0    ; address ''TABLE'' → %0
```

We can now shorten, simplify, and speed up a previous example:

```
S1:     .ASCIZ  /This is a test/
S2:     .BLKB   100      ; 100 oct
        .EVEN
        ...
        MOV     #S1,%1 ; adr ''S1'' to %1
        MOV     #S2,%2 ; adr ''S2'' to %2
LOOP:   MOVB    (%1)+,(%2)+
        BNE     LOOP
        ...
```

You may be tempted to write expressions such as

```
ADRX:   .WORD #ABC
```

to attempt to make the address ABC available as a constant. This attempt to "force" ABC to be used as an address will be thwarted by the assembler, since it has no meaning. All the addressing modes of the PDP-11 are meaningful only when applied to addresses appearing as part of instructions. A .WORD is a mere directive, not an instruction. When an addressing mode is used within an instruction, the assembler substitutes the appropriate 3-bit mode code in the M field of the instruction. There are no M (mode) fields in which to "hide" mode codes when using a .WORD. Furthermore, they are not needed. A simple

```
ADRX:   .WORD ABC
```

is sufficient to have the address corresponding to ABC stored in the location labeled ADRX.

The assembler encodes a request for an immediate operand, #m, by using mode code two, and it sets the register field code to 7, to signify use of the PC. The constant m is itself placed in the next word. We recognize mode code two as the same auto increment mode we saw earlier.

Addressing Mode Summary

When writing data-oriented instructions, we can now choose among the following addressing modes, in addition to the simple memory references we have been using all along, such as INC ABC. The basic addressing modes we have seen are summarized in figure 6.1. The important points to remember about these addressing modes are:

1. They can be used freely only with data-oriented instructions (e.g., MOV, ADD, CMP, INC, CLR, ...).
2. A control-oriented instruction such as JMP cannot use register mode; it can use index, register-deferred, and auto-increment modes.
3. The control-oriented branch instructions cannot use any of these addressing modes.
4. Of modes (a)–(d), which use register R, only the auto-increment mode modifies the register which it uses.
5. Immediate addressing makes sense only in the two-operand instructions.
6. With auto-increment addressing, an increment of 1 will be used to increment the register after it is used, if a byte-oriented instruction is executed.

There are other addressing modes used on the PDP-11. We will be describing these later, after we have had a chance to use the ones we have just introduced.

Mode	Code	Symbolic form	Effective adr	Example
(a)	0 register	OP R	R	CLR %4; 0→%4
(b)	6 index	OP d(R)	d+(R)	CLR 6(%4); 0→6+(%4)
(c)	1 reg deferred	OP (R)	(R)	CLR (%4)+; 0→(%4)
(d)	2 auto increment	OP (R)+	(R)	CLR (%4) ; 0→(%4) and (%4)+2→%4
(e)	2 immediate	OP #m,d	PC	MOV #77,%4; 77→%4

Figure 6.1 Basic addressing modes.

Exercises

6.1 Let each instruction use the following initial conditions (that means the results of one instruction will not be used by another instruction). Fill in the contents of the indicated locations after the instruction has been executed. Write — if no change occurs.

	%2	%3	1050	1052
Initial Conditions	1050	1052	2052	35

(a) `MOV %2,%3`

(b) `CLR (%2)`

(c) `MOVB (%2)+,%3`

(d) `CMP (%2)+,(%2)+`

(e) `MOV #1050,%3`

6.2 Write a program using correct MACRO-11 syntax which takes a string that ends in a 0 byte and removes all the spaces from it. Thus,

```
.ASCIZ /TEST ONE  TWO/ becomes
.ASCIZ /TESTONETWO/.
```

6.3 Each of the following may be acceptable or not acceptable to the assembler or to the CPU, depending on other parts of the program in which they are used. Specify under what circumstances each statement or the corresponding code is not accepted (use a reason only once). Example:

`LOOP  MOV A,B` uses illegal label syntax.

(a) `BR  START`

(b) `JMP  %4`

(c) `MOV  #99,%1`

(d) `BR  (%5)`

(e) `JMP  (%5)`

(f) `BEQ  A+700`

6.4 Show the machine language the assembler generates given the following text.

```
          CONST = 10
          R0    = %0
START:    MOVB   (%1),R0
          CLR    44(%1)
          ADD    #CONST, (%0)+
          HALT
A:        .WORD  27.
          .END
```

6.5 Write a PDP-11 symbolic program using 10 or fewer instructions which, given a character string ending with a 0 byte, creates a new string with all the characters reversed. Thus, given

```
A: .ASCII /ABCDEFGHZXC/<0>
```

the new string would be

```
CXZHGFEDCBA
```

6.6 Given the following as initial conditions before each part of this problem, fill in the contents of each specified register or memory location after execution of each indicated instruction

	Loc 2000	Loc 3000	Loc 4040	%1	%2
Initial values	4040	6001	501	2000	3000

(a) `MOV (%1),(%2)`

(b) `CMPB (%1)+,(%2)`

(c) `ADD %1, (%2)`

(d) `SUB 1000(%1), (%2)`

6.7 True (T) or False (F)?

(a) Uninitialized memory locations are assumed to have the value 0.
(b) BR should execute almost twice as fast as JMP.
(c) `MOV #10,%1` uses 2 memory references during its fetch-execute cycle.
(d) Some conditional branch instructions can modify the CC.
(e) `DEC %3` and `SUB #1,%3` have the same effect.

6.8 Briefly explain what the following program segment does. Write down the final content of AL, AL+1,...,AL+4.

```
START:   MOV #LINE,%1
         MOV #AL, %2
         MOV #15, %3
L1:      CLR (%2)+
         DEC %3
         BNE L1
L2:      CMPB (%1), #12
         BEQ    FND
         MOVB (%1) +,%3
         SUB A, %3
         BMI L2
         INC AL(%3)
         BR L2
         ...
A:       .ASCII /A/<0>
AL:      .BLKW 15
LINE:    .ASCII /THIS IS A TEST SEQUENCE/<12>
         ...
```

6.9 For each of the following instructions, assume the indicated initial conditions. Indicate all changes to these items produced by executing these instructions. The initial conditions apply to each instruction. Each one is assumed to begin at location 1000. Use — to indicate no change.

Initial Conditions	%1	%2	1200	1202	PC
	1200	1202	1202	1200	1000

(a) `ADD (%1)+,(%2)`

(b) `ADD #1200,%1`

(c) `CMP (%1)+,(%1)+`

(d) `062121`

(e) `060102`

6.10 When execution reaches the instruction at ABC, %1 contains an integer between 0 and 4, inclusive. Assume that labels L0, L1, L2, L3, and L4 are properly defined at various points in this program. Write no more than five instructions at location ABC that will pass control to the appropriately labeled instruction. Any needed items (but no instructions) may be inserted at the beginning of the program.

```
GO: ...

ABC:


.END      GO
```

6.11 (a) What does `MOV #1, ABC` do?
(b) What does `MOV ABC, #1` do?

6.12 For each of the instructions listed below, assume that the initial content of registers 1 and 2 and memory locations 1112 and 2020 are as follows:

%1: 001112 %2: 002020 1112: 054326 2020: 000123

Match each instruction below with the correct final content of %1 and %2 after the execution of that single instruction. Ignore the results of any other instructions.

Instruction		Final Content	
		%1	2%
(a) `MOVB (%1)+,(%2)`	(i)	001110	002020
(b) `MOV (%2)+,%1`	(ii)	001111	002021
(c) `SUB %1,%2`	(iii)	000123	002022
(d) `MOVB %1,(%2)+`	(iv)	001113	002017
(e) `MOVB (%1),%2`	(v)	001113	002020
(f) `ADD #10,(%1)`	(vi)	001112	002022
	(vii)	001112	002021
	(viii)	001112	000706
	(ix)	001112	000130
	(x)	001112	002020

6.13 Write a program which counts the occurrences of the letter "L"in a given string which ends with a 0 byte. Leave your answer in register 1. Use correct PDP-11 syntax and add useful comments to your code.

6.14 Define each item; why are they important (be brief)?

(a) PS.
(b) PC-relative address.
(c) Effective address.

6.15 (a) Briefly explain what the following program does. Assume that execution of the code corresponding to the .PRINT macro displays all the characters starting with the one whose address was provided to .PRINT and stops when a code 12 (line feed) is seen.

```
            .MCALL      .PRINT
START:      MOV         #ABC, %2
OVER:       MOV         #STR, %1
LOOP:       CMPB        (%1), (%2)
            BEQ         REPLAC
            ADD         #1, %1
            CMPB        (%1), #12
            BNE         LOOP
            ADD         #1, %2
            CMPB        (%2), #12
            BNE         OVER
PRNT:       .PRINT      #ABC
            .PRINT      #STR
            HALT
REPLAC:     MOVB        STR,(%1)
            MOVB        STR,(%2)
            BR          PRNT
ABC:        .ASCII          /ANSWER/<12>
STR:        .ASCII     /QUESTION/<12>
            .END        START
```

(b) If

```
STR:    .ASCII     /LOOK OUT/<12>
```

is substituted and the program rerun, then what changes?

6.16 Outline how you could implement the program which loads the PDP-11 machine-language programs prepared according to the specifications given in chapter 3.

6.17 What is left in registers 1 and 2 after

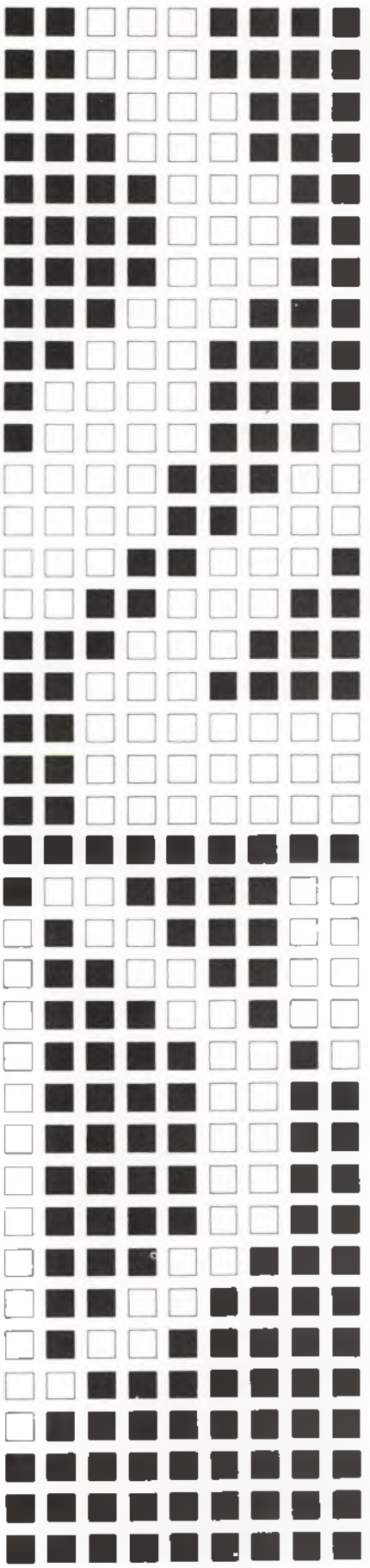
```
(a) MOV   #300,%1
(b) MOV   #07217,%2
    MOVB  #100,%2
```

6.18 What would be left in registers 1 and 2 if, after the MOVs of problem 6.11, we had

```
(a) INCB    %1
(b) CLRB    %2
```

7

computer arithmetic



The arithmetic capabilities of a computer are usually determined by its arithmetic logic unit (ALU). On the PDP-11, hardware support for integer arithmetic operations such as ADD, SUB, INC, etc. is provided by its ALU. Hardware support for arithmetic with floating point numbers is provided by optional, extra-cost hardware, which will be described in a later chapter. There are many ways of representing signed numbers in binary. The most important are discussed next.

Negative Numbers

We have said very little about negative numbers beyond the fact that they have their high order bit set. The time has come to discuss all the relevant properties of numbers, positive and negative; what can go wrong; and how to take care of it.

Sign Magnitude Representation

The simplest way to represent negative numbers is to attach a minus sign to the corresponding magnitude (i.e., the unsigned value). On a binary computer we could set aside a bit—say, the high order bit—to serve as a sign bit, and the remaining bits can be used to represent the magnitude in binary.

Some years ago computers were built to process numbers represented in the sign-magnitude system directly by their ALUs. More recent computer designs have adopted other representations for a number of reasons, including faster processing. We will discuss the two most widely used representations in the coming sections. We will, of course, use sign magnitude representation for decimal numbers as they are fed to computers, and that representation will be used for the numbers we eventually print. Realize, though, that we will have to write programs to convert the representation used outside the computer into the representation used inside the computer and back again, as shown in figure 7.1.

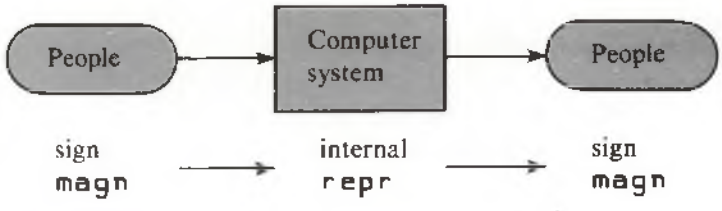


Figure 7.1 External and internal representations.

Two's-Complement Representation

The most widely used representation for signed binary numbers in computers is the *two's-complement* representation. If we are using an n-bit word, then the number x and its negative, X, always obey the relation

$$x + X = 2^n$$

Figure 7.2 Signed and unsigned values.

Binary number	Unsigned value	Signed value
000	0	0
001	1	1
010	2	2
011	3	3
---	-	-
100	4	-4
101	5	-3
110	6	-2
111	7	-1

in a two's-complement system. Suppose we are dealing with 3-bit words, so we can keep the examples short and simple yet illustrative. Given the number 2 (for x), or 010 binary, its two's complement is 110 (for X), since

$$010 + 110 = 1000 = 2^3$$

In figure 7.2 we have written down all possible 3-bit numbers and illustrated the relationships between the unsigned values and the signed values. The unsigned value is the quantity we expect when using an ordinary 3-bit number. The signed value is the quantity we obtain if we are using the two's-complement representation for signed numbers.

In this case we are dealing with what is known as a *modulo 8* number system. This means that any multiple of eight will be taken to be equivalent to 0. So we find the following happening:

$$3 + 5 = 8 \rightarrow 0, \text{ modulo } 8$$

which we can rewrite as:

$$3 + 5 = 0, \text{ modulo } 8$$

This being the case, it follows that

$$3 = -5, \text{ modulo } 8$$
$$5 = -3, \text{ modulo } 8$$

If you check the relationship between the columns of signed and unsigned values in figure 7.2, you will see this relationship always holds. For instance,

unsigned 0 = signed 0, modulo 8
unsigned 1 = signed 1, modulo 8
. . .
. . .
unsigned 7 = signed -1, modulo 8

You can generalize this modulo arithmetic for any desired binary word length, and it also holds for number bases other than binary. If it seems vaguely familiar to you, it is, in fact, how most automobile odometers record mileage. With a five-digit odometer, you can record up to 99,999 miles. As you log mile 100,000 it is recorded as mile 00,000, because the fixed five-digit precision of the odometer enforces a modulo 100,000 addition.

What can we learn from this table? Three important observations should be singled out, as they generalize to two's-complement representations for any specified number of bits. In particular, they will apply to working with 16-bit words on a PDP-11.

Different Orderings Apply

Going from small to large, the signed and unsigned numbers do not follow the same pattern. We can see the differences below: The bit patterns for the positive numbers 0, 1, 2, and 3 are identical. The other bit patterns—represented by a, b, c, and d—appear in quite different positions.

Note that the sum of the pair of bit patterns for a, for b, for c, and for d always produces 1000, which in a modulo 8 system (implied by the use of 3-bit numbers) reduces to 0. When two numbers sum to 0, we like to think of one of the pair as being the negative of the other.

Lack of Symmetry

The numbers -1, -2, and -3 can have their negations represented as positive numbers if desired, since $+1$, $+2$, and $+3$ appear in the table. However, there is no way of representing $+4$, since it does not appear in the table. The negatives of -1, -2, and -3 can be used, but there is no way of negating -4, even though -4 is represented. Consider: $-(-4)$ is $-(100)$, or 100, since $100 + 100 = 2^3$. So the negative of -4 is -4—not a very satisfactory state of affairs, but one we will have to live with.

What of -0? $0 + 1000 \rightarrow 1000 = 2^3$. Reducing 1000 to 3 bits, we are left with $-0 = 000$. So $+0$ and -0 have the same representation. This property of *unique representation* for 0 makes two's-complement representation very attractive, because most other number systems in use (such as sign magnitude) allow two representations for 0; $+0$ and -0 have different bit patterns. This is ambiguous, and so it often leads to errors being made.

Pseudo Sign Bit

We notice in figure 7.2 that all the numbers that we consider negative have their high order bit set to 1. So we can always determine whether a number is positive or negative simply by examining its high order bit.

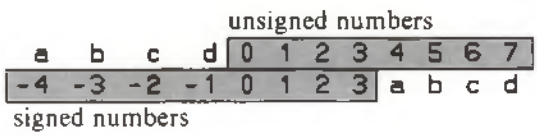


Figure 7.3 Different ordering of signed and unsigned numbers.

Figure 7.4 Positive and negative numbers in the two's complement representations.

Positive		Negative	
+0	000	-0	000
+1	001	-1	111
+2	010	-2	110
+3	011	-3	101
+4	???	-4	100

Unlike the sign-magnitude system, where negating a number is accomplished merely by changing its sign bit, with the two's-complement representation, it is necessary to change almost every bit of a number when negating it. The only exception to this is zero; +0 and −0 have the same representation, 000. Figure 7.2 shows the positive numbers and their negative counterparts. Observe that the negative numbers furthest from 0 (closest to minus infinity) have more leading 0's (after the sign bit), whereas the negative numbers nearest 0 have more leading 1's. This is more striking when dealing with 16-bit numbers. A −1 in a PDP-11 word looks like 1111111111111111 (or 177777 in octal). All the properties we discussed when dealing with 3-bit numbers, which is a modulo $2^3 = 8$ number system, apply to the PDP-11 where we use 16-bit numbers in a modulo $2^{16} = 64K$ number system.

One's Complement Representation

The one's-complement representation is another commonly used representation for signed numbers. Using n-bit numbers, the one's-complement of x is the number X, such that

$$x + X = 2^n - 1$$

Using 3-bit numbers, we can illustrate its use. Given x=2, then −2 is −(010), or 101. We need only invert or *complement* each bit individually. We thus see that −0 is 111.

The one's-complement representation is important even on computers that use two's-complement representation for their signed numbers, as does the PDP-11. This stems from the fact that the negative of a number in the one's-complement system is identical to its *logical complement* (often called the Boolean negation).

We will be dealing with logical manipulations in which each bit is more or less independent of its neighboring bits. The logical complement becomes very useful to us then.

We can conclude this discussion of one's-complement numbers by noting that, as a practical matter, it is very easy to derive the one's-complement of a number by hand (or mentally, if you prefer). If you are working in binary, replace each 0 by 1, and vice versa. If working in octal, replace each octal digit by its complement: 0 → 7, 1 → 6, 2 → 5, 3 → 4, 4 → 3, 5 → 2, 6 → 1, and 7 → 0. For instance, the one's-complement of 123456 is 054321. (Note the leading 0. When 16-bit values are written in octal, the leading digit is actually just a single bit, so in that position 0 → 1 and vice versa.)

How are the one's- and two's-complement representations related? Let us use the notation TC(x) to designate the two's-complement of x, and OC(x) for its one's-complement. Since the two's-complement of x is $TC(x) = 2^n - x$, and the one's-complement of x is $OC(x) = 2^n - 1 - x$, by rearranging this slightly we get $OC(x) = 2^n - x - 1$, or $TC(x) - 1$. So if you want to easily find the two's-complement of a number, take its one's-complement and add 1 to that.

Arithmetic Using Two's-Complement Numbers

How do we perform addition or subtraction when dealing with signed numbers using the two's-complement representation?

Addition

With sign-magnitude numbers, you have to check the signs when starting an addition. If they are both +, add the numbers and attach a + to the result. If they are both −, add the numbers (i.e., their magnitudes) and attach a − to the result. If the signs differ, subtract the magnitudes and attach the sign that came from the larger magnitude to the result.

```
  +12     −13     −14     +14
+       +       +       +
 +10     −11     +20     −20
 +22     −24     +6       −6
```

With two's-complement numbers, addition is performed by adding the numbers without any consideration whatsoever for their signs! There is no need to examine signs. Consider:

```
(a)  001   1   (b)  010  2    (c) *  111  −1   (d)  011   3
     100  −4        001  1           111  −1         010   2
     101  −3        011  3          1110  −2         101  −3?
```

* keep 110, drop high 1

How does 111 + 111 work? Consider the defining relation $x + X = 2^n$. 111 comes from 1000 − 1 $(2^3 - 1)$. So 111 + 111 can be evaluated as (1000 − 1) + (1000 − 1), or (1000 + 1000) − (1 + 1), or 10000 − 10, which is 1110. Since we can retain only 3 bits in modulo 8 arithmetic, we discard the high order bit. The discarded bit is called the *carry bit*.

Example (d) requires some explanation. The result 101 represents −3, which is not the correct result (+5). What is going on? Given a 3-bit word, with one bit effectively dedicated to being a sign bit, we have two bits left for the magnitude. With two bits, the largest number you can represent is 3, so we can't handle a result as large as 5!

This brings us back to the PDP-11. What assistance does it provide in coping with such problems?

Carry and Overflow

Whenever the ALU performs arithmetic (this includes executing CMPs) on operands, the result is reflected in the four bits of the CC. We have previously discussed under what conditions the Z and N bits of the CC are set or cleared. The other two bits—C for carry, and V for overflow—are now of immediate concern.

Signed Numbers

We will use the phrases "signed number" and "number in its two's-complement representation" interchangeably when speaking of the PDP-11. When signed numbers are being added, it is perfectly natural for an extra bit to be generated in the two's-complement system. You perhaps only notice it when it is a 1. When performing an ADD, the extra bit is copied into the CC's C bit. So, following an ADD, the C bit indicates whether the addition had a carry of 1 or 0. In subtractions with either a SUB or a CMP/CMPB, since the analog of the carry is the "borrow," the C bit will be cleared if a 1 carry was generated, and the C bit will be set if a 0 carry was generated. You generally ignore the C bit when adding signed numbers, since a carry is not unusual.

When you add two signed numbers which are positive and get a negative result, as we saw in example (d), you should be concerned. Similarly, you should be concerned if you add two negative numbers and get a positive result, as occurs here:

```
      101    −3
      101    −3
     1010    −6
Keep  010,   giving +2
```

The generation of the carry = 1 here is not necessarily a sign of trouble; getting the "wrong" answer is certainly troublesome. The potential for this happening when performing arithmetic on a computer is quite high, and it is a nuisance to monitor. Failure to check for it, though, can lead to disasters (we will see one that affected millions of people in a few moments).

Monitoring the possibility of getting the wrong result involves asking the following questions:

1. Do operands of the ADD have the same sign? If so, verify that the result has that sign.
2. Do operands of the ADD have opposite signs? If so, don't worry.

Once again, the computer designers have anticipated providing needed information in a useful way. The V bit will be set by the ALU if the two operands of the ADD had like signs, and the result had the opposite sign. In other words, the V bit is set when the "wrong" result is generated. Otherwise it is cleared. Since the problem is caused by being unable to store the correct result for lack of space (too few bits in a word), the result is said to "overflow." The setting of the V bit corresponds to the problem of *integer overflow* which you may have encountered when using high level languages.

What happens after the V bit is set? The fetch-execute cycle continues as if nothing significant had happened. If your program doesn't examine the V bit in the CC, and take appropriate action, no one else will. On most computers, if you are adding large integers and the result won't fit in one word, a V bit or some equivalent will be set, but if your program does not test for this condition, the program will continue, oblivious to the nonsense it is now processing.

BVS, BVC

You can test the setting of the V bit by using one of the following two conditional branch instructions:

BVS do ; if V=1, then da → PC
BVC do ; if V=0, then da → PC

Here "do" is an 8-bit destination offset, and "da" is the 16-bit effective destination address regenerated from "do" by the control unit. Using one of these, you can write

```
          ...
AGAIN:    ADD   (%1)+,TOTAL
          BVS   ERROR     ; test for overflow
          DEC   %2        ; count down
          BNE   AGAIN
          ...
```

You might be tempted to try

```
          ...
AGAIN:    ADD   (%1)+,TOTAL
          DEC   %2        ; count down
          BNE   AGAIN
          BVS   ERROR     ; test for overflow
          ...
```

to avoid executing the BVS over and over again. However, the V bit, if it is going to be set at all due to a "bad" ADD, will be set/cleared by the following DEC. So you have to check for overflow as soon as it might happen, certainly before you execute another instruction which uses the ALU and which can set or clear the V bit.

You should take this discussion of overflow very seriously. Once upon a time this author was asked to write a program for a large company, one of the largest in its field of business. Given historical data on the size of numbers which were to be processed, he wrote a program, debugged it, tested it, and revised it—in assembler, of course, since FORTRAN was still a curiosity back then, and COBOL had not yet been invented. The customer then tested the program and deemed it acceptable. Some nine or ten months later, late in the year, there were screams of alarm.

Figure 7.5 Tiny computer snag. (Courtesy The Associated Press.)

The Virginia and Electric Power Co. reactors at Surry, Va., were among five nuclear power plants shut down by the Nuclear Regulatory Commission this week.

Tiny computer snag has domino effect on energy

By EVANS WITT
The Associated Press

WASHINGTON—Something just didn't add up.

And the result is: Five nuclear power plants are shut down; millions of Americans may pay higher utility bills; and a sizable blow may have been struck to President Carter's efforts to reduce the use of imported oil and to control inflation.

The immediate source of all this is part of the federal bureaucracy—the Nuclear Regulatory Commission which ordered the shutdowns.

But in one sense, the ultimate culprit was "Shock II," a tiny part of a computer program used by a private firm to design the power plants' reactors.

Shock II was wrong and that means parts of the five reactors might not survive a massive earthquake. Shock II was the weak link that could have allowed the chain to snap.

In between Shock II and the shutdowns were a public utility, a private engineering firm and the NRC staff. It was really the judgments of the dozens of scientists and engineers, not elected or appointed officials, that led to the shutdowns.

Perhaps as a result, the decision's impact on the nation's energy situation was not even considered until the very last moment—when the commission itself was faced with the final decision.

And at that point, the NRC said, it had no choice. It said the law was clear: Serious questions about the reactors had been raised and the reactors had to be turned off until answers were found.

The specific questions are arcane engineering issues, but the explanation is straightforward: Will some of the systems designed to protect the reactor survive an earthquake—or will they fail, and possibly allow radioactive death to spew into the air?

The regulations say the reactors must be able to withstand a quake equal to the strongest ever recorded in their area. The regulations don't allow any consideration of the likelihood of a major quake. All four states where the reactors are located—New York, Pennsylvania, Maine and Virginia—have had minor quakes in this decade and damaging quakes at least once in this century.

The only way to test them—short of having a massive earthquake—is to test a model of the reactor. The "model" is actually a set of mathematical formulas in a computer that reflect how the reactor and its parts will behave in a quake.

The model used for the five reactors came from Stone and Webster, the large Boston engineering and architectural firm that designed the plants. The Stone and Webster model indicated how strong and well supported pipes had to be and how strong valves had to be.

The problem apparently cropped up after Stone and Webster suggested within the last few months more pipe supports in the secondary cooling system of the reactor at Shippingport, Pa., operated by Duquesne Light Co. in Pittsburgh.

But why were the supports needed? "This was not clear to us, looking at the

calculations done by the models," said Gilbert W. Moore, Duquesne's general superintendent of power stations.

So Duquesne—and Stone and Webster—sent the computer models through their paces again, having them calculate and recalculate what would happen to the pipes in an earthquake.

"We came out with some numbers which were not in the range we would like," Moore said.

That made the problem clear—the model now said the pipes might break in an earthquake. The previous analysis indicated an adequate safety margin in the pipes, and Stone and Webster's explanation was: "One subroutine may not give uniformly conservative results."

The problem was in a "subroutine," a small part of the computer model, called "Shock II," said Victor Stello, director of NRC's division of reactor operations.

"The facts were that the computer code they were using was in error," said Stello. "Some of the computer runs were showing things are OK. In some cases, the piping systems were not OK.

"We didn't know the magnitude of the error or how many plants might be affected," he said.

It was on March 1 that Duquesne told the NRC of the problem by telephone and asked for a meeting to discuss it. The same day, Energy Secretary James R. Schlesinger was telling Congress that unleaded gas might cost $1 a gallon within a year and service stations might be ordered shut down on Sundays because of oil shortages.

The meeting took place Thursday, March 8, in Washington with NRC staff, Stone and Webster engineers and Duquesne Light people on hand.

Through the weekend, Stello said, engineers from NRC, Duquesne and Stone and Webster worked at the private firm's Boston office, analyzing the severity of the problem.

"By the middle of Sunday (March 10) we begin to get a pretty good idea of what it meant for the systems," Stello said. **"Monday, we got the latest information from our people at the Stone and Webster offices. It became clear that there would be a number of the safety systems that would have stresses in excess of allowable limits. The magnitude of the excess was considerable."**

Tuesday, members of the NRC were briefed by their staff of engineers and scientists. They asked for an analysis of the economic impact of the decision, and then ordered the plants closed within 48 hours.

And the five reactors shut down: Duquesne Light Co.'s Beaver Valley plant at Shippingport, Pa.; Maine Yankee in Wiscasset, Maine; the Power Authority of New York's James Fitzpatrick plant at Scriba, N.Y.; and two Virginia and Electric Power Co. reactors at Surry, Va.

It may take months to finish the analysis of the potential problems and even longer to make changes to take care of the situation.

Until the reactors start generating again, the utilities will have to turn to plants using oil or coal. This may cost more, and that cost may be borne by the millions of utility customers.

To replace the power from these nuclear plants could require 100,000 barrels of oil a day or more. And this at a time when President Carter has promised to cut U.S. oil consumption by 5 percent—about 1 million barrels a day—and when the world's oil markets are in turmoil because of recent upheavals in Iran.

Why were the year-to-date sales figures for December lower than those of November? It turned out that no one associated with the company had ever expected the sales volume to be so large! So when the program specifications were drawn up almost a year earlier, the innocent programmer, trusting to a sufficiently large word size in the light of the largest expected data, never tested for overflow. No person was permanently hurt because of this error, although it might have precipitated some heart attacks. Nowadays a programming oversight of this type in a hospital application, in an airborne computer system, in a nuclear power plant control system, etc. could have disastrous consequences. The incident reported in the newspapers regarding the shutting down of five power plants serving millions of people, because of a computer programming error made years earlier, is unfortunately not as rare as it should be.

The moral of this story is: expect the specifications to be wrong. Check for every conceivable error. The error you don't check for is the one most likely to occur.

Subtraction and Comparisons

As might be expected, subtraction is a variant of addition. Given the symbolic form

SUB s,d ; (d) − (s) → d

subtracting (s) from (d) is accomplished by having the ALU add the two's complement of (s) to (d). Once again, carry is not of overwhelming importance. The carry will, of course, be recorded in the CC. Overflow detection will take place according to the following rule:

If (s) and (d) have opposite signs, and if the result of (d) − (s) should have the sign of the source operand, then overflow has occurred, and the V bit will be set. Otherwise the V bit will be cleared.

Consider the following examples:

(a)			(b)			(c)		
011 →	011	3	011 →	011	3	011 →	011	3
−011	+101	−3	−100	+100	−(−4)	−111	+001	−(−1)
	1000	0		111	7		100	4
	000	OK		−1	X		−4	X

In examples (b) and (c), the operands have opposite signs, and the sign of the source operand appears as the sign of the result. In those two cases the result is wrong (in terms of signed numbers), so the overflow bit V will be set. In case (a), V will be cleared. For the record, the 1 carry in (a) *clears* the C bit, whereas the 0 carries in (b) and (c) *set* the C bit because this is a subtraction not an addition.

Keep in mind that the order of subtraction in a CMP s,d is the opposite of that in a SUB s,d. The ALU compensates for this and sets the V bit should the implied subtraction of a CMP warrant it.

MOVB Revisited

Recall that a MOVB copies 8 bits from a source to a destination, and updates the CC. Nothing else is affected *unless* a register has been specified as the MOVB destination address. In a

MOVB source, register

after the 8 bits are copied from the source into the low 8 bits of the register, the sign of the byte just copied is propagated through the other 8 bits (the top half) of the register, either setting them all to 1 or all to 0. This *sign extension* preserves the value of the 8-bit byte as a 16-bit word in the two's complement representation. So if you had to add data stored in bytes, you could easily do so even in the absence of a byte-add instruction. Consider:

```
MOVB   B1,%1
MOVB   B2,%2
ADD    %2,%1
```

For (B1)=001 (+1), and (B2)=377 (−1)

		Before	After
MOVB	B1,%1	%1: 123456	%1: 000001
MOVB	B2,%2	%2: 007000	%2: 177777
ADD	%2,%1		%1: 000000

If the MOVB did not automatically force sign extension in the registers, we would have had 007377 in %2, after the MOVB B2,%2 which is clearly not very close to the 16-bit representation of −1.

Caution Regarding the C Bit

In an ADD instruction, if the ALU generates a carry of 1, the C bit is set, C=1; otherwise it is cleared. In the cases of both the SUB and CMP,the setting of the C bit is based on the opposite situation. If the ALU generates a carry of 1, the C bit will be cleared, C=0; a carry of zero will set the C bit, C=1.

Why is this so? The C bit can be thought of as representing a "borrowed" bit when doing subtractions. Consider SUB A,B where A and B have the value 1: 001 − 001 → 001 + 111 → 1000 → result 000, with a carry of 1, setting C=0. Since subtracting 1 from 1 can be done without "borrowing," we expect and get C=0. If we had A at −3 and B at 1; (B)−(A) → 001 − 101 → 001 + 011 → 0100, which is recorded by setting C=1. Subtracting 101 from 001 requires "borrowing" a leading 1, so we can subtract 101 from 1001. The borrow is reflected in the C-bit setting.

BCS, BCC, ADC, SBC

If the carry bit is so innocuous, why does it deserve a place in the CC? One good reason is to facilitate operating on arbitrarily large numbers.

Reverting to 3-bit numbers again, suppose you expected to have to use numbers larger than +3, say as large as +25. You could use pairs of 3 bit "words" for each number in this application. Then the numbers could be represented as shown in figure 7.6. Adding such numbers involves first adding the pair representing the low order half of each operand. So in adding 7 and 8, we first add low halves 111 + 000 → 111 = low half of result, then we add the high halves, getting 000 + 001 → 001 = high half of result. So the final result is (high half, low half), which is 001111.

Number	Representation High	Low
1	000	001
-1	111	111
7	000	111
8	001	000
12	001	100
-12	110	100

Figure 7.6 Some positive and negative binary numbers.

In adding 7 and 7, we start with 111 + 111, getting 1110. This sets the C bit, C=1. We next test the C bit (using BCS or BCC, described below) and if it is set, we add one to the high order part of the result operand. We can then finish by adding the two high order parts. The original high order parts for 7 + 7 were 000 and 000. After taking the carry into account, one of these is incremented (by us; it is not done automatically) to 001, so our final result is:

001110 (7 + 7 → 14)

The two instructions which can directly examine the C bit are:

BCS do ; if C=1, then da → PC
BCC do ; if C=0, then da → PC

Neither change the C bit or any other CC bit. As it will develop, we won't need these instructions for the previous use in performing multiple precision integer arithmetic, but they will nonetheless be useful.

Realizing that 16-bit numbers are too small for some applications (unsigned numbers have a range of 0 to 65,535 while signed numbers run from −32,768 to 32,767), the PDP-11 designers provided a way of efficiently supporting the handling of large integers using multiple words. A pair of instructions is provided for this purpose.

ADC d ; (d) + (C) → d
SBC d ; (d)−(C) → d

Rather than having to test the C bit after adding the low order operands, and adding one to the next higher order operand if C was set, the ADC does all of that for you. Consider:

```
A:    .WORD   1,100000; =98,304
B:    .WORD   2,100000; =163,840
      ...
      ADD     A+2,B+2; add low halves
      ADC     B      ; add carry to high half of B
      ADD     A,B    ; add high halves
      BVS     ERROR  ; check for overflow
```

Similarly, if you wished to perform multiple-precision integer subtraction, the SBC instruction would eliminate testing for a "borrow," which is reflected in the C bit.

The above technique can be extended to handle n-word numbers. After adding each corresponding pair of the n−1 lower order words, you perform an ADC. Obviously, multiple-precision integer arithmetic takes longer than ordinary addition, which we can now call single-precision integer arithmetic. N-precision arithmetic (using n words per number) would be more than n times slower than single-precision arithmetic.

CC Bit	Bit Values 0	1
Z	BNE	BEQ
N	BPL	BMI
V	BVC	BVS
C	BCC	BCS

Figure 7.7 Conditional branches on individual CC bits.

Condition Code Combination	Result 0	1
C	≥ BCC BHIS	< BCS BLO
CvZ	> BHI	≤ BLOS

Figure 7.8 Branches for unsigned numbers.

All Branches for Unsigned Numbers

Figure 7.8 summarizes *all* the conditional branches applicable when comparing or subtracting unsigned numbers. The normal sequence of affairs in using these is:

```
CMP     s,d ; pair of unsigned operands
Bxx         ; xx from figure 7.8
```

What happened to the BMI? As you may recall, in an earlier example in chapter 6 we had a loop which kept repeating as long as the current address in register 1 was less than the limit address in location LIM. A simple BMI was naively used to test for termination.

This represents a pernicious bug which acts as a time bomb. Since addresses on the PDP-11 are 16-bit unsigned numbers, when comparing them, beware. If you are stepping through a set of words which begin at address A and end just before address B, you might use the instruction `CMP #B,%1`, where %1 holds the current address. Consider:

```
        MOV     #A,%1
LOOP:   CLR     (%1)+
        CMP     %1,#B
        BMI     LOOP
        ...
A:      .BLKW   5
B:      ...
```

As %1 is incremented, the effective addresses take on the successive values A, A + 2, A + 4, ... It would appear that if you want to stop after using the effective address A + 10, then a simple BMI LOOP would keep looping as long as the effective addresses are less than address B. The loop will terminate when (%1) = A + 12 (i.e., A + 10.) equals the address B. You can write and test a program using this CMP/BMI combination and it will probably work! However, the next day or the next year, after you make a small totally unrelated change in the program,

possibly to some part far removed from this one, the program may inexplicably fail. Your well-tested program no longer runs correctly.The sequence CMP/BMI will function correctly so long as both the address B and the ones being compared with it are all positive. As soon as any of them become negative, because a change in the program forced data or instructions into slightly higher memory so that some of these addresses cross over the 16K word boundary, the relationship being tested for by the BMI no longer holds. Addresses in the range 0 to 16KW are positive numbers. Those above 16KW look like negative signed numbers.

The moral might be: don't use addresses to control loops. If you do decide to compare addresses, then treat them as the unsigned numbers they are, and use the appropriate conditional branches for unsigned numbers. In this case, a **BLO LOOP** would be correct. You might also consider testing for equal addresses. You could use

```
CMP    #B,%1
BNE    LOOP
```

without any fear, provided that address B corresponds to the last effective address to be generated, which is two greater than the last effective address used.

SOB

We would be less inclined to use addresses in controlling loops if we had better support for controlling loops by using a count. The following instruction is not found in the earliest models of the PDP-11 (models 11/04, 11/05, and 11/20). The SOB (subtract one and branch) combines a count down with a conditional branch. Instead of writing

```
        MOV     #10, %1 ; count = 10
        MOV     #LN, %2 ; base adr
NEXT:   CLR     (%2)+
        DEC     %1      ; count down
        BNE NEXT
```

we can now write

```
        MOV  #10, %1
        MOV  #LN, %2
NEXT:   CLR  (%2)+
        SOB  %1,NEXT   ; count down using %1
```

The symbolic form is

```
        SOB     r,adr
```

Sign Extension SXT

The MOVB instruction forces sign extension when its destination specifies a register. If you wish to perform multiple-precision arithmetic, and have some single-precision data on hand, how do you make these single-precision items compatible with the double-precision ones? The key is to perform sign extension analogous to that performed by the MOVB, but not limited either to bytes or registers.

Suppose SP1 and SP2 are two single-precision data words:

```
SP1:    .WORD   001234
SP2:    .WORD   -1      ; 177777 oct
```

In order to use these with other double-precision integers, we have to enlarge each one so that it uses two words, while preserving the correct sign. Let SP1 be transformed into DP1, and SP2 into DP2:

```
DP1:    .WORD   0,001234 ; keep it +
DP2:    .WORD   -1,-1    ; keep it -
```

You could accomplish this using:

```
        CLR     DP1         ; set high part=0
        MOV     SP1,DP1+2   ; copy low part
        BPL     OK          ; done if +
        DEC     DP1         ; force high part -
OK:     ...
```

Similar instructions can be used to transform SP2 into DP2.

The instruction SXT d is a single operand instruction provided to assist in performing these transformations. It tests the N bit setting in the CC, as it reflects the sign of the item just operated on and extends it through all 16 bits of its destination. So we can now write:

```
MOV     SP1,DP1+2 ; copy low part
SXT     DP1       ; extend its sign
                  through high part
```

Dealing with Unsigned Numbers

Consider 3-bit unsigned numbers. We can rank all 8 possible numbers from low to high:

```
lowest=000 001 010 011 100 101 110 111=highest
         0   1   2   3   4   5   6   7
```

Suppose the unsigned number a is "lower" (less than) the unsigned number b; consider the case a=001 and b=101. We normally begin to test for "lower" by using a CMP instruction. If we used

```
CMP item.a, item.b
```

this implies a subtraction item.a − item.b. or 001 − 101 which is 001 + 011 → 100. The ALU notices that the operands had opposite signs and that the result sign and the destination signs agree, so it sets V = 1 and C = 1 (C is set to 1 because the carry was 0 in doing a subtraction).

For a = 101 and b = 111, a is still lower than b. The CMP leads to 101 − 111 → 101 + 001 → 110, leaving V = 0 and C = 1.

For a = 001 and b = 001 ("same"), 001 − 001 → 001 + 111 →1000, leaving V = 0, C = 0, and Z = 1.

These unusual combinations of CC bits could be tested directly, since the PDP-11 supports direct tests of each CC bit. Some of these we have seen before (e.g., BVS, BMI, etc.). The full set is shown here in figure 7.7.

So if we wanted to branch to AHA if an unsigned number at location A was less than or equal to an unsigned number at location B, we could write:

```
...
CMP    A,B    ; (A)-(B)
BEQ    AHA    ; if (A)=(B), then Z=1
BCS    AHA    ; if (A)<(B), then C=1
...
```

Since the condition "lower than or same" is a common one, a new instruction, the conditional branch BLOS, tests both the Z and C bits simultaneously. If either bit is set, symbolized by CvZ, ("v" is the logical "or," which is 1 if either of its arguments is 1; otherwise it is 0), the branching will occur.

The complement of BLOS is the BHI (branch if higher). It branches if neither C or Z is set; it looks at C and Z and computes (CvZ) = 0. Here the "=" is used as a logical relational operator. In a statement such as x = y, it means "is x the same as y?". So x = y has a value 1 if the answer is yes, and 0 otherwise. So (CvZ) = 0 is shorthand for "branch if neither C or Z is 1." Two more conditional branches are provided for use with unsigned numbers. These two are actually only new names for BCC and BCS, respectively:

```
BHIS do   ; branch if higher or same
BLO  do   ; branch if lower
```

Using these new names is recommended if you are comparing unsigned numbers. If you are checking the C bit for some other purpose, use the BCx form. The assembler will not care, but it may make understanding and debugging your program simpler.

The assembler will convert the symbolic address "adr" into a 6-bit *unsigned* offset "sbo" and place it and the 3-bit code for register r in the machine-language form:

```
077   R xx
SOB   r sbo
 7    3  6  (bits)
```

When it is executed, the effect is:

1. (R) − 1 → R
2. if (R) not 0, PC − (2 * sbo) → PC

So here we see a conditional branch which is only capable of branching *backward!*

The SOB is not classified as a double or a single operand instruction. Offset fields found in branches do not count as operands (operands are intended to refer to data, not instructions, as a rule). Although the SOB instruction has an R field that looks suspiciously like one that designates an operand, it is still not classified as a single operand instruction. With the previously defined single operand instructions, their operand address fields could use any valid addressing mode (e.g., indexing, auto-increment, etc.). The SOB R field can only be a plain register designation; no addressing modes are permitted. Obviously, if some instruction within an SOB loop modifies the register being used for count control by the SOB, strange things may happen. For instance, if this register holds a negative value when the SOB is reached, the loop will go on for a long time.

We see a trend away from the full generality of the source- and destination-addressing instructions toward special-purpose instructions with special constraints on their addressing flexibility. This is the price one must pay to gain execution speed.

Comparing Signed Numbers

Signed numbers deserve special consideration because they represent the principal data object the computer was designed to process in the first place. Clearly, when signed numbers are compared, the N bit should play a central role. As we shall see, it does.

Let us recall that when executing a CMP s,d, the V bit will be set:

if the signs of the s and d operands differ; and
if the signs of the result and the d operand are the same.

Consider the following three cases in which (s) = S, (d) = D, and assume for cases 1 and 2 that $S<D$.

Case 1. Suppose S − D overflows, setting V = 1. Since $S<D$ (by assumption), then necessarily $S<0<D$ (i.e., they had to have opposite signs in order for their difference to overflow). Since the V bit is set (by assumption), then the sign of the result must be the same as the d operand's sign. But we have just seen that (d) = D must be >0. So this forces N = 0.

Case 2. Suppose S − D does not cause overflow, so V = 0. Then S − D<0, leaving N = 1.

Case 3. Suppose S = D, so S − D = 0, leaving V = 0, N = 0, and Z = 1.

The conclusion we can draw is that in comparing two signed numbers S and D, S is less than D if either

V = 1 and N = 0; or
V = 0 and N = 1.

This can be reduced to (V≠N). The symbol ≠ (not equals) is also called the "exclusive-or" (with the symbol ∨ sometimes used). The expression x≠y has the value 1 if x and y are not equal; otherwise it has the value 0.

A pair of signed numbers can be examined to see which is larger or smaller by using the conditional branch instructions shown in figure 7.9. Simultaneously, the relevant CC bits are examined and the correct combinations of these bits evaluated.

Signed and Unsigned Conditional Branches

We can summarize the preceding discussions regarding comparisons of pairs of signed or unsigned numbers in figure 7.10. Since the ALU does not know whether the operands it processes are "really" signed or unsigned numbers, it processes *all* numbers as if they were signed numbers and sets or clears the relevant CC bits accordingly. Whichever branch instruction you use will be executed. Figure 7.10 shows all the conditional branches appropriate for comparisons. Study it well.

Figure 7.9 Branches for signed numbers.

Condition Code Combination	Result 0	1
N≠V	≥ BGE	< BLT
(N≠V)∨Z	> BGT	≤ BLE

Figure 7.10 Branches for signed and unsigned numbers.

Condition Code Combination		Result 0	1
C	≥ BCC BHIS	< BCS BLO	Unsigned Numbers
C∨Z	> BHI	≤ BLOS	
N≠V	≥ BGE	< BLT	Signed Numbers
(N≠V)∨Z	> BGT	≤ BLE	

TST and Testing

If you need to check an item at some time other than immediately after it is generated, or copied with a MOV, it must be forced through the ALU again in order to update the CC bits, so these in turn may be examined. The only mechanisms we have for doing this are clumsy.

Suppose several hundred instruction executions earlier, you had computed a result stored in RES. You now wish to have a 3-way branch, such as:

```
MOV     RES,RES; force it through ALU
BEQ     RESZ
BMI     RESM
BR      RESP
```

You could use a CMP RES, #0 instead of the MOV. In order to simplify things, a new instruction is provided:

```
TST  d ; update CC depending on (d)
TSTB d ; update CC depending on byte (d)
```

The TST and TSTB fetch the destination operand (d) so that the ALU may set/clear the N and Z bits. Since these instructions have no information as to the original computation which generated (d), they cannot provide meaningful information about either carry or overflow. So V and C are unconditionally cleared.

Note that a TST/TSTB might change something besides the Z and N bits. A

```
TST    (%3)+
```

will update the CC and add 2 to %3. There will be times when such an instruction is used principally for its effect on a register, with no use being made of its effect on the CC. This would not have happened if there were instructions such as INC2 to add 2 to its destination.

Which Branch Should I Use?

If you are copying a value with a MOV, or testing one with a TST, then only those Bxx branches which examine N or Z are applicable.

If you have just executed a CMP, then you should select the appropriate Bxx for what you are comparing, signed or unsigned numbers. If you are comparing addresses, treat them as the unsigned numbers which they are.

If you have just executed an ADD, SUB, INC, or DEC, then you should test for overflow if it is at all likely. If you are doing multiple-precision integer arithmetic, then you can either test the C bit (or use ADC, SBC) following use of the ADD or SUB, but not with the INC or DEC. These last two instructions *do not* set or clear the C bit. So

```
ADD    #1,X
INC    X
```

Figure 7.11 Effects on condition codes.

	N	Z	V	C
MOV	*	*	0	-
TST	*	*	0	0
ADD	*	*	*	*
INC	*	*	*	-

(* = set or cleared, 0 = cleared, - = not affected)

are not completely equivalent, because they do not always have the same effects on all of the CC bits.

Figure 7.11 summarizes which instructions affect which of the four CC bits. Both SUB and CMP affect the same CC bits as the ADD; the DEC affects the same ones as the INC.

It should now be apparent that attention has to be paid to how each and every instruction can affect the CC bits. Appendix 1, which summarizes the PDP-11 instruction set, provides for each instruction the kind of information we have in figure 7.11.

Summary

The PDP-11 uses the two's-complement representation for signed numbers. This makes it possible to guarantee that only one bit pattern corresponds to the value 0. The two's-complement representation of a negative number is closely related to another representation used on some computers, the one's-complement representation. Given a negative number, its one's-complement representation, plus 1, equals its two's-complement representation. Taking the one's-complement of a number is also identical to logically complementing (inverting) each bit.

When signed numbers are added by use of the two's-complement representation, it does not matter whether the pair of operands have the same or opposite signs. The correct sum will be generated by the ALU, provided that the result can be expressed as a 16-bit number (in the two's-complement representation if negative). When this is not possible, the ALU will set the overflow bit V in the CC. The programmer should check for possibility of integer overflow by using a BVS or BVC instruction.

Arithmetic operations normally set or clear the CC's carry bit C. It can be used to facilitate performing multiple-precision integer arithmetic, in which each operand is represented by several words. In such a multiword operand all 16 bits of each of the words contribute to the magnitude, except for the most significant word. Thus we do not sacrifice 1 bit per word as a pseudo sign bit. The BCC and BCS instructions can be used to test the settings of the carry bit C. When multiple-precision integer arithmetic is being performed, the ADC and SBC instructions can simultaneously test the C bit and increment or decrement the desired operand.

In the two's-complement system, a negative number always has its most significant bit set to 1. Extending a negative number from one byte to a full word, or from one word into two words, requires propagating the sign bit. This is done automatically whenever a MOVB has a register as its destination. Extending one word's sign through another word is done using the SXT instruction.

Condition Code Combination	Result 0	1
C	≥ BCC BHIS	< BCS BLO
CvZ	> BHI	≤ BLOS

Figure 7.12 Branches for unsigned numbers.

Condition Code Combination	Result 0	1
N≠V	≥ BGE	< BLT
(N≠V)vZ	> BGT	≤ BLE

Figure 7.13 Branches for signed numbers.

The SOB (subtract and branch) instruction is very unusual both in its purpose (count-down loop control) and in the constraints on its two operand fields. The first field must specify a register, and the second field will hold a 6-bit unsigned offset. The SOB is the only instruction which always insists on branching backward, if its count is not down to 0. The TST and TSTB test instructions update the N and Z bits if you wish to subsequently branch on the outcome of a result other than the last one to flow through the ALU.

The distinction between arithmetic with signed and unsigned numbers has been treated at length. When unsigned numbers are subtracted or compared (such as memory addresses), it is necessary to use the branch instructions specifically reserved for this purpose. Their importance warrants their being repeated here, in figure 7.12. Similarly, when signed numbers are subtracted or compared , it is necessary to use the appropriate set of conditional branch instructions from figure 7.13. Using the wrong kind of conditional branch may not lead to an immediate program failure, if by coincidence the operands were both positive, but this lack of attention plants the seeds for a latent bug which may be apparent only months or years later.

Exercises

7.1 The parity of a group of bits is said to be odd if it contains an odd number of 1 bits. Write a program which can find the parity of a 3-bit nibble, where the nibbles are stored in the low part of a word.

7.2 Write a program to find the parity of 3-bit nibbles which appear in the high part of a word.

7.3 The MACRO-11 assembler is written in MACRO-11. How does an assembler assemble itself? Outline a plausible scenario for constructing an assembler.

7.4 Write each of the required representations in octal.

(a) −49 in the two's-complement system, in a 16-bit word.
(b) The most negative integer representable in two's-complement form by a 16-bit word.
(c) The largest positive integer representable in the two's-complement system, in a 36-bit word.
(d) The logical complement of the 16 bits in the octal number 102345.

7.5 Prove or disprove: every occurrence of a "BR x" (where x is a properly defined label) can be replaced by the pair of instructions BNE x and BPL x without the flow of control being altered. Disregard the case when x is such that x+2 would be out of range.

7.6 The instruction `ADD %1,%2` could result in the condition code having various bit patterns, depending on the values in %1 and %2. For each bit pattern shown below, give a pair of values for %1 and %2 which would cause an ADD %1,%2 to produce the specified pattern in the condition codes.

Condition Code Pattern					%1	%2
	N	Z	V	C		
(a)	0	0	0	0	_____	_____
(b)	0	1	1	1	_____	_____
(c)	1	0	0	1	_____	_____
(d)	0	0	1	0	_____	_____

7.7 What is the result produced by an `"ADD P,Q"` where the contents of P and Q were 032710 and 060212, respectively?

(a) The result?
(b) Describe one situation where this result is desirable and meaningful.
(c) Describe one situation where this is indicative of an error.
(d) Show the contents of the condition code bits.

7.8 A certain computer uses an 8-bit word. Answer the following about this machine in binary.

(a) If an address fills one word, and each address is a word address, how many words can be addressed?
(b) Assume that it uses two's-complement representation for signed numbers. What is the largest (most positive) number that can be represented in one word?
(c) What is the smallest (most negative) number that can be represented in one word?

7.9 Convert the following 7-bit binary numbers to decimal, in sign-magnitude form (assume that negative binary numbers use two's-complement representation).

(a) 0010000
(b) 1000000
(c) 1111111

7.10 Show the source operands for which MOV instructions, if any, can lead the N and Z bits of the CC to simultaneously assume the indicated values:

	N	Z	Results from:
(a)	0	0	-
(b)	0	1	-
(c)	1	0	-
(d)	1	1	-

7.11 Assume that each instruction takes 1 microsecond to fetch-execute, and that fetching or storing 1 word of data also takes 1 microsecond. Write the code to perform a triple-precision-integer addition from memory to memory. Assume that the operands are at memory locations A and B and the result goes to memory at location C. How much time does it take?

7.12 For instructions `INC X` and `INCB X`:

(a) Show when they produce the same result.
(b) Show when they produce different results.
(c) Show how one of these instructions could cause the program to abort.

7.13 For each combination of N and Z bits shown, write one PDP-11 instruction which forces the N and Z bits into the indicated values. Each one must use one or the other (or both) of A and B, where:

```
A:      .WORD   5
B:      .WORD   4
```

	N	Z
(a)	0	0
(b)	0	1
(c)	1	0

7.14 Suppose you needed to report what the setting of the four CC bits was, without changing them. Keeping in mind that even a MOV can change the N and Z bits, show how you could "copy" the CC bits into memory, using only MOVs and the set of branch instructions we have seen. Show how you "restore" the CC bits if they were changed in the process of recording their state.

7.15

(a) What is the result produced by `ADD A,B` where A and B have the values 021450 and 075102, respectively?
(b) Under what interpretations is this result desirable?
(c) Under what interpretation is this result indicative of an error?
(d) What will be left in the CC?

7.16 True or False?

(a) `INC %2` and `ADD #2,%2` produce the same result.
(b) `.EVEN` is a subroutine called at runtime to force the PC to the next even address.
(c) Registers can be initialized at assembly time.
(d) A two-operand instruction always requires three words of memory.
(e) The assembly language statements

```
.BYTE   130,131,132
.ASCII  /XYZ/
```

produce the same code within a program.
(f) If your program executes an illegal instruction, the system will always display the address of the guilty instruction by means of an M-TRAP message.
(g) Conditional branch instructions test the values in the CC to decide whether or not to branch.
(h) `BR (%3)` is a useful way to dynamically transfer control within a program at run time.

7.17 The following are all in octal:

177777	6775
000000	42765
103542	236
100000	

Suppose that the above are addresses:

(a) Which is the smallest address?
(b) Which is the largest address?

Suppose that the above are PDP-11 (two's complement) representations of numbers:

(c) Which is the smallest signed number?
(d) Which is the largest signed number?
(e) What is the sum of 103542 and 100000, and what (if anything) can we say about the result?
(f) If you want to compare addresses and branch accordingly, list two permissible branch instructions; list two that are not permissible.
(g) Repeat (f) for comparison between signed numbers.

7.18 Assume that when the following program gets to location CASES, R0 contains either 0, 2, 4, 6, or 8, and that ZERO, TWO, FOUR, SIX, and EIGHT are symbolic labels that may be scattered throughout the program. Write one single instruction at location CASES that will cause the program to go to one of locations ZERO, TWO, FOUR, SIX, or EIGHT, depending upon the value of R0. You may add as much data at the beginning of the program as you like.

```
DATA:   ;add data here



START: .
       .
       .     ; inst. to branch to ZERO,
               ...,
CASES:         ; EIGHT depending on value
                 of R0
```

7.19 Prove the statement made in case 1 of the section "Comparing Signed Numbers" that, given two numbers S and D, with $S < D$, then it must be true that $S < 0 < D$ if their difference causes overflow.

7.20 Many different things can be seen in a memory dump. If you had no idea what one or several consecutive octal words were supposed to represent, name 6 different kinds of things they might represent. Name things such as signed integers, for example; not data structures.

7.21 Consider the instruction sequence given below where the "---" stands for a conditional branch instruction:

```
MOV    #170605, %1
MOV    #107173, %2
ADD    %1, %2
---    LOOP
```

(a) Give the values of the condition code bits after the ADD is executed.

(b) For each conditional branch instruction listed below, write a Y underneath it if that instruction used in place of the "---" would transfer control to LOOP; write an N otherwise.

```
BCS BCC BVS BVC BMI BPL BEQ BNE

BLE BGT BLT BGE BLO BHIS BLOS BHI
```

7.22 Assume that register 1 contains 177774 and register 2 contains 100050, and that they are reset to these values after each of the following instructions. Fill in the spaces indicating the results of each instruction. Use * to indicate "no effect."

	N	Z
(a) `TST %1`		
(b) `CMP %1, %2`		
(c) `MOV %1, %2`		
(d) `ADD %1, %2`		
(e) `BNE (%2)`		
(f) `SUB %1, %2`		

7.23 Write the code to perform a double-precision-integer addition of two double-word operands, from memory to memory.

7.24 Fill in the indicated representations:

(a) −33 as a two's complement number, in a 16-bit word, in octal.
(b) The largest positive integer representable in two's-complement form by a 16-bit word.
(c) The most negative number representable in two's-complement, in a 12-bit word, in octal.

7.25 Describe the similarities and the difference, if any, among:

(a) `.BYTE 0,0,0`
(b) `.ASCII /000/`
(c) `.BLKB 3`

7.26 Assume that the specified registers and memory locations are reinitialized to the values shown prior to executing each of the following instructions (so that it does not matter in what sequence they are executed). Show the content of each register, memory location, and condition code bit changed by each instruction. Items which do not change should be marked with a dash (—).

	%0	*%2*	*%5*	*776*	*1000*	*1002*	*2000*	*C V N Z*
Initial Conditions	1000	467	1000	1002	2000	0000	177776	0001

(a) `MOVB %2, %0`

(b) `ADD (%0),(%0)`

(c) `CMP %0, 2000`

(d) `MOVB (%0)+, (%0)+`

(e) `MOVB %2, (%5)+`

(f) `SUB #776, %0`

(g) `MOV +1000(%5),1000`

7.27 Which four branch instructions, after arithmetic involving two's-complement numbers, will branch as desired even after overflow has occurred?

7.28 Which two branch instructions, after arithmetic involving two's-complement numbers, will branch as desired except when overflow has occurred?

7.29 Which four branch instructions are appropriate for use when unsigned integers (such as addresses) are compared, and which will always work as desired, even when a carry has occurred?

7.30 The following situations each describe how one bit of the condition code (CC) is affected by arithmetic operations. Identify the condition code bit described by each line.

(a) Set if the sign bit of the result is 1, cleared otherwise, after an add or a subtract.
(b) Set during a subtract if there is a borrow into the MSB (most significant bit, bit #15), and cleared otherwise.
(c) Set during an add if two two's-complement numbers of the same sign are added but the result is of the opposite sign.
(d) Cleared if the result of an add is 0, set otherwise.
(e) If the operands of an add are two's-complement (signed) and this bit is set, it indicates that some sort of error has occurred; but if the bit is cleared, then no error has occurred.
(f) If operand A is subtracted from operand B, this bit is set only if A and B are of different sign (two's-complement) and the result is the same sign as A.
(g) Set after an add if there has been a carry out of the MSB (bit #15), cleared otherwise.
(h) If the operands of an add or subtract are considered to be unsigned and this bit is set, it indicates that an error has occurred; if the bit is cleared, it indicates that no error has occurred.

7.31 The following are all octal numbers:

000000	100001
177776	006775
127003	042765

(a) Which is the largest unsigned number?
(b) Which is the largest signed number?
(c) Which is the smallest unsigned number?

(d) Which is the smallest negative signed number?
(e) What is the sum of 127003 and 100001?
(f) What can you say about the result of (e)?

7.32 Write a program segment which can add two 48-bit integer operands, updating the second operand, and which transfers control to a statement labeled ERROR if the result is invalid. Assume two's-complement representations.

7.33 Perform the following operations using two's-complement arithmetic. Show your work. If the result is negative, then also show the result in sign-magnitude form (as well as in two's-complement form). As an example, the first problem has been done for you. (Assume a 16-bit word, all words in octal.)

(a) $14 - 26$

$$\begin{array}{r} 000014 \\ +177752 \\ \hline 177766 = -12 \end{array}$$

(b) $1421 - 10$

(c) $-1246 - 27$

(d) $1073 - 64712$

7.34 Assume that after each instruction, the values of A and B are reset to their original contents before the next instruction is executed. To the right of each instruction, show the value of the result of that instruction (in octal) and of each bit of the condition codes. As an example, the first line has been done for you.

```
B:    .WORD    103724
A:    .WORD    070632
```

		Value of result	N	Z	C	V
	`ADD A,A`	161464	1	0	0	1
(a)	`ADD A,B`					
(b)	`ADD #071234,A`					
(c)	`SUB B,A`					
(d)	`CMP B,B`					
(e)	`ADD B,B`					
(f)	`ADD #16,B`					
(g)	`SUB B+2,A`					
(h)	`ADD #177772,A`					

7.35 The following segment of code is supposed to print GOOD NEWS. Instead it is printing G*******. Correct the code and explain the nature of the problem. Hint: assume the program is relocated by octal 1000.

```
076726 012701          MOV    #MESS,%1
076732 012702          MOV     #OUTPUT,%2
076736 112122    LOOP: MOVB    (%1)+,(%2)+
076740 020127          CMP     %1,#ENDMES
076744 002774          BLT     LOOP
076746                 .PRINT #OUTPUT
076764 000137          .EXIT
076770     107 MESS:   .ASCII /GOOD NEWS/<12>
077011     000 ENDMES: .BYTE   0
077012     052 OUTPUT: .ASCII /********/<12>
```

7.36 The V bit of the CC (condition codes) indicates whether or not an error (overflow) has occurred. An error is considered to have occurred only when the operands are treated as (choose the correct answer or answers):

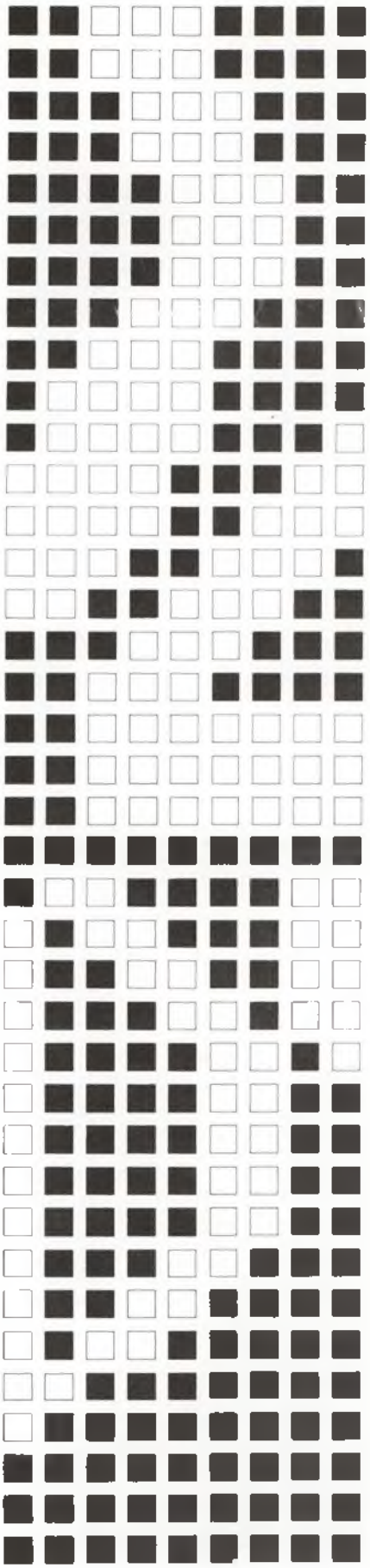
(a) unsigned numbers (such as addresses)
(b) two's-complement numbers
(c) either of the above

7.37 Fill in the blanks below so that each line contains the positive, the one's-complement negative, and the two's-complement negative forms of the same integer value. Write all values in octal, using a 16-bit word. As an example, the first line has been done for you.

	Positive	**Negative** (one's-complement)	**Negative** (two's-complement)
	001234	176543	176544
(a)		103610	
(b)			100003
(c)	000002		
(d)			177777
(e)	070372		
(f)		103164	

8

subroutines and stacks



Subroutines, or Necessity Is the Mother of Invention

The jump and conditional transfer instructions allow us to include decision making in our programs. Coupled with indexing and auto-increment addressing modes, we can reuse instructions in a productive way. Now a small program can process a very large quantity of data.

Unfortunately, small programs easily turn into large programs almost spontaneously, as the problems being solved become more complex from one day to the next. Large programs are very much harder to design, write, test, document, improve, and maintain—more so than the enlarged size might lead you to believe.

Much of the success of modern science and engineering is due to the systematic application of the maxim "Divide and conquer." Complex problems are analyzed and decomposed into a set of simpler subproblems. If the subproblems are still too complex to grasp easily, these in turn can be decomposed into subsubproblems. This process of successive problem analysis and decomposition can be illustrated by using a "problem tree" structure. Computer scientists often see the world inside out, or upside down, as is the case with the tree depicted in figure 8.1.

A problem, represented by P, is shown as being decomposed into the three subproblems P1, P2, and P3. The first two of these are, in turn, further decomposed into subsubproblems P11 and P12, and P21 and P22, respectively. Note that some subproblems may not require further subdivision, and that the number of subdivisions at any level may be as large or as small as necessary. Furthermore, the number of levels of subdivision may be quite deep. In figure 8.1, we have a total of 3 levels of subdivision (including the original problem). Levels two and three each include three and four subproblem modules respectively. You might see some similarity between a problem tree and the classical organizational chart many companies proudly display.

What is the connection between problem trees and computing? The blocks involved in the decomposition (usually called the *nodes* of the tree) can be viewed as self-contained segments of instructions which solve a particular part of the problem program P is attempting to solve. The mental process is: we could write program P if we had a subprogram P1 doing this, and a subprogram P2 doing that, and a subprogram P3 doing the rest. And P1 would be easier to write if we had subsubprograms P11 and P12; and so on.

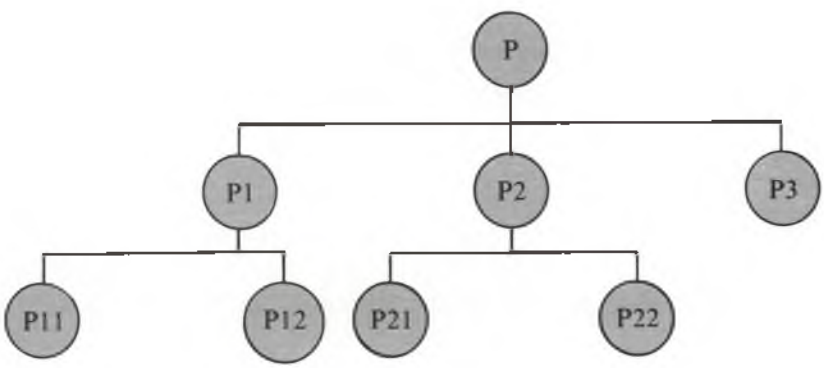


Figure 8.1 Problem decomposition as a tree.

A subprogram is usually called a "procedure" or a "subroutine." The name function is also used. We won't make any particular distinction between the names procedure and subroutine. A subroutine provides a self-contained segment of instructions which can be invoked (used) by another program.

Suppose we had to multiply by ten twice in a program (and we don't yet know how to use the MUL instruction):

```
                ...
X:              MOV     A,%1
                ADD     %1,%1   ; 2X
                ADD     %1,%1   ; 4X
                ADD     A,%1    ; 5X
                ADD     %1,%1   ; 10X
XX:             ...
                ...
Y:              MOV     Q,%1
                ADD     %1,%1   ; 2X
                ADD     %1,%1   ; 4X
                ADD     Q,%1    ; 5X
                ADD     %1,%1   ; 10X
YY:             ...
```

If we had written the instruction sequence:

```
                ...
MUL10:          MOV     %1,%2   ; make a copy
                ADD     %1,%1   ; 2X
                ADD     %1,%1   ; 4X
                ADD     %2,%1   ; 5X
                ADD     %1,%1   ; 10X
                ...
```

then the previous sequences at X and Y could be shortened to:

```
                ...
X:              MOV     A,%1
                JMP     MUL10
XX:             ...
                ...
Y:              MOV     Q,%1
                JMP     MUL10
YY:             ...
                ...
```

Unfortunately, MUL10 doesn't return control to resume the program at the instruction with the label XX, so the shorter code sequence provided by MUL10 won't be used by the invocation at Y, since we have no way to get there.

We need a way for *one* copy of the instructions for MUL10 to be usable from many different points of a program. This means we have to provide MUL10 with a way of returning control to the right place at the right time. In our example, we have to be able to get back to XX and YY at the right times.

We could try the following:

```
X:      MOV     A,%1
        MOV     #XX,MULR ; return adr
        JMP     MULR+2
XX:     ...
        ...
Y:      MOV     Q,%1
        MOV     #YY,MULR ; return adr
        JMP     MULR+2
YY:     ...
        ...
MULR:   .BLKW   1           ; save return adr here
        MOV     %1,%2
        ADD     %1,%1       ; 2X
        ADD     %1,%1       ; 4X
        ADD     %2,%1       ; 5X
        ADD     %1,%1       ; 10X
        MOV     MULR,%2
        JMP     0(%2)
```

The instructions from MULR to the "JMP 0(%2)" constitute a subroutine. Using the subroutine, as is done in the sections labeled X and Y, is called *invoking* or *calling* the subroutine. The first invocation of MULR (just before XX) places the address XX in location MULR. The MULR code returns control using index mode, with the effective address 0 + XX, which is the desired return point. Similarly the second invocation, just before YY, places the address YY in location MULR, and it finally gets into %2, yielding an effective address of 0 + YY for the JMP which ends the MULR routine.

This subroutine return mechanism was used on early computers and is still in use. There should be a better way. Since the appropriate use of subroutines is not only an excellent tool for problem decomposition, but also a very effective way of reducing the memory needs of a program (a very important consideration when you are designing a "mini" computer), the designers of the PDP-11 provided addressing modes that are very useful in supporting subroutine use. We shall examine these, and then reconsider subroutine mechanisms for the PDP-11.

Figure 8.2 General register addressing modes.

mode		mode	
0 OPC R	register	1 OPC (R)	reg deferred
2 OPC (R)+	auto-inc	or @R	
4 OPC -(R)	auto-dec	3 OPC @(R)+	auto-inc def
6 OPC X(R)	indexed	5 OPC @-(R)	auto-dec def
		7 OPC @X(R)	indexed def

Deferred-Address Addressing Modes

The MACRO-11 assembler recognizes all the address expressions shown in figure 8.2 as valid. They are called the "general register addressing modes" because they are used in conjunction with registers 0 to 6 (once again you are cautioned to avoid using %6 until further notice).

We have already discussed using modes 0, 1, 2, and 6. As is evident in figure 8.2, each even mode has a corresponding deferred mode which is assigned an odd mode code. Mode 0, OPC R, uses (R) as the operand. Mode 1, OPC (R), or OPC @R, uses (R) as the address of the operand for OPC.

Similarly, mode 3, OPC @(R)+, uses (R) to locate an address, and then it uses that address to find its operand, leaving (R) two greater than before, because of the auto-increment request.

Mode 4, auto-decrement , OPC −(R), modifies R by −1 or −2, depending on whether OPC operates on a byte or a word. Then it uses the *new* value in R to locate its operand. With auto-decrement, you can now easily step backward over consecutive bytes or words.

The deferred counterpart of mode 4 is mode 5, auto-decrement deferred. The form OPC @−(R) causes register R to be decremented by two; then the new value in R is used to locate an address. This last address is used to locate the operand for OPC.

The last of the general register addressing modes, mode 7, is called index deferred. As with mode 6, index mode, the base address or full offset may have a + or − sign. If the base address is zero, it may be omitted. The form OPC @X(R) leads to finding the effective address (R)+X, and this address is used to locate a second address. The second address is used to locate the operand for OPC. Nothing has happened that would change (R).

We will demonstrate the use of these modes in context, after we discuss subroutines more extensively. Figure 8.3 summarizes the actions of the addressing modes. Note that all of them can be used in double-operand instructions—e.g., `MOV @ABC(%3),-(%2)`—as well as in the single-operand instructions. When used in a double-operand instruction, the effective source address is evaluated first, the effective destination address is evaluated next.

In the cases of modes 2 and 4, the change in (R) is governed by the nature of the OPC. A byte operation will force a change by 1; a word operation forces (R) to change by 2. In the cases of modes 3 and 5, (R) will *always* change by 2, since R is always used to locate an address with these modes, and the address of an address must always be even.

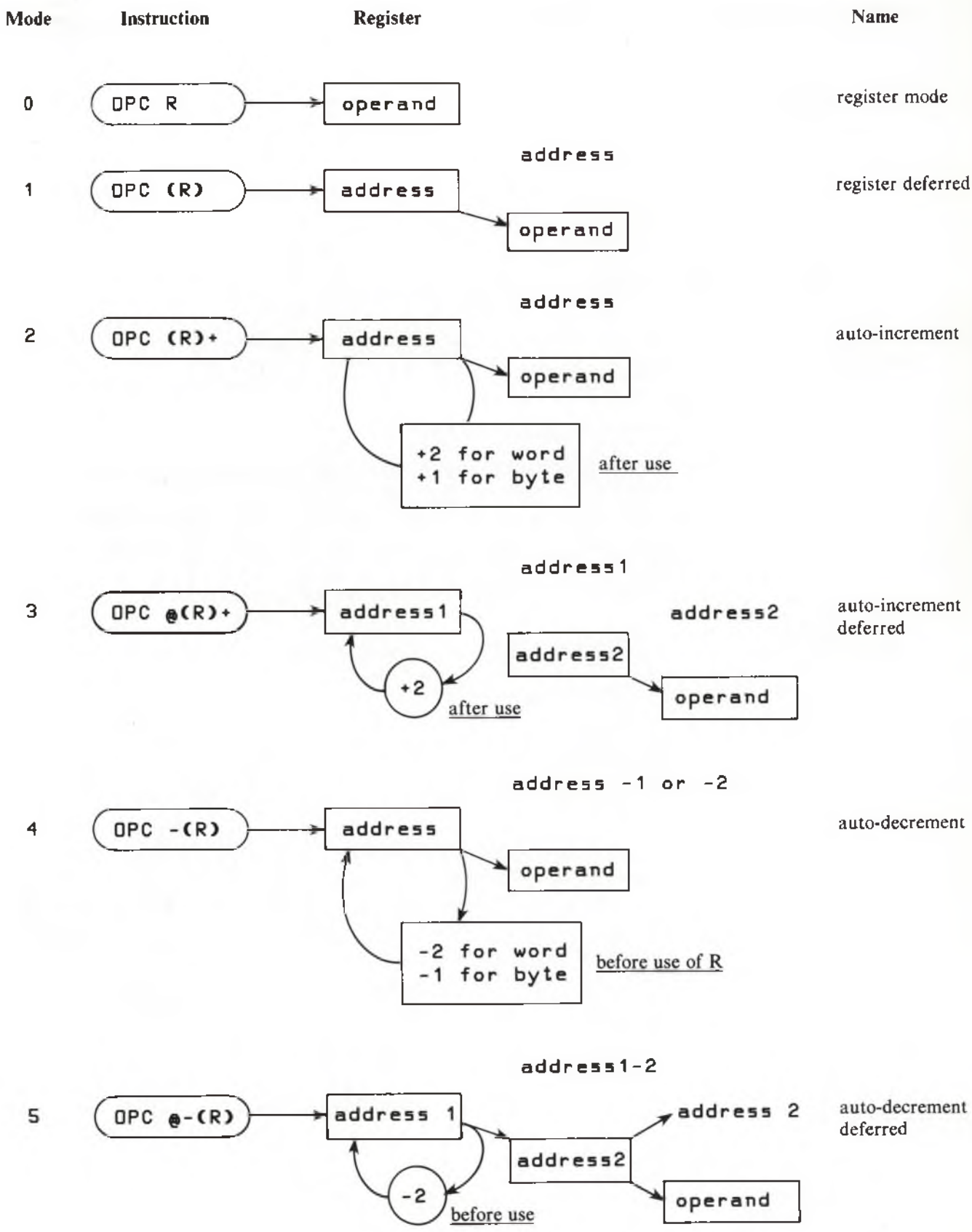


Figure 8.3 General register addressing modes in action.

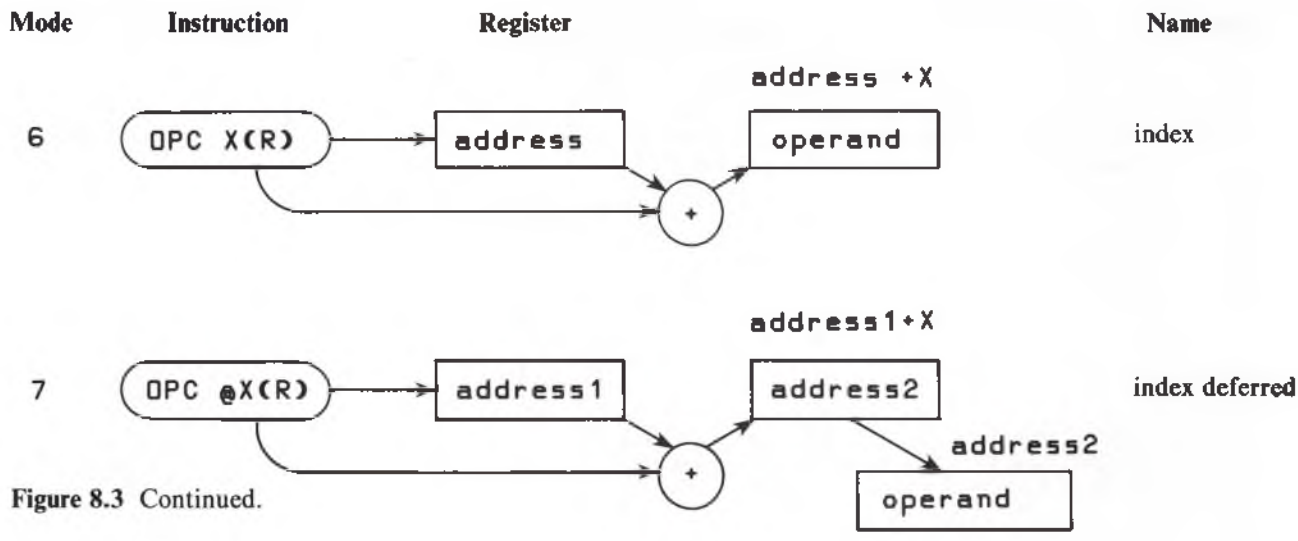


Figure 8.3 Continued.

Program Counter Addressing

We can complete the examination of all the remaining PDP-11 addressing modes in this section. There are only four more addressing modes, and they are all associated with the use of the PC. When the program uses these modes, no register is named; the assembler recognizes that the PC register must be used. With these modes, the use of the PC is always implied; it is never made explicit in the symbolic program.

Relative Addresses

When we write the most common form of memory reference:

OPC ADR

the assembler treats this as if we had written

OPC OFFSET(%7)

This is a use of mode 6, index addressing, with the PC being used as the index register. It is called *relative addressing*. The OFFSET is a 16-bit offset calculated by the assembler so that the effective address computed at execution time, OFFSET+(%7), will be the same as ADR. Consider the following:

```
Adr          Content          Comment
010          005267           INC ABC
012          000002
014          000000           HALT
016          000001   ABC:    .WORD    1
```

The opcode for INC is 0052. The "67" in "005267" represents addressing mode 6, using register 7. The content of location 12 is set to be 000002 at assembly time. The assembler assumes that, when the program is executed, when the number from location 12 is in the control unit, the PC will have been updated, and its value will be 14. So, in evaluating the effective address, the control unit will take (12)+(PC) → 2+14 → ABC.

Even if the program has been relocated by octal 1000, or any other amount, say RELAMT, both the value in the PC and the address ABC would be increased by the same amount RELAMT and the offset would not change, so the effective address would still be the correct one. Similarly, suppose we had:

```
000     000123  LAST:   .WORD  123
        ...             ...
020     021167          CMP    (%1),LAST
022     177754
024     ...             ...
```

Here the 16-bit offset 177754 is negative because the symbolic reference LAST is a backward reference. At execution time, after fetching 177754 from 022, the PC will be updated to 024. In completing execution of the `CMP (%1),LAST`, the control unit will evaluate:

```
177754 = offset
   24 = (PC)
000000 = LAST
```

Note that the encoding in the first word of the CMP instruction is

```
  02    11    67
  OP    SS    DD
 CMP    MR    MR
```

The destination coding is 67; mode 6, register 7 (=PC). Relative addressing always uses a 16-bit word to store the 16-bit offset. So a double-operand instruction may be one, two, or three words long, depending on how many relative addresses it uses (0, 1, or 2).

If we are careful in using relative addresses whenever memory references to operands are made, we will have gone a long way toward ensuring that our programs have the property of being position independent, since all of these references to memory are PC-relative.

Immediate Operands

The "immediate operands" that we have been using, such as in `MOV #77, %3`, or `CMPB A(%0),#40`, are treated as mode 2 requests, and the use of the PC is always implied. The actual operand itself will be stored as either the second or the third word of the instruction, so the implied auto-increment reference using the PC will find just what we are looking for, and then update the PC appropriately.

Relative Deferred Addressing

Both the relative and immediate operand modes have their deferred counterparts. OPC @ADR, relative deferred addressing, leads to the assembler working with OPC @OFFSET(%7). We could have used this for the subroutine MULR:

```
MULR:   .BLKW   1       ; save return address
        ADD     ...
        ...
        ADD     %1,%1   ; 10X
        JMP     @MULR
```

This will force the desired return address (one of XX or YY, if you recall) previously stored in location MULR, to be forced into the PC. This type of addressing is often called *indirect addressing*.

Deferred Indexing

You may recall that when we discussed the program segment dealing with an automated bank teller station, we promised that a better way of handling a "jump table" was forthcoming. Promises must be kept.

A "direct" JMP must have an instruction in the location which its destination address specifies:

```
JMP ABC; expect some instr. at loc ABC
```

An indexed JMP must also have an instruction as its "target":

```
      JMP TIN(%1) ; (%1)+TIN → PC
TIN:  JMP A
      JMP B
      ...
```

With deferred indexing, we can have the initial target of the JMP be a table of addresses. Thus

```
      JMP    @VECTOR(%2)     ; 0,2,4,... in %2
VECTOR:.WORD  CASE1,CASE2,CASE3, ...
```

The addresses beginning at location VECTOR form a *jump table*. For an initial value of 4 in %2, the sequence of addresses evaluated at execution time is:

VECTOR+(%2) = VECTOR+4
(VECTOR+4) = CASE3
CASE3 → PC

So we effect a `JMP CASE3` when %2 holds a 4.

Absolute Addressing

The last addressing mode we have to examine is actually the first one we ever used! In writing machine-language programs a long time ago, we wrote (see figure 3.7):

```
1000: 013737     ; MOV a,c
1002: 1016
1004: 1022
      ...
```

The addresses 1016, 1022 were the addresses for items a and c. Mode 3 is called the *absolute addressing* mode. Writing machine-language programs with relative addresses would have required performing octal additions and subtractions by hand in order to compute the necessary offsets while keeping track of the expected value in the PC. It would have been too much for us to keep track of at the time.

Figure 8.4 Program counter addressing modes.

called "RELATIVE ADDRESS." We recognize it as mode 6, index mode. Assembler generates offset value so the OFFSET + [PC] = ADR

which is called "IMMEDIATE OPERAND." We recognize use of autoincrement mode 2.

called "relative deferred address," it uses index deferred mode 7

called "absolute address"; it uses autoincrement deferred mode 3

The assembler recognizes the notation OPC @#ADR as a use of absolute addressing. It will take the actual numeric value associated with ADR and place it in the second or third word of the instruction, and encode the corresponding mode and register fields as 3 and 7, respectively (remember all the xx3737 machine-language instructions?).

Most programs have no need to use absolute addressing. Why lock your program into a fixed section of memory if you don't have to? There are times when it may be necessary to refer to particular absolute memory locations; then you can use this mode of addressing.

Figure 8.4 sums up this discussion of PC addressing modes.

Relocation

If we have the following statements in a source program:

```
LL:       .WORD   CASE1
            ...
CASE1:      ...
            ...
X:        JMP     CASE1
            ...
```

the address assigned to CASE1 at assembly time may be 000200. The offset computed by the assembler, at line X, might be 177450 (if X+4−CASE1 is 177450). Suppose the linker places the first word of this program at location 1000. Then the address 0 assigned by the assembler becomes 0+1000, and the other locations are also changed by 1000.

This process is called *relocation*. What does the linker actually change in the program? The offset used in the JMP at location X need not be changed, because (X+4+1000) − (CASE1+1000) reduces to (X+4−CASE1), which is what we already have. However, the word at LL contains the address CASE1. If it were left as the number 000200, it would no longer be the correct address. So the linker adjusts it by adding the relocation amount 1000 to it. All direct memory references which appear as constants (e.g., as operands of a .WORD) must be modified by the linker. This is part of the relocation process performed at link time.

How does the linker know which words to modify, and which to leave intact? It is the assembler's responsibility to identify which words are subject to the relocation transformation and which are not. The assembler encodes this information using relocation bits, which the linker subsequently interprets and then strips away. An assembler that can produce relocatable code (as does MACRO-11) is called (somewhat misleadingly) a relocatable assembler. One which does not is called an absolute assembler.

The object code produced by a relocatable assembler is called a *relocatable module*. A linker or a loader which can accept relocatable modules as input is called a relocating linker or loader. The loaders and linkers which cannot accept relocatable modules are called absolute loaders or linkers. An assembler which does not produce relocatable code and a linker which does not accept relocatable modules are considered unsophisticated.

The linker will make a point of not modifying any absolute addresses (those written with a @# prefix).The other "ordinary" addresses will be increased by a relocation constant, by the linker. The linker will not change any absolute addresses or any offsets.

Do not be surprised if the octal information found in the assembly listing for your program and the octal words found in a memory dump of the same program do not coincide, even though your program did not modify the words that differ. It should be the case that the linker modified those words. If so, the difference in each case should be the same.

We now have completed our examination of all of the PDP-11 addressing modes and their implications regarding program relocation. It is time to resume investigating some powerful concepts.

Stacks

A stack is one of those very simple ideas that turns out to have far-reaching consequences. In everyday life, a stack is a place where you put things (bills, letters, messages) on top of older things, perhaps sitting on a desk or in an "in-basket." When you have a moment, you look at the thing on the top of your stack and take care of it. You then look at the

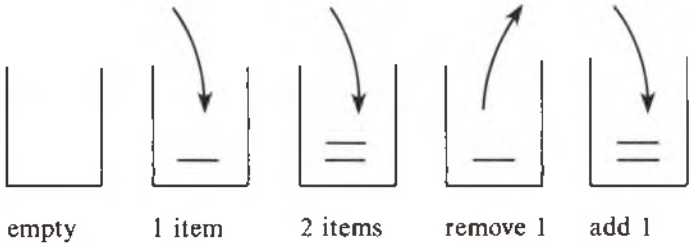


Figure 8.5 Stacking.

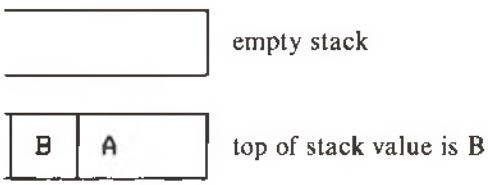


Figure 8.6 Stacks lying on their sides.

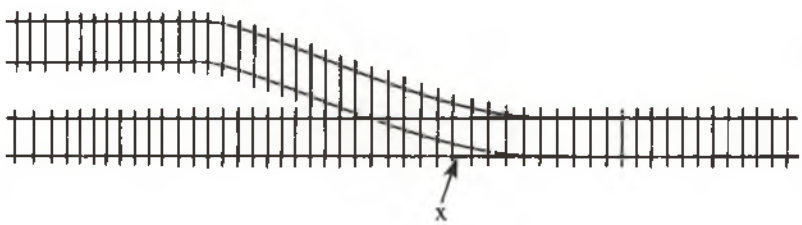


Figure 8.7 A train using a stack.

next thing, which is now on top of the stack, and take care of it. If you proceed in an orderly fashion, you never reach into the middle of the stack. You always put new things on top of the stack, and you always remove things from the top of the stack. It will be convenient for us to picture our stacks lying on their sides.

Railroads use stacks to hold boxcars temporarily when making up a train. A train E-C1-C2-C3-C4 may arrive, as depicted in figure 8.7. To remove boxcar C2, you back into the stack

E-C1-C2-C3-C4

uncouple C1 from C2, drive away with E pulling C1. Switch the track at X and uncouple C2 from C3. Let gravity cause C2 to roll off the stack. Then after switching X again, E-C1 can be coupled back to C3-C4, and the train may leave the stack.

Stacks in Computers

Stacks were not common in computing in the early 1960s, except notably in Burroughs computers. In the 1970s, stacks became a necessity for most minicomputers and almost all micros, and fairly common on most large computers.

The PDP-11 supports stacks by using some of the addressing modes we have seen, without any necessity for new hardware or new instructions. Suppose you want a stack with a maximum capacity of ten words. You can use one of registers 0–5 for your personal stack pointer. Let us call it PSP, and use %4 as our stack pointer:

```
        PSP= %4              ; personal stack pointer
BOT:    .BLKW  9.            ; stack space
MAX:    .BLKW  1             ; end of stack
        ...
A:      MOV    #BOT,PSP      ; initialize your st ptr
        ...
B:      MOV    X,(PSP)+      ; push (X) into st
        ...
C:      MOV    Y,(PSP)+      ; push (Y) into st
        ...
D:      ADD    -(PSP),SUM    ; pop (Y) off st
        ...
E:      ADD    XXX,-2(PSP)   ; (XXX)+(TOS)→TOS

F:      MOV    -(PSP),RES    ; pop TOS into RES
        ...
```

If we let (X) be x, (Y) be y, and x + (XXX) be z, then we can illustrate the behavior of our stack in figure 8.8. The abbreviation TOS stands for the current top-of-stack address. Since the stack is just a section of memory, when we say the stack is empty, or that one item was just popped off the stack, clearly that section of memory still contains what was there. We simply agree not to use anything beyond the current top of stack. Should the value in PSP ever exceed the address MAX, we will expect trouble.

The System Stack Pointer SP

We can finally explain why we had to avoid using %6. This register is reserved for use by the hardware and support software as the system stack pointer, and we will often use the name SP for it. Certain instructions such as a JSR imply use of the PC, even though no reference to the PC appears in the instruction. Similarly, certain actions of the control unit (e.g., detection of an illegal instruction) will lead to use of the SP in reporting the error.

Normally, to use a stack, one must:

1. Reserve a block of memory for it.
2. Designate some register for use as a stack pointer.
3. Initialize that register to point to the bottom of the stack.

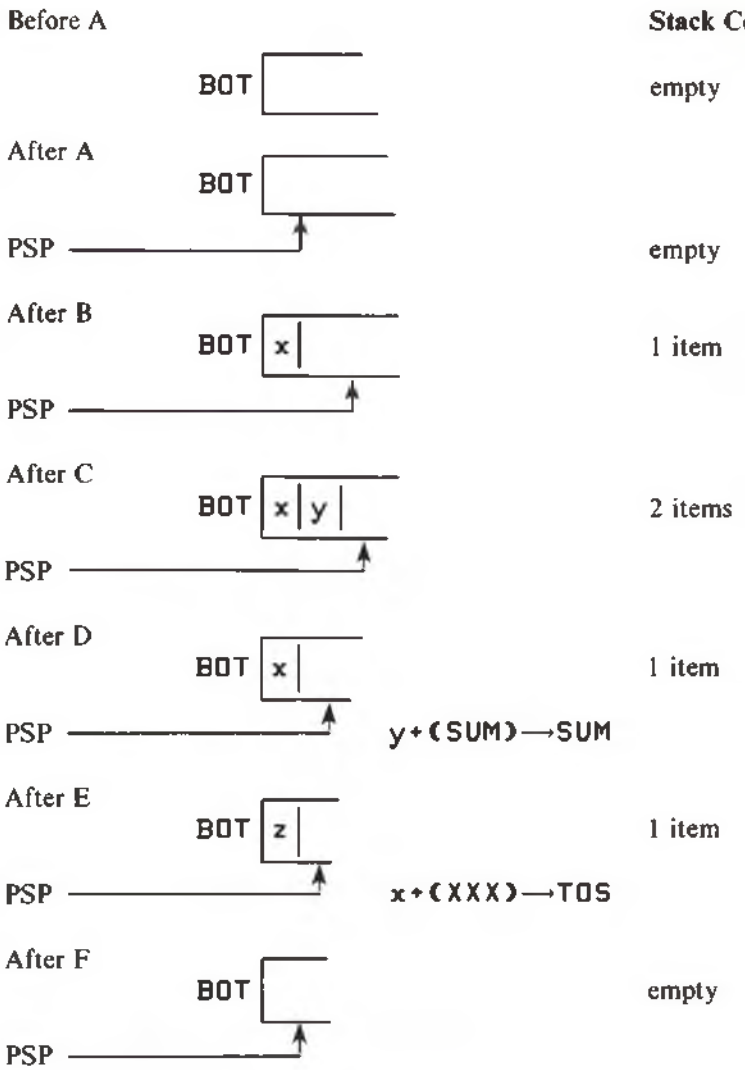


Figure 8.8 History of a personal stack.

You don't do any of this if you wish to share the system stack with the support software. It has been done for you, by whichever operating system you are using. With the single-user RT-11 operating system, the system stack pointer %6 is initialized to the value 1000 octal when RT-11 initiates execution of your program. Of course, if you are working on a bare machine (i.e., one without any operating system), you will be responsible for initializing %6, even if you do not plan to use it. The symbol SP is *not* predefined in MACRO-11, so you will need the statement SP=%6 if you intend to use the name SP; or, if you prefer, you can use the .MCALL .REGDEF to provide this definition. On a multi-user system, the memory management unit will usually have some other segment of memory assigned for use as your system stack, with a relatively large bottom-of-stack address.

If you plan to share something provided by the operating system, you must observe the ground rules. The ground rules governing use of the system stack are:

1. All pushes into the system stack are performed using auto-decrement addressing.

2. All pops from the stack are performed using auto-increment addressing.

This is just the opposite of the earlier example we worked out using our personal stack pointer. The system stack *grows down* into lower memory. Logically you could have stacks growing up or down, but since the system insists on doing it downward, life will be simpler if all stacks grow downward.

We can redo the previous example, taking advantage of the system stack which we will share with the operating system. Note that when our program begins execution, the system stack is always empty:

```
        ...
        SP=%6
A:      ; no need to initialize SP
        ...
B:      MOV     X,-(SP) ; push (X) into st
C:      MOV     Y,-(SP) ; push (Y) into st
        ...
D:      ADD     (SP)+,SUM ; pop (Y) off st
        ...
E:      ADD     XXX,(SP) ; (XXX)+(TOS)→TOS
F:      MOV     (SP)+,RES ; pop TOS into RES
        ...
```

Using the same notation as before, we can illustrate the behavior of the system stack and the SP in figure 8.9. The stack pointer should always be pointing to a valid top-of-stack item. If the stack is empty, then the SP cannot be pointing at the first word of the stack; this would be misleading. So it points to the word just above the bottom of the stack. If your program is loaded by the RT-11 linker, then the bottom of the stack is at location 776 (so BOT would be 776) and the SP is initialized (by RT-11) with the value 1000.

If you are using the RT-11 operating system, the allocation of memory is shown in figure 8.10.

If you fail to abide by the rules for using the SP and the system stack, you could easily arrange for the stack to overwrite your program, or parts of the library routines used by your program.

In spite of all warnings to the contrary, some people wind up doing things like this:

```
CLR     %6
ADD     X,%6
ADD     Y,%6
MOV     %6,RES
```

They use %6 as if it were just another general purpose register. This program segment might actually run correctly most of the time, but sooner or later during some execution, it will abort, even though no changes have been made to it, to the operating system, or to the hardware. How can a program which "ran correctly" suddenly fail, if nothing has changed?

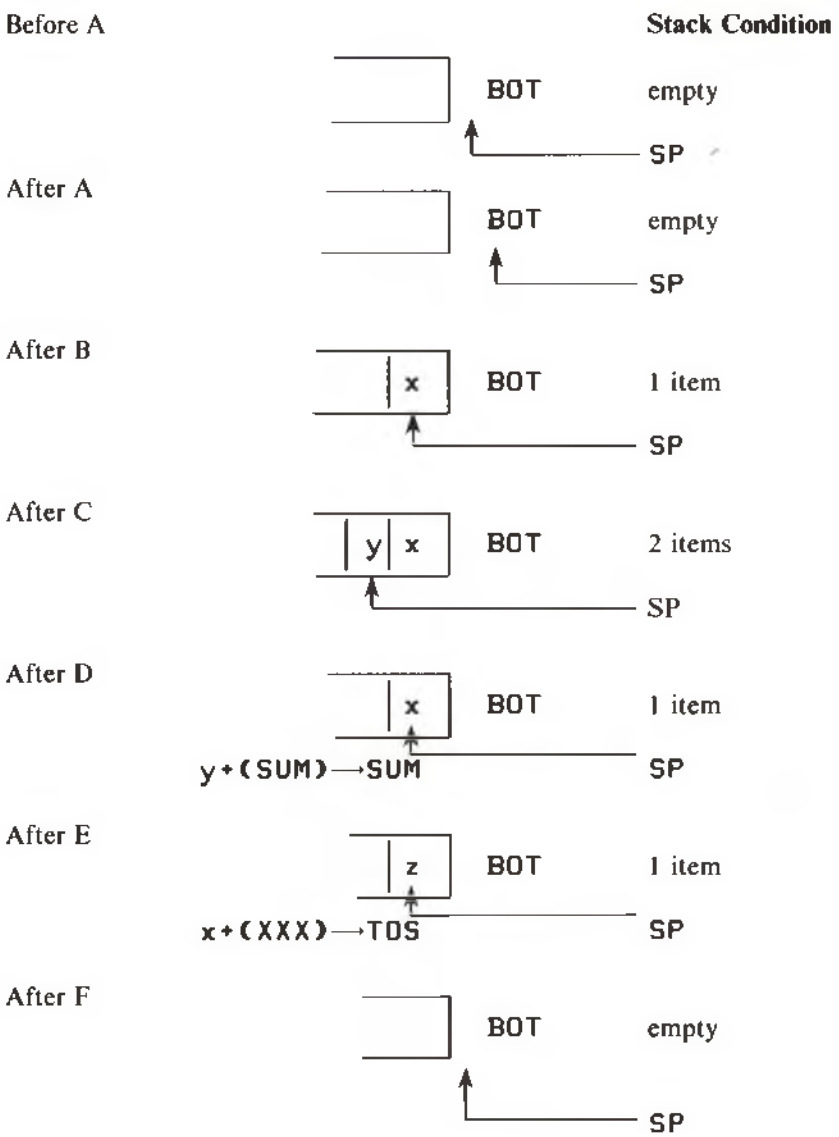


Figure 8.9 History of the system stack.

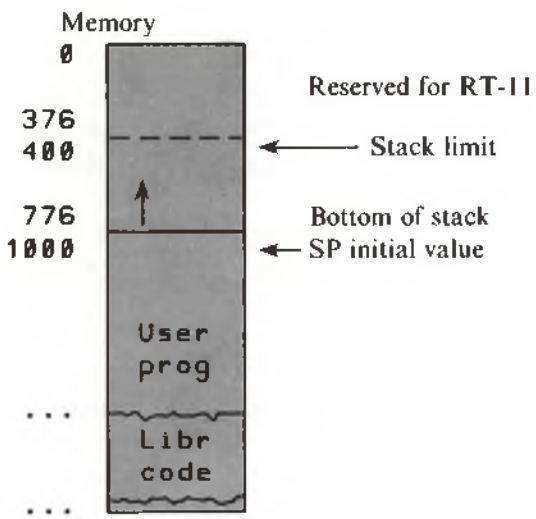


Figure 8.10 Memory allocation in an RT-11 system.

The key to understanding this somewhat mysterious phenomenon is an awareness of the computing context in which you operate. The system services that a user's program requests from the support software (e.g., output) use the system stack *while* your program is executing. There are conditions that may have absolutely no connection with your program that may cause the control unit to temporarily interrupt your program so it can let the operating system use the system stack while your program was running.

Consider the following cases:

1. The program uses some system service routine (e.g., a dump, a "read," a "print," etc.).
2. The program attempts to execute an illegal instruction or use an illegal address.
3. The system supports a clock of some type.
4. The memory has parity error checking hardware.
5. A momentary power fluctuation occurs.

If your program is using %6 in any manner inconsistent with its use as *the* system stack register, when you come to invoke the services of case 1, you will be in for a surprise, since such services cannot be provided without using the SP. If your program is not error-free, when case 2 occurs, you will get another surprise. The control unit relies on the SP to get to the service routines that produce error messages such as "M-TRAP TO -." In such a case, you will not even be told why your program aborted.

With case 3, the computer's clock may be ticking at the rate of 60 times per second, if not much faster. Each time it ticks, it needs to use the system stack to update the current time. Some computers use an interval timer in addition to or instead of a clock. Such timers can set their alarms to go at arbitrary times, and if it happens when your program is running, they will use the system stack. Under normal conditions, these clock or timer interruptions of your program are totally transparent (invisible) to your program, but not if you were using %6 improperly.

Case 4, memory parity error, might occur once a day or once a month, as a transient error. If your computer does not have the hardware to automatically check parity for each byte or word, then you will simply use a "bad" word or byte, and if it is an item of data, too bad. If it is part of an instruction, it may change a valid instruction into some other valid instruction (e.g., MOV, code 01 → MOVB code 11), or into an invalid instruction. In the former case, too bad for you. In the latter, you might have gotten an M-TRAP message to warn you, if only you had respected the rules for use of %6. If your computer does have the hardware to check for memory parity errors, upon detecting such an error, the hardware will attempt to use %6 in order to report the parity error.

Case 5 involves a standard feature of all PDP-11s. If the computer's power supply detects a sufficiently large deviation from the normal 110 volts and the normal frequency of 60 Hertz (220V and 50Hz in some countries) for a sufficiently long time (a few milliseconds), it (the power supply) will inform the control unit about the situation, and the control unit has some 2 milliseconds to complete a "graceful" power failure shutdown procedure—which, of course, uses the system stack.

When and if power is restored, the computer is able to sense that fact, and it will attempt to restore normal operation triggered by a power-up interrupt.

We will be looking into the details of these "interrupts," particularly when we discuss input and output operations, but also when we discuss error detection features of the hardware.

Couldn't you cheat "just a little" by using %6 for just a few instructions? Suppose you tried:

```
        ...
        MOV     SP,SAVE
A:      CLR     SP
        ADD     X,SP
        MOV     SP,SUM
B:      MOV     SAVE,SP
        ...
```

and you never used the SP anywhere else in your program. Since you have no way of guaranteeing that cases 3, 4, or 5 cannot occur, the hardware could interrupt your program between points A and B. Since these hardware-generated interrupts require the use of the SP, you have a disaster on your hands. The probability of it happening while you are in the midst of a sequence of five instructions may be very low, but anyone who knowingly does it in spite of these warnings is planting a time bomb in his program.

How Does the System Share the Stack?

If you are using the system stack properly, you may be at a point in your program with the system stack looking like figure 8.11. For any of the reasons mentioned earlier, the operating system may get to use the stack temporarily, and while the system has it in use, the stack may come to look like figure 8.12. Presumably items P, Q, and R were pushed into the stack by the operating system or its service routines. It is the system's responsibility to leave the stack as it found it when it began sharing it with your program. So at the point the system permits your program to resume, the stack will be back to its original configuration, figure 8.11. This may explain why, if your program happens to never put more than three words in the stack (e.g., a, y, and x), the content of the next several words may be changing even though you never touched them.

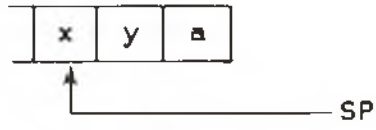


Figure 8.11 Original condition of stack.

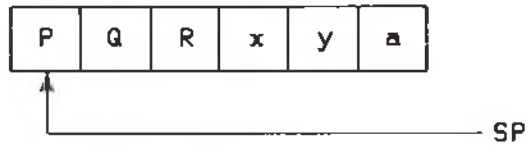


Figure 8.12 Stack during use.

Rules for Using the System Stack

We can repeat and expand the rules regarding the use of the system stack:

1. Push items into the system stack using auto-decrement mode.
2. Pop items from the system stack by using auto-increment mode.
3. If you explicitly increase the SP value, expect the items at locations (SP), (SP)−2, . . . to vanish.
4. A push or a pop of the SP will always change its value by 2, even if you are executing a byte operation. The SP must always contain a word address, and any change in its value which is implied by an auto-increment or auto-decrement operation will change the SP value by 2 even when performing a byte operation.

In rule 3, the values may not exactly vanish, but you cannot guarantee that they won't be overwritten, so you should assume that these values are no longer accessible.

Recycling Memory with Stacks

Why bother using stacks? Careful use of a stack can reduce a program's memory requirements. Suppose you are in the middle of writing a lengthy program and all registers are being used. You need three registers temporarily, to help you copy a group of words:

```
            MOV     %1,SAVER1
            MOV     %2,SAVER2
            MOV     %3,SAVER3
A:          MOV     #ARRA,%1
            MOV     #ARRB,%2
            MOV     #50.,%3
LOOP:       MOV     (%1)+,(%2)+
B:          SOB     %3,LOOP
            MOV     SAVER1,%1
            MOV     SAVER2,%2
            MOV     SAVER3,%3
            ...
SAVER1:     .BLKW   1
SAVER2:     .BLKW   1
SAVER3:     .BLKW   1
```

The locations SAVER1, SAVER2, SAVER3 will be reserved at assembly time, and those three words will be allocated to your program for the duration of its execution, in spite of the fact that you may be using them for only a brief period of time.

The allocation of storage, when it occurs at assembly time, is called *static* memory allocation. Static allocation remains in effect for the duration of a program's execution.

You could avoid tying up these three words of memory indefinitely by using the system stack instead.

Consider the following :

```
        MOV     %1,-(SP)
        MOV     %2,-(SP)
        MOV     %3,-(SP)
A:      MOV     #ARRA,%1
        MOV     #ARRB,%2
        MOV     #50.,%3
LOOP:   MOV     (%1)+,(%2)+
B:      SOB     %3,LOOP
        MOV     (SP)+,%3
        MOV     (SP)+,%2
        MOV     (SP)+,%1
        ...
```

Since the registers were pushed onto the stack in the sequence 1, 2, and 3, they must be popped off in the reverse order. First 3, then 2, then 1. This illustrates the LIFO characteristic of a stack. The last-in item *must* be the first-out item.

If it is necessary to use a larger number of words for a short period of time, you can "borrow" the space from the system stack. If you needed five words, for instance:

```
A:    MOV    SP,%1 ; save SP
B:    SUB    #10.,SP ; push 5 empty wrds in st
      ...
      MOV    X1,-2(%1)
      MOV    X2,-4(%1)
      ...
C:    MOV    X5,-10.(%1)
      ...
```

The stack history is shown in figure 8.13. Your program and the system can continue using the stack as if nothing had happened. When your program no longer needs the space in the stack *and* the SP is pointing just beyond the fifth word (i.e., the stack is otherwise empty), then you can "release" the space by, in effect, popping off five words by executing an ADD #10., SP, which takes us back to the stack configuration in effect "before A."

In this example, %1 has not been changed, so you could restore the SP by using a MOV %1, SP. Note that according to rule 3, you can expect those five words to "vanish." If you change the SP by 10 prematurely, you will have thrown away the wrong items!

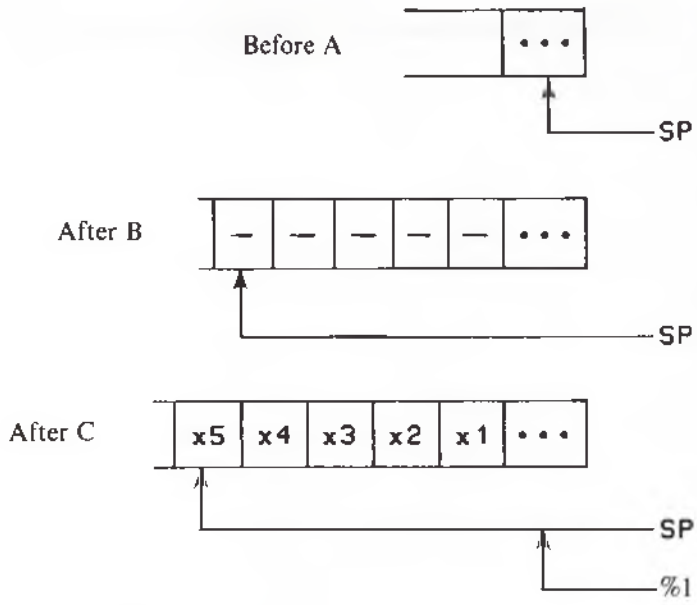


Figure 8.13 Borrowing space from the stack.

Subroutines and Stacks

The mechanism we discussed earlier for supporting return of control from a subroutine is clumsy. It wastes space, because a word has to be set aside in each subroutine to save the return address. It wastes more space and time because the user of the subroutine needs to write a "MOV #RA, sub" to provide the return address "RA."

JSR PC,sub; RTS PC

The PDP-11 eliminates this waste of time and memory space by providing a pair of instructions to support subroutine *linkage*, the mechanism which permits you to invoke a subroutine, and which provides the subroutine with the return address so it can return control to the point immediately following the subroutine's invocation. The first of these instructions is called the JSR. One common form of its use is:

```
JSR PC, sub
```

The execution of JSR PC, sub has the following consequences:

1. The PC is saved in the system stack: (PC) $\rightarrow$ stack;

   (x $\rightarrow$ stack means "push" x into the stack).

2. The effective address resulting from sub (which may use the addressing modes we have previously discussed) is copied into the PC.

So the single instruction `JSR PC, ABC` seems equivalent to

```
MOV     PC,-(SP)
JMP     ABC
```

This is almost correct (the PC value saved by the MOV is not quite the desired address). The other instruction needed for subroutine linkage is the RTS. One common form of its use is:

```
RTS     PC
```

The instruction `RTS PC` pops the stack into the PC: stack $\rightarrow$PC; (stack $\rightarrow$ x means "pop" the stack into x). We can demonstrate how these instructions work by looking at a simple subroutine, one which multiplies its argument by 5.

```
MUL5:
        MOV     %1,%2     ; save original value
        ADD     %1,%1     ; 2X
        ADD     %1,%1     ; 4X
        ADD     %2,%1     ; 5X
        RTS     PC        ; leave MUL5
        ...
A1:     MOV               A,%1
A2:     JSR               PC,MUL5
A3:     ...
        ...
B1:     MOV               B,%1
B2:     JSR               PC,MUL5
B3:     ...
```

If we begin execution of the above program segment at location A1, it places a value which is to be multiplied by 5 in %1, the address in the PC (which now has the value A3) is saved (by virtue of the JSR at A2) by copying it into the system stack (with a push, automatically). We now enter subroutine MUL5. As we leave MUL5, upon reaching the RTS PC, the top of the stack still holds the address A3. It is popped off the stack and copied into the PC, by the RTS PC. This gets us back to location A3.

Similarly, the JSR PC, MUL5 at B2 will push the content of the PC into the stack. This will be the value "B3," which is the desired return address. After entering MUL5, we quickly reach the instruction RTS PC. This time, since the top of the stack value is B3, it is popped into the PC, effecting the desired return.

What if register 2 was not known to be free at all the times MUL5 might be used? We can have a philosophical discussion about the responsibilities of the user of a subroutine to be informed thoroughly about which registers a subroutine uses, and to prepare himself accordingly. You will always have to know what the effects of a subroutine are before you can use it, but it is generally accepted practice that a subroutine should have no unnecessary side effects (such as "destroying" a register). If a subroutine is specified as requiring an input value in %1, and producing a result in %2, then it should not alter the other registers without advance notice.

Reconsidering our previous example, we should change MUL5 to:

```
MUL5:   MOV     %2,TMP
        ADD     %1,%1
        ... ; as before
        MOV     TMP,%2
        RTS     PC
TMP:    .BLKW   1
```

The invocations of MUL5 are not affected. The careful reader will object, noting that the word at TMP is not necessary.

Why not use the stack fully?

```
MUL5:   MOV     %2,-(SP)
        MOV     %1,%2
        ADD     %1,%1 ; 2X
        ADD     %1,%1 ; 4X
        ADD     %2,%1 ; 5X
        MOV     (SP)+,%2 ; restore %2
        RTS     PC
```

The very careful reader will think: "Why use %2?" Once again:

```
MUL5:   MOV     %1,-(SP)
        ADD     %1,%1 ;
        ADD     %1,%1
        ADD     (SP)+,%1 ; 5X
        RTS     PC
```

We can in a single instruction both add and pop. Keep in mind that when a subroutine uses the stack, it is responsible for restoring the stack to its original configuration. Otherwise the RTS will not find the desired return address where it should be—namely, at the top of the stack.

Passing Parameters

You can look at a subroutine as if it were a black box (BB) which performs a specified function without its user being aware of its inner mechanism, only of its result. The black box is given some inputs, called its *input parameters*, or *input arguments*. It produces results, called outputs, often called *output arguments*. A subroutine becomes a general-purpose subroutine to the extent to which it avoids making any symbolic references to instructions or data within the calling routine, and also to the extent that its function is of use to other problem solvers.

How do we communicate the inputs to a subroutine, and how does it make its result(s) available to the calling routine?

Passing Values as Parameters

In the MUL5 examples, we placed the *value* we wished to have multiplied in %1, and when the subroutine returns, we look in %1 to find the *value* of the result.

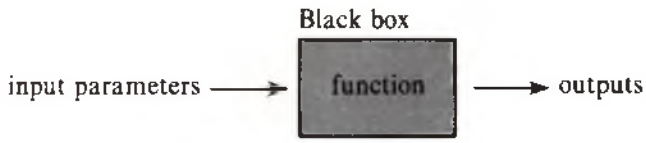


Figure 8.14 A subroutine as a black box.

Many of the mathematical functions provided in some high level languages can be supported by this technique. For instance, the function MAX, which has three inputs, can work nicely with only the value of its arguments, and it can return a value as the result. In a high level language, we might write

```
M = MAX(A,B,C), or
BIG := MAX(X,Y,Z)
```

or something similar. In MACRO-11, the first of these could be written:

```
        MOV     A,%1
        MOV     B,%2
        MOV     C,%3
        JSR     PC,MAX
        MOV     %1,M
        ...
MAX:    CMP     %1,%2   ; find maximum of %1 %2 %3
        BGE     M1      ; place max in %1
        CMP     %2,%3   ; %1 must be low
        BGE     M2
M3:     MOV     %3,%1   ; %3 is max
        RTS     PC
M1:     CMP     %1,%3
        BLT     M3
        RTS     PC      ; %1 is max
M2:     MOV     %2,%1
        RTS     PC      ; %2 is max
```

The name *function* is sometimes used to describe a subroutine which returns a single value, as in this example.

Passing Addresses as Parameters

It is not always practical to pass values as parameters. Suppose you want to use a subroutine which sums arrays. You do not want to have to copy the whole array just to have it summed. Consider how we could write a subroutine to find the total of any array of words whose address is placed in %0, and whose size (number of words) is placed in %1:

```
Invocation
        MOV     #A,%0       ; array address
        MOV     #50.,%1     ; array size
        JSR     PC,SUMIT
        MOV     %2,TOTAL    ; sum
        ...
A:      .BLKW   50.
```

```
Subroutine
SUMIT:  CLR     %2
S1:     ADD     (%0)+,%2
        SOB     %1,S1
        RTS     PC
```

The distinction between arguments passed as values and those passed as addresses is a very important one. Remember how important the difference between a $\longrightarrow$ and (a) $\longrightarrow$ has been?

If you want a subroutine to perform some action in "your" memory space rather than in "its" memory space, then you have to provide the subroutine with addresses, not values. The classical problem in high level languages which is used to illustrate the difference between "call by value" and "call by address" (also referred to as "call by reference") is the SWITCH(A,B) subroutine, which will either interchange the values associated with locations A and B, or it will not, depending on whether A and B are passed by value or by address.

More General Parameter Passing Techniques

If we had a long list of things to pass to a subroutine, we could do just that: give it a list of things. Consider:

```
        MOV     #LIST,%0
        JSR     PC,SUB
        ...
SUB:    MOV     (%0)+,%1 ; get 1st arg
        MOV     (%0)+,%2 ; get 2nd arg
        ... ; compute
        RTS     PC
        ...
LIST:   .WORD   A
        .WORD   12
        ...
```

We could use several lists if we wanted to. The lists of addresses or values can be placed in any convenient location.

Let us see what a summing subroutine looks like if we use an argument list to pass parameters:

```
        MOV     #L,%0
        JSR     PC,SUMX
        ...
  SIZE = 50.
L:      .WORD   ARR,SIZE
ARR:    .BLKW   SIZE
        ...
SUMX:   MOV     (%0)+,%1 ; adr of array
        MOV     (%0)+,%2 ; length
        CLR     %3 ; running total
SUMXL:  ADD     (%1)+,%3
        SOB     %2,SUMXL
        RTS     PC
```

What if the list L had been the following?

```
L:      .WORD   ARR,ARRLN
ARRLN:  .WORD   SIZE
```

Then SUMX would have to be changed to be:

```
SUMX:   MOV     (%0)+,%1 ; adr of array
        MOV     @(%0)+,%2 ; length
        ...     as before
```

We finally had an opportunity to use the auto-increment deferred addressing mode!

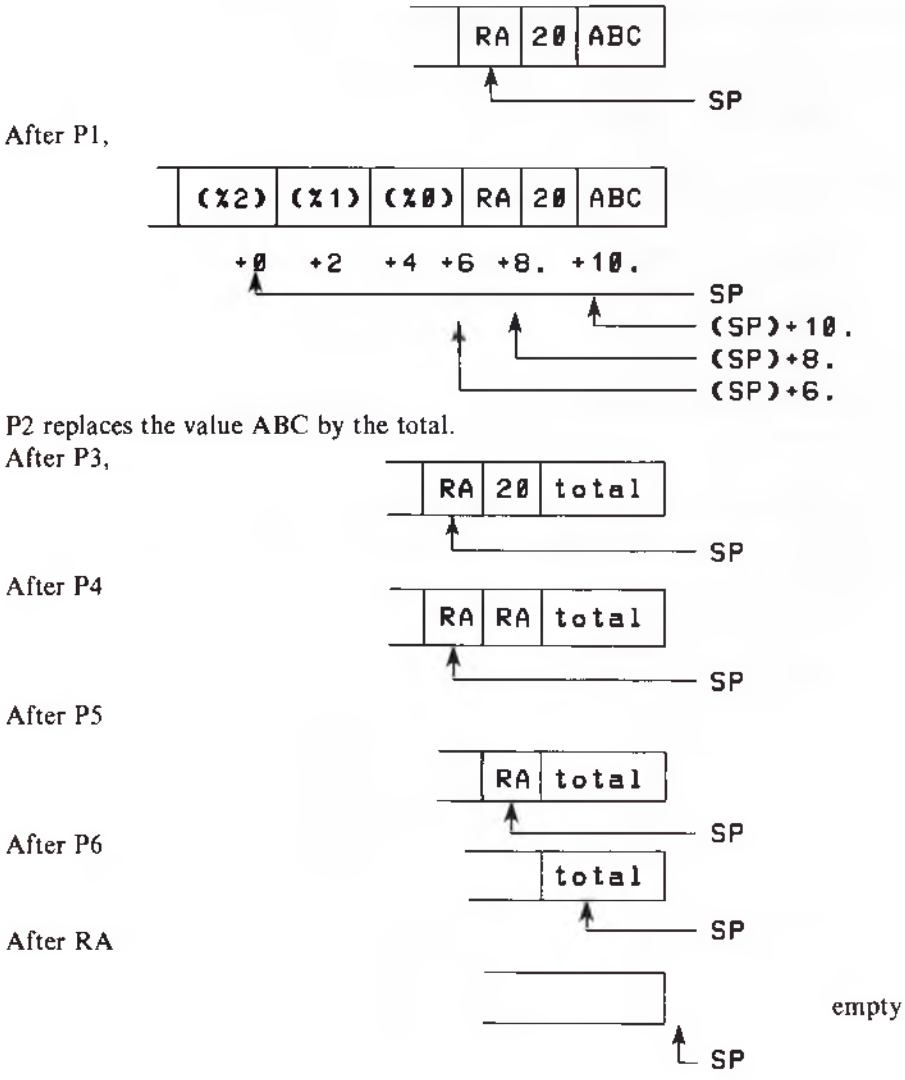
Passing Arguments in the Stack

You can also have the calling routine place its arguments in the stack instead of using registers or an argument list. Consider yet another summing program:

```
        SABC = 20
ABC:    .BLKW   SABC
        ...
        MOV     #ABC,-(SP)
        MOV     #SABC,-(SP)
        JSR     PC,SUMSTRA:
        MOV     (SP)+,RESULT
        ...
SUMST:  MOV     %0,-(SP)        ; save regs
        MOV     %1,-(SP)
P1:     MOV     %2,-(SP)
        MOV     10.(SP),%0      ; get base adr
        MOV     8.(SP),%1       ; get count
        CLR     %2              ; running total
SUMSTL: ADD     (%0)+,%2
        SOB     %1,SUMSTL
P2:     MOV     %2,10.(SP)      ; copy res over base adr
        MOV     (SP)+,%2
        MOV     (SP)+,%1
P3:     MOV     (SP)+,%0
P4:     MOV     (SP),2(SP)      ; cpy rt adr ovr cnt
P5:     TST     (SP)+           ; pop stack
P6:     RTS     PC
```

The stack history is depicted in figure 8.15.

Figure 8.15 Passing arguments in the stack.



JSR R,sub; RTS R

So far only the PC and the stack have been involved in subroutine linkage. We will be considering the case where a general register (one of %0–%5) is used instead of the PC as the linkage register R. First let us note that if it is necessary or advantageous to have a parameter list isolated from the subroutine invocation, then you can use the technique we just described. A subroutine invocation is also spoken of as a subroutine call, in deference to FORTRAN usage, where a subroutine named ABC is invoked by the verb CALL, as in CALL ABC(X,Y).

In most instances there is no advantage to isolating the argument list from the subroutine invocation, so a more general form of the JSR/RTS combination is provided to accommodate this situation. A JSR R, sub where R is *not* the PC has the following effect:

1. (R) is saved in the stack.
2. (PC) is copied into R.
3. The effective address for sub is copied into the PC.

Whenever a subroutine is called by this more general form JSR R, sub, that subroutine *must* exit using RTS R; the same R must be used. The effect of an RTS R is:

1. Copy (R) into PC.
2. Pop stack into R.

The net effect of calling a subroutine with JSR R,sub and having the return effected by RTS R is that (R) does not change. The reason for using this more general form of linkage is that the so-called linkage register R is left pointing to the word immediately following the JSR R,sub. If you have your argument list in that position, then the subroutine can use R to pick successive arguments in the list, one after another.

When the subroutine has done its work, and it is about to return to its caller, it is the subroutine's responsibility to ensure that R holds the desired return address. Under normal circumstances, since the point of return immediately follows the argument list, the by-product of stepping through the argument list using auto-increment addressing with R is to leave R holding the correct return address. Since the RTS R will copy (R) into the PC, the return is effected. At the same time, the stack is popped into R, restoring the stack pointer to its former value. Consider the following example:

```
            ...
            TSIZE = 25.
TABLE:      .BLKW   TSIZE
A:          JSR     %4,SUMY
B:          .WORD   TSIZE,TABLE ; 1 val, 1 adr
C:          MOV     %2,TOTAL
            ...
SUMY:       MOV     (%4)+,%0    ; length
            MOV     (%4)+,%1    ; base adr
            CLR     %2          ; running total
SUMYL:      ADD     (%1)+,%2
            SOB     %0,SUMYL
            RTS     %4
```

The JSR %4,SUMY results in the following:

%4	PC	Stack
B	SUMY	(%4) ...
		SP

As SUMY does its work, it auto-increments %4 twice, leaving it with the value B+4, which is the desired return address C. When SUMY exits with the RTS %4, we get C → PC, Stack → %4. This leaves us with %4 as it was originally, and the caller is back at C, storing its result in TOTAL.

Which Linkage Register Should I Use?

We used register 4 as the linkage register in the previous example. Since correct use of JSR R,sub and RTS R leaves R unaffected, you can use any R other than %7 (if you want to have an argument list pointer). There is no point in using %6 as the linkage register, since it is in use as a stack pointer. We could use any of %0 through %5 as the linkage register. Since none is any better or worse for the job, and it is annoying to have to choose one each time, the custom has evolved that %5 will be used as the linkage register. The hardware is not predisposed toward %5 as a linkage register, as it is towards %6 as a stack pointer; merely by convention is the reason. Henceforth when using argument lists, we will adopt %5 as the linkage register. You can still use %5 between subroutine calls, because it is preserved by the subroutine linkage mechanism.

Suggestions for Subroutine Design

A subroutine should be implemented as a black box. We discussed this earlier, but now that we know more about subroutine linkage, it bears repeating. The *only* things a subroutine should know about its caller are the input and output arguments. The very important consequence of this requirement is that, since a subroutine has *no* knowledge of any of the symbolic names used by the calling program, the subroutine can be written and assembled *independently* of the calling program.

The possibility of assembling subroutines independently of the calling routines, or of any other subroutines which the subroutine in question calls, is a significant breakthrough. Among other things, it means the many modules that a large application is decomposed into can be designed and implemented simultaneously and independently, subject to appropriate coordination.

We are now able to define the function of a desired subroutine, document it, test the code, fix the code, optimize it, then file the relocatable module in a *subroutine library*. Then any new problem can use the subroutine library as a source of modules, as building-blocks, to solve the subproblems into which the new problem has been decomposed.

Internal versus External Subroutines

If you have a "small" problem, you may be able to solve it by writing one source module. It may still be convenient for you to prepare and invoke a subroutine written as a part of this module. Such a subroutine is called an internal (or local) subroutine, because it is used only within the source module of which it is a part. Its name and existence are known to no other modules.

So far as the assembler is concerned, it has no knowledge that part of a module is an internal subroutine.

For instance, it may be helpful to have a multiply-by-4 subroutine because such an action may be required several times in a given application. Since we are assuming that this is a small application, let us define and use this subroutine internally:

```
        ...
        MOV     X,%1
        JSR     PC,MUL4
        MOV     %1,PROD
        ...
MUL4:
        ADD     %1,%1    : 2X
        ADD     %1,%1    ; 4X
        RTS     PC
```

It is of course assumed that the only way to reach MUL4 is via a JSR. The ... preceding the MUL4 line could be for a .BLKW or .WORD. If it is used for any instruction, it must be a JMP, BR or a return-to-system code.

.GLOBL

If you have a larger problem, you may want to write and assemble some subroutines independently of the main program. Or you may wish to have access to an existing library of subroutines you want to invoke. To do either of these, you have to use the MACRO-11 directive .GLOBL.

Normally when the assembler encounters a user-defined symbol, it is assumed to be of only local interest; that is, of interest only to other parts of the same source module. The mechanism for telling other modules that the ABC defined in this module can be reached from other modules is provided by the .GLOBL directive. Similarly, the mechanism for saying "I need to use an externally defined subroutine whose name is HELP" is also provided by the same directive. The .GLOBL performs two distinct functions. One of them is to pass on requests from the current module—requests for definitions of names of subroutines and/or data items which this module needs but which are not defined within this module. For instance:

```
        .GLOBL  HELP,YEAR
ST:     JSR     %7,HELP
        HALT
YR:     .WORD   YEAR
        .END    ST
```

This program has two undefined symbols: HELP and YEAR. The .GLOBL directs the assembler to remember these two names so the linker can be asked (at link time) to find definitions for each of them. The name YR will not be saved by the assembler beyond assembly time, since it is purely of local interest. The second function embodied by the .GLOBL is the act of informing the linker that *this* module has the definition for the name other modules may be looking for. To continue our previous example, you could write:

```
        .GLOBL  HELP,YEAR
YEAR:   .WORD   1984.
HELP:   NOP     ; helpless
        RTS     %7
        .END
```

Except for the presence of the .GLOBL, the text of an externally defined subroutine such as the HELP subroutine is no different from an internal subroutine. Since, as a general rule, a general-purpose internal subroutine should not make any symbolic references to its calling routines, a general-purpose internal subroutine can trivially be converted into an external subroutine, should it be desirable.

Linking Externally Defined Subroutines

When you are using a system-defined subroutine, you still need a .GLOBL to indicate that your module needs an externally defined subroutine, as in:

```
.GLOBL   DUMP
...
JSR      %5,DUMP
...
```

The linker will automatically search the system subroutine library in an attempt to satisfy your request, when you invoke the linker.

What if you wanted to fetch your own externally defined subroutine—say, one called FIX? The source module which needs it would, of course, have the following statements in it:

```
.GLOBL   FIX
...
JSR      PC,FIX
...
```

Suppose the object module file name corresponding to this source module is MAIN, and the object module name for FIX is FIXM. We can use the RT-11 linker to construct an executable program from MAIN and FIXM, as follows:

```
LINKER
*PROG=MAIN,FIXM,LIB
*^Z
```

The linker brings in your MAIN code, then it satisfies MAIN's need for FIX by bringing FIXM and filling in the address used by the JSR in MAIN. If MAIN or FIXM have any unsatisfied .GLOBL requests, the linker will next search for them in the file LIB. Because the linker uses the first definition it encounters, you can provide your own definitions in place of those provided in the system subroutine library, if you desire.

Transparency

When you invoke a subroutine, you may have certain expectations. You don't want any surprises. A subroutine is "surpriseless," or *transparent* to the calling routine to the extent that it is "invisible" to the calling routine. It has no effects other than those directly expected by the caller. If a subroutine is not completely transparent, then its description must specify all the other effects which it has. Unspecified side effects of a subroutine are similar to time bombs; sooner or later they will cause trouble.

Other than using the parameter passing techniques we have described, can a caller and a subroutine exchange information in any other way? The JSR and the RTS have no effect on the CC bits. This makes it possible to use the CC bits as a special way of communicating information to or from subroutines. It is fairly common for a subroutine to return information using the CC bits. Sometimes this happens as a natural by-product. Sometimes it is explicit. If things have gone well, a subroutine might by agreement clear the V bit; if things went wrong, it would set the V bit. So the calling routine might appear as follows:

```
...
JSR     %5,SOLVE
.WORD   A,X,Y ; arg list
BVS     ERROR ; no solution found
...
```

Subroutine Nesting

At the beginning of the chapter we discussed problem decomposition, a process which might lead to many levels of decomposition. If we think of the name P as representing the main program, and P1, P2 as subprograms called by P, then P11 and P12 could be the subprograms called by P1, and so on, as shown in figure 8.16.

```
        P                P1                P2
        --               --                --
        ...              ...               ...
A1:     JSR PC,P1
        ...              B1: JSR PC,P11
        ...              ...
A2:     JSR PC,P2        ...
        ...              B2: JSR PC,P12
        ...              ...
                         RTS PC
```

Figure 8.16 Problem decomposition by subroutines.

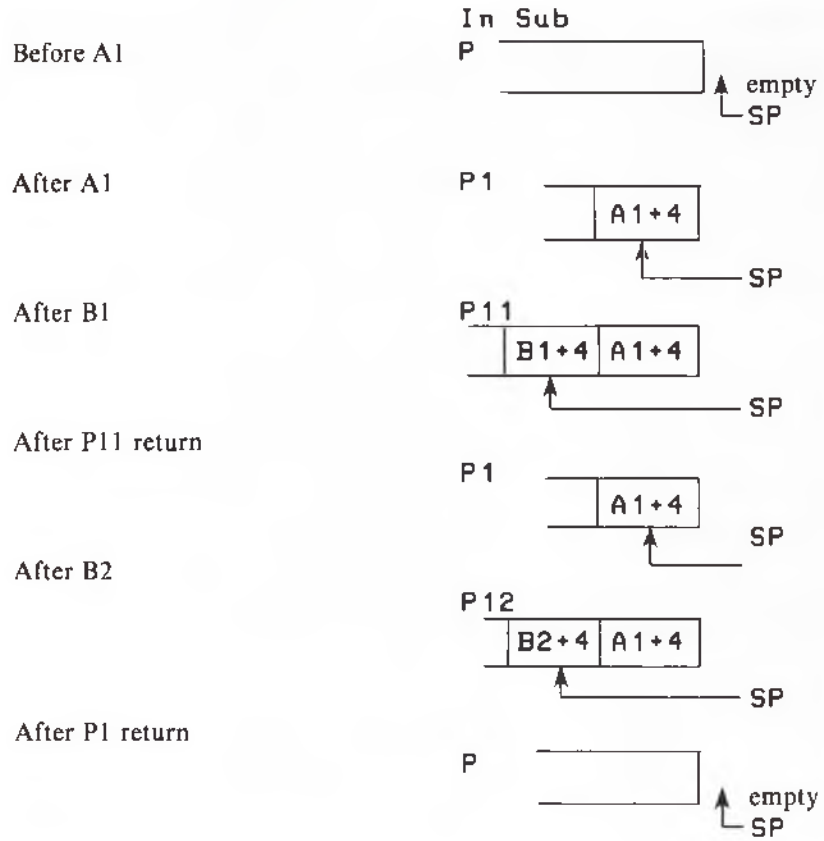


Figure 8.17 Stack history for nested subroutines.

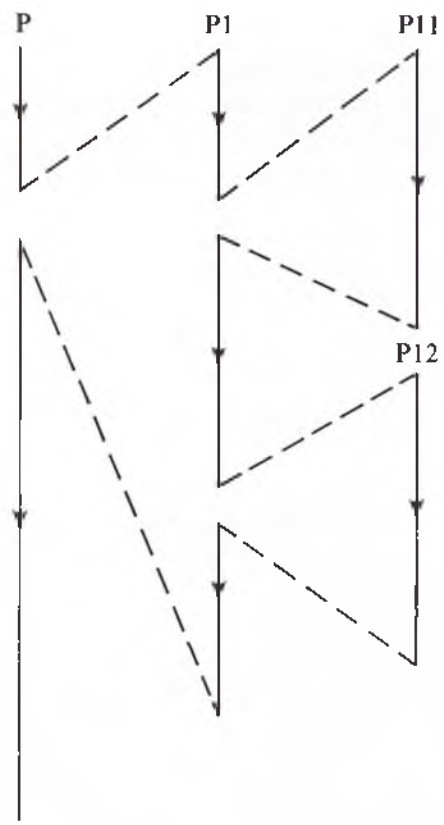


Figure 8.18 Execution chronology.

The stack history (with no regard to other uses of the stack) is shown in figure 8.17. Schematically, we can follow the chronology of execution, as depicted in figure 8.18.

Let us examine a simple example. You want to write a program which reads numeric data from a file and adds these items. Suppose each record has the form dd, dd , . . ., dd where dd are pairs of decimal digits. You are provided with a library subroutine called "reader." The program you write might be:

```
        .GLOBL  READER ; external sub
READ:   JSR     %5,READER ; copy line to REC
        .WORD   REC,EOF ; if none,→EOF
        MOV     #REC,%0
NEXT:   JSR     %5,GETNUM
        BMI     READ ; done if N=1
        ADD     %1,SUM
        BR      NEXT
EOF:            <output result and exit>
REC:    .BLKB   81.
        .EVEN
SUM:    .WORD   0
;
GETNUM: TSTB    (%0)            ; quit on 0 byte
        BNE     XX
        SEN                     ; set N
        RTS     %5
XX:     MOVB    (%0)+,%1        ; get 1st d
        JSR     %7,TENS
        MOVB    (%0)+,%1
        JSR     %7,ONES
        ADD     %2,%1
        CLN ; N=0
        RTS     %7
TENS:   SUB     #60,%1          ; reduce to binary
        ADD     %1,%1
        MOV     TENTBL(%1),%2
        RTS     %7
TENTBL: .WORD 0,10.,20.,30.,40.,50.,60.,70.
        .WORD 80.,90.
ONES:   SUB     #60,%1
        RTS     %7
        .END    READ
```

We began by writing the main program. As we reached NEXT, it seemed like a good idea to let a subroutine handle the details of processing independent numbers. Later, as we wrote GETNUM, it seemed like a good idea to let some new subroutines (TENS, ONES) take care of even lower level details. Afterward when reviewing our efforts, we may well decide to eliminate the subroutine ONES, and incorporate its one productive instruction where it is needed.

There is nothing wrong with admitting that some subroutine turned out to be unnecessary. It is not always clear before the fact how things will turn out. Use the concept of subroutines; sketch your program; test it; fix it. When it works correctly, you can see how it can be improved. There is no point in speeding up or otherwise improving a program that does not work.

What Can Go Wrong?

Managing a stack introduces several interesting possibilities for getting into trouble.

Stack Overflow

When using a stack, you have to take care not to push too many things into it. Its depth is finite. In a one-user system, the stack usually starts at 776 octal and goes down to 400 octal. If you push one thing too many into the stack, causing the stack pointer to auto-decrement from 400 to 376, you will get an M-TRAP TO 4 on a single-user PDP-11/34. The same thing done on an LSI-11 or PDP-11/03 would not be detected and acted upon by the control unit. On these systems, you might continue pushing things into the stack well below octal 400, until you destroyed some vital system information.

In a multi-user system (a PDP-11/34 or other PDP-11 with a memory management unit) the operating system sets a stack limit register, which a mere user can't see or touch. If, in using the SP, you force it below the preset stack limit, you might get an M-TRAP-TO 4. It depends on the operating system in use. Some will abort your program; others will try to find more space and dynamically extend the stack, increasing the stack limit register.

Stack Underflow

Stack underflow is the converse of the stack overflow problem. If you somehow lose track of how many things you pushed into the stack, you might begin popping more out than you pushed in. On a single-user system, you will begin using locations 1000, 1002, . . . as if they had been part of your stack. The hardware will not detect this problem. You will discover it the hard way.

On a multi-user system, popping too many things off leads to the SP having a value which is lower than that which the stack limit register will allow. The hardware will detect this and abort your program with an M-TRAP TO 4.

Making the Stack Deeper

On a single-user system you expect the SP to be set at 1000 when the operating system initiates execution of your program. The stack is considered empty at that instant. If you need a stack deeper than 128 words (400 to 776), you can do the following:

```
        .BLKW   50.         ; 1st line of prog
START:  MOV     #START,%6
        ...
        .END    START
```

The stack is now 50. words deeper than it used to be. You can do this safely only when the stack is empty.

On a multi-user system, since the stack limit register is "off-limits" to nonprivileged users, you would have to invoke a system service request specific to the operating system you are using in order to request a bigger stack, unless the operating system is clever enough to keep enlarging the stack (within reason) as you put more and more things in it.

Summary

The ability to have a sequence of instructions used from various points in a program, to accomplish a specific function, is very important. As we first examined this idea—called a subprogram, subroutine, function, or procedure—it can be implemented without any new instructions. It is such an important idea that on most computers special instructions are provided to support it efficiently. The PDP-11 has the JSR and RTS instructions to provide for efficient subroutine calls and returns.

Indirect addressing is a very useful concept, supported on the PDP-11 with the deferred addressing modes. The addressing modes that implicitly use the PC are called PC addressing modes; the most frequently used is called PC relative addressing. It has the appearance of a direct memory address, but the assembler transforms it into a PC relative address.

The notion of a memory stack, accessed on a LIFO (last in, first out) basis, has been introduced. Stacks make it easy to reallocate memory dynamically. This is especially important for mini- and microcomputers.

On the PDP-11, a register is specifically assigned to point to the top of an area of memory designated as the system stack. Register 6, %6, the system stack pointer SP, should always have an even (word) address in it. The PDP-11 hardware expects the SP to always hold an address which points to the top of the system stack. Any program which violates this expectation is courting trouble. On the PDP-11, a push into the stack corresponds to auto-decrementing the stack pointer; thus the system stack grows from high memory into low memory.

Subroutine linkage can be implemented very nicely on computers which support stacks. The JSR PC, sub saves its return address in the stack, and subroutine arguments can be passed in the stack. The various ways of passing arguments and obtaining results have been described, including use of the general registers, use of argument lists, and use of the stack. In all cases, it is important to distinguish between passing addresses and passing values.

The ability to independently assemble modules complements the organizational advantages of subroutines. The .GLOBL directive serves the dual role of presenting requests for externally defined subroutines and also making the linker aware of the presence of subroutine definitions.

The use of stacks can lead to their misuse, leading to stack overflow or stack underflow. Since the manner in which these problems can be detected and reported depends upon the specific model of computer you are using, and on which operating system, if any, you are using, you will be well advised to determine how these problems are manifested in your particular situation.

Exercises for Chapter 8

8.1 If a program began with the following statements, what would the assembler generate as the third word of the first instruction?

```
        MOV #4,ABC
        HALT
ABC:    .WORD 2
```

Select one of the following:

0 ; 10 ; 2 ; 4

8.2 How many memory references are made in fetching and executing the following?

```
(a) MOV   ABC, (%0)      ; (itemize them)
(b) CMP #123,@(%1)+
```

8.3 The first word of a MOV instruction has the general form 01SSDD. Match the following machine-language instructions with their corresponding symbolic representations.

```
(a) 012304              (i)   MOV 2000(%3),2000(%4)
(b) 015344              (ii)  MOV (%3)+, %4
(c) 016374,2000,2000    (iii) MOV 2000,2000
(d) 012767,2000,772     (iv)  MOV 2000,2000
(e) 017737,772,2000     (v)   MOV -(%3),-(%4)
```

8.4 After the first word of the instruction `MOV 0(%0), (%1)` has been fetched, what does the PC (program counter) contain?

(a) The next instruction to be executed.
(b) The address of the next instruction to be executed.
(c) The address of the source operand.
(d) The address of the destination operand.
(e) An offset.
(f) The address of an offset.
(g) It cannot be determined from the given information.

8.5 How many memory references are made in fetching and executing the instruction `MOV #A, B`? (Assume that A and B are properly defined labels.) Describe each reference.

(a) 3 (b) 4 (c) 5 (d) 6 (e) 7

8.6 The following PDP-11 program is loaded starting at location 1000, with the stack pointer (SP) initially containing 1000. Draw a "snapshot" of the system stack after the execution of each of the following instructions.

```
                MOV     #1,-(SP)
                MOV     #2,-(SP)
                JSR     PC,SUB
X:              HALT

SUB:            MOV     (SP)+,RETAD
                MOV     (SP)+,JUNK1
                MOV     (SP)+,JUNK2
                MOV     RETAD,-(SP)
                RTS     PC

RETAD:          .BLKW 1
JUNK1:          .BLKW 1
JUNK2:          .BLKW 1
                .END 1000
```

8.7 What is the final value in %4 after executing the following?

```
MOV #1000, %4
MOV (%4)+, -(%4)
```

8.8 True or False?

(a) `.BYTE 101,102,103` and `.ASCII /ABC/` are equivalent.
(b) The loader initializes all registers to zero.
(c) The .END statement produces an executable run-time instruction.
(d) The .EVEN directive is required after all .ASCII directives.
(e) Arithmetic overflow always clears the V bit.
(f) Care must be taken to avoid having a conditional branch instruction affect some of the condition codes.
(g) The destination of a BR instruction cannot be more than 300 words decimal from the BR instruction.
(h) A BHI should not be used following comparisons using signed numbers.

8.9 In the worst case, how long does the following subroutine take? Assume that *each* memory reference (for fetching instructions, address, or data or storing operands) takes 1 μsec, and that instruction execution times are 100 nsec.

(a) How many memory references occur?
(b) How many instructions are executed?
(c) What is the total time taken?

```
MAX:            MOV R0,-(SP)
                MOV #LAST,R0
                MOV -(R0),BIG
```

```
NEXT:       CMP -(R0),BIG
            BLE DUN
            MOV (R0), BIG
DUN:        CMP R0, #VEC
            BHI NEXT
            MOV (SP)+, R0
            RTS PC
VEC:        .BLKW 50
LAST:
BIG:        .WORD 50
```

8.10 In the last example of this chapter, the statement

```
MOV #REC,%0
```

occurs. Beginning programmers are tempted to write

```
MOVB #REC,%0
```

instead, reasoning that REC is the address of a string of bytes, therefore they should use a MOVB, not a MOV. What is the error in this reasoning? What are the likely consequences of this error? In some cases it would be easy to detect; in others, it might be very difficult. Explain this carefully.

8.11 Let EACH instruction use the following initial conditions (that means the results of one instruction will NOT be used in any other instruction). Fill in the contents of the indicated locations AFTER the instruction has been executed. Write—if no change occurs.

Instruction	%2	%3	1050	1052
Initial Conditions	1050	1052	1052	35

(a) `MOV %2,%3`
(b) `CLR (%2)`
(c) `MOVB (%2)+,%3`
(d) `MOV -(%3),%2`
(e) `MOV @(%2)+,-(%3)`
(f) `CMP (%2)+,(%2)+`
(g) `MOV @-(%3),2(%2)`
(h) `ADD -2(%3),(%2)+`
(i) `MOV @-2(%3),(%2)+`
(j) `MOV #1050,%3`

8.12 Match each addressing mode syntax representation on the right with its correct name on the left and indicate how many memory references are needed to completely interpret this addressing mode (assume that the first word of the instruction has been fetched).

Memory Cycles

(a) relative	(i) @100
(b) auto-decrement deferred	(ii) %1
(c) index	(iii) @(%2)+
(d) auto-increment	(iv) @#173160
(e) register	(v) #1000
(f) index deferred	(vi) 200
(g) auto-decrement	(vii) @50(%0)
(h) auto-increment deferred	(viii) @-(%3)
(i) immediate	(ix) (%1)+
(j) relative deferred	(x). 100(%4)
	(xi). -(%5)

8.13 Consider the following subroutine invocation:

```
MOV #ARY1, -(%6)
MOV #ARY2, -(%6)
MOV #ARYSUM, -(%6)
JSR %7, ARYADD
ADD #6, %6
```

ARY1 and ARY2 are the beginning addresses of two word arrays of equal length. The end of each is marked by a sentinel value of 0 (the number, not the character). You are to write the subroutine ARYADD which will sum corresponding elements of ARY1 and ARY2 and place the results into the corresponding location of ARYSUM, which is also the beginning address of an array. Your subroutine must not have any side effects (e.g., changing registers), and the only information your subroutine has about the main program is that which is passed as arguments. Show clearly how the stack is used.

8.14 On some computers it is possible to execute instructions such as `BMI (%1).` Why is it not possible on the PDP-11? How could you obtain the desired effect on the PDP-11?

8.15 Write a program using correct MACRO-11 syntax which takes a string that ends in a zero byte and replaces all the spaces in it with minus signs. Thus the text in `.ASCIZ /A BB C/` becomes A−BB−C.

8.16 Given the following symbolic program, show the addresses assigned and the numeric code generated by the assembler:

```
L:      MOV A, %0
        BR L
A:      .WORD -1
        .END
```

8.17 The following is a code segment:

```
DATA1:      .WORD 17
DATA2:      .WORD 20
FI:         MOV     #7,%5
            MOV     #DATA1,ARG1
            MOV     #DATA2,ARG2
                                        ← (a)
            JSR     %5,FOO
ARG1:       .BLKW 1
ARG2:       .BLKW 1
                                        ← (b)
            HALT
FOO:        MOV     (%5)+,FIR
            MOV     (%5)+,SEC
                                        ← (c)
            MOV     @FIR,WHO
            MOV     @SEC,@FIR
                                        ← (d)
            MOV     WHO,@SEC
            RTS     %5
FIR:        .BLKW 1
SEC:        .BLKW 1
WHO:        .BLKW 1
            .END    FI
```

If this program is loaded starting at location 1000, give the values stored in DATA1, DATA2, ARG1, ARG2, %5, WHO, FIR and SEC when execution reaches the points labeled by (a), (b), (c) and (d).

8.18 Examine the following program segment (assume that it is loaded at location 100):

```
D1:         .WORD   D2
D2:         .WORD   D1
GO          MOV     #D1,A1
            MOV     #D2,A2
            JSR     %4,S1
A1:         .BLKW   1
A2:         .BLKW   1
B:          HALT
S1:         MOV     (%4)+,S11
            MOV     (%4)+,S12
```

```
A:          MOV     @S11,S13
            MOV     @S13,S12
            RTS     %4
S11:        .BLKW   1
S12:        .BLKW   1
S13:        .BLKW   1
            END     GO
```

(a) What are the values stored in the following locations when execution just reaches the indicated points (use ? for "unknown"). You may use numeric or equivalent symbolic addresses.

	D1	D2	A1	A2	S11	S12	S13	%4
A								
B								

(b) Rewrite S1 so that it can be invoked using `JSR %7,S1`, making as few changes as possible in all of the above code.

8.19 Write a single PDP-11 instruction which uses five memory references in the process of fetching and executing it.

8.20 Assume that the following program is assembled and loaded beginning at memory location 0. Write down the contents (in octal) of memory locations 10 and 12 immediately before program execution and immediately after program execution.

```
ST :        MOV #14, @A
            HALT
A :         .WORD B
B :         .WORD A
            .END ST
```

8.21 Find the errors the assembler should detect in this code:

```
A:          .BYTE 18
START       MOV A, R5
            JNQ END
            TEST Q
            R5 = %5
            BGZ START
DONE:
   Q:       .WORD START
            END START
```

8.22 How many memory references are made in fetching and executing CMP ABC(%3),@X? Describe each reference.

8.23 When a program aborts, you may get to look at the registers and some memory if you were using a .PMD. Similarly, a .SNAP displays the registers. What could you do so you could also see what is in the stack?

8.24 Given the following as initial conditions *before each part* of this problem, fill in the contents of each specified register or memory location after execution of each indicated instruction:

	Loc 2000	Loc 3000	Loc 4040	%1	%2
Initial values	4040	6001	501	2000	3000

(a) `MOV (%1),(%2)`
(b) `CMPB (%1)+,(%2)`
(c) `ADD %1,@(%2)`
(d) `SUB 1000(%1),(%2)`

8.25 Write a subroutine which passes the address of a character string in R0 and returns with the same string (in the same place) having all contiguous groups of the same character replaced by one occurrence of that character (thus OCCURRENNNNNCE would come back OCURENCE). The string is terminated by an actual zero (0), not the code for the ASCII character zero. You may assume that the string is nonempty (so the first character in the string is not 0), and you may ignore the saving of registers. This segment can be written with nine instructions.

8.26 The code for MULR in section 1 returned control using a `JMP 0(%2)`. Find a more appropriate addressing mode which still uses register 2.

8.27 Consider the following program excerpts:

```
START:    MOV   #13, %5
CALL:     JSR   %5, A
          ...
A:        JSR   %5, B
          ...
          RTS   %5
B:        ...
B1:       JSR   %5,C
          ...
          RTS   %5
C:        ...
          RTS   %5
```

Assuming the stack is used only for subroutine linkage, and the indicated program flow beginning at START is not misleading, write down the exact content of the stack on entering subroutine C.

8.28 True or False?

(a) Uninitialized memory locations are assumed to have the value 0.
(b) A `MOV #10,%1` uses two memory references during its fetch-execute cycle.
(c) Some conditional branch instructions can modify the CC.
(d) `DEC %3` and `SUB #1,%3` have the same effect.

8.29 Indicate the addressing mode(s) which can be used when a register functions as:

(a) an accumulator (b) a pointer (c) an index

8.30 Write a subroutine that is given, in r0 and r1, the addresses of two strings of alphabetic characters (each string is terminated by an actual zero (0), not an ASCII zero, and returns with the Z bit set if the two strings are equal and with the N bit set if the one pointed to by r0 alphabetically precedes the one pointed to by r1. Don't worry about saving registers. What problems would there be if you wanted to save registers?

8.31 Explain briefly the *differences* between the system stack and user-defined stacks. When should one be used rather than the other?

8.32 What will be in %1 and %2 after each instruction is executed? (Treat each instruction independently.)

location	%1	%2	40	100	200	300
content	100	100	3	200	40	50

(a) `MOVB @(%2)+,%1`
(b) `SUB (%2),%1`
(c) `CMP @100(%2),%1`

8.33 Explain the difference at assembly time and execution time between these two sets of instructions:

(a)
```
NEAR=100
FAR = 200
MOV @#NEAR,@#FAR
MOV NEAR,FAR
```

(b)
```
MOV 100,200
MOV @#100,@#200
```

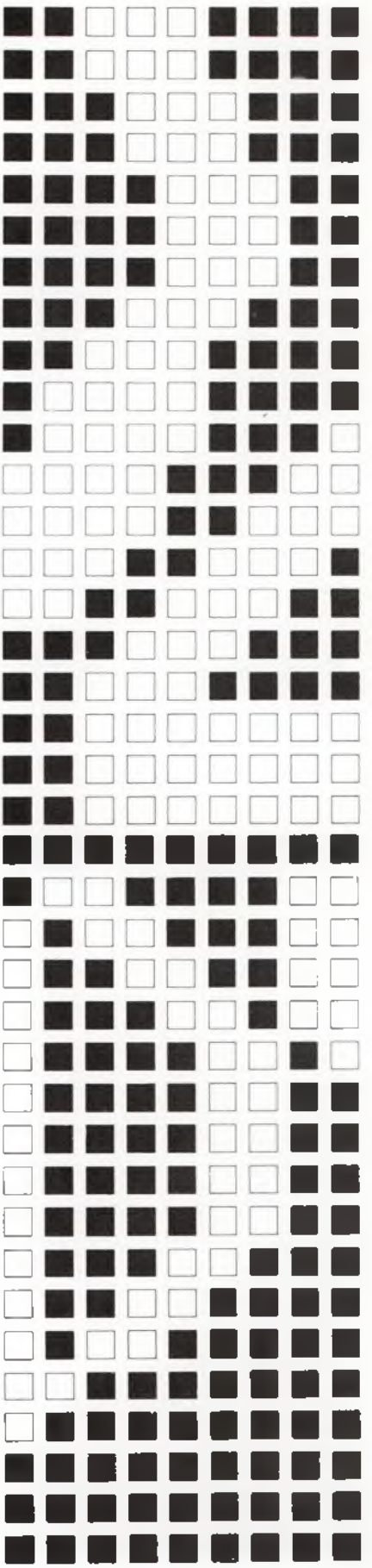
8.34 For each of the following instructions, assume the indicated initial conditions. Indicate all changes to these items produced by executing these instructions. The initial conditions apply to each instruction. Each one is assumed to begin at location 1000. Use—to indicate "no change."

Initial conditions	%1	%2	1200	1202	PC
	1200	1202	1202	1200	1000

(a) `ADD (%1)+,(%2)`
(b) `ADD #1200,%1`
(c) `CMP (%1)+,(%1)+`
(d) `SUB %1,-(%2)`
(e) `CLRB -(%2)`
(f) `062121`
(g) `060102`
(h) `105021`

9

the remaining general-purpose instructions



We will discuss the remaining general-purpose instructions of the PDP-11 in this section. Before doing so, we need to consolidate what we have seen. There are two organizing concepts that will be helpful. The first considers the instruction set classified according to format. The second groups instructions according to function.

Format Classification

Classifying instructions by format means identifying all the instructions that use the same encoding for their operand fields. Then if you can understand the possible variations and interpretations for one instruction from each group, it becomes relatively easy to learn all about the other instructions which share the same format.

The six different formats are shown in figure 9.1. The instructions we have seen, plus those we have yet to discuss, are shown in figure 9.2. The shortest opcode field of any format is that associated with the double-operand instructions; the opcode field is only 4 bits wide. Since the opcode fields for the entire instruction set vary in length from 4 to 13 bits (for group 6), how does the control unit know which format is involved when it copies an instruction into its IR?

If the octal digit encoded in bits 14–12 is one of 1–6, the format is double-operand. If the digit is 0, then it might be either the single-operand format or one of the branch instructions. These last two groups are differentiated by the bits in positions 11–10, and so on. We can see this more clearly by examining a table in which all the instructions have been sorted numerically (figure 9.3). The control unit can inspect the leading bits of the instruction in the IR, determine unambiguously which format applies, and then isolate the particular opcode and its operand fields.

Classification by Function

A functional classification groups instructions according to the nature of the work they perform. This is perhaps the most useful classification to have at hand when you are writing programs. It helps you decide things such as: "Which arithmetic instruction should I use here?" An alphabetic listing of instruction mnemonics may be helpful if you have forgotten what a particular instruction does, but it is very clumsy to use when writing programs.

We can classify the instructions into two major functional categories, as shown in figure 9.4. We then classify the PDP-11 instructions according to the minor functional categories. Figure 9.5 shows the data-oriented instructions; figure 9.6, the control-oriented instructions.

Placing some of these instructions in one group or another is somewhat arbitrary. You can see why the grouping by format has some advantage when you are first learning about these instructions. If we had proceeded on a functional basis, we would, for instance, have been looking at the double operand ADD, the single operand INC, and the special operand MUL (to be discussed soon) simultaneously. We can now proceed to discuss the instructions that are new to us and contrast them with the familiar ones.

The following legend describes the symbols and abbreviations used in the descriptions associated with each instruction.

(XXX)	The contents of XXX
Sa or SS	Source address
dst or DD	Destination address
∧	Boolean AND function
∨	Boolean OR function
≠, ∀	Boolean EXCLUSIVE OR function
~	Boolean NOT function (complement)
⟶	"becomes"
↑	"popped from stack"
↓	"pushed onto the stack"
R	Register
B	Byte
■	O for word, I for byte
X	Relative address
%	Register definition
Condition Codes- N,Z,V,C	* = Conditionally set or cleared
	— = not affected
	0 = Cleared
	1 = Set

Instruction Formats

The following formats include most instructions used in the PDP-11. Refer to individual instructions for more detailed information.

1. Single Operand Group CLR.CLRB.COM.COMB.INC. INCB.DEC.DECB.NEG. NEGB.ADC.ADCB.SBC. SBCB.TST.TSTB.ROR. RORB.ROL.ROLB.ASR. ASRB.ASL.ASLB.IMP. SWAB.MFPS.MTPS.SXT. XOR)

OP Code	DD
15 6	5 0

2. Double Operand Group (BU.BITB.BIC.BICB.BIS. BISB.ADD.SUB.MOV. MOVB.CMP.CMPB.)

OP Code	SS	DD
15 12	11 6	5 0

3. Program Control Group

a. Branch (all branch instructions)

OP Code	offset
15 8	7 0

b. Jump To Subroutine (JSR)

0	0	4	R	DD

c. Subroutine Return (RTS)

0	0	0	2	0	R

d. Traps (break point, IOT.EMT.TRAP.BPT)

OP CODE

e. Mark (MARK)

0	0	6	4	NN

f. Subtract I and branch (if = 0)(SOB)

0	0	7	R	NN

4. Operate Group (HALT,WAIT.RTI.RESET.RTT.NOP)

OP CODE

5. Condition Code Operators (all condition code instructions)

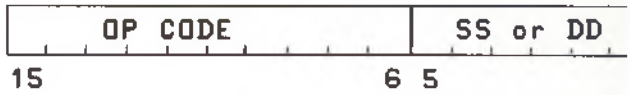
0	0	0	2	4	N	Z	V

6. Fixed and Floating Point Arithmetic (optional EIS/ FIS)(FADD. FSUB.FMUL. FDIV.MUL. DIV.ASH. ASHC)

OP CODE	R

Figure 9.1 PDP-11 instruction formats.

Single Operand



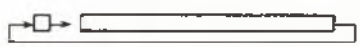
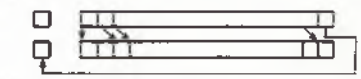
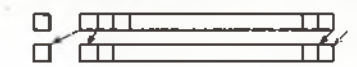
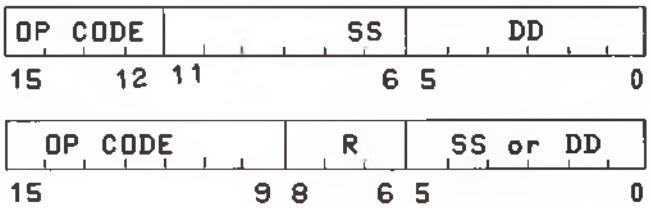
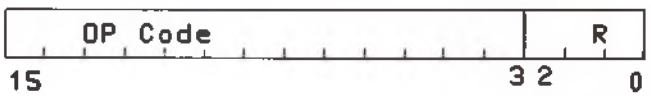
Type	Mnemonic	Octal Code	Instruction	dst Result	NZVC
General	CLR(B)	■050DD	clear destination	(dst)←0	0100
	COM(B)	■051DD	complement dst	(dst)←~(dst)	**01
	INC(B)	■052DD	increment dst	(dst)←(dst)+1	***-
	DEC(B)	■053DD	decrement dst	(dst)←(dst)-1	***-
	NEG(B)	■054DD	negate dst	(dst)←-(dst)	****
	TST(B)	■057DD	test dst	(dst)←(dst)	**00
Shift and Rotate	ROR(B)	■060DD	rotate right		****
	ROL(B)	■061DD	rotate left		****
	ASR(B)	■062DD	arithmetic shift right		****
	ASL(B)	■063DD	arithmetic shift left		****
Byte Manipulation	SWAB	0003DD	swap bytes		***0
Multiple Precision	ADC(B)	■055DD	add carry	(dst)←(dst)+(C)	****
	SBC(B)	■056DD	subtract carry	(dst)←(dst)-(C)	****
	SXT	0067DD	sign extend	(dst)←0 if N bit clears -1 if N bit set	-*0-
PS word operators	MFPS	1067DD	move byte from PS	(dst)←PS	**0-
	MTPS	1064SS	move byte to PS	PS←(src)	****

Figure 9.1 Continued.

Double Operand



Type	Mnemonic	Octal Code	Instruction	Operation	NZVC
	MOV(B)	■1SSDD	move source to destination	(dst)←(src)	**0-
General	CMP(B)	■2SSDD	compare source to destination	(src)-(dst)	****
	ADD	06SSDD	add source to destination	(dst)←(src)+(dst)	****
	SUB	16SSDD	subtract source from destination	(dst)←(dst)-(src)	****
	BIT(B)	■3SSDD	bit test	(src)∧(dst)	**0-
Logical	BIC(B)	■4SSDD	bit clear	(dst)←~(src)∧(dst)	**0-
	BIS(B)	■5SSDD	bit set	(dst)←(src)∨(dst)	**0-

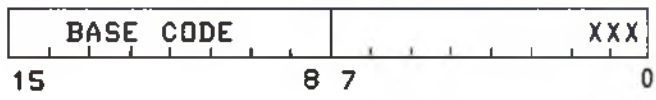


Type	Mnemonic	Octal Code	Instruction	Operation	NZVC
	MUL	070RSS	multiply	R,RV1←Rx(src)	**0*
	DIV	071RSS	divide	R,RV1←R,RV1/(src)	****
	ASH	072RSS	shift arithmetically	R←R(shifted NN places left or	****
Optional	ASHC	073RSS	arithmetic shift combined	right)	****
EIS/FIS	FADD	07500R	floating add	R,RV1←R,RV1	**00
	FSUB	07501R	floating subtract	Aarg←Aarg+Barg	**00
	FMUL	07502R	floating multiply	Aarg←Aarg-Barg	**00
	FDIV	07503R	floating divide	Aarg←AargxBarg	**00
				Aarg←Aarg/Barg	

Figure 9.1 Continued.

Branches

B.....location
Octal Code = Base Code+XXX



If condition is satisfied: Branch to location,
New PC←Updated PC+(2X offset)

address of branch instruction+2

Type	Mnemonic	Base Code	Instruction	Branch Condition
Branches	BR	000400	branch (unconditional)	(always)
	BNE	001000	branch if not equal (to 0)	Z=0
	BEQ	001400	branch if equal (to 0)	Z=1
	BPL	100000	branch if plus	N=0
	BMI	100400	branch if minus	N=1
	BVC	102000	branch if overflow is clear	V=0
	BVS	102400	branch if overflow is set	V=1
	BCC	103000	branch if carry is clear	C=0
	BCS	103400	branch if carry is set	C=1
Signed Conditional Branches	BGE	002000	branch if greater or equal (to 0)	N∀N=0
	BLT	002400	branch if less than (zero)	N∀V=1
	BGT	003000	branch if greater than (zero)	ZV(N∀V)=0
	BLE	003400	branch if less or equal (to 0)	ZV(N∀N)=1
Unsigned Conditional Branches	BHI	101000	branch if higher	CVZ=0
	BLOS	101400	branch if lower or same	CVZ=1
	BHIS	103000	branch if higher or same	C=0
	BLO	103400	branch if lower	C=1

Figure 9.1 Continued.

Type	Mnemonic	Octal Code	Instruction	Operation
Jump and Subroutine				*Notes*
	JMP	0001DD	jump	PC←dst
	JSR	004RDD	jump to subroutine	Use same R
	RTS	00020R	return from subroutine	
	SOB	007RNN	subtract 1 and branch (if=0)	(R)−1, then if (R)≠0: PC←updated PC−(2xNN)
Trap and Interrupt				
	EMT	104000/104377	emulator trap	PC at 30, PS at 32
	TRAP	104400/104777	trap	PC as 34, PS at 36
	BPT	000003	breakpoint trap	PC at 14, PS at 16
	RTI	000002	return from interrupt	
	RTT	000006	return from interrupt	inhibit T bit trap
	IOT	000004	input/output trap	PC at 20, PS at 22
Miscellaneous				
	HALT	000000	halt	
	WAIT	000001	wait for interrupt	
	RESET	000005	reset external bus	
	NOP	000240	(no operation)	

Type	Mnemonic	Octal Code	Instruction	NZVC
Condition Code Operators	CLC	000241	clear C	---0
	CLV	000242	clear V	--0-
	CLZ	000244	clear Z	-0--
	CLN	000250	clear N	0---
	CCC	000257	clear all CC bits	0000
	SEC	000261	set C	---1
	SEV	000262	set V	--1-
	SEZ	000264	set Z	-1--
	SEN	000270	set N	1---
	SCC	000277	set all CC bits	1111

Figure 9.2 PDP-11 instructions.

Numerical OP Code List

OP Code	*Mnemonic*	*OP Code*	*Mnemonic*	*OP Code*	*Mnemonic*
00 00 00	HALT	00 60 DD	ROR	10 40 00	EMT
00 00 01	WAIT	00 61 DD	ROL	↕	
00 00 02	RTI	00 62 DD	ASR	10 43 77	
00 00 03	BPT	00 63 DD	ASL		
00 00 04	IOT	00 64 NN	MARK		
00 00 05	RESET	00 65 SS	MFPI	10 44 00	TRAP
00 00 06	RTT	00 66 DD	MTPI	↓	
00 00 07	(unused)	00 67 DD	SXT	10 47 77	
00 00 77					
		00 70 00	(unused)	10 50 DD	CLRB
00 01 DD	JMP	↕		10 51 DD	COMB
00 02 0R	RTS	00 77 77		10 52 DD	INCB
				10 53 DD	DECB
00 02 10	(unused)	01 SS DD	MOV	10 54 DD	NEGB
↕		02 SS DD	CMP	10 55 DD	ADCB
00 02 27		03 SS DD	BIT	10 56 DD	SBCB
		04 SS DD	BIC	10 57 DD	TSTB
00 02 3N	SPL	05 SS DD	BIS		
00 02 40	NOP	06 SS DD	ADD	10 60 DD	RORB
				10 61 DD	ROLB
00 02 41	cond codes	07 0R SS	MUL	10 62 DD	ASRB
↕		07 1R SS	DIV	10 63 DD	ASLB
00 02 77		07 2R SS	ASH		
		07 3R SS	ASHC	10 64 00	(unused)
00 03 DD	SWAB	07 4R DD	XOR	↕	
00 04 XXX	BR	07 50 0R	FADD	10 64 77	
00 10 XXX	BNE	07 50 1R	FSUB		
00 14 XXX	BEQ	07 50 2R	FMUL	10 65 SS	MFPD
00 20 XXX	BGE	07 50 3R	FDIV	10 66 DD	MTPD
00 24 XXX	BLT				
00 30 XXX	BGT	07 50 40	(unused)	10 67 00	(unused)
00 34 XXX	BLE	↕		↕	
		07 67 77		10 77 77	
00 4R DD	JSR				
		07 7R NN	SOB	11 SS DD	MOVB
00 50 DD	CLR			12 SS DD	CMPB
00 51 DD	COM	10 00 XXX	BPL	13 SS DD	BITB
00 52 DD	INC	10 04 XXX	BMI	14 SS DD	BICB
00 53 DD	DEC	10 10 XXX	BHI	15 SS DD	BISB
00 54 DD	NEG	10 14 XXX	BLOS	16 SS DD	SUB
00 55 DD	ADC	10 20 XXX	BVC		
00 56 DD	SBC	10 24 XXX	BVS	17 00 00	floating point
00 57 DD	TST	10 30 XXX	BCC, BHIS	↕	
		10 34 XXX	BCS, BLO	17 77 77	

Figure 9.3 Instructions in numeric order.

Figure 9.4 Major functional categories.

Data oriented	**Control oriented**
arithmetic	branches
data handling	loop control
logical	subroutine linkage
test	input/output
miscellaneous	interrupts
	miscellaneous

Figure 9.5 Data-oriented instructions.

1. Arithmetic
   - 1.1 integer (single precision)
     ```
     ADD SUB ASH ASHC ASR ASL NEG INC DEC
     MUL DIV
     ```
   - 1.2 integer (multiple precision)
     ```
     ADC SBC SXT
     ```
   - 1.3 floating point
     varies according to model; discussed later
2. Data handling
   ```
   MOV SWAB CLR
   ```
3. Logical
   ```
   BIT BIC BIS ROR ROL COM XOR
   ```
4. Test
   ```
   CMP TST
   ```
5. Miscellaneous
   ```
   all CL- and SE-
   ```

Figure 9.6 Control-oriented instructions.

1. Branches, jumps
   ```
   JMP, all B-- and BR
   ```
2. Loop control
   ```
   SOB
   ```
3. Subroutine linkage
   ```
   JSR RTS
   ```
4. Input-output
   None required for PDP-11
5. Interrupts
   ```
   BPT EMT IOT RTI RTT TRAP
   ```
6. Miscellaneous
   ```
   HALT WAIT RESET NOP MFPS MTPS
   ```

Logical Operations

There are some fundamentally important mathematical operations without which one could not even construct a computer. The seemingly essential operations of addition and subtraction are actually derived from these more fundamental operations. We are referring to the functions AND, OR, NOT, and EXCLUSIVE-OR.

These functions are called logical operations because their operands can only assume the logical values T and F (true and false). We will equate T with a binary 1 and F with 0. They are also called Boolean operations because George Boole worked out many useful relationships in this area over one hundred years ago.

NOT, COM

The simplest logical operator is NOT. It has a single operand; this unary operator is often represented by the tilde (~) sign. Given a logical variable x (i.e., x can assume only one of two values 0 and 1), then NOT x, or ~x, is defined by the relationship:

$$\sim 0 = 1 \quad ; \quad \sim 1 = 0$$

Among other deductions, it is always the case that $\sim(\sim x) = x$.

The PDP-11 handles logical variables in groups of eight or sixteen at a time. The logical operation NOT is implemented on the PDP-11 as the COM (complement) instruction. The instruction COM d computes:

$$\sim(d) \rightarrow d$$

Each bit of (d) is NOT-ed, or complemented. If you recall the earlier discussions of number representation, you will recognize that the logical complement produced by COM is identical to the one's complement of a number.

Given the statement `X: .WORD 123450,` the instruction `COM X` produces the result 054327 and stores it back at X. You can complement a whole word or just a single byte. These may be in memory or registers. As expected, the CC will be updated. The N and Z bits will have appropriate settings; V will always be cleared; and C will always be set. We will be saying "the N and Z bits will be given the appropriate settings" when it is the case that they will be set or cleared according to the result of the operation, as expected.

AND, BIT

The logical operator AND—symbolized by $\wedge$, an inverted "v"—is a function with two operands. We can define it using a truth table (a table in which all the entries are T/F or 1/0), as shown in figure 9.7. This table represents all possible values for (y AND x). The similarity between this table and that of the binary addition table is striking; the difference is very important. We can rewrite this table as in figure 9.8, in a form which is more convenient because it generalizes to logical functions of more than two operands. On the PDP-11 the nearest equivalent to the AND is the BIT instruction. Its format is:

```
BIT s,d ; (s) AND (d) updates CC without changing (d)
```

AND	x = 0	x = 1
y = 0	0	0
y = 1	0	1

Figure 9.7 Truth table for x AND y.

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 9.8 Column truth table for x AND y.

It operates on pairs of words (or bytes). It *does not* modify either operand. It is similar to the CMP in this way. The purpose of using BIT is to update the CC.

BIT s,d computes (s) ∧ (d), which means that corresponding pairs of bits from the source and destination operands will be ANDed together. If the *entire* result is 0, then the CC's Z bit will be set; otherwise Z will be cleared. The sign of the result will be copied into the N bit. The V bit will always be cleared, and the C bit is not affected.

The BIT instruction is often used with one of its operands being called a "mask." Suppose you need to know if bit 5 of item Q is set; item Q might hold the T/F answers to sixteen questions. A mask 000040 will "select" bit 5 (the PDP-11 bits are always numbered from right to left beginning at 0):

```
←                        one word                        →
15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
                    Bit Numbers
```

So the instruction `BIT #000040,Q` has the following effect. If (Q) was 102006, then (Q) ∧ 000040 ⟶ 0, leaving Z = 1. If (Q) was 012076, then (Q) ∧ 000040 ⟶ 40, leaving Z = 0. In both cases, we get N = 0.

You can, if you wish, examine several bits simultaneously. Thus, to see if any of the three low bits of WIZ are set, you can write:

```
    BIT     #7,WIZ
    BNE     SOMEON
```

or:

```
    BIT     WIZ,#7
    BEQ     NONEON
```

OR, BIS

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 9.9 Truth table for x OR y.

The "logical or," operation OR, is represented by ∨, a small "v". It has two logical variables. The truth table in figure 9.9 defines the logical OR. This OR is often called the INCLUSIVE-OR, because all but one case results in a one, and because the name emphasizes the contrast with another OR operation we will be examining, the EXCLUSIVE-OR. Figure 9.10 shows an everyday use of the OR.

The OR operation is provided by the PDP-11 BIS instruction. Its general form is:

BIS s,d ; (s) v (d) ⟶ d

The BIS instruction is useful when one wishes to be certain some set of bits are "turned on." The instruction `BIS #100001,A` will set both the high and low bits of (A); the other bits of (A) will remain as they were. The CC is updated as expected.

Bit Clear BIC

The BIC instruction combines a NOT with an AND:

BIC s,d ; ~(s) ∧ (d) ⟶ d

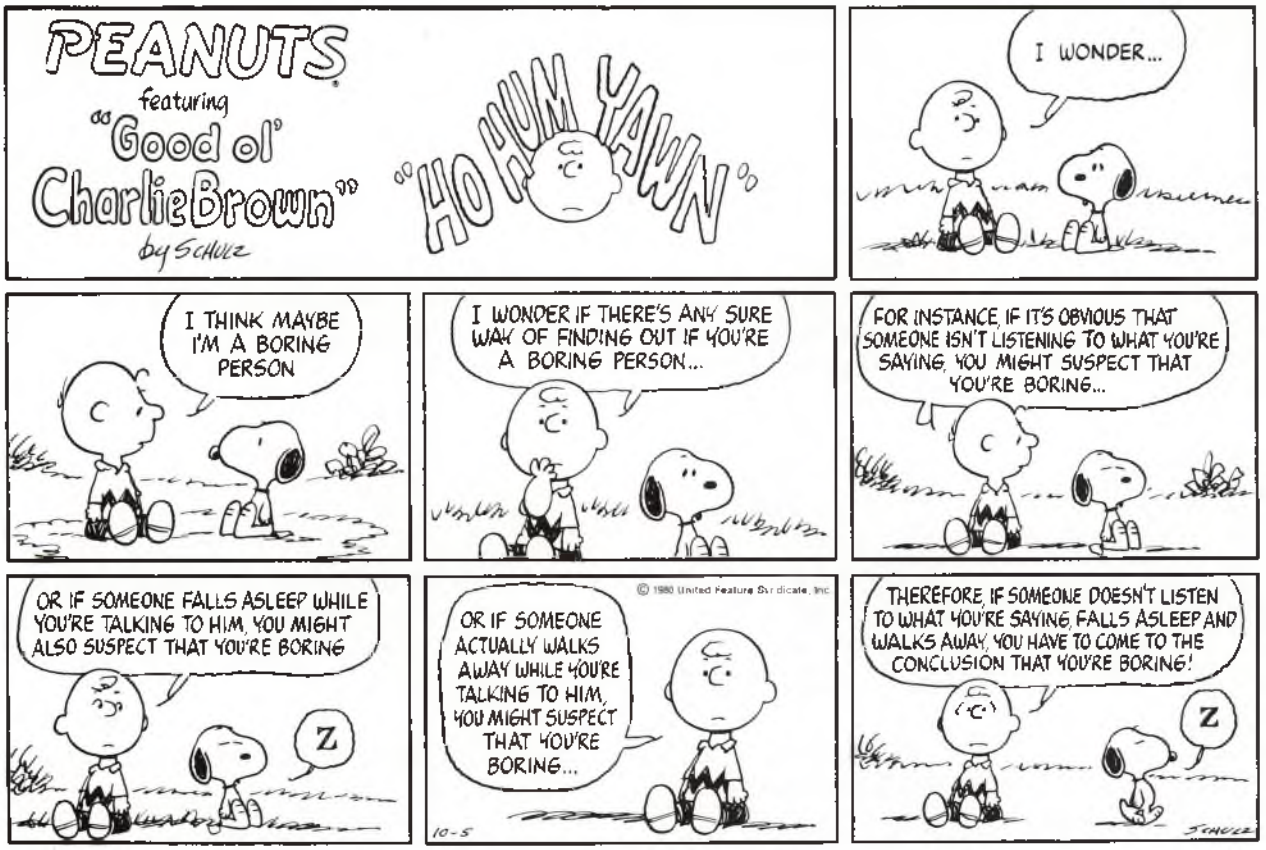


Figure 9.10 A very, very inclusive-OR. © 1980 United Feature Syndicate, Inc.

If you think of the source operand as a mask, then each "1" bit of the mask will clear the bit in the corresponding position in the destination operand. Suppose you want to clear every other octal digit in WOW. You can write:

```
BIC #070707,WOW
```

For (WOW)=123456, the result is ~(070707) AND 123456

```
  107070  AND 123456 → 107070
                       123456
                       103050
```

EXCLUSIVE-OR, XOR

The EXCLUSIVE-OR is a logical function of two variables. It is often represented using a NOT-ed OR ∀ (V with a -); we will more often than not use the "not-equal" symbol ≠ for it. We can define this logical operation as in figure 9.11. Whenever x and y are not equal, then x ≠ y is true. Similarly, whenever one of x or y but not both are true, then x ≠ y is true; thus the "exclusive" part of the "or." It follows that x ≠ x is always 0.

x	y	x ≠ y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 9.11 EXCLUSIVE-OR (not-equals).

The ≠ is implemented as the XOR instruction. This instruction is not available on all models of the PDP-11. It also deviates from the usual generality of double-operand instructions. Its format is:

```
XOR R,d ; (R) ≠ (d) → d
```

Consider this example of its use. Suppose you are machine-grading a true/false test. You can compare responses to sixteen questions at a time. Let SA be a location holding a student's answers to sixteen questions. Suppose the correct answers to those questions are in %1. Then

```
XOR %1,SA
```

will transform (SA), leaving a bit set wherever the student and the grader came up with opposite answers. You can count the 1's in SA to see how many wrong answers were found. For instance, with (%1) = 014072 and (SA) = 114022:

```
  014072
≠ 114022
  100050 ; 3 incorrect responses
```

If you inadvertently interchange the operand field ordering, the assembler will detect the error: i.e.,

```
XOR #123,%1 ; error
```

will be flagged as an "A" (addressing) error by MACRO-11. Correcting this by writing

```
XOR %1,#123 ; not much better
```

is accepted by the assembler, but results in changing the constant 123 at execution time, which is probably not what you intended. So remember that XOR R,d needs the register designation R first, and that the destination operand is modified.

Rotate Instructions

The rotate instructions shift the bits of a word or byte left or right, in a circular fashion. It may seem strange to think of these as logical operations, but the fact is that the rotate instructions treat neighboring bits independently of each other; that is, they are treated as independent logical values. The other rotatelike instructions we will be seeing soon treat the bits as part of a signed binary number, and the effect is quite different.

ROR and ROL

The instruction ROR d rotates (d) one bit to the right. The bit that is "pushed out" of (d) is copied into the carry bit C in the CC. The bit position vacated by the high order bit of (d) is filled in with the value previously found in the carry bit C (prior to executing the ROR). So a circular shift of a word, using ROR, always affects 16 + 1 bits, and a circular shift of a byte, using a RORB, always involves 8 + 1 bits. Consider:

```
MOV     #123456,%1
ROR     %1
```

We begin with %1 holding the bits 1 010 011 100 101 110. We finish with %1 holding:

? 101 001 110 010 111 and C = 0

The ? is whatever was in the carry bit C before the MOV (recall that a MOV never changes the C bit).

The ROL d instruction rotates (d) one bit to the left, with the C bit involved in filling in the vacated bit position, and itself being refilled with the bit pushed out of (d). A ROL %1 in the preceding example (in place of the ROR %1) would change %1 as shown in figure 9.13. A ROL ABC followed by a ROR ABC will restore ABC to its original value and only affect the CC bits.

You now have a simple way of counting the number of wrong answers that were found in an earlier example:

```
        CLR   COUNT
        XOR   %1,SA
        BEQ   DONE
        MOV   #16.,%2    ; check 16 answers
NEXT:   ROR   SA
        BCC   OK
        INC   COUNT      ; count errors
OK:     SOB   %2,NEXT
```

A ROL could just as well have been used above.

Condition Code Instructions

In the normal course of events, the condition code bits are expected to reflect the result of some operation such as a MOV, a CMP, etc. The setting of the CC bits is supposed to be a side effect of those instructions which use the ALU. So it may appear strange that the PDP-11 supports ten instructions which directly set or clear one or more CC bits.

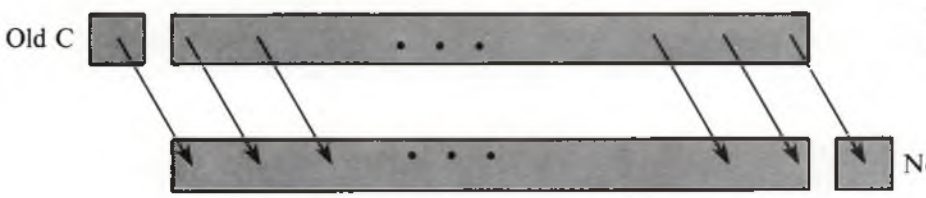


Figure 9.12 The ROR instruction.

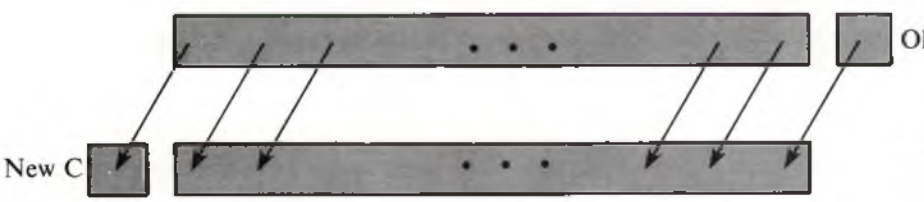


Figure 9.13 The ROL instruction.

The "clear" instructions have a mnemonic beginning with the letter C. The first four of these clear a single bit of the condition code, as indicated. The fifth clears all four of the condition code bits:

```
CLC     ; CL C   0 → C
CLN     ; CL N   0 → N
CLV     ; CL V   0 → V
CLZ     ; CL Z   0 → Z
CCC     ; C CC   0 → C,N,V,Z
```

We could have used the CLC instruction in the example which illustrated the ROL and ROR instructions. That would have ensured that the C bit starts out with the value 0.

The next five instructions set one or all of the condition code bits.

```
SEC     ; SE C   1 → C
SEN     ; SE N   1 → N
SEV     ; SE V   1 → V
SEZ     ; SE Z   1 → V
SCC     ; S CC   1 → C,N,V,Z
```

There is a deeper reason for allowing direct manipulation of these bits. When you are designing a computer, it is reasonable to allow users to manipulate anything they can see. This design principle allows users to "artificially" simulate conditions which would otherwise only be obtainable by means of complex test cases. It also simplifies saving the entire state of the computer (i.e., saving every special purpose and general purpose register, as well as any other volatile information). If this design principle is adhered to, then any desired state can be achieved without standing on your head.

MFPS, MTPS

We have discussed how the CC bits can be tested, set, or cleared. How can they be saved or restored as a group? The CC bits occupy the low four bit positions of a special register known as the PS word, for processor status, as shown in figure 9.14. The instruction MFPS d (only available on the PDP-11/34a) will copy the low byte of the PS word into the specified destination location. After doing so, N and Z will be updated in the usual way (in the real PS, not the copy just stored).

The converse instruction, MTPS d (also available only on the PDP-11/34a), restores only the low four bits of the PS (the CC portion of the PS), using the low four bits of (d).

We will be discovering other ways in which the CC is saved and restored.

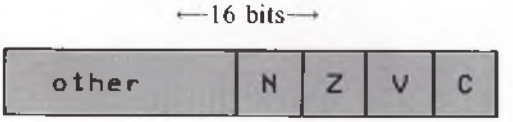


Figure 9.14 Processor status word (PS).

Arithmetic Shifts

When a word or a byte holds a signed number, it is possible to halve it rapidly, using a 1-bit shift-right instruction.

ASR

The ASR d instruction effects (d)/2 → d. In order to divide by two correctly, a negative operand should produce a negative result, and a positive operand should lead to a positive result. The ASR maintains the correct sign for the result by forcing sign extension. The sign bit of the operand is shifted right one position, as are all the other bits of the operand. The vacated sign bit is filled with a copy of the old sign bit. In this way a negative operand remains negative, and likewise for a positive operand. Consider the following examples:

```
                %1
                Before    After
ASR %1          000001    000000      1 → 0
                177774    177776     -4 → -2
                177777    177777     -1 → -1
                000004    000002      4 → 2
```

The N and Z bits are affected in the usual way. The bit that is forced out (bit position 0) is copied into the carry bit C. The overflow bit V is made equal to N ≠ C as they appear after the shift. This is not particularly useful (the setting for V), but it is consistent with its setting for the ASL, the instruction we will see next.

It is interesting to note how computer arithmetic differs from normal arithmetic. Here we have a situation where you can repeatedly divide something by two without changing it!

ASL

The arithmetic shift left instruction ASL doubles the value of its operand. The vacated low order bit is always filled with a 0 bit. The high order bit is copied into the C bit. Consider the following examples:

```
                %2
                Before    After
ASL %2          000002    000004      2 → 4
                1777777   1777776    -1 → -2
                100000    000000     -2^15 → 0
```

Whenever the original operand is negative, its double will also be negative, if it can fit in 16 bits. Attempting to double the largest negative number (furthest from 0) gives the result zero. Fortunately the V bit is always made equal to N ≠ C as they appear after the shift. This is the appropriate signal for overflow if we try to double the most negative or the most positive numbers.

Multiplication, MUL

The first PDP-11, the model 20, had no multiply or divide instructions. The arithmetic shift instructions were used to simulate a multiplication. The fastest way to multiply a number in X by five was apparently:

```
MOV     X,%1
ASL     %1      ; 2X
ASL     %1      ; 4X
ADD     X,%1    ; 5X
```

In actual fact, the execution time of an `ADD %1,%1` is the same as that of an `ASL %1` on the PDP-11/20; on a PDP-11/34, the ADD is faster than the ASL. Later models of the PDP-11 (such as the PDP-11/34) provide a hardware multiplication capability via the MUL instruction.

The symbolic representation

MUL s,R

computes

(s)*(R) $\longrightarrow$ R,R $\vee$ 1

which certainly needs some explaining. First of all, if you look up the MUL instruction in the appendixes, you will find its machine representation is 070 R SS. Do not let the reversal of the R and SS fields fool you. The assembler expects the source operand name first, then the register designation next. It will take care of providing the instruction encoding which the control unit expects at execution time.

If the register R is an even-numbered register (one of 0, 2, or 4), then MUL s,R produces a 32-bit result (s)*(R). The high order 16 bits are placed in R; the low order 16 bits are placed in register R v 1 (or R + 1, since R is assumed to be even here).

Consider what happens to registers 2 and 3 in the following examples, if we execute a MUL A,%2 with (A) having the specified values.

		Before		After	
	A	%2	%3	%2	%3
1.	0	10	20	0	0
2.	2	10	20	0	20
3.	2	077777	20	0	177776
4.	2	177777	20	177777	177776
5.	020000	030000	20	003000	000000
6.	160000	137777	20	004000	020000

What is going on here? Think back to the hypothetical computer of chapter 2. It had a MPY instruction which computed (AC)*(s) $\longrightarrow$ AC, and you were warned to avoid generating results that could not fit in the AC. Few real computers impose such a restriction.

As a general rule, the product of two n-digit numbers may require 2n digits for its representation (e.g., 400 * 300 dec → 120000, 3 digits * 3 digits producing a 6-digit result). The same rule applies in binary: the product of two 16-bit operands will be 32 bits long. This being the case, two words are required for the result of a multiplication of single-word operands. So the MUL will use a pair of registers, R and R + 1, for these two-word results. As a consequence, such a multiplication can never generate an overflow, and the V bit will always be cleared following a MUL. The N and Z bits are affected in the normal way.

To avoid unnecessarily checking R to see if the product is actually larger than 16 bits (e.g., are the high order 16 bits of a positive 32-bit product, all 0?), the C bit is set if the high order part is significant.

The form

```
MUL s,R; R odd
```

produces a 32-bit result (s)*(R), but only the low 16 bits are placed in R. If this leads to losing some significant bits, the C bit can be checked; V will always be cleared.

If we had used a `MUL A,%1` in the previous examples, with %1 having the values previously assumed by %2, the following results would appear:

		Before	After
	A	%1	%1
1.	0	10	0
2.	2	10	20
3.	2	077777	177776
4.	2	177777	177776
5.	020000	030000	000000
6.	160000	137777	020000

Execution time for a PDP-11/34 MUL is approximately four times greater than that of an ADD or an ASL.

Processing Numeric Data

Numeric information is usually transmitted to a computer by means of some character code such as ASCII to represent each digit. The decimal number 123 is encoded as the series of octal bytes 61, 62, 63. If we use an ASCII CRT and type the following line (followed by a Return)

```
123 + 304
```

the corresponding string of bytes generated by the CRT is (in octal):

```
61 62 63 40 53 40 63 60 64 15
```

Note that on some systems the Return code 15 (for CR) is automatically replaced (by the system software) with a line feed code 12 (LF). If these bytes are delivered to your program, how can you perform the indicated "+" operation?

There are two widely-used techniques that we will examine. The first of these involves transforming the external representation into its binary equivalent, and then using the ALU's binary arithmetic instructions, such as ADD, SUB, etc. The second technique involves performing arithmetic directly with the external representation, as a digit string.

Decimal-to-Binary Conversion

We can scan the byte string, isolate each numeric substring, and convert each digit code to its binary equivalent. We can then build up the binary numeric equivalent of the original character string.

Given the three bytes 61, 62, and 63, we can proceed from left to right:

```
61 − 60 → binary 1 → DTB
done? no, so compute (DTB) * 10. → DTB
next digit.
62 − 60 → binary 2
                    2 + (DTB) → DTB
done? no, so compute (DTB) * 10. → DTB
next digit.
63 − 60 → binary 3
                    3 + (DTB) → DTB
done? Yes; result in DTB
```

We can trace the values recorded in DTB:

Processing	DTB	
Digit	Decimal	Octal
"1"	1	1
"2"	10.+2	12+2
	12.	14
"3"	120.+3	170+3
	123.	173

Using the same algorithm, we can process the bytes representing the number 304 decimal, which results in octal 460. Completing the desired "+" operation is then trivial.

As a general rule, if you are only going to use a number once, it may not be worth converting it from its external representation as a sequence of character codes into its binary equivalent. In such a case, you would use the technique we will examine next. If you expect to use a number several times, then the conversion pays off.

BCD Arithmetic

How can you add two numbers without first converting them to binary? The technique involves going back to first principles. How do little children add "big" numbers? They process them a digit pair at a time. (Remember how we first did binary addition?)

Given the ASCII encoded equivalents for the decimal numbers 123 and 304,

```
  61        62        63        123
+ 63      + 60      + 64      + 304
                    147
                  - 60
                    67

          142
        - 60
          62

  144
- 60
  64                                427
```

The result is the string of ASCII codes representing 427. It may not always turn out so neatly. Consider 345 + 507.

```
  63        64        65        345
  65        60        67      + 507
 150       144       154
- 60       60        60
  70        64        74        84< !
```

The octal bytes 70, 64, and 74 represent the character string 84<, which is not quite what is desired. The problem clearly is that we must deal with the possibility that the sum of two digits cannot be represented with a single digit. This is an overflow situation which is not directly detectable by the PDP-11 overflow detection hardware, because that circuitry is concerned only with overflow from 8- or 16-bit operand operations. Here we are dealing with 4-bit operands (after you remove the bias of 60). It is up to the user to check each digit-pair sum and apply a correction, if necessary. So in the preceding example we should have proceeded as follows:

```
    63        64        65          345
    65        60        67        + 507
   150       144       154
-  60        60        60
    70        64        74
  > 71 ?    > 71 ?    > 71 ?  does it exceed 9 ?
    no        no        yes
-   0       - 0       - 12        correct by 0 or 10.
    70        64        62
              + 1
    --        --        --        propagate the carry
    70        65        62        result 852
```

Carry propagation might trigger additional carries (work out the details for adding 567 and 894). So, in general, it is better to proceed as follows.

Let the two-digit strings be referenced symbolically as $a_3\ a_2\ a_1$ and $b_3\ b_2\ b_1$.

$0 \rightarrow$ carry
$a_1 + b_1 \rightarrow c_1$
$c_1 - 60 \rightarrow c_1$; normal correction
is $c_1 > 71$? If yes, then $1 \rightarrow$ carry, $c_1 - 12 \rightarrow c_1$
$a_2 +$ carry $\rightarrow a_2$; propagate carry

$0 \rightarrow$ carry
$a_2 + b_2 \rightarrow c_2$
$c_2 - 60 \rightarrow c_2$; normal correction
is $c_2 > 71$? If yes, then $1 \rightarrow$ carry, $c_2 - 12 \rightarrow c_2$
$a_3 +$ carry $\rightarrow a_3$; propagate carry

$0 \rightarrow$ carry
$a_3 + b_3 \rightarrow c_3$
$c_3 - 60 \rightarrow c_3$; normal correction
is $c_3 > 71$? If yes, then $1 \rightarrow$ carry, $c_3 - 12 \rightarrow c_3$
carry $\rightarrow a_4$; propagate final carry

Some computers simplify life by providing a "half-carry" bit in their condition code register. The half-carry is set whenever the result of an addition exceeds 10 decimal. See figure 17.2 (page 445) for another nice way of handling BCD arithmetic.

The kind of processing we have done here is serial in nature. Corresponding series of digits from each operand are processed one pair at a time. There is no reason why you cannot process extremely large numbers this way.The same algorithm will handle one-hundred-digit-long numbers just as well as it handles three-digit numbers; it just takes longer.

The name BCD Arithmetic is often used to describe this kind of serial arithmetic (as opposed to parallel arithmetic, which is what a binary ADD does). BCD stands for binary-coded decimal; any character set encoding can easily be used for serial arithmetic if the codes assigned to the decimal digits 0 to 9 are consecutive and go from low to high.

In some data-processing applications (e.g., preparing a payroll) the numeric information tends to be used very few times. Computers intended for use in data processing may have special hardware and the instructions to use it, so you might have a "hardware" BCD-add instruction, a BCD-subtract instruction, etc. This is much better than having a "half-carry" bit, but a bit more expensive.

The PDP-11/44 (introduced in 1979) has a commercial instruction set (CIS) which is an extra-cost option. It supports 27 instructions, some of which operate on decimal digit strings as long as 65,535 digits. Among other features, it provides instructions to convert decimal to binary and vice versa.

Division, DIV

Division on a computer would not lead so many people into error if calculators were not so common. We are so accustomed to getting a single number as the result of a division performed by a calculator that we have forgotten that the result of dividing an integer by another has two parts, not just one. Most calculators immediately slip into a floating point mode (which we will soon discuss) when division is used so they can provide the result as one number. Divide 5 by 2 on a calculator and you get what you expect, namely 2.5. Not so on most computers.

Recall from previous years that, given a positive dividend x and a positive divisor y, the result of the division x/y is q + r/y. Little q is called the quotient and r the remainder. The remainder r satisfies the relationship $0 \leq r < y$. So the quotient q is the largest whole number such that $q * y \leq x$. For x = 17 and y = 4, 17/4 = 4 + 1/4. Most calculators would present the result as 4.25. A computer will handle the quotient 4 and remainder 1 as separate items. In general, since the dividend and divisor may be positive or negative, the remainder produced by a division will always have the same sign as that of the dividend. So for x = −17 and y = 4, −17/4 produces a quotient of −4 and a remainder of −1, instead of −5 and 3, respectively.

The PDP-11 DIV instruction is used for dividing integer operands. It assumes that you have a 32-bit signed number as the dividend, sitting in a pair of registers R and R + 1. The symbolic form

```
DIV s,R
```

computes

(R,R + 1)/(s) → R, R + 1

The quotient is placed in R and the remainder in R + 1.

In the case of a DIV instruction, R must always be even (the assembler fails to check this), and both R and R + 1 can be expected to change; (s) is never modified. Suppose you have something in location X which you wish to divide by 16 (decimal). You can write

```
MOV     X,%1
DIV     #16.,%0
...
```

Note that (X) → %1, and that %0 is specified in the DIV instruction; this is only the first point of confusion. The above code might produce the right result, but it would be mainly luck. Recall that a `DIV #16.,%0` divides the number 16. into the 32 bits found in both %0 and %1. Since we failed to put anything in %0, the result is probably wrong (unless %0 held a zero, just by coincidence), *if* the DIV even gets to complete.

It might not, because an unknown value in %0 might lead to a quotient which is too large to fit into a single word. In such a case, the V bit will be set. If you know that (X) is positive, then you could correct the problem by doing the following:

```
MOV     X,%1
CLR     %0
DIV     #16.,%0
MOV     %0,Q        ; quotient
MOV     %1,REM      ; remainder
...
```

Instead of worrying about the sign of (X), it is simpler to use the instruction we saw some time ago:

```
MOV     X,%1
SXT     %0          ; extend (X)'s sign
DIV     #16.,%0
...
```

As mentioned above, if you attempt a division which generates a quotient which cannot be represented as a 16-bit signed number, the V bit will be set. So trying to divide 2^{18} by 2 leads to V = 1, since 2^{17} won't quite fit in one 16-bit word. If you attempt to divide anything (including 0) by 0, the C bit will be set to indicate a problem, since x/0 is not defined for any value of x, inside or outside a computer (try it on a calculator).

To sum up, the DIV instruction is the trickiest instruction you are likely to encounter. You can get in trouble (WA = wrong answer) by doing any of the following:

1. DIV s,%1 ; odd numbered reg (M-TRAP).
2. Failing to place dividend in even reg (WA).
3. Failing to extend its sign (WA).
4. Dividing too large a # by too small a # (V = 1).
5. Dividing by 0 (C = 1).

The earliest PDP-11 had no DIV instruction. It was necessary to implement division by going back to first principles. Just as a multiplication is a shortcut for repeated addition, a division is a shortcut for repeated subtraction. ("How many times can I subtract 4 from 17, and how much will be left over?") So DIV was simulated using SUB instructions in a counting loop. The DIV instruction is not a simple one to execute as far as the hardware is concerned; it takes two to three times longer than a MUL. So an `ASR %0` would be much quicker than a `DIV #2,%0`, besides eliminating the sign extension before division.

Long Shifts; ASH, ASHC

The arithmetic shift instructions we have seen (ASL, ASR, ASLB, ASRB) only shift their operands by one bit. The pair of instructions we are about to look at permit you to provide a shift count between 0 and 31 or 32; the shift count is variable at run time.

ASH

The instruction ASH s,R allows you to provide a 6-bit signed count in location s. Since s can use any standard addressing mode, you clearly could have the shift count in memory or in a register and set it as desired at run time. If the count is positive, the item in R is shifted left arithmetically—think of (R) being multiplied by 2^{count}. If the count is negative, the content of R will be shifted right, arithmetically, by as many bits as are specified in the count value. Consider the following examples:

```
ASH CNT,%1
                    Before    After
          Count     %1        %1
          1         000001    000002
          2         000001    000004
          102       000001    000004
          177777    000001    000000
          177001    000001    000002
```

Only the low 6 bits of the count will be used. Should the result have a sign which differs from the sign of the original operand, V will be set; C will reflect the last bit shifted out of R.

You can if you wish use any of the standard addressing modes with the first operand of an ASH:

```
ASH A(%1),%2      ; get cnt from A(%1)

ASH #-3,%3        ; shift %3 right 3 bits
```

ASHC

The ASHC s,R instruction is called the arithmetic-shift-combined instruction. It is very similar to the ASH instruction except that it uses a *pair* of registers: R and R + 1. It still uses a 6-bit shift count, so we can shift the 32 bits of (R,R + 1) arithmetically left by as many as 31 bits, or right by as many as 32 bits. As with the ASH, V is set if a sign change occurs, and C will hold the last bit shifted out. As you might expect, register R *must* be even numbered. The assembler does not flag a violation of this rule. Consider:

```
                        Before          After
ASHC    CNT     %0      %1      %0      %1
CNT,%0
        1       000001  000001  000002  000002
        1       177777  177777  177777  177776
        177777  000001  000001  000000  100000
```

This completes our introduction to the most frequently used general-purpose instructions which are common to almost all models of the PDP-11 and LSI-11 family of computers. We will, of course, be seeing numerous other examples of their use.

Summary

We have seen two ways of classifying instructions: by format and by function. The PDP-11 control unit examines the leftmost bits of an instruction to determine which format is applicable. The remaining general purpose instructions have now been discussed. The logical instructions are: COM (take the one's-complement); BIT (which does not modify its operands); BIS (logical OR); BIC (clear bits of (d) using (s) as a mask); and XOR (EXCLUSIVE-OR).

The rotate instructions shift groups of bits as independent logical values: ROR (rotate right), ROL (rotate left). Condition code operators set or clear individual CC bits, or all four CC bits: SE−, SCC, CL−, CCC, where the − is one of C, N, V, or Z. The MFPS and MTPS (only for the 11/34) can save and restore the CC bits.

The shift instructions shift groups of bits as part of signed arithmetic values. ASR and ASL shift a word right or left by 1 bit. ASH and ASHC allow you to specify a 6-bit shift count; ASH operates on one 16-bit operand in a register. ASHC operates on a 32-bit operand residing in a register pair (even register, odd register). Integer multiplication and division are supported by the MUL and DIV instructions. Each has its special conventions regarding the use of registers. A MUL normally produces a 32-bit result, while a DIV expects to start with a 32-bit dividend.

Conversion of numeric data from its external representation as a digit string into its internal binary equivalent has been illustrated. This is especially important in applications which make repeated use of the numeric data. When that is not the case, arithmetic can be performed directly using the external representation; this programmed arithmetic is called BCD arithmetic. On some computers, special hardware is available to support BCD arithmetic.

Exercises

9.1 For the 16-bit binary number 1001 0101 1100 0001, show the effect of

(a) Four consecutive ASR instructions.
(b) Four consecutive ROL instructions.
(c) Four consecutive ROR instructions.
(d) Four consecutive ASLB instructions.
(e) Four consecutive RORB instructions.

9.2 Write a subroutine to simulate a restricted form of the PDP-11 instruction BIT. The subroutine call should look like

```
JSR      R5,BITTY
.WORD    NUM,BIT
```

where

```
NUM:   .WORD    —
BIT:   .WORD    n
```

and the value n of BIT is between 0 and decimal 15. Example: if NUM had the value 1... 101, then calling the subroutine with n=2 sets C=1; calling it with n=1 sets C=0. Bits are numbered from the right 15...2,1,0. The subroutine should return condition code C=1 if and only if the corresponding bit in NUM is set. You are not allowed to use the BIT instruction.

9.3 True or False?

(a) MUL can never produce overflow.
(b) A subroutine return RTS updates the CC.
(c) The remainder from a DIV can never be negative.
(d) BHI can be used following comparison of signed numbers.
(e) BCD arithmetic is preferred in computing-intensive applications.

9.4 Consider the following code:

```
1          MOV        A,%1
2          MOV        B,%2
3          MOV        #C,%3

4          CMP        %1,%2
5          BLE        Z
6          MOV        %2,-(%6)
7          MOV        %1,%2
8          MOV        (%6)+,%1

9   Z:     TST        %1
10         BEQ        Y

11         BIT        ONE,%1
12         BEQ        X

13         ADD        %2,(%3)
14  X:     ASL        %2

           ASR        %1
           BR         Z

Y:         ... MORE CODE ...

A:         .WORD      10
B:         .WORD      20
C:         .WORD      0
ONE:       .WORD      1
```

The following questions refer to the previous code.

(a) What does the code above do?
(b) In terms of speed, what happens when %2 has a larger value than %1 if lines 4 through 8 were removed?

(c) What problems arise if either %1 or %2 are less than 0 (but not both)?
(d) Given that A is 1, how large can B become before an overflow occurs?
(e) What is the value of C if A=56 and B=2?

Please state all assumptions that are made.

9.5 How can the execution of a single ASH instruction start with a positive operand, finish with a positive operand, yet set off the overflow alarm?

9.6 How could you save and restore the CC bits without using MFPS and MTPS? Try using nothing but branch instructions, MOVs, and CC manipulating instructions.

9.7 Why is it that in the suggested algorithm for BCD arithmetic, it is not necessary to see if propagating the carry does not itself trigger a carry? After all, couldn't "a + carry $\rightarrow$ a" leave a $>$ 71 octal?

9.8 Explain what happens when you execute

```
ASHC #32.,%0
```

when %0 and %1 contain 100000 and 000001, respectively.

9.9 If %2 and %3 contain 170100 and 000001, respectively, and you execute `ASHC %5,%2`, what will happen to the V bit if:

(a) %5 contains 3?
(b) %5 contains 7?

9.10 Taking advantage of the instructions introduced in this chapter, write the code for computing the parity of an 8-bit byte.

9.11 Outline an algorithm to convert the internal representation of a number to its external representation (i.e., a binary to decimal conversion).

9.12 Write a subroutine that takes a 16-bit word in %1 and generates its 6 octal character ASCII representation, placing it in memory using a pointer found in %2. Full credit will be given only to correct solutions using 20 lines of code or less. The character string must be what you would expect to see in an octal dump.

Case Study

In discussing machine-language programming for the PDP-11, back in chapter 3, we used a program-loading program. As you recall, it processed lines such as:

```
1000:013737,1010,1012
1006:0
1010:4
1000
```

Given these lines, memory locations 1000, 1002, . . . , 1010 octal would be loaded with the binary equivalents of the octal numbers 013737, 1010, 1012, 0, and 4. Then the loading program would stop loading on encountering the first line without a colon, and it would use the number on that line as the program's entry point. In this example the loader would execute a `JMP 1000,` since this is the entry point specified here.

Let us try to write the program which loads these machine-language programs, and let us do it in assembly language. As we proceed, we will call upon whatever system services we may need.

We can outline what needs to be done in a few lines of English. We can then refine our outline, over and over again, until we are left with well commented assembly-language statements that perform the desired processing. This is sometimes called the method of successive refinements. In order to keep things simple, we will not worry about some of the nonessential features of the chapter 3 loader, such as allowing for comments, ignoring blanks or blank lines, etc. These can be grafted in later, if desired. So we can develop an outline such as:

```
Repeat until entry point line is read:
   Read a line
      if no line is available, exit with a message.
   Process a line
End of Repeat block.

Entry-point line reached: print initial memory values
                          and copy Entry-point address into PC.
```

We could proceed in a top-down fashion (the name given to the method of decomposing a major problem into a series of simpler subproblems), but a digression into some bottom-up analysis will be interesting. In the bottom-up analysis you are concerned with the details of how a particular operation can be performed with a particular machine. Top-down analysis deals with a series of abstractions: bottom-up tries to implement a specific abstraction on a real computer.

One major concern, as we look at the loading problem from the bottom up is that we must convert an octal number string, presented to us as an ASCII character string, into its 16 bit-equivalent. How can we do this? Consider the statement:

```
A:   .ASCIZ  /304/
```

If we are given an ASCII string of numeric characters selected from the characters 0, 1, . . . , 7, that is terminated by a zero byte (courtesy of the .ASCIZ), we can generate its 16-bit numeric equivalent as follows:

```
        MOV     #A,R0    ; use R0 as a string pointer
        CLR     R1       ; build the 16 bits in R1
LOOP:   MOVB    (R0)+,R2          ; get a character
        BEQ     DONE     ; was it zero?
        SUB     #60,R2   ; no, so assume it is octal
; now we have three bits in R2, and expect more digits, so
        ASL     R1       ; make room for 3 more bits
        ASL     R1
        ASL     R1
        ADD     R2,R1
        BR    LOOP; do it again
```

Given the octal digits 3, 0, and 4, what we are doing is equivalent to evaluating the arithmetic expression

$$(3*8 + 0)*8 + 4$$

which is the value of the octal number 304. The *8 is performed here by three successive *2, each one accomplished by using an ASL instruction. Now that we feel comfortable with this octal-to-binary conversion, we can resume our top-down development.

We can refine what "process a line" means:

Echo the line
Scan it for a number

(1) if no colon is found, set EP flag, exit
(2) if colon is found, use the number as the
load address "adr"
and scan remainder of line for numbers,
storing them at addresses adr, adr+2, . . .
Also keep track of high and low addresses.

We can now consider what our memory needs may be, which constants may be needed, etc., as we begin writing some of the higher level code:

```
MEM:     .BLKW   1000     ; memory for prog being loaded
;
START:   .PRINT  #STRTM   ; "starting" message
REPEAT:  TST     EPFLAG   ; entry point reached?
         BNE     YES
                          ; no, keep reading lines
         .READ   #LINE,#NONE
; process a line
```

```
          .PRINT  #LINE;    echo it
          MOV     #LINE,R0 ; use R0 as pointer
          JSR     PC,SCAN
          TST     ERFLAG   ; any error?
          BNE     QUIT     ; bad line
          BR      REPEAT
;
YES:      .PRINT  #DMPM1   ; entry point reached
          .PMD    LOW,HIGH ; set up for PMD
          JSR     R5,DUMP
SHOW:     .WORD   LOW,HIGH
          JMP     @EP      ; start user's program
EP:       .WORD   0        ; entry pt adr
EPFLAG:   .WORD   0        ; entry pt flag
ERFLAG:   .WORD   0        ; error flag
;
NONE:     .PRINT  #NOEP
          .EXIT
NOEP:     .ASCIZ  /NO ENTRY PT GIVEN/
STRTM:    .ASCIZ  /LOADER WORKING/
DMPM1:    .ASCII  /ABOVE ADR IS ENTRY POINT/<12><15>
          .ASCIZ  /INITIAL MEMORY DUMP FOLLOWS/<12>
LINE:     .BLKB   82.
```

We of course have to fill in the missing .EVENs, .MCALLs, etc. But first let us see how the subroutine SCAN can be written. Notice how natural it was to think of getting a number from a line by having a subroutine do it. A sketch for SCAN follows:

Given a character string with its address in R0:

Call FINDNO, which processes octal digits, building a 16-bit
  number in R1, leaving the first nonoctal character in R2.
Is R3 0? If so, ignore line and repeat.
Does R2 contain a colon?
  If yes, save R1 as "adr"
    and if adr < LOW, set new LOW
    or if adr > HIGH, set new HIGH
  Repeat until line is finished
    Call FINDNO
      Is R2 ok?
      If not, bad line, so terminate.
      Otherwise store data from R1 in
      memory, at location adr and
      increment adr by 2.

This outline can be expressed as the following code:

```
SCAN:   CLR     ERFLAG  ; assume line is ok
        JSR     PC,FINDNO       ; get first #
        TST     R3      ; line empty?
        BNE     SCANA   ; no
        RTS     PC      ; yes- ignore it.
SCANA:  CMPB    R2,#COLON; first non-octal char in R2
        BNE     SCANNO  ; was not a colon
        MOV     R1,ADR  ; is a colon, so save load adr
;
SCANLP: JSR     PC,FINDNO       ; get next #
        TST     R3      ; no #?
        BNE     SCANB
        RTS     PC      ; none, so ignore
SCANB:  CMPB    R2,#COMMA; any comma?
        BEQ     SCANC
        CMPB    R2,#EOL; end-line code?
        BNE     SCAND   ; no
;
SCANC:  MOV     R1,@ADR; store number in memory
        JSR     PC,HILOW ; update high or low adr
        ADD     #2,ADR  ; adjust load adr for next word
        CMPB    R2,#COMMA
        BEQ     SCANLP  ; repeat after a ,
        RTS     PC      ; exit at end-line
; not a : or , or end-line
SCAND:  INC     ERFLAG; set error flag
        RTS     PC
; first # was not followed by : so use as entry pt
SCANNO: CMPB    R2,#EOL
        BEQ     SCANOK
        CMPB    R2,#SPACE
        BEQ     SCANOK
        INC     ERFLAG  ; found invalid char
        RTS     PC
SCANOK: MOV     R1,EP
        INC     EPFLAG  ; set entry pt reached flag
        RTS     PC
;
HILOW:  CMP     ADR,LOW ; new low adr?
        BHIS    HILOWA
        MOV     ADR,LOW ; new low
HILOWA: CMP     ADR,HIGH
        BLOS    HILOWB
        MOV     ADR,HIGH ; new high
HILOWB: RTS     PC
HIGH:   .WORD   0       ; init high very low
LOW:    .WORD   -1      ; init low very high
```

We are now down to the FINDNO level. Since registers 3 through 5 are not yet in use, we might exploit them if we need them here. Even if they were in use, we could always free them up by saving their values in the stack. We can write FINDNO as follows:

```
FINDNO: CLR     R1          ; build up 16 bit # in R1
        CLR     R3          ; number-was-found flag
FNDNXT: MOVB    (R1)+,R2    ; get next char
        CMPB    R2,#60      ; is it an ASCII digit?
        BMI     FNEND       ; in the range 0-7?
        CMPB    #67,R2
        BMI     FNEND
; have a good digit (octal)
        SUB     #60,R2      ; reduce to 3 bits
        ASH     #3,R1       ; make room for it
        ADD     R2,R1
        INC     R3          ; found a digit
        BR      FNDNXT      ; repeat
FNEND:  RTS     PC
```

Notice that here we are using a single instruction, ASH, instead of three ASLs, to multiply by 8. This is a slight improvement. You should usually avoid making improvements, called optimizing your code, unless you are certain your code works correctly. If it does not work correctly, there is no need to worry about making it go faster. Once it works correctly, worry only about speeding it up, or making it smaller, but only if it is too slow or too large.

We can collect all the pieces of code we have written above and insert the necessary definitions and directives, and that leaves us with the following program:

```
        .MCALL  .READ,.PRINT,.PMD,.SNAP,.EXIT,.EGDEF
        .GLOBL  DUMP
        .NLIST BEX
FORCE:  .BLKW   1000    ; forces MEM to adr 1000
MEM:    .BLKW   1000    ; memory for prog being loaded
;
START:  .PRINT  #STRTM  ; "starting" message
REPEAT: TST     EPFLAG  ; entry point reached?
        BNE     YES
                        ; no, keep reading lines
        .READ   #LINE,#NONE
; process a line
        .PRINT  #LINE   ; echo it
        MOV     #LINE,R0; use R0 as pointer
        JSR     PC,SCAN
        TST     ERFLAG  ; any error?
        BNE     QUIT    ; bad line
        BR      REPEAT
```

```
;
YES:    .PRINT  #DMPM1  ; entry point reached
        JSR     R5,DUMP
SHOW:   .WORD   LOW,HIGH
        JMP     @EP     ; start user's program
EP:     .WORD   0       ; entry pt adr
EPFLAG: .WORD   0        ; entry pt flag
ERFLAG: .WORD   0        ; error flag
HIGH:   .WORD   0        ; init high very low
LOW:    .WORD   -1       ; init low very high
ADR:    .BLKW   1
;
NONE:   .PRINT  #NOEP   ; no entry point
        .EXIT
QUIT:   .PRINT  #QUITM  ; illegal char
        .EXIT
QUITM:  .ASCIZ  /ILLEGAL CHAR; LOADING STOPPED/
NOEP:   .ASCIZ  /NO ENTRY PT GIVEN/
STRTM:  .ASCIZ  /LOADER WORKING/<12>
DMPM1:  .ASCII  /ABOVE ADR IS ENTRY POINT/<12><15>
        .ASCIZ  /INITIAL MEMORY DUMP FOLLOWS/<12>
LINE:   .BLKB   82.

        .EVEN
COLON   = 072   ; ASCII code for colon
COMMA   = 054   ; comma
EOL     = 12    ; end-of-line code
SPACE   = 040   ; space or blank
;
SCAN:   CLR     ERFLAG  ; assume line is ok
        JSR     PC,FINDNO       ; get first #
        TST     R3      ; line empty?
        BNE     SCANA   ; no
        RTS     PC      ; yes- ignore it.
SCANA:  CMPB    R2,#COLON; first non-octal char in R2
        BNE     SCANNO  ; was not a colon
        MOV     R1,ADR  ; is a colon, so save load adr
;
SCANLP: JSR     PC,FINDNO       ; get next #
        TST     R3      ; no #?
        BNE     SCANB
        RTS     PC      ; none, so ignore
SCANB:  CMPB    R2,#COMMA; any comma?
        BEQ     SCANC
        CMPB    R2,#EOL ; end-line code?
        BNE     SCAND   ; no
;
SCANC:  MOV     R1,@ADR ; store number in memory
        JSR     PC,HILOW ; update high or low adr
        ADD     #2,ADR  ; adjust load adr for next word
        CMPB    R2,#COMMA
```

```
        BEQ     SCANLP  ; repeat after a ,
        RTS     PC      ; exit at end-line
; not a : or , or end-line
SCAND:  INC     ERFLAG; set error flag
        RTS     PC
; first # was not followed by : so use as entry pt
SCANNO: CMPB    R2,#EOL
        BEQ     SCANOK
        CMPB    R2,#SPACE
        BEQ     SCANOK
; invalid char
        INC     ERFLAG
        RTS     PC
SCANOK: MOV     R1,EP
        INC     EPFLAG  ; set entry pt reached flag
        RTS     PC
;
HILOW:  CMP     ADR,LOW ; new low adr?
        BHIS    HILOWA
        MOV     ADR,LOW ; new low
HILOWA: CMP     ADR,HIGH
        BLOS    HILOWB
        MOV     ADR,HIGH; new high
HILOWB: RTS     PC
FINDNO: CLR     R1      ; build up 16 bit # in R1
CLR     R3      ; number-was-found flag
FNDNXT: MOVB    (R0)+,R2
        CMPB    R2,#60  ; is it an ASCII digit?
        BMI     FNEND   ; in the range 0-7?
        CMPB    #67,R2
        BMI     FNEND
; have a good digit (octal)
        SUB     #60,R2  ; reduce to 3 bits
        ASH     #3,R1   ; make room for it
        ADD     R2,R1
        INC     R3      ; found a digit
        BR      FNDNXT  ; repeat
FNEND:  RTS     PC
        .END    START
```

There is a subtle error in this program. It will load what was intended correctly, but at the point where you expect an octal memory dump, you won't see what you want to see. Requesting a dump with the arguments LOW, HIGH as we have it at the line labeled SHOW will display the low and high addresses encountered in loading the user's machine-language program. If you want to display the program that was loaded, as was done in chapter 3, you should replace the SHOW line with the lines

```
LOW:    .WORD   -1
HIGH:   .WORD   0
```

The line labeled FORCE will assure that the address assigned to MEM is 1000, if you are using a computer in which the first address assigned to your program is 0. If you are using RT-11, then leave out the FORCE line, since the desired address of 1000 will be the first assigned anyway.

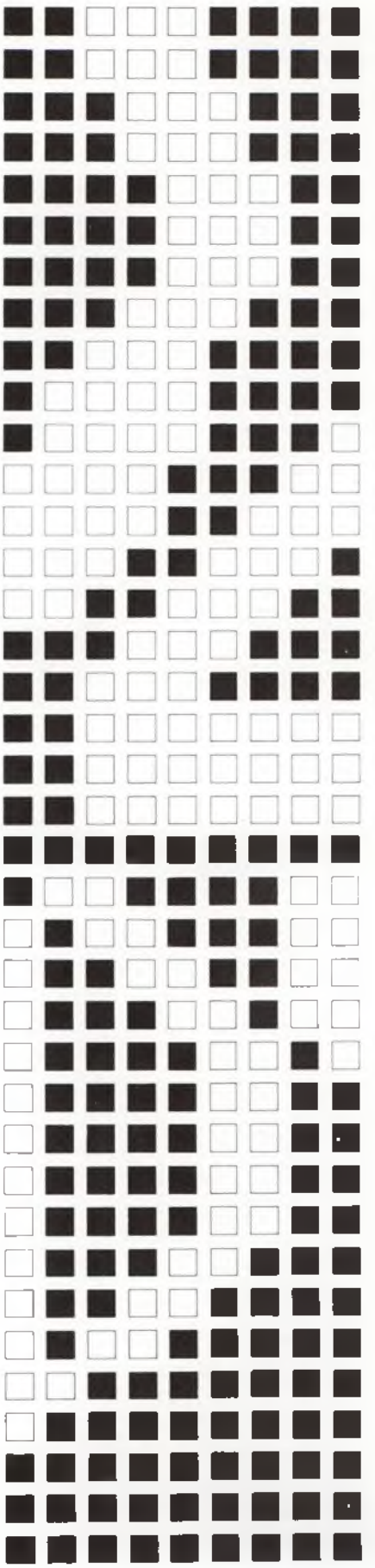
The simple implementation we produced in this case study can be enhanced in many ways. Some of the features which could be added are listed here:

1. Accept blanks as delimiters, as in 1000: 1, 2.
2. Accept tab codes as delimiters.
3. Accept blank lines.
4. Accept comments, preceded by a semicolon.
5. Check for too many octal digits, as in 1234567.
6. Warn a user that a memory location is being loaded twice.
7. Check that only addresses in the desired range (e.g., 1000–1776) are being loaded.
8. Accept standard branch instruction opcodes, with full-width memory addresses as operands, and have the loader generate the 8-bit branch offset. Thus to obtain the effect of a branch to 1012, to be loaded in location 1234, you could type 1234: 400 > 1012. The > would request conversion of the address which follows it into a branch offset.
9. Accept instruction mnemonics, so that instead of typing 1000:013737,1200,1106, you could type 1000:MOV 1200,1106.
10. Drop the requirement that each line begin with a load address; the loader could assign these.
11. Allow the use of symbolic names. At this point your loader is turning into an assembler. It would assemble, load, and execute symbolic programs.
12. Allow access to a library of predefined system services. At this point, your loader also provides some features of a linker.

This case study might leave you unsatisfied in that the program we developed is not 100 percent self-contained. After all, it did use system services for its input and output, and for dumps. If you wish to write your own input and output subroutines, even your own dump and post-mortem, you should be able to do so with very little effort after reading chapter 11, which deals with how the PDP-11 really does input and output.

10

keyboards, codes, and terminals



We have yet to discuss how a PDP-11 performs input or output operations. Before we do so, it will be helpful to first look into the details of the devices most people use when communicating with a computer. These devices have evolved rapidly in the last ten years. We will take a brief look at the simpler ones.

Punched Cards and Paper Tape

Many computing facilities use punched cards as the preferred medium for preparing machine-readable input. We described the standard punched card back in chapter 1; it has a capacity for 80 characters, each being expressed as one 12-bit code. As a general rule, after being keypunched, data are passed through a verification process done with another machine, called a verifier, which closely resembles a keypunch. The original data are entirely rekeyed, each new keystroke being compared with what had been punched in the card. In this manner, only good data cards should ever reach the computer.

A different approach is used when data are prepared on paper tape. If an error is discovered when the tape is being typed and punched, the tape can immediately be backed up and the erroneous tape frame overpunched with a DEL (delete) code. This explains why the DEL code is 177; it can overwrite any other code. A program which expects to process data coming from a paper tape has to expect to see and discard characters such as DEL. This simply could not happen with punched cards. As we will now see, the situation is even more interesting with CRTs.

Glass Teletypes

The venerable model 33 Teletype was discussed briefly in chapter 1. Well before interactive computing came into existence (usually in the form of time-sharing systems), teletypelike devices—teletypewriters—were used for nonvoice communications by the military and by common carriers (e.g., Western Union telegrams), and as (rather primitive) "automatic" typewriters (so called because of their ability to generate and replay paper tapes).

When computer time-sharing became feasible in the midsixties, the least expensive device which allowed a user to interact with a time-sharing computer system was the model 33 Teletype, so many of them were pressed into service as computer terminals. Non-time-sharing computers also used the Teletype as the interactive device of choice, especially on the then new breed of so-called minicomputers. The first PDP-11s used the model 33 Teletype; the DECwriter and DEC VT52 and VT100 CRTs had not yet been invented.

Other manufacturers, sensitive to the growing market for teletypewriters, and to the weaknesses of the TTY, built new products which emulated the TTY in an upward compatible fashion; i.e., they could do everything a TTY does, and then some. The phrase "glass teletype" is used to describe a CRT which does little more than emulate a TTY. Such CRTs are typically the least expensive terminals.

The TTY operates at 10 cps (characters per second). This is very slow speed for dealing with computers. The TTY is an electromechanical device, so it tends to be noisy; it needs oiling and cleaning; it consumes paper and ribbons. Other manufacturers eliminated the paper and ribbons and substituted a CRT display. The only major mechanical component left in may have been a fan for internal cooling. The new products provided selectable speeds from a low of 10 cps to a high of 960 cps. More recently, 1,920 and 3,840 cps have become common speeds for terminals. The simple CRTs emulate the TTY so that they can be substituted for TTYs (even to the extent of emulating the TTY bell signal), but they provide other features in addition to a range of speeds. The TTY has only a 72-character line; most CRTs accommodate an 80-character line, some even more. The TTY does not support lowercase letters (part of the reason why MACRO-11 is biased toward uppercase only). Most CRTs support upper- and lowercase letters, since they are included in the ASCII character set. What about keyboards?

Keyboards

Except for the QWERTY arrangement of the letters of the alphabet, not much else has been standardized on the typical CRT keyboard. The location of punctuation characters, control keys, etc. varies from one model to another. One manufacturer may have several different keyboard layouts, even for the same model of terminal. The keyboard for a "typical" CRT is depicted in figure 10.1. For most of the keys, pressing results in transmitting the corresponding ASCII code. Some of the keys transmit no code but have a local significance which may affect future code transmission. Those keys which cause an ASCII code to be transmitted when depressed are referred to as encoded keys.

The Shift key does not transmit a code (it does if you are using a Baudot code, an IBM PTTC code, etc., but not ASCII). Any other character key pressed while Shift is being pressed (or while it is latched, in the case of shift-lock keys) will transmit the code corresponding to either the alternate marking on the key caps or an uppercase letter code, as appropriate.

The nonprinting keys such as Backspace, Return, Line Feed, Tab, etc. each transmit one ASCII code (BS, CR, LF, and HT, respectively). Perhaps the most misunderstood keys are the CTRL and Break keys. The CTRL (or CTL) key is the Control key; it is very much like a shift key.

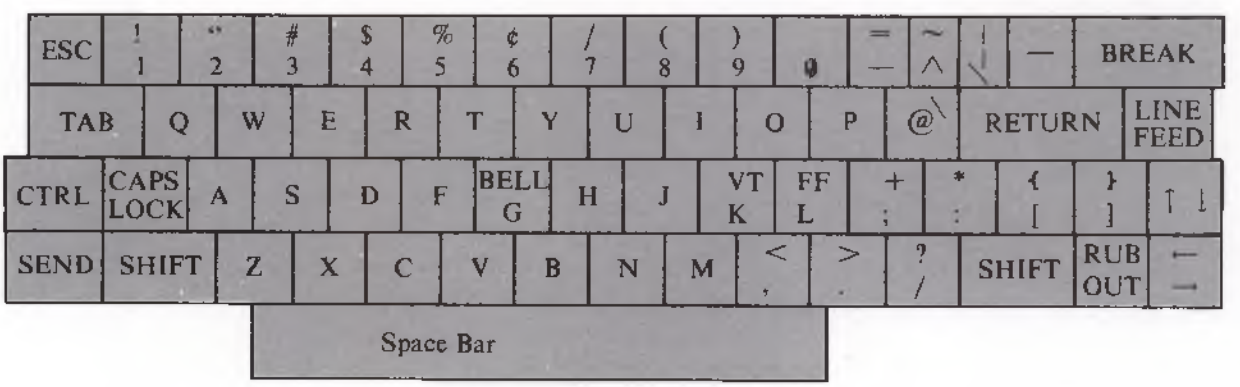


Figure 10.1 A typical CRT keyboard.

Any other of the transmitting keys pressed while CTL is held down is assigned an alternate meaning. On some keyboards the alternate meaning is shown on the keycaps. CTL-h generates a Backspace code, CTL-g sends a Bell code which rings a bell (once per code), CTL-i sends a Tab (HT). This is true whether or not these alternate meanings appear on the keycaps. These are so defined in the ASCII code set. Pushing and releasing CTL has no effect unless some other key is pressed at the same time.

The Break key is distinguished by the fact that it is the only key which sends a code which is not part of the ASCII code set! One of the uses of the Break key is to allow you to "break in" in the middle of a transaction. A user may wish to send a signal to whoever or whatever is sending him a long message, to cease transmitting it. If the user U and other party P share an absolutely minimal communication line, one pair of wires, it is not possible to reliably send an ASCII code from U to P while P is in the process of sending ASCII codes to U. If you try it, both parties will get garbled messages. The Break key provides a solution by forcing all bits, and all inter-bit gaps, to the same voltage level for an interval of time which encompasses at least one character (e.g., 0.1 seconds or more on a 10 cps terminal). This means the Break signal will override any in-progress transmission, and break-detection circuitry recognizes the break signal as such. Both parties in a communications hookup (TTY to TTY, or CRT to CPU) can easily detect or generate a break signal. If you receive one, by convention it is interpreted as a request to cease sending until further notice, or to revert to some previously agreed-upon mode.

CRT Displays

Many names are used for computer terminals which display information on a glass screen: VDU and VDT (for video display unit or terminal), CRT (for cathode ray tube). Such display terminals are manufactured in hundreds of different models with dozens of different attributes. We will describe a typical "composite" CRT.

Our composite can display 24 lines of 80 characters; this is a maximum of 1,920 characters/page. The word "page" is often equated with the amount of information displayable at one time on one screen. The information you see at one time is being dynamically regenerated at the rate of 60 times per second (much as a television image is). Without the regeneration of the image, it would decay rapidly and disappear. As a consequence, a CRT needs a display memory capable of storing at least one page of information (1,920 bytes). It was this need for memory that kept the cost of CRTs so high even into the early 1970s.

The correspondence between the display memory and the display image is worth looking into. Suppose our CRT screen was displaying the image shown in figure 10.2. A particular screen image may have arisen from quite different keystroke sequences (or none at all, if the image was generated by received codes). The keystroke sequence

A,B,C,RET,LF,HT,D,E,F,RET,LF,LF,1,2,3

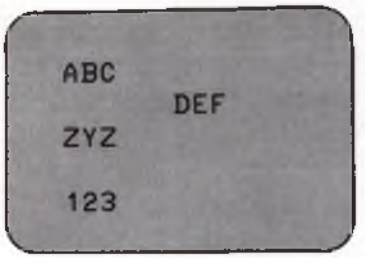


Figure 10.2 Display image on screen.

could have produced this image. The eight spaces in front of the DEF could be produced either by one Horizontal Tab character, as in the preceding keystroke sequence, or by typing eight spaces, or by typing two HTs and eight Backspaces, etc. The keystroke sequence

A,B,C,RET,HT,D,E,F,...

may lead to the display memory storing the earlier string, if the CRT hardware interprets the Return key as a request to send a Return code (octal 15), automatically followed by an LF line feed code (octal 12). Sending a code from the CRT does not necessarily imply storing it in its display memory. After all, does your password ever appear on the screen? Only those things echoed at the CPU end will be received and stored in display memory if you are operating in the full-duplex mode we will discuss shortly. So the first of the keystroke sequences would be stored if the operating system echoed all it received. It might choose to echo only the line feed upon receipt of a Return-Line feed pair.

Typing a Backspace may appear to erase a character, or at least let you write over it. Whether it does so at the CPU end is a function of the CPU's software; beware. The CPU could elect to interpret "!" or "#" or anything else as an "erase previous character" request.

We may draw the conclusion that the behavior of a CRT may vary greatly, depending on the software running on the CPU it is attached to. To compound the problem, the characteristics of CRTs vary greatly themselves.

What happens when the display memory is full? As a general rule, the oldest information is at the top of the screen, and when a new line is received, all lines are pushed up by one line position, freeing the bottom line for the incoming text. This action is called *scrolling*. If a terminal has several pages of display memory, then it would have forward and reverse scroll function keys, so you could browse.

Hard-Copy Terminals

The TTY is an early example of a hard-copy terminal. As mentioned, it was limited to uppercase only, with lines of 72 characters and a speed of 110 baud (10 cps). Popular terminals such as DEC's DECwriter family have selectable speeds, either in the 10–30 cps range, or as high as 120 cps. Line lengths of 80, 120, or 132 characters are common; and the full ASCII character set is supported.

Forms control is part of the ASCII code set, principally for the benefit of hard-copy terminal users. The notion of forcing a continuous-form, fan-folded paper source of 2,500 sheets to eject so that a new page may begin printing at the precise "top-of-form" has significance for hard-copy terminals. Some terminals interpret the HT tab code as a signal to skip to the next "tab stop" position (analogous to those on typewriters), with an implied tab stop set at every eighth character position. Other terminals have no interpretation for the Tab code and may ignore it or replace it with a single blank.

Good system software accepts a description of each terminal's characteristics, so that it can cooperate with this peripheral hardware. If the system is told that you (your CRT) has a hardware tab feature, it will send tabs to save time (yours, its, and line charges). If it knows you do not have this feature, the software can simulate it for you, substituting an appropriate number of blanks in place of any tab code it would otherwise have sent you. In this last case, observe that the number of blanks, or space codes, necessary to replace one tab code may vary from tab to tab even though the tab stops are fixed and uniformly spaced.

The phrase "soft copy" is sometimes used to describe CRTs to emphasize the contrast with "hard-copy" paper-using terminals. Of course, for those who can afford them, or cannot afford to be without them, there are soft-copy CRTs with built-in hard-copy printers; you only print what you wish to keep.

Most hard-copy interactive terminals print one character at a time, as they are received. These are character-printers, as opposed to line-printers, which print a whole line at one time. Few interactive hard-copy terminals use this technique. It is fairly expensive, and it is used where high-speed printing is needed (e.g., 1,000 lines/minute, or 2,000 cps for 120 char/line).

Intelligent Terminals

TTYs and "glass" TTYs are examples of so-called "dumb" terminals. A "glass" TTY is a low-cost CRT that can replace a TTY. The other kind, called smart or intelligent, usually have a built-in microprocessor (or two). Some even use an LSI-11 just for this purpose. Not all terminals with a built-in microprocessor are categorized as "intelligent." CRT designers sometimes find it is cheaper to build a dumb terminal with a microprocessor than without one (some microprocessors retail for less than $10.00; they cost much less when purchased in quantity). The designers do not necessarily provide the additional features that could easily have been provided. It is interesting to note that when the PDP-11/20 was first built, the cheapest way to provide the hardware for registers 0–7 was to use a chip that supported 16 registers. So every buyer of a PDP-11/20 paid for a 16-register machine but could access only eight of them. Likewise in the world of terminals.

An intelligent terminal offers a variety of special services. The variety is enormous and bewildering. We will mention just a few.

Every CRT has a cursor, a visible symbol which indicates where the next displayable character received by the CRT will be displayed. When a character is received, the cursor moves right one character position so the new character can be displayed. When the cursor would otherwise pass beyond the end of the line, it proceeds to the beginning of the next line. If necessary, all preceding lines will be scrolled up by one line. Programmable cursor control allows the distant CPU to send codes to the CRT so that the cursor may be placed at any desired spot on the screen, within the characters/line-lines/page resolution. Often a simple coordinate scheme is used.

DC4, binary x, binary y may place the cursor at line x in character position y. DC-n is the ASCII device control code. Some CRTs might use an "escape sequence" instead. ESC is an ASCII code that can be interpreted as a generalized "shiftlike" code. It does transmit an ASCII code. The receiver presumably treats the following characters in a very special way.

Local editing may be supported. An "Insert line" function key may be available. After the cursor is moved to the desired line (by means of the cursor-movement-control keys), pushing "Insert line" forces a blank line to be squeezed in after the current line (or in front of it). You may then enter a new line over it. Several other editing functions would normally accompany the insert function. Presumably all of these would be used with your terminal in local mode. When you have a "perfect" screenful of input, you place the cursor in the "home" position (top left corner of screen). Pushing the "block transmission" or Send key then places your CRT in remote-mode (on-line, not local), transmitting the whole screenful as one "block."

Data-processing-oriented systems tend to operate in this one-screen-at-a-time fashion as opposed to the character-at-a-time action we described earlier. The phrase "transaction-processing" is often used to describe systems in which the common mode of interaction is a screen-at-a-time, as in a bank teller system (the ones used by the clerks, not the customers).

Some terminals have none of the switches described earlier (e.g., speed selection). The switches have been replaced by software; the functional capability is still there, but the mechanical switches are gone. You enter a special "terminal configuration mode" by pushing a strange key combination, then select from a built-in menu the desired speed, parity, behavior on hitting Return, etc. This phenomenon of replacing hardware by software is not a new one. Remember the CPU front-panel emulators we discussed when looking at loading machine-language programs? It is exactly the same kind of substitution of software for hardware that took place then.

Parity Selection

Most terminals have a switch-selectable parity option. If it is set to odd-parity, then an eighth bit will be placed in the high-order position of each 7-bit ASCII code, and its value will always be such that there are an odd number of 1 bits. Even parity works in a similar fashion. Zero parity always fills in a zero, in effect wasting a bit. When set to even or odd parity, your terminal will generate the correct parity bit for each outgoing character. It will check each incoming character for correct parity. Finding a bad parity bit should result in some signal to you.

Half-Duplex Communication

The two-wire communication link we mentioned above allows reversible one-way communication; this is called a *half-duplex* communications circuit. That is, one of the two parties sharing the link may send while the other listens, and vice versa. Attempts at simultaneous transmission result in garbled messages. This requires a certain discipline on the part of the users, not unlike that used on walkie-talkies, or more recently on CB radios. The word "Over" is used to indicate that one party has finished sending for the moment and is disposed to listen for messages from the other party.

When communicating with a computer in half-duplex mode, the character you intend to transmit should be echoed on your CRT display by *your* CRT hardware; that is the purpose of the Full-Duplex/Half-Duplex switch which you will find on almost all CRTs (if you look hard). With a half-duplex line, you cannot afford the luxury of having the CPU echo what you transmit. That would reduce the line's communication capacity by more than half.

Full-Duplex Communication

An interactive terminal incorporates two distinct communication functions: sending and receiving. There are noninteractive terminals. An RO (receive-only) printer probably has no keyboard. An SO (send-only) device might be a sensor in an environmental-monitoring system; if a temperature exceeds a particular level, the SO sensor sends an ASCII-coded message such as "sensor 13 detects a temperature of 200; call the fire department." By definition, an interactive terminal is an SR (send-receive) device. Model 33 Teletypes equipped with a paper tape reader are called ASR devices (automatic SR) because the reading of a prepositioned paper tape can be started and stopped at will by another TTY, distantly located.

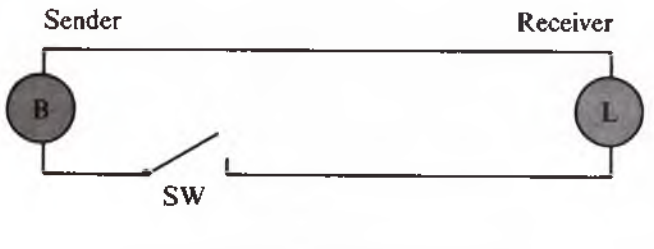
If you have an interactive terminal, why not allow it to receive independently of any sending which may be in progress, without necessarily relating the two functions? This can be done, and it is quite common, but it requires more than a two-wire link. Since one-way communication needs at least one pair of wires (see next section), it stands to reason that simultaneous two-way communication probably needs four or more wires. Such a simultaneous two-way communication link is called a *full-duplex* communications circuit. That was a very good reason for staying with half-duplex telegraphy lines when the transcontinental telegraph lines were being installed. The cost of doubling the quantity of wire to support full-duplex communication was simply too high. Besides, sending a telegram is not really an interactive situation.

With a CRT set to full-duplex mode, connected to a full-duplex line, the codes you send will as a general rule not be displayed unless the receiving CPU forces echoing. This has the virtue of showing you what the CPU thinks it received, giving you an opportunity to correct it if a transmission error occurred. It is worth spending a little time to discuss how half-duplex and full-duplex links can be implemented. Many of the mysteries in computing hardware and software stem from a poor understanding of how bits travel.

Suppose you wanted a one-way communication link from your house or apartment to a friend's house (perhaps to serve as an alarm or a call signal). How could you build something simple, reliable, and inexpensive without involving sophisticated electronics? Let B be a battery (maybe a six-volt lantern battery), and L a six-volt light bulb (or bell or buzzer), and SW a switch (or a push-button). With two wires you could connect the sender S and receiver R as shown in figure 10.3.

Such a one-way communication system is called a simplex circuit. By closing the switch at SW, the sender S can alert the receiver R. With a little training, S can send Morse encoded messages to R; this is but one step removed from sending bits. What we have here is properly called a digital communications circuit. We will be examining how a typical computer communications circuit behaves shortly.

What if you wanted to allow the receiver R to send to the sender S? The wealthy person's solution is to duplicate the simplex system, reversing placement of the light L and the switch SW, as depicted in figure 10.4. Site 1 can send to site 2, while site 2 is sending to site 1. This is the essence of a full-duplex communication circuit. The thinking person on a tight budget will reason: "Do I really need simultaneous two-way communication? Could we not agree to take turns sending and receiving?" It is possible to get along with just two wires, as shown in figure 10.5.



Simplex Communication Circuit

Figure 10.3 Simplex communications.

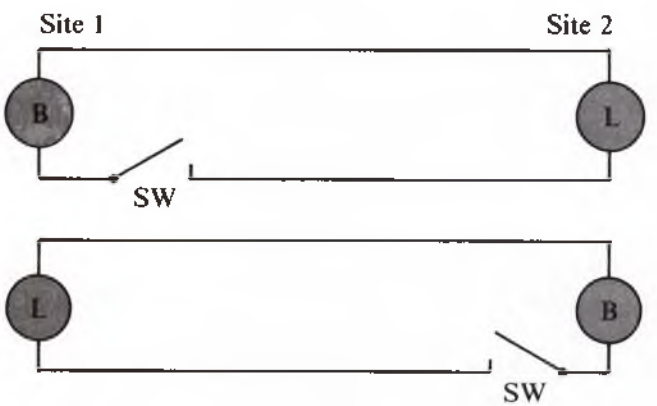


Figure 10.4 Full-duplex communications.

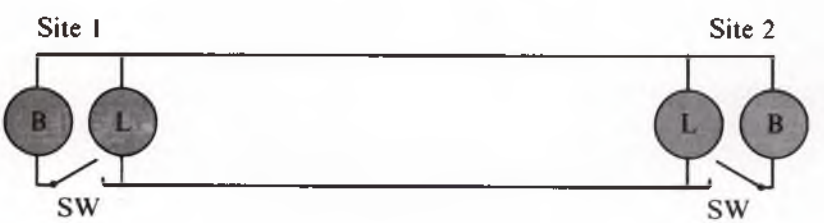


Figure 10.5 Half-duplex communications.

S1 can send to S2. While doing so, S1 immediately sees on his light what he is sending to S2; this is a *local* echo. S2 can send to S1; S2 also gets a local echo while sending. If S1 and S2 try to send at the same time, both messages will be garbled. This simple circuit depicts the essence of a two-wire half-duplex communication circuit. If you were to build one, wire which comes as twisted pairs to increase immunity to electrical noise would be preferred. So in talking about CRTs which are "hard-wired" (i.e., they are directly and permanently connected) to a CPU, one often hears of so many twisted pairs being used.

Binary Transmission Codes

The practical code to use in these simple systems is the International Morse code; every character is represented by a combination of short and long light flashes or buzzer signals, etc. By convention, we use dots (.) and dashes (-) to represent the two digits of this strange-looking binary code. The Morse code is given in conjunction with one of the problems at the end of this chapter. You may be familiar with ... --- ... (SOS), the international distress signal (now vocalized as "Mayday"). The Morse code is practical for nonvoice people-to-people communication. You can even buy hardware to let a computer transmit and receive Morse code. In Morse code, the time intervals are critical. A . must result in a signal of such-and-such duration; a dash must last so many times longer than a dot; the dot and dash separation interval must be such-and-such; and the gap between character codes must be a certain duration. How is it with standard transmission of ASCII bytes?

The technique used to transmit ASCII bytes is also used with other binary codes (EBCDIC, etc.). We will illustrate the technique using ASCII codes in our examples. A standard technique used for transmitting bits is to associate with each 1 bit a "mark" signal corresponding to a voltage level in a particular range, say -5 to -15 volts. A 0 bit could then be represented as a signal corresponding to a positive voltage level—say in the range $+5$ to $+15$ volts. A 0 is traditionally called a "space" as opposed to the "mark." The voltage levels in either case must be held at those levels for a specified period of time (a "bit" time). Bit-signal separation time is also prescribed. When the bits of a byte have been shipped out, the time interval before sending the first bit of a new byte is sent out must exceed a given duration. It suffices to understand how bits flow in a simplex system; all the other circuits can be understood as supersets of simplex circuits.

How does the receiver R on a simplex circuit know that a byte is being sent by the sender S? Every ASCII code will be preceded by one mark bit. The receiving hardware at R is continuously monitoring the communication line, looking for what may be a "mark" voltage level. If we were mentally simulating what the R hardware is doing, the script would read as follows:

1. Am I seeing a voltage level which represents a "1"? No; so keep looking; go back to step 1.

Maybe; let's keep looking for a few milliseconds; and if the level is still holding in the right range, go to step 2. If not, let's write it off as noise and resume step 1.

2. At the instant the presumed initial mark bit ends, let's start our timing circuits to expect a fresh bit every 10 milliseconds and begin counting the bits. When the appropriate count is reached, and nothing but nice mark and space signals were seen, at the expected times, present the byte to the Receiver's processing unit, and resume step 1.

The process of transmitting an ASCII character can be illustrated by showing how an ASCII "A" would be transmitted:

```
Character                "A"
ASCII code               101 octal
  "    "                   1 000 001 binary
With odd parity            11 000 001
Add framing bits        11 11 000 001 0
                  Stop      ...      Start
```

The communications hardware then converts this bit stream, beginning with the start bit, into the voltage levels depicted in figure 10.6.

The general case is illustrated in figure 10.7. Transmission of the ASCII code bits is preceded by sending the start bit, then it goes from the least significant bit to the most significant bit, followed by the parity, followed by the stop bits.

By convention, not only is each information byte preceded by a space bit, which acts as a "start" bit, it is also followed by one or more "mark" bits acting as "stop" bits. These extra bits in front of and behind each ASCII byte transmitted are called "framing" bits. They are totally transparent to the user. Your program never sees them; you never create them. The communications hardware puts them in at the transmitting site and strips them off at the receiving site. These start-stop bits make it possible for R and S to communicate without a common clock. They each need clocks of course, since timing is of the essence here, but they need not be synchronized so long as they each measure time intervals with sufficient accuracy. If you can communicate without clock synchronization (usually provided by having a common clock and special wiring just to share the clock, an expensive proposition) you have a "self-clocking" system. We refer to these as supporting *asynchronous* communication. If the framing bits are totally transparent to computer users, why drag them into our discussions? Well, how do you explain to someone that a 110-baud transmitter (in computing, the word *baud* is used to designate bits-per-second) sending 10 characters per second is using a 7-bit ASCII code with a parity bit? At 8 bits per character, and 10 characters per second, you can account for only 80 baud! Where did the other 30 baud go? Enter the framing bits: at 110 baud, there are 3 framing bits per character, 1 start bit, and 2 stop bits. Thus, each ASCII code results in 8+3 bits being transmitted. At 10 cps, this comes out to 110 baud.

At higher baud rates, 300 baud and up, only 2 framing bits are used: 1 start bit and 1 stop bit. So a 300-baud ASCII terminal can send or receive 300/(2+8), or 30 cps.

Keep in mind that the framing bits are tacked on and removed by the serial communications hardware. A user's program does not see these bits, even though they clearly influence the data communications rate.

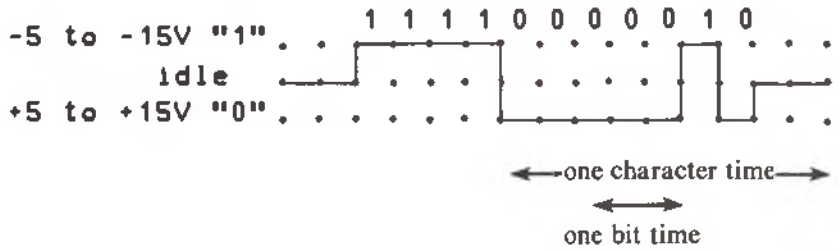


Figure 10.6 Sending an ASCII "A."

Figure 10.7 Bit transmission sequence.

Your programs might or might not ever see a nonzero parity bit, depending on the communications interface software handler being used. Some handlers make the parity handling transparent to you, so you always see the parity bit as having the value 0. Other handlers pass on to you whatever they found as the eighth bit.

The voltage-level conventions, framing-bit conventions, etc. that we have used here are those associated with the most frequently used serial-data communications standard, which goes under the name RS-232C.

Baud and Bits/Second

The name "baud," in honor of Emil Baudot, is used to represent transmission rates in both the computing business and the telecommunications business. The last group used it first, and we have to appreciate how their definition is slightly different from common use in computing. In telecommunications, a "baud" is the number of signal events per second. It might be so many light flashes per second (which we would view as being identical to bits per second). It might be the rate at which the positions of a flag used in ship-to-ship communication change. In this case, since a flag can have several positions (up, down, left, right, top-right quadrant, etc.), the bit rate may be considerably higher than the baud rate. Whenever a "signal event" is capable of conveying more than one bit of information, baud and bps (bits per second) are no longer equivalent.

In most instances in computing the baud rate is equivalent to bits per second. We will discuss an important exception when we get to telecommunications.

We have dwelt on serial communications in this section, where we transmit our bytes or words bit-by-bit, one bit at a time. Clearly you can transmit many more bits per unit of time if you transmit several of them at a time. This involves using parallel communications such that if you wish to send n bits at a time, you need n pairs of wires. We will discuss the use of parallel interfaces later.

Caution. As we proceed to finally discuss how the PDP-11 performs I/O operations in the next chapter, keep in mind that many problems encountered in learning I/O stem from not understanding what the I/O device itself is capable of doing; what switch settings it is capable of accepting; what their significance is; and so on.

Summary

Character-oriented devices play an important role in computing. Almost all the interactive devices we work with are character oriented. Punched cards, paper tape, printers, and other devices are also character oriented. Of all the character-oriented devices, the CRT has come to play a central role. We briefly examined the characteristics of a composite CRT. We stressed the importance of distinguishing between the encoded keys and keys such as Shift, Control, etc.

Simple communications schemes used to link CRTs and other devices to each other as well as to computers have been discussed. The relationship between simplex, half-duplex, and full-duplex communications has been developed. The role of the parity bit and framing bits and their relationship to the common measure of transmission speed, the baud, have been defined.

Prerequisite to understanding our next topic—computer input and output—is a clear understanding of what devices can or cannot do, and how they are linked to a computer.

Exercises

10.1 If you are using a CRT, perform the following experiment. Put the CRT in local mode (i.e., off-line or not-remote). Type a ctl-g. What happens? See how long a line can be (how many characters?). How many lines can be displayed? What is the response to each of the following?

(a) ctl-g
(b) ctl-h
(c) ctl-i
(d) ctl-j
(e) ctl-k
(f) ctl-m

Can you, without using the ctl key, locate keys which evoke any similar responses? After this experiment, put the CRT back into on-line mode.

10.2 If you are using a hard-copy interactive terminal, put it in local mode and see how it responds to the ASCII control characters such as Vertical Tab, Horizontal Tab, Form Feed, Line Feed, Carriage Return, and Bell. Does a carriage return force a line feed? What are the line length and printing speed?

10.3 Test the operating system you are using in the following ways.

(a) Does it provide any commands which permit you to
  (i) determine what information the operating system has about your terminal?
  (ii) change such characteristics as you make the corresponding physical changes to your terminal (e.g., switch from 300 baud to 110 baud, from full-duplex to half-duplex, etc.)?

(b) Use the command for (i) above to see what the system knows about your terminal. Use the command for (ii) above to temporarily change the following:
   (i) your duplex mode
   (ii) your "erase" character
(c) Account for the behavior you observe when you tell the system your terminal is in half-duplex mode but you don't actually switch the terminal's mode into half-duplex. What should happen if the situation were reversed?

10.4 Write a program that translates from Morse code to ASCII and from the corresponding ASCII subset to Morse. The program must use a table which represents each of the codes for Morse and use a bit-count field.The Morse code is given in figure 10.8.

10.5 Write another Morse-ASCII translator but do not use a bit-count field. You may use one 16-bit word for each Morse character.

10.6 Using either of the techniques from the previous two exercises, generalize your program so that it may accept or generate all printing ASCII characters. Carefully describe the mechanism you are using for mapping a rich character set into a poor one, and vice versa.

10.7 IBM computer systems have traditionally (since the first IBM 360) used the 8-bit EBCDIC code. Find a reference which gives you the EBCDIC code. Compare EBCDIC to ASCII. Then answer the following questions for each code set:

(a) How many printing characters are supported?
(b) How many nonprinting characters are supported?
(c) Which printing characters are unique to each set?
(d) Does either code support some unique function?
(e) Are the codes for alphabetic characters contiguous (i.e., do they follow each other without intervening nonalphabetic codes)?
(f) Are the codes for alphabetic characters in lexicographic order?

10.8 Outline a program which could perform ASCII-EBCDIC conversion. What is the trickiest part, if any?

A .-	B -...	C -.-.	D -..	E .	F ..-.
G --.	H	I ..	J .---	K -.-	L .-..
M --	N -.	O ---	P .--.	Q --.-	R .-.
S ...	T -	U ..-	V ...-	W .--	X -..-
Y -.--	Z --..				
1 .----	2 ..---	3 ...--	4 ....-	5	
6 -....	7 --...	8 ---..	9 ----.	Ø -----	
Period .-.-.-			Question mark ..--..		
Colon ---...			Semicolon -.-.-		
Double dash -...-			Slash (/) -..-		
End of message .-.-.					

Figure 10.8 International Morse code.

10.9 Suppose you are sending visual messages from ship to ship, using flags. Your flagman has a red and a green flag (one in each hand). The legal flag positions are: up, down, left, right, hidden. Each flag operates independently. It takes two seconds to move from one flag setting to the next.

(a) What is the baud rate?
(b) What is the information transmission rate?

10.10 If a CRT has a 24-line-by-80 character display, how long does it take to fill the screen at the following speeds?

(a) 300 baud
(b) 1200 baud
(c) 9600 baud

10.11 If you are using a CRT:

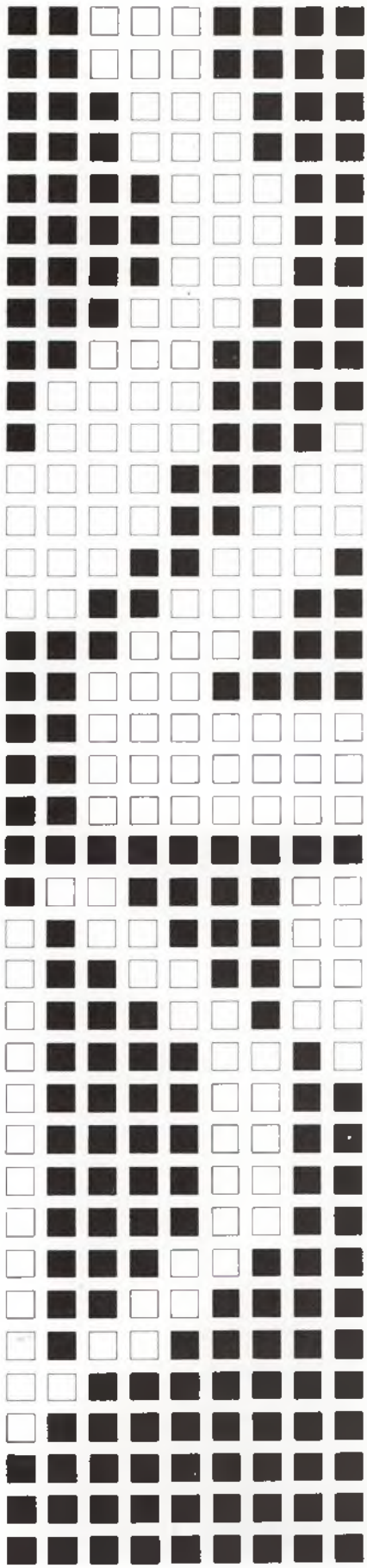
(a) What speed is it set at?
(b) What is the shortest time to completely fill the screen?
(c) How large a variance do you actually experience? Explain why this is so.

10.12 What is the parity of each of the following numbers?

(a) 132 octal
(b) 101 binary
(c) 1011 binary
(d) 101 decimal

11

character-oriented input and output



The input and output operations visible to an assembly-language programmer on a multi-user system are not much different from those seen by users of high level languages:

```
Read a line, or
Read item 1, item 2, ...
Get a character
...
Print a line, or
Print item 1, item 2, ...
Put a character
```

In a high level language a great deal of support software (the run-time I/O library) is provided so that I/O is "made easy." If you are working on a bare machine, I/O seems much more complicated. Yet it really is not.

We tend to lump together two closely related activities: (1) the transformation from external to internal representation, or vice versa; and (2) the actual operation of getting bits into a computer, and vice versa, for output.

If we treat the data-representation transformations as the separate problem they really are, I/O suddenly seems a lot simpler. It is, except for some entirely new phenomena which we will get to shortly. Let us begin by looking at the hardware involved in linking a CRT with a computer. First we see the layperson's simple view of such a link (the "executive's view") depicted in figure 11.1.

Our view includes a few more details, as seen in figure 11.2.

A CRT can be connected to a computer by installing a *serial interface controller* in the CPU. This interface hardware communicates with the CRT by means of two or more wires to handle the bit-serial input of characters from the CRT, and two or more other wires to handle the bit-serial output of characters to the CRT. Note that from now on our world becomes CPU-centered. The word "output" will mean "output-from-the-CPU-to-..." unless it is otherwise qualified. Similarly, "input" means input for the CPU. A minimum of four wires will support full-duplex communications. More wires are desirable to provide other useful functions; some of these will be discussed in the section on "Telecommunications, Teleprocessing" in chapter 17.

Figure 11.1 Layperson's view of an I/O link.

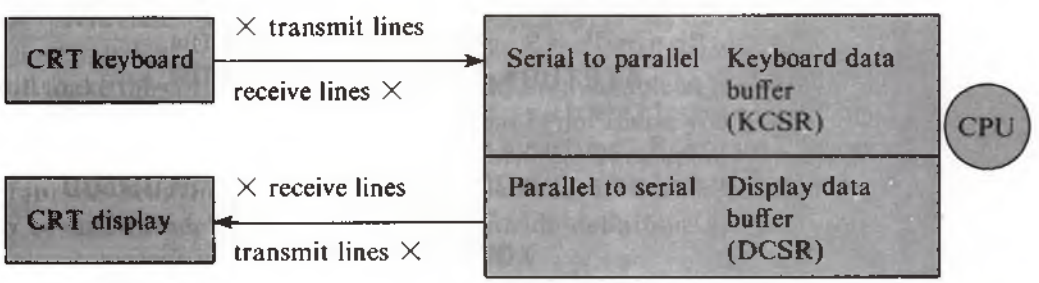


Figure 11.2 View of communications interface.

Data Registers, Control and Status Registers

The computer side of the serial interface involves many more wires because it uses byte-at-a-time transfers, which are bit-parallel, not bit-serial. The interface hardware makes two special pieces of information accessible to the CPU by supporting two kinds of register: (1) data-buffer register (DBR); and (2) control and status register (CSR).

The data register, often called a buffer register, usually has a 1-byte capacity. When the CPU wishes to transmit one character to the CRT, the CPU deposits the character's 7- or 8-bit code in this data register (we will refer to it as the DBR). Conversely, a character sent from the CRT must find its way into the DBR if the CPU is ever to access it. It seems we may have a conflict here.

Since the full-duplex CRT and the full-duplex circuit permit simultaneous two-way communication between the CRT and CPU, a data-buffer register must be provided for each of the send and receive functions. Since the send and receive functions are not necessarily synchronized, each of them also requires a control and status register. In the following sections the abbreviations KCSR and KDBR will be used to refer to the control and status register and the data-buffer register, respectively, when associated with a device capable of sending bytes to the CPU, such as a keyboard. Similarly, the abbreviations DCSR and DDBR will be used to refer to the corresponding registers associated with a device which is capable of receiving bytes from the CPU, such as a CRT display. In those cases where we need not distinguish between the sending and receiving functions, the shorter abbreviations CSR and DBR are used.

The incoming bit stream goes through a serial-to-parallel conversion unit, which accumulates an 8-bit byte in the KDBR. The parallel-to-serial conversion unit likewise shifts out the 8-bit byte presented to the DDBR by the CPU and transmits them in bit-serial fashion over its transmit lines. Note that the interface's transmit lines connect to the CRT's receive lines, and vice versa. A cable which connects the CRT to the interface must provide for this crossover of s-to-r and r-to-s. This is quite different from extending a short cable by inserting an extension cable. The extension cable would then have to provide a "straight-through function." If it also provided for crossover, we would have an amusing situation.

The CRT and the CPU should be using the same display code if useful interaction is to take place. This does not preclude connecting an EBCDIC terminal to the CPU with one interface and an ASCII CRT with a separate but identical interface. The code conversion can be handled as a CPU software function. The CRT and the interface it is connected to must be operating at the same baud rate. For hard-wired terminals (the ones permanently connected to a particular CPU), this is not a problem, for the speed-selection switches on the CRT and its interface can be set to match and be left that way. For non-hard-wired terminals—that is, for telephone dial-up lines, speed detection must be accomplished somehow so that the interface can set its baud rate to match that of the dial-up terminal. We will discuss speed detection in the telecommunication section. Note that the CRT's send-baud-rate and its receive-baud-rate can be different. That is a freedom allowed over full-duplex lines.

Input Output Instructions

The traditional approach to implementing I/O functions in hardware was to provide a special set of instructions for I/O. It can be assumed that all I/O devices are known to the CPU by device number. To send a byte out of the CPU to some device, we can use a hypothetical command such as:

```
ODB %1,7 ; hypothetical output inst.
```

This could mean "transmit an output-data-byte from CPU register 1 to device 7's output data register." Once a byte gets to an output data register, the device's interface assumes responsibility for having the bits sent to the device. The actual data transfer, from the CPU's register to the device's DDBR, can be accomplished very quickly. It should not take more than a few microseconds, since it is essentially a register-to-register transfer.

Conversely, a command such as:

```
IDB %2,6 ; hypothetical input inst
```

could input a data byte from device 6's KDBR and copy it to %2. What if device 6 was still transmitting the byte when you executed the IDB? Conversely, what if you executed an ODB to device 7 while it was still transmitting the previous byte? Both of these situations are possible, and we have to avoid them, somehow.

The problem we just alluded to is called "data overrun." It results in "bad" data, or data loss. It can usually be avoided by using the device's CSR to check the device's status. If the device is still busy transmitting a byte, and you (the CPU) want to send the next byte, then you can either wait until it becomes nonbusy (i.e. "ready" for more work) or go away and do something else for a while before you come back and check again. The DCSR has a "busy/done" bit which is automatically set (by the hardware, not your program) when the CPU loads the device's data register with a new byte to be output. The device's transmitter will clear this "busy/done" bit in the DCSR as soon as the last framing bit is shifted out. The interface is then ready to accept a new byte for output.

The KCSR similarly has an "empty/full" status bit. When a byte is received from a device, the hardware automatically sets the "empty/full" status bit for that device's CSR. The act of fetching the new data byte from the interface's KDBR, so a program can process this new byte, automatically clears the "empty/full" status bit.

A beginner's attempt to use the traditional hypothetical instructions ODB and IDB could lead to the following code segment:

```
        MOV     #HELPM,%1
NEXT:   MOVB    (%1)+,%2
        BEQ     DONE
        ODB     %2,6 ; not a PDP-11 instruction
        BR      NEXT
        ...
HELPM:  .ASCIZ  /HELP/
```

This is intended to send out the message "HELP" if ODB was a PDP-11 instruction. Unfortunately the message received will be very garbled. No sooner is the first framing bit for the "H" on its way to device 6, than the CPU overwrites the partially shifted bits for "H" with the code for "E," and so on. The problem could be resolved by using something like:

```
        MOV     #HELPM,%1
NEXT:   ISW     %3,6 ; input status (hypothetical inst)
        BMI     NEXT
        MOVB    (%1)+,%2
        BEQ     DONE
        ODB     %2,6    ; hypothetical instruction
        BR      NEXT
        ...
```

The "ISW" instruction (a hypothetical one) could fetch the content of DCSR for device 6 and copy it to register 3. Assuming its high bit is set while transmission (bit-serial) is in progress, the two-instruction loop

```
NEXT:   ISW     %3,6    ; hypothetical instruction
        BMI     NEXT
```

could synchronize the CPU's ultra-high speed with the device's generally much slower transmission speed. This very tight loop is called a *busy-wait* I/O loop. The CPU does nothing productive during this time. If you took this two-instruction loop "out-of-context," it is very puzzling. It is an infinite loop. Neither of the two instructions involved changes anything (except that the ISW eventually changes the N bit, you hope). Obviously some external agent is responsible for updating device 6's CSR. That agent is its interface controller.

Many computers have instructions similar to the ISW, ODB, and IDB in order to perform I/O. Many have no I/O instructions of any kind; the PDP-11 is one of these. Fortunately we have not been wasting our time looking at the above "traditional" approach. Everything we have done has a counterpart on the PDP-11.

I/O without I/O Instructions

In all our previous work with the PDP-11 we have dealt with addresses of only two kinds: memory addresses and register addresses. The mode bits in instructions capable of using either type of address keep the distinction clear.

If you expect to attach a number of I/O devices to a computer, it seems reasonable to assign a device identification number to each one of these devices. Such device numbers could be thought of as a new kind of address. Rather than use this approach, and rather than have a special class of instructions just for I/O, the PDP-11 partitions its address space so that addresses below 28KW correspond to ordinary memory addresses, and those above 28KW correspond to the I/O device register addresses reserved for registers specifically associated with particular I/O devices.

This provides 4KW of I/O address space, as illustrated in figure 11.3. Instead of assigning a device a single address on the PDP-11, each of its interface controller registers will be assigned a word address in the 4KW region shown in figure 11.3. This approach to supporting I/O has been adopted by many microcomputers. It is known as memory-mapped I/O.

A typical assignment of I/O addresses for a device such as a CRT is shown in figure 11.4. As a general rule, the device buffer registers will be assigned word addresses even if they support only byte data. If a device is a receive-only device, it would be assigned only two addresses (its CSR and DBR addresses); similarly for a send-only device. Device register addresses are usually contiguous, with send-receive devices having the Input-CSR, DBR first, followed by the Output-CSR, DBR.

We can now write a "real" PDP-11 program, figure 11.5, which performs a useful I/O function. This code segment will display the message "HELP" on the CRT whose interface has the specified device register addresses. The implication here must be that bit 7 of a device's CSR provides the "busy/done" status. As long as the low byte of the CSR appears positive, we will stay in the busy-wait loop. Presumably when the device's transmitter is done, bit 7 will be set, and the code segment will then proceed to load another data byte for transmission, which has the side-effect of clearing CSR bit 7, and so on. We have to be careful in specifying the device register addresses as absolute addresses using @#.

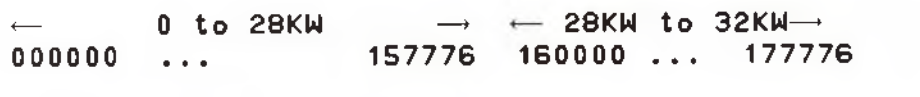


Figure 11.3 PDP-11 memory-mapped I/O addresses.

Keyboard control & status register	177560
Keyboard data buffer register	177562
Display control & status register	177564
Display data buffer register	177566

Figure 11.4 Device register addresses.

```
       DCSR = 177564 ; CRT display CSR
        DDBR = DCSR+2 ; CRT display   DBR
        MOV     #HELPM,%1
SEND:   TSTB    @#DCSR     ; test for busy
        BPL     SEND     ; busy-wait loop
        MOVB    (1%)+,@#DDBR      ; send a byte
        BNE     SEND     ; try sending next byte
        ...
HELPM:  .ASCIZ  /HELP/
       ...
```

Figure 11.5 Output code.

Figure 11.6 I/O echo code.

```
            KCSR = 177560    ; input CSR
            KDBR = KCSR+2    ; input DBR
            DCSR = KDBR+2    ; output CSR
            DDBR = DCSR+2    ; output DBR
;
INPUT:      TSTB     @#KCSR     ; is Kbd buffer empty?
            BPL      INPUT    ; if empty, try again
;
OUTPUT:     TSTB     @#DCSR     ; is display busy?
            BPL      OUTPUT   ; if busy try again
; echo new keyboard input
            MOVB     @#KDBR,/@#DDBR        ; in → out
            BR       INPUT
```

Let us examine the program in figure 11.6—a code segment which performs input and output. This is the kind of test program one uses when trying to determine whether a CRT, its interface, the CPU, or a cable may be responsible for a communications failure.

It is important to know clearly who does what and when to either a device's DBR or CSR in order to understand how I/O really works.

Input CSR

1. User strikes a key on CRT keyboard
2. Serialized bits in transit
3. KDBR being filled
4. KCSR "done" bit 7 is set when KDBR is filled
5. MOVB KDBR, d triggers 0 → KCSR bit 7

Output CSR

1. Device transmitter shifts bits out of DDBR to a device and sets "ready" bit 7 in DCSR when transmitting completed
2. Program filling DDBR (say, with a MOVB s, DDBR) triggers 0 → bit 7 of DCSR

A perceptive reader may wonder why it is necessary to have two busy-wait loops in the echo program. This clearly is necessary when the send and receive baud rates are different, as is often the case in data processing systems. A 300-baud keyboard input rate and a 1200-baud display output rate are not an unusual combination. Clearly, if the sender goes faster than the receiver, or vice versa, one of them has to wait for the other.

What if the send and receive data rates are identical? Recall that in an asynchronous communications link the sender and the receiver do not have a common clock. So slight variations in the data rate could occur, leading to inconsistencies. It is wiser to be safe than sorry. We do need to worry about how an interface reports to you that it just received "bad" data, so we now must take a closer look at the CSR.

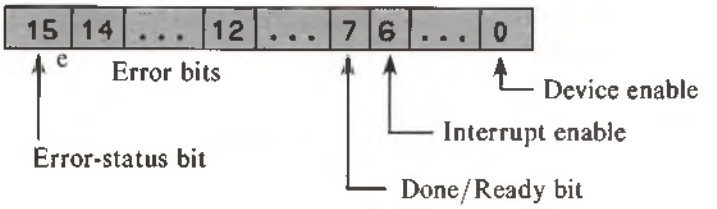


Figure 11.7 Typical-control & status-register bit meanings.

Error bit	Meaning
12	Parity error on received character
13	Framing error; STOP bit not found
14	Data overrun
15	Error status

Figure 11.8 Error status bits.

Control and Status Bits

The CSR has two functions. We have already seen part of one of these functions, dealing with status information (i.e., "ready" bits, etc.). The other function pertains to accepting commands from the CPU. With very simple devices such as a CRT, there are not many commands to present to the CRT's interface controller. With the more sophisticated devices we will examine later, there are so many command variations possible that a single CSR no longer suffices. The typical CSR for a simple device has the layout shown in figure 11.7. Bit 15, the error-status bit, is the logical OR of bits 12 through 14. These bits report specific error situations. For a serial interface input CSR, their meanings are given in figure 11.8.

The CSR registers are unusual in that some bits are "read only" and others are "write only." No user or program ever sets or clears bit 7; it is always set or cleared by the hardware as a side effect of sending or receiving. Only the interface controller hardware and some CPU hardware can set/clear this bit; if you try a CLR DCSR, it will have no effect on bit 7.

The "write-only" bits are set or cleared by software. A TTY's CSR has its bit 0 as a write-only bit. If you try to look at it, even immediately after you set it, it will always appear to have the value 0. That being the case, an INC KCSR has the strange property of always setting bit 0 of (KCSR). This is used as a signal to an ASR TTY to have its paper tape reader read one more character, then stop.

It is unfortunate that the read-only/write-only distinctions appear in the CSR. To save/restore or simulate all the states a computer system could conceivably get into is thereby made more difficult.

Overlapped I/O and Processing

The necessity to synchronize input with output, or input with processing, or processing with output, is tricky and time consuming. The time spent in doing

```
IN:     TSTB    KCSR    ; busy-wait loop
        BPL     IN
```

might have been used more productively.

Consider an application which processes one data item per punched card, and suppose that the various timings shown below apply:

1. A card reader operating at 1,000 cards/minute (60 msec/card)
2. Processing time for data on one card: 50 msec
3. Printer operating at 600 lines/minute, one line/card: (100 msec/line)

The simple way to handle this application is:

Read card 1, process it, print result.
Read card 2, process it, print the result.
Read ...
...

If we let R be the time to read one card, P the time to process the data on it (the compute time), and O the time to output the result (the print time), then the total time to process three cards is:

$$(R+P+O)+(R+P+O)+(R+P+O) \rightarrow 3(R+P+O)$$

For n cards, the total time is $n(R+P+O)$. Suppose we had 10,000 cards to process; then the total time would be $10{,}000\ (60+50+100)=10{,}000$ (.210 sec) = 2,100 seconds = 35 minutes.

What if we could arrange things so that, while results from the data for the first card were being printed, the data from the second card were being processed, and the third card was being read, all simultaneously? If we could do this systematically, we could have the following time line:

Read 1	Read 2	Read 3	Read 4	Read 5	...	
	Proc 1	Proc 2	Proc 3	Proc 4	Proc 5	...
		Outp 1	Outp 2	Outp 3	Outp 4	Outp 5 ...

Since the read, process, and output times are 60, 50, and 100 msec, respectively, the time to process 10,000 records will be approximately 10,000 (max of (60, 50, 100)), or 10,000*100 msec, or 1,000 seconds, or 17 minutes—which is about twice as fast as the time taken when no overlapping of I/O and processing took place.

It is possible to synchronize I/O and processing by means of busy-wait loops; but it is complicated. A better technique has evolved, which can be described in terms of "management by exception." The management technique known as "management by exception" can be paraphrased as: "Don't bother me unless something important happens." In computing terms, it translates as follows:

1. The CPU instructs a device to read a card and to let the CPU know when the card has been read.
2. The CPU instructs the printer to print a line and to let the CPU know when the line is printed.
3. The CPU instructs device three to display one character and to let the CPU know when that character has been displayed.
4. The CPU instructs device three's interface to let the CPU know when device three has a new input character ready for the CPU.

You can appreciate item 4. Instead of waiting (perhaps indefinitely) for a key to be struck on device three, the CPU initializes the device's CSR appropriately (which is how the CPU "instructs" a device), and then the CPU goes on to other activities.

The actual mechanism which supports this process by which an I/O device informs the CPU that "something important has happened" is called *interrupt handling.*

Interrupt Handling

The CPU can instruct a device to "interrupt" the CPU by setting the *interrupt enable* bit in the device's interface controller CSR. This is usually in bit position 6 in the CSR. Since each device has a unique CSR associated with it, we often speak loosely and say "the device is enabled for interrupt," when, of course, it is the interface controller which actually is enabled for interrupt.

It is understood that the device will not interrupt the CPU until something important happens. Usually the important event is the normal completion of the output operation initiated by the CPU, or availability of input anticipated by the CPU. Occasionally the important event will be reporting that the expected I/O activity did not complete normally; it may have completed, but an error was detected (e.g., a parity error) or it may have aborted in midoperation (e.g., the card reader jammed, the printer ran out of paper, etc.).

Processing an Interrupt

Suppose you have set the interrupt enable bit (bit 6) in the CSR for keyboard input from a CRT. When and if a key is pressed on that CRT's keyboard, if the CRT is not in local mode, and the communication line to the CRT's interface on the CPU is in working order, and that key generated a code to be transmitted, then, when the interface receives all the bits transmitted from the CRT, it will request an interrupt. That is, the interface will signal the CPU that a device needs service.

If the CPU "honors" the interrupt service request, the CPU will save what it needs to remember in order to resume whatever it is currently doing, and the CPU will fetch two items which lead it directly to the device's interrupt service software handler. What must be saved in order to resume execution at a later time? No less than what is saved in calling a subroutine, certainly. The PC must be saved. In addition, the PS will be saved (that is the simplest way to save the CC bits). If the CPU honors an interrupt request, the control unit automatically pushes the current PC and PS values into the system stack. The control unit then immediately fetches two words associated with the interrupting device and loads these into the PC and PS. Giving the PC a new value has the effect of forcing a jump to the desired interrupt service handler.

At the time a hardware interface for a new device is installed, it is necessary to designate addresses in the 28KW–32KW address range for the device's data-buffer register and its control-status registers. It is also necessary, if the device is ever to interrupt, to specify an address for a two-word block specific to that device. This address is called the device's *interrupt vector address.*

Figure 11.9 Some interrupt vector addresses.

Address	Content
60	New PC value, for input interrupt
62	New PS value, for input interrupt
64	New PC value, for output interrupt
66	New PS value, for output interrupt

Figure 11.10 Simple instruction fetch-execute cycle.

1.	Mem(PC) → IR	Fetch instruction
2.	(PC)+2 → PC	Update PC
3.	Execute (IR)	Execute instruction and go to step 1.

Figure 11.11 Instruction fetch-execute cycle with interrupt handling.

```
0.   Any interrupts requested? If yes, go to
     step 4.

1.   Mem(PC) → IR    Fetch instruction.

2.   (PC)+2 → PC     Update PC.

3.   Execute (IR)    Execute instruction
                     and go to step 0.

4.   Is the device's priority high enough?
     No: ignore request, resume step 1.
     Yes: push PS, PC in system stack, and
          copy (IV) → PC, (IV+2) → PS,
          and resume step 1.
```

We can finally resolve completely why it was necessary to avoid using the lower part of memory on a single-user system. The interrupt vectors are placed in the lower part of memory. Locations 60 and 62 are normally used for the console's hard-copy terminal input-interrupt vector; locations 64 and 66 for its output-interrupt vector, etc.

The PDP-11 hardware assumes that, if an interrupt for a device is honored, the device's interrupt vector has previously been initialized so that the first word of the word pair holds the value to be loaded into the PC, and the next word holds the value to be loaded into the PS. For the console device, we would have the interrupt vectors shown in figure 11.9. What does it mean to say "the CPU has interrupt-handling hardware"? We have to reexamine the instruction fetch-execute cycle to answer this question. As we saw this cycle a long time ago, it appeared as shown in figure 11.10. The purpose of having interrupt-handling hardware is to avoid having to execute the instructions to keep asking, "Is this done?" or "Is that still busy?" etc. So the hardware is modified to automatically keep asking these questions for us, far more often than we could ever imagine doing it ourselves. The innermost part of the control unit is modified so that the instruction fetch-execute cycle now appears like that shown in figure 11.11. The device controller's interrupt vector address is IV. Obviously, if there were several devices capable of interrupting the CPU, the hardware would choose the appropriate IV address.

Some computers funnel all interrupt requests through one special memory location; it is then necessary for the software to poll each device to find out which one just asked for interrupt service. The purpose of the interrupt vectors is to eliminate this polling, since it is time consuming. To "vector" (meaning "go in this direction") directly to the interrupt service handler uniquely associated with the interrupting device means that you can respond to interrupts faster.

What if several devices send requests for interrupt service at the same instant. That literally means within a few microseconds of each other. This is not as farfetched as you might think. Sooner or later, if several devices are active on one system, simultaneous interrupt service requests will occur. How do you resolve the potential conflicts?

Interrupt Priorities

Any well run organization sets priorities. In day-to-day activities most people would interrupt reading the day's mail if there were a knock on the door. While talking to whoever knocked on the door, some people would consider interrupting that conversation if the telephone rang. After the telephone conversation was completed, you would presumably resume talking to the knocker (possibly subject to another telephone call interruption). Afterward you would finish reading the mail. And finally, you would get back to what you were supposed to be doing. If you get too much mail, or too many visitors, or too many telephone calls, you may never get anything else done. This is known as interrupt overload, and it can happen to computers as well. It means that your priorities are wrong, or your interrupt handling is too slow, or you need a faster CPU with better hardware to offload some of the interrupt-handling overhead.

Single-Level Hardware Priority

Some simpler computers have a one-level priority scheme: a device's interrupt service priority is determined by its physical proximity to the CPU. This distance would be measured by looking at the interrupt-request line that every device capable of generating interrupts is connected to. The nearer the device's connection to that line, the higher its priority in the event of a conflict due to simultaneous interrupt requests.

Consider figure 11.12. The interrupt-request logic box (IRL) determines which one of the devices that may simultaneously request interrupt service on the interrupt-request line will have its request passed through to the CPU. The presumption in this diagram is that D3 will override a simultaneous request from D2 and D1. Similarly, D2 prevails over D1, if D3 had no request at that instant.

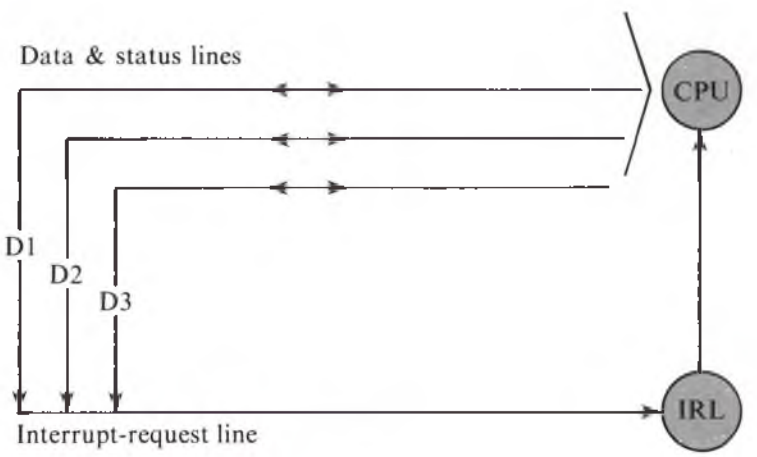


Figure 11.12 One-level-interrupt priority system.

What happens if the lowest priority device here, D1, has its request for an interrupt deferred because D2 or D3 also needed interrupt service at the same instant? In a well-designed system which is properly programmed, device D1's interrupt request will be considered pending, and it will be honored when all higher priority interrupt requests have been honored. If this takes too long, the reason for D1's original interrupt request may be superceded by more recent activity at D1. Suppose D1 is a CRT; the CPU happens to be very busy when you type an "A." It would not respond immediately to D1's interrupt request if it is still taking care of D2's or D3's interrupt requests. Since D1 is a full-duplex CRT on a full-duplex line connected to a full-duplex interface, you feel free to "type ahead," and you strike the "B" key. It is conceivable that the code for "B" might override the code for "A," which was sitting in D1's interface controller data buffer register, waiting for the CPU to fetch it as soon as its interrupt request was honored. In this case the interrupt request to process "A" is lost as the request to process "B" overwrites the older pending request. The interface's CSR will report data overrun error. Should this happen too often, it indicates that the whole computer system is in trouble, and it needs to be corrected. It may be indicative of a system for which the workload has grown beyond projections made when the system was designed and implemented. This may be the problem affecting the worldwide military command and control system known as WIMICS, as reported in the news item in figure 11.13.

Interrupt Handlers; RTI

When the CPU honors an interrupt request, it in effect forces a subroutine jump, the destination address being provided by the device's interrupt vector. In order to maintain transparency, the PS is preserved, in addition to the PC.

When the interrupt has been serviced or handled by the interrupt service handler, an RTS instruction is not quite enough to resume the processing that was interrupted. The instruction RTI is used to exit from an interrupt service handler. It pops the system stack into the PC and pops it again, restoring the PS.

Clearly, an interrupt service handler could be less than transparent if it failed to save and restore something it used. In particular, each interrupt service handler is responsible for saving and restoring any registers it uses.

Multi-Level Priorities

PDP-11 computers support a four-level hardware interrupt priority mechanism (except the LSI-11, which has only one level of priorities) and an eight-level software interrupt priority mechanism.

For the hardware (or hard-wired) priorities, devices are placed in four priority groups or levels:

level 7: top priority
level 6: second priority
level 5: third priority
level 4: lowest priority

War exercise leaves U.S. with black eye

WASHINGTON—The Defense Department has experienced another fiasco in its attempts to mobilize itself and the rest of the federal government for a major war.

In an exercise simulating wartime mobilization called "Proud Spirit," which ended Nov. 26, the Pentagon and 35 other federal agencies that would participate in a wartime mobilization experienced several severe shortcomings, according to sources who participated. They include:

- A major failure of the computerized worldwide military command and control system that left military commanders without essential information about the readiness of their units for 12 hours during the height of the "crisis" as Pentagon programmers found themselves locked out of one of their own computers.
- A shortfall of around 1 million tons of ammunition and military equipment that is supposed to be in the war reserve stocks in Europe. The shortfall was far beyond the military's capability to make up during the 20-day exercise.
- A shortage of 350,000 trained soldiers to fill up units leaving the United States and an inability to bring frontline U.S. Army units in Germany up to their authorized wartime strength during the exercise.
- Evidence that the U.S. industrial capability to resupply the Pentagon with basic items of military hardware, such as tanks and ammunition, continues to decline.
- Among the eight major federal agencies that would be needed to carry out a mobilization for war, not one department secretary responded to an invitation to spend two hours studying his agency's role in the scenario for Proud Spirit.

Proud Spirit and its civilian counterpart, code-named "Rex 80 Bravo," are an updated version of a 1978 exercise called "Nifty Nugget," the first simulated governmentwide war mobilization exercise since World War II.

Nifty Nugget exposed enormous shortfalls in munitions, equipment, manpower and planning. The results of its successor, Proud Spirit, indicate that while the planning has improved somewhat, the shortages, the confusion and the dismal results of Nifty Nugget have been re-enacted.

Defense Department officials would not talk about Proud Spirit, saying it was too sensitive because of the transition of administrations.

A memorandum of an Army briefing on the exercise suggests that the Pentagon has seen better days than those during the Proud Spirit exercise.

"There is a doubt in my mind, in view of our generalized objectives, as to whether we obtained success," the memo quotes retired Gen. Walter T. Kerwin, as concluding. Kerwin, former vice chief of staff of the Army, was the leader of a team of retired military officials overseeing the game.

"The exercise seemed to highlight the fact that we just have a long way to go," said Frank Camm, an associate director of the Federal Emergency Management Agency, organizer of the civilian agencies that played the game.

The emotional high point of the game, according to some of the players, was the failure of one of the major subsystems of the worldwide command and control system, believed to be the largest and most expensive computer system in the world. It "just fell flat on its ass," was the way one of them put it.

One of the roles of the system is to give top generals and admirals an up-to-the moment report on the readiness of their units. However, the computer system in charge of doing that during Proud Spirit became overloaded, so the updated information was temporarily stored in an interim memory bank called a "buffer."

When the time came for the buffer to feed the information into the worldwide system, however, it balked. The result was that the Army's manpower- and equipment-related computer systems went silent for six hours while programmers struggled to find the right code sequence to release the readiness information. That information surfaced 12 hours later.

After that the system went in fits and starts, providing some information a bit too late. One result was that Air Force transports were given simulated orders to land at military bases two days before the troops assigned to board the transports were told to move out.

At other times, however, the computers went the other way, inundating top officials with trivia.

There is some evidence suggesting that the Defense Department tried to avoid some of the problems of Nifty Nugget by making the crisis envisioned in Proud Spirit easier. Instead of the multiwar scenario used during the 1978 exercise, which included the opening sequence of a major war in Europe, Proud Spirit only envisioned rising tension in Europe and a resulting mobilization for one war.

Figure 11.13 An example of an overloaded computer system. Reprinted from The Washington Star, copyright reserved.

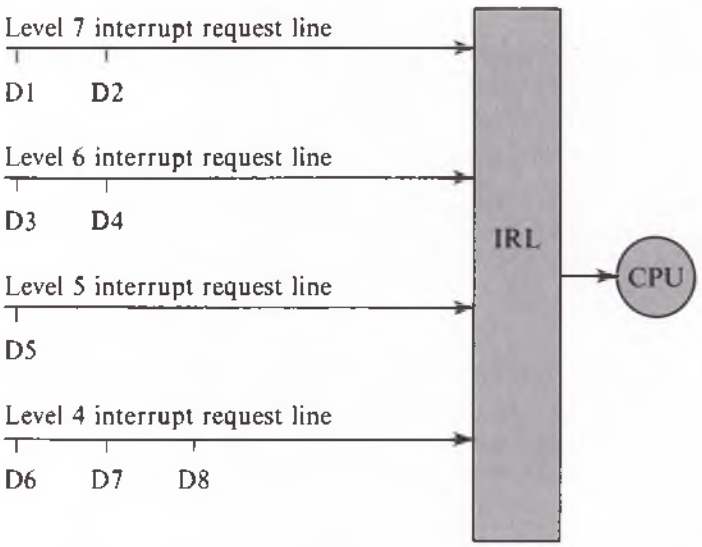
Figure 11.13 Continued.

Several other problems were reportedly assumed away. Unlike Nifty Nugget, the Proud Spirit mobilization assumed that huge amounts of ammunition and equipment had been shipped to Europe to replenish U.S. war reserve stocks which, according to well-placed sources, are now more than a million tons short of their authorized levels of ammunition and equipment.

The sources made it clear that the shortfall is not just the problem of the outgoing Carter administration, but the result of 15 years of neglect and borrowing during the Vietnam and 1973 Israeli-Arab wars. The only form of ammunition the Army has to satisfy its requirements for a NATO war is said to be rifle ammunition.

Another problem that surfaced during the game was that the capability of U.S. industry to replenish military arsenals with tanks, missiles, aircraft and ammunition continues to decline. It now takes longer to order a tank, for example, than it did during Nifty Nugget. One way Proud Spirit planners considered to meet a shortfall in M-16 rifles was to order them from a plant in South Korea.

Figure 11.14 PDP-11 four-level priority mechanism.



At any particular level, priorities are assigned as they were with the one-level priority mechanism we saw earlier. This leads to configurations such as the one shown in figure 11.14. Any device at a higher level can block a lower level device's interrupt request if they both appear simultaneously. For devices on the same level, the device nearest the CPU can block its neighbor's interrupt request if they are simultaneous.

All the priority mechanisms we have discussed so far are hardware priorities. They are set when the equipment is first hooked up, and usually they are not touched thereafter.

We now come to *software priorities*, which introduce an element of flexibility. The software priority is determined by a 3-bit field in the processor status word register PS; these are called the priority bits. If a device has a higher hardware priority than all other devices requesting interrupt service, it must pass one other test. Is the device's hardware priority level (4, 5, 6, or 7) *greater than* the software priority currently in the PS? If not, the interrupt request will be deferred. If yes, the interrupt request will be honored.

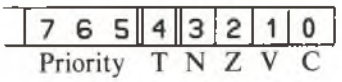


Figure 11.15 Processor status word (PS).

```
        MOV     #CLOCK,@#100        ; set locations 100
        MOV     CLKPS,@#102         ; and 102
        BIS     #100,@#177546       ; enable for interrupt
; continue computing
        ...
CLKPS:  .WORD   200         ; PS priority 4
CLOCK:  INC     TIME+2
        BEQ     UP
        RTI                 ; return from int.
UP:     INC     TIME
        RTI
TIME:   .WORD   0,0
```

Figure 11.16 Clock interrupt handler.

The process of honoring an interrupt request loads a new PS value from the device's interrupt vector. This provides a new 3-bit software priority value in the PS (the T bit will be discussed later). The software priority might be higher than, lower than, or the same as the device's hardware priority. An interrupt handler could inadvertently cause a whole system to shut down because no new interrupts were being honored due to the fact that the interrupt vector's PS priority field was accidentally set to 7! It is hard to imagine that a mere three bits can shut down a whole computer system (actually it only takes one bit).

Clock Interrupt Handler

The simplest example for a complete interrupt handler involves the simplest device found on a PDP-11, the 60 hertz clock. It "ticks" 60 times/second. If its CSR interrupt-enable bit in position 6 is set, the clock will request an interrupt each time it ticks.

The first part of the program in figure 11.16 will initialize the clock's interrupt vector, which is at location 100. It then enables the clock interrupt by setting bit 6, and it goes on to do other useful work. At any time one can look at locations TIME and TIME+2 to see the current tick count since the clock-enable was turned on. If, for some reason, the PS priority field was set at five or higher for a sufficiently long time, the counts in TIME would not be correct; they would be too low.

CRT Input-Interrupt Handler

Using the KDBR and KCSR addresses we had earlier, and the input-interrupt vector address for the CRT, we can write a simple input-interrupt handler which will place incoming characters in consecutive byte locations beginning at location BUF; it will set a flag in LINE when it encounters the end-of-line code 15. Someone else must look periodically to see if the LINE flag is set. That higher level routine would reset the LINE flag and the BUF pointer (in BUFPTR) when it finished processing the current line of text. Note that the interrupt handler does not test the CSR at KCSR to see if the data is in the DBR at KDBR. The very fact of being in the interrupt handler guarantees that either the data have just arrived in the KDBR or they won't arrive because an error has occurred.

Figure 11.17 Simple input interrupt handler.

```
KCSR = 177560    ; interface register adrs
KDBR = KCSR+2
KIV  = 60        ; interrupt vector adr
...
MOV     #KCRT,@#KIV ; initialize int vector
MOV     KPS,@#KIV+2 ;  init vector PS word
BIS     #100,@#KCSR ; enable interrupt bit 6
; start computing
...
KPS:    .WORD   300      ; PS priority 6
KCRT:   TST      @#KCSR    ; begin interrupt handler
        BMI     ERROR ; error bit was set
        MOVB     @#KDBR,@BUFPRT
        CMPB    @BUFPTR,#15 ; is it Return?
        BEQ     ENDLN    ; yes
        CMPB    @BUFPTR,#'!      ; is it "!"
        BEQ     BACKUP
        INC     BUFPTR
        RTI              ; return from int
ENDLN:  INC     LINE     ; set flag
        RTI
;
BACKUP: DEC     BUFPTR   ; treat "!" as a char erase
        RTI
ERROR:  ... check error bits
        RTI
;
BUF:    .BLKB   80. ; line buffer
BUFPTR: .WORD   BUF
LINE:   .WORD   0        ; flag
        ...
```

A more sophisticated handler would echo the input and check that a user does not accidentally back up too far by typing too many "!"s. You can see clearly how the "erase-the-last-character" function can be implemented, and changed. You can use registers in an interrupt handler, provided that the handler saves and restores them to maintain transparency.

Summary

The key to understanding I/O is to disassociate the actual transfer of data from any data representation transformations which may precede or follow the data transfer. On character-oriented devices, data is typically transferred one byte at a time. Many computers have special instructions for I/O; the PDP-11 has none. Ordinary PDP-11 instructions are interpreted as referring to I/O devices, instead of memory locations, by treating all addresses greater than 28KW as addresses of I/O device control and status and data registers.

Since data transfer speed of an I/O device is usually much slower than that of the CPU, the CPU can be forced to wait for I/O completion with a busy-wait loop. Interrupt-controlled I/O operations are provided to permit the CPU to continue performing useful processing while one or more I/O tasks are in progress. A device's busy or ready status can be determined by examination of a bit in its status register (usually bit 7).

An error associated with an I/O activity is also reflected in the device's status register (usually bit 15). A device is permitted to interrupt the CPU only if its interrupt-enable bit is set; this is usually bit 6 in the device's control register. Simple device controllers such as those for a CRT usually have a combined control and status register.

Adding an interrupt capability to a CPU involves a number of things. The instruction fetch-execute cycle must be modified so that it checks for interrupt requests prior to each instruction fetch. Each I/O device controller must have an interrupt request line into the CPU. Relative positions along that line will determine the hardware priority levels. One must associate with each device a pair of memory locations called the device's interrupt vector. These should hold the values to be loaded into the PS and PC registers if that device's interrupt request is honored. When the CPU honors an interrupt request, it saves the current PC and PS values in the system stack. A device's interrupt-handler software should terminate with the RTI instruction, which pops the stack into the PC and PS registers.

The PDP-11 has four interrupt-request lines. This makes it possible to assign a device a hardware interrupt priority ranging from a low of 4 to a high of 7 when it is connected to a PDP-11. Its PS register has a 3-bit interrupt priority field. An interrupt request will be honored only if its hardware priority exceeds the current PS priority.

The ideas we have discussed in this chapter will be seen again when we come to examine high-speed non-character-oriented I/O.

Exercises

11.1 Code segments beginning at CLK and OUTPUT are interrupt handlers for a clock and an output device.

(a) How should the symbols which are not otherwise defined be defined?
(b) What does each interrupt handler do?
(c) How do they interact— that is, what does the whole program do?

Note that the clock status register and vector addresses are 177546 and 100; those for the output device are 177564 and 64.

```
CLK:     INC     %1
         CMP     #59.,%1
         BGT     RETURN
         CLR     %1
         INCB    MBOX
         MOV     #100,@#OUTSTA
RETURN:  RTI

OUTPUT:  CMPB    #72,MBOX
         BNE     NOWORK
         MOVB    #57,MBOX
```

```
NOWORK:MOVB    MBOX,@#OUTBUF
       CLR       @#OUTSTA
       RTI

MAIN:  CLR        %1
       MOV     #OUTPUT,@#OUTADR
       MOV     #CLK,@#CLKADR
       MOV     #100,@#CLKSTA
X:     BR      X
MBOX:  .BYTE   57
       .END    MAIN
```

11.2 (a) Describe two ways of blocking interrupts selectively.
(b) Which kinds of interrupts cannot be blocked?
(c) Why are vectored interrupts desirable?

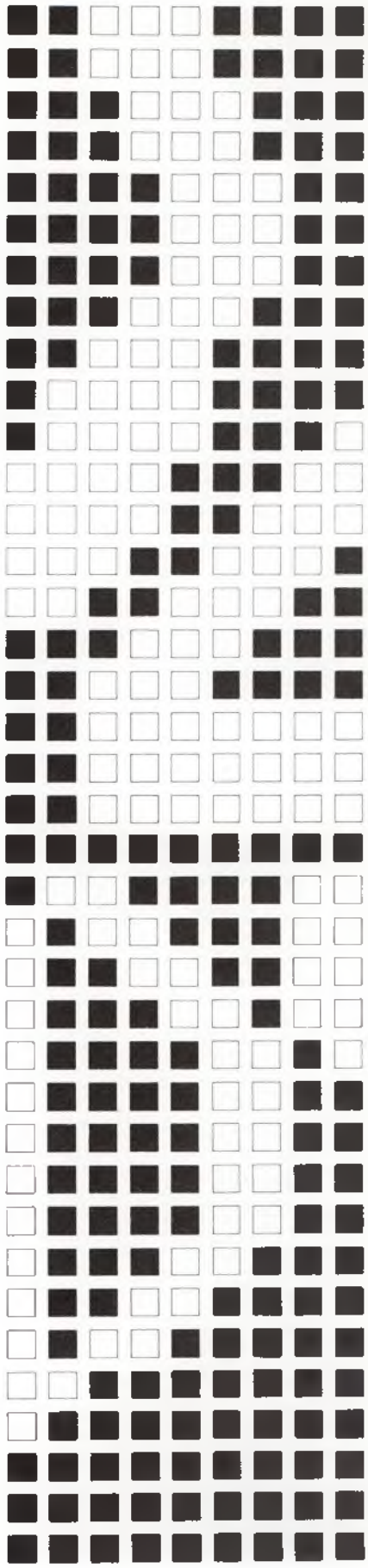
11.3 (a) Why are interrupts used in conjunction with I/O?
(b) Can an interrupt handler call subroutines?

Explain briefly why or why not.

11.4 Since interrupts are very much like forced subroutine calls, why is it necessary to use an RTI rather than an RTS to exit from an interrupt handler?

12

the assembler revisited



Surely there must be more to MACRO-11 than just supporting the use of instruction mnemonics, the various symbolic representations for the addressing modes, handling labels, and the few directives we have seen, such as .WORD, .BLKW, etc. After all, MACRO-11's reference manual has some 150 pages, and the MACRO-11 assembler uses some 15 KW of memory (MACRO-11 is written in approximately 4,400 lines of MACRO-11 statements). What have we been hiding? Not much, really, so we can finish most of it here.

Housekeeping Directives

MACRO-11 is a child of the time when "serious" computer professionals still regarded high level languages as "toys." Truly important programs would always be written in assembly language, because it was thought that no high level language compiler could generate machine-language code as compact or run-time efficient, or both, as could an experienced assembly-language programmer. That view of high level languages may have been justified some ten years ago. It cannot be defended now—not that there are no horrendously poor high level language compilers in use. But other important reasons lead one to use high level languages today.

A consequence of the early view was that MACRO-11 was thought to be appropriate for writing large programs involving several programmers. Since our purpose in this book is better served by writing small one-person programs, we have not felt a compelling need for the features we are about to see.

Documentation Support

Properly documenting one's programs is necessary, and one should use software tools that assist in documentation, but it is not an issue central to understanding computing. It does relate to understanding people who compute and to the sound management of computer-based services. The following directives can help make your programs look more professional and easier to understand, maintain, and enhance.

The statement

```
.TITLE PROG3 SORTS USING BUBBLE ALGORITHM
```

causes the first "word" (PROG3) in the .TITLE directive's operand field to appear in the header line reproduced at the top of each page of a multi-page assembly listing (only the first six characters will appear).

Somewhat more descriptive information may be included if the directive .SBTTL is also used. Its entire operand field is reproduced as the second line of the header lines which appear at the top of each page of multi-page assembly listings. You could use something like this:

```
.SBTTL HEAP SORT; author J. DOE; 80-9-28; see TR3-80
```

The presence of a `.LIST TOC` will produce a table of contents that lists all the subtitles in the given module. A typical table of contents is shown in figure 12.1. When paper was inexpensive, the .PAGE directive was used to request page ejects in order to visually separate different parts of the program listing.

Figure 12.1 Typical table-of-contents.

```
MACRO Version ...                    <date assembled>
Page    Line    Subtitle
1       1       Part 1: read input
1       32      Output section
...
13      1       Constants
13      20      Uninitialized space
13      33      Last line
```

Some problems cannot be readily solved by just one module. Even if they could, lengthy modules are not wise. Some people argue that a module should fit on a single sheet of paper. Therefore, you wind up with many modules to solve one problem. The linker and the assembler can help you keep track of things, if you use an .IDENT directive in each module. The statement

```
.IDENT /MODUL3/ ; ...
```

makes the name "MODUL3" available to the linker. If you subsequently request a "load map" (a link-time option), you will not only be shown the global names defined by virtue of .GLOBL statements, and the addresses assigned to them, you will also see the name MODUL3 appear, so you will be reminded that global symbols A, B, . . . are defined in the same module.

The directives .TITLE and .IDENT will only accept an argument with six or fewer characters. These may be alphanumeric, blank, the dollar sign, or a period.

The pair of directives

```
.ENABL arg, and .DSABL arg
```

control a set of ten assembly-time functions. For instance,

```
.ENABL LC
```

prevents lowercase ASCII characters from being automatically converted into uppercase characters at assembly time. The other functions are somewhat esoteric, and the reader should refer to the MACRO-11 reference manual for further information.

.LIST, .NLIST

We saw the statement ".LIST TOC" earlier. The directive .LIST, and its companion .NLIST allow you to control some fifteen assembly-time features. If you want to suppress the statement line numbers printed by MACRO-11, you can so specify by using

```
.NLIST SEQ
```

You can have their printing resumed by using

```
.LIST SEQ
```

Similarly, if it bothers you to see the octal addresses printed as MACRO-11 assigns them,

```
.NLIST LOC
```

will suppress them. If you don't want to see parts of your program echoed, you could insert

```
.NLIST SRC
...
.LIST SRC
```

to have MACRO-11 omit echoing the statements implied by . . . Similarly,

```
.NLIST BEX
```

suppresses the printing of all but the first octal word or byte generated by those statements which involve multiple bytes or words (e.g., `.WORD 1,2,3` or `.ASCII /ABC/`, etc.).

Conditional Assembly

This is perhaps the most misunderstood aspect of working with assembly language, yet the basic idea is simple, and failure to understand it clearly and fully is symptomatic of not really knowing how hardware and software interact.

When Do Things Happen? Assembly Time Versus Run Time

If you see the following statements in a friend's program

```
MOV #5,%1
ADD %1,%1
...
```

you would shake your head saying "Why didn't he simply write a `MOV #10.,%1`?" When there is a clear choice between computing something at assembly time (2*5 here) versus computing it at run time, if computing it at assembly time does not close off any options, then it is silly not to do it at assembly time.

On the other hand, suppose you saw

```
    MOV 2X,%1
    ...
X:  .BLKW 1
```

Besides realizing that the assembler will complain about 2X (and 2*X is no improvement), you will know that the presumed intent to multiply X, or (X), by 2 at assembly time is ludicrous because the .BLKW implies that the value (X) won't be available until run time. The general rule is: if it can be done early (i.e., at assembly time) with no loss in generality, then it should be done early to obtain a more space- and time-efficient program. If it can't be done at assembly time, but it could be done at link time, then do it then (e.g., defer the choice of version A or version B of a particular subroutine until link time).

Computing literature refers to "binding time." The time at which some value or attribute is decided ("frozen") is the time of binding a value to a variable. Deferring binding enhances flexibility, usually at the expense of greater run-time overhead.

```
X: .WORD 100. ; bind now (asm time)
Y: .BLKW 1 ; bind at run time
```

The preceding discussion of binding time may seem somewhat abstract. Let us relate it to the topic at hand by looking at a specific application.

Debugging Statements

The .SNAP debugging tool was described in an earlier problem. As you recall, its occurrence in the opcode field (if preceded by an earlier statement .MCALL .SNAP) leads to instructions which, if executed, print the content of the eight registers and the PS. When you believe your program has passed all your tests and it should be released for use by others, you can remove all the .SNAP debugging statements you were using. However, anticipating that something may go amiss after the program is put to productive use, you may prefer to leave the .SNAP statements in place but "disable" them by surrounding each .SNAP as follows

```
    TST DEBUG
    BEQ     S1
    .SNAP
S1:...
```

The production version of the program is then assembled with DEBUG defined by

```
DEBUG: .WORD 0 ; skip debug code
```

The production-test version can either be assembled using

```
DEBUG: .WORD 1
```

or arrange to set (DEBUG) $\neq$ 0 at run time.

In any case, all the debug instructions are assembled and loaded as part of the production program. The run-time overhead for all the TST/BEQ instructions is still incurred, even when you are not using the debug code. Yet, the alternative of maintaining two source files, one with and one without the debug statements, is not appealing. Inevitably, inconsistencies will creep in. A correction made to the debug version will not be made to the other version, due to simple human error; or it might be made, but incorrectly. Even if perfect consistency is maintained, the storage costs are doubled. There must be a better way.

Include/Exclude at Assembly Time

Consider having a single source module for the preceding application. If you have only one source module, presumably it will contain the debug statements. By using *assembly-time conditional assembly directives*, we can decide, at assembly time, whether or not the assembler should generate instructions, or *no* instructions, for the statements qualified by assembly-time conditional directives.

.IF and .ENDC

The .IF and .ENDC are conditional assembly directives, and they must always be used as a pair. .IF introduces a "conditional block" and .ENDC signals its end. The general form used is

```
.IF condition arguments
conditional block
.ENDC
```

The conditional block is a set of one or more MACRO-11 statements of any kind (even including nested .IFs and .ENDCs, but not including .END). The simplest condition one can use is represented by the symbol EQ. For example,

```
.IF EQ SW1
.SNAP
.ENDC
```

If, at assembly time, SW1 has the value 0, then the EQ (read as "equals zero") is said to be satisfied, and so the conditional block will be *included*, and the assembler may assemble it. Otherwise, if the condition is not met, then the conditional block will be excluded (i.e., MACRO-11 will pretend it did not even see it); thus it will occupy no space in the object module and create no run-time overhead.

The key phrase here is "SW1 has the value zero." This must be an assembly time value. What if SW1 is defined by

```
SW1:   .WORD 0
```

Does SW1 have the value zero at assembly time? No it does not. This SW1 has a non-0 value at assembly time, since it represents an address (unless by coincidence it was the very first location assigned in the whole program, leaving it with the address value 0).

The condition to be met in an .IF requires values defined at assembly time. The correct way to assign SW1 a value at assembly time is:

```
SW1 = 0 ; include the debug code
or
SW1 = 1 ; any value ≠ 0 excludes it
```

Examine carefully the two ways of getting "the same result":

```
          A                          B
1.  SW1 = 0              DEBUG:  .WORD 1
     ...                          ...
2.  .IF EQ SW1                    TST DEBUG
3.    .SNAP                       BEQ S1
4.  .ENDC                         .SNAP
     ...                  S1:     ...
```

In column A, lines 1, 2, and 4 generate no code, so no run-time overhead is incurred by them. In column B, each of lines 1, 2, 3, and 4 leads to code being generated or storage being allocated, with commensurate overhead at run time.

A further point of confusion should be dispelled. At run time, a TST instruction examines the hardware's CC bits. When the assembler evaluates assembly-time conditions, the CC bits are not involved in any way visible to you. Some section of the code in MACRO-11 examines the specified condition, and the arguments provided, and concludes, "The condition is satisfied (or not)."

.IF Conditions

The principal conditions used with an .IF are shown in figure 12.2. The arguments used in the .IF may involve an assembly time arithmetic expression. Thus you could write:

```
.IF NE ABC+SW7
BLAH
.ENDC
```

and if both the symbols ABC and SW7 had assembly-time values of 0 (or opposite signed values) then the condition would not be satisfied, and "BLAH" would be skipped by the assembler.

Figure 12.2 Conditions used with .IF directive.

Conditions		Arguments evaluating as
EQ	NE	zero/not zero
GT	LT	greater/less than zero
GE	LE	greater/less than or equal to zero

Life would be simpler if all assembly-time variables were colored red and the "usual" variables manipulated at run time were colored green; then they would be less likely to be confused. Some high level languages allow compile-time variables analogous to our assembly-time variables. These are usually required to have a special character prefix (e.g., % in PL/1), so you can't possibly confuse them with "ordinary" variables. This is a roundabout way of saying that the concept of assembly-time versus run-time variables is of such value that it is (or should be) supported within high level languages. It is a vehicle for managing immediate or deferred bindings.

Reusing Labels

It is a nuisance having to make up names for things that have no universal significance. Why not allow simple names to be used where needed and reused elsewhere if needed, with a connection made between the various uses only if you intended it. MACRO-11 attaches a special significance to names of the form 1$, 2$, . . ., 63$; they are called *local labels*. You can use these names as labels of words. If you run out after using all 63 such names, you can begin reusing them and ensure that the new definitions are not in conflict with the old definitions. Consider the following program skeleton:

```
1$:     MOV X,Y; (a)
        ...
        BEQ 1$ ; (b)
        ...
        BMI 1$ ; (c)
        ...
1$:     ADD A,B; (d)
        ...
        BEQ 1$ ; (e)
        ...
```

We see here two definitions for the label "1$". We see three uses of it. What sense can we make of it?

If the assembler finds a "normal" label being defined between points (c) and (d), then the 1$ references at (b) and (c) are associated with its definition at (a), and the (e) reference is tied to the (d) definition. If no ordinary label is defined between points (a) and (d), then "1$" will be regarded as multiply defined.

Instead of writing statements such as

```
NEXT:   TST KCSR
        BPL NEXT
```

if "NEXT" has no truly transcendental significance, we could write

```
3$:     TST     KCSR
        BPL     3$
```

assuming that the last used local label was 2$. We don't want to worry about the "scope rule," which requires that two definitions for the same local label be separated by an ordinary label definition.

Local labels of the form 64$, 65$, . . ., 127$ can be used, but since they are also used by some macros (coming up soon), it is advisable to avoid using these in order to preclude conflicts. A final comment: local labels can be attached only to items on word boundaries.

Assembler Location Counter

The assembler maintains an item called the location counter. The assembler keeps track of which memory location it will be assigning next by keeping that number in its location counter. The location counter plays a role analogous to that of the PC, but two crucial distinctions apply:

1. The PC is a piece of hardware, a 16-bit register, managed by the control unit. The location counter is just a name for a variable which is managed by a particular software product, the assembler.
2. PC values are crucial to you during your program's execution. Location counter values are crucial to you while your program is being assembled.

That being understood, what can you do with the location counter? Knowing that its real name is "." (yes, period), you can write lines such as:

```
        CMP     A,B
        BEQ     .+4
        BHI     XYZ
```

instead of

```
        CMP     A,B
        BEQ     SKIP
        BHI     XYZ
SKIP:   ...
```

or

```
        CMP     A,B
HERE:   BEQ     HERE+4
        BHI     XYZ
```

When an *instruction* is being assembled, the value of "." is the address assigned to the first word of the current instruction. In an I/O driver, you might write

```
        TST     @#KCSR
        BPL     .-4
```

instead of

```
LOOP:   TST @#KCSR
        BPL LOOP
```

When a *directive* is being assembled, the "." value is the address of the current word or byte being processed. Thus

```
A:  .WORD .,.
```

and

```
A:  .WORD A,A+2
```

are equivalent.

If you want to, you can perform an assembly-time addition and assignment

```
. = . + 10.
```

to increase the location counter by 10. This has exactly the same effect as

```
.BLKW 5 ; or .BLKB 10.
```

Each of these has only one effect: increasing the location counter value by 10.

We have avoided using ".", the location counter symbol, for a number of reasons. The "." is already in use in two entirely different ways. One is as the decimal radix request (e.g., 10.); the second is as part of symbolic names or directives (e.g., .PMD, .BLKW). Finally, the risk of miscounting a .+offset is simply too great, and it is too likely to introduce future problems. Consider

```
            ...
            CMP     A,B
L0:         BEQ     .+18.
L1:         MOV     A,C
L2:         ADD     B,C
L3:         MOV     C,X
L4:         ...
```

The `BEQ .+18.` may get you to the right place, but it is very obscure—if, in fact, it is correct. If the program is subsequently "improved" by inserting or removing any instructions between lines L0 to L4, then the .+18. has to be corrected. If you optimize the program by inserting a statement such as

```
C = %0
```

three pages back, then .+18. is guaranteed to be wrong.

.ASECT

On a single-user system you sometimes want to "force" the assembler to assign particular absolute memory locations. The .ASECT directive conditions the assembler to assign absolute addresses. So if you want the assembler and linker to cooperate in loading an interrupt vector for you at location 100, you can write

```
        .ASECT
        . = 100
        .WORD   CLCK  ; new PC
        .WORD   300   ; new PS
    . = 1000          ; leave room for stack
CLCK:   TST     KCSR
        ...
```

In conclusion, you now know what "." and the assembler location counter can be used for. It is advisable not to use "." if you can avoid it. Use local labels liberally if ordinary labels are inappropriate.

Immediate ASCII Constants

When you are scanning an ASCII string for a particular character, as in the following:

```
        ...
        CMPB    (%1),COMMA
        ...
COMMA:  .ASCII  /,/
        ...
```

the only alternative is to look up the ASCII code for "," and write

```
        CMPB    (%1),#54  ; COMMA
```

MACRO-11 supports a single-character immediate operand. The # operator followed by an apostrophe "'" causes the next character (the one immediately following the ') to be treated as a numeric constant equivalent to the corresponding ASCII code. Thus, the preceding example and

```
        CMPB    (%1),#',
```

are equivalent.

If you want to use a *pair* of ASCII codes as immediate operands, you can use the quotation mark operator " (not '). If you were writing an assembler for MACRO-11 programs, you could write

```
        CMP     #"EQ,(%1)
        BR      EQUALS
        CMP     #"NE,(%1)
        BR      NOTEQ1
        ...
```

The 'char and " char-pair notation can be used independently of immediate mode.

```
        .WORD   "HI,"BY,"E ;
        .BYTE   'A,22,'X,-7
        QM = '?             ; why not
        ...
        .BYTE   QM
        ...
        CMPB    (%2)+,#QM
```

Simple Macros

We have been using the assembly-time assignment operator "=" for some time. It allows us to write a short (or long) name in place of some other name, number, or expression. We can write the following:

```
        K = 1024.
        SIZE = 100.
        ...
        MOV     #K,%1
        ...
ABC:    .BLKW   SIZE
        ...
```

As we shall see, macros are a generalization of the assembly-time assignment operator.

A "macro" is a body of text to which you assign a name. When you use that name (referred to as using or invoking the macro), the meaning (i.e., the body of text you previously associated with that name) is substituted, at assembly time, for the name used.

We repeat: all of this happens at assembly time. Macros are often confused with subroutines. A macro is an assembly time "artifice." It may or may not lead to anything at execution time.

Macro Definitions

You assign a macro a name in the act of defining the macro. Its definition always involves the pair of directives .MACRO and .ENDM.

If you wanted to define a macro to help you more easily "save" registers 1 and 2, you could write

```
.MACRO  SAVE12    ; name SAVE12
MOV     %1,-(%6)  ; two lines
MOV     %2,-(%6)  ; of text
.ENDM
```

Then you can invoke this macro at any point *following* its definition simply by writing its name in the opcode field, as if it were an ordinary instruction. The assembler, upon encountering this name in use as an opcode, will refer to the table of macro names it has been collecting and, finding the name "SAVE12" in this macro name table, the assembler will proceed to replace the invocation of "SAVE12" by the body of the definition. If, on the other hand, you invoke a macro and forget to define it, the assembler will issue a diagnostic regarding an undefined opcode.

The consequences of a misplaced macro definition may be far more dramatic. Use of a macro prior to its definition may trigger an avalanche of error flags, the "P" flag error, for phase errors. The explanation for this will be discussed when we look inside the assembler to see how it works.

Processing Macros

When a .MACRO statement is encountered, the assembler places the name of the macro in a macro name table; we will call it MNT. The body of the definition—that is, the text between the .MACRO and its matching .ENDM—is copied to a safe place, and a pointer to this saved text is recorded next to the macro's name in the MNT.

When the assembler is processing what it takes to be an opcode, because of the context, it searches up to three tables, looking for a matching name. The tables the assembler searches, beginning with the first one it examines, are:

1. MNT (macro name table)
2. PNT (permanent name table, for mnemonics, directives)
3. UDNT (user defined name table)

If the opcode field in the statement being processed holds a name matching one in the MNT, then the text of the macro definition will be copied from the area in which it was earlier saved, and this text will replace the invocation. If you inadvertently assigned a name to a macro without realizing that the name was also being used as an instruction mnemonic (or as a directive), the meaning you assigned via your macro definition will override the normal "built-in" meaning.

```
.MACRO  ADD
NOP
.ENDM
```

would cause every "ADD" instruction following this definition to be assembled as if it were a NOP.

If you define a macro but never get around to using it, no harm is done. No flag will be raised by the assembler. If you invoke one of your macros and it happens to have some nonsense in the midst of the definition, then under the normal default listing option, since the macro's expansion is not included in the listing, the "bad line" will appear out of context, flagged with an error. Consider the following segment of a source module:

```
        MOV     #1,%1
        BLAH
        MOV     %1,ABC
```

You would be at least a little surprised if the assembly listing came back with

```
        MOV     #1,%1
        BLAH
****A
        INC #%1
        MOV     %1,ABC
```

You look back at your source module. Since the only statement between the two MOVs is BLAH, then the only conclusion to be drawn happens to be the correct conclusion: BLAH must be a macro, and its expansion generated an erroneous line `INC #%1`. As we just mentioned, the assembler does not display the macro's expansion unless you ask it to—except, of course, if it finds an objectionable statement. You can request printing of the macro expansion code by using

```
.LIST ME
```

As a general rule, if you use `.LIST ME` indiscriminately, you will waste far more paper (or CRT display time) than was the case with `.LIST BEX`. You can selectively list only those macro expansions of interest by using pairs of `.LIST ME` and `.NLIST ME` directives.

.MCALL Revisited

As we saw long ago, macro definitions may be collected in a macro library which is accessible to the assembler at assembly time. The operating system maintains a special collection of system macros, any of which can be used by having their names listed as operands of an .MCALL directive.

Thus, if your system provides debugging macros such as .PMD and .SNAP,

```
.MCALL .PMD,.SNAP
```

will make the corresponding macro definitions available at assembly time.

If you happen to ask for something that is not there, you will be given a null (blank) definition, without any warning. Once upon a time a program was behaving very strangely. It seems that its author, in an attempt to be safe rather than sorry, tried to provide definitions for everything.

The following .MCALL appeared in his program:

```
.MCALL .PMD,.LIST,.NLIST,.SNAP
```

The null meanings for .LIST and .NLIST were extracted from the system macro library and overrode their normal meanings as built-in directives, with very strange consequences. Of course, the innocent looking .MCALL was the last statement to be suspected.

Macros with Arguments

The reason for creating and using macros is to reduce *your* effort. Let the assembler do the writing (of the macro's expansion). It may or may not have much effect on the work done at run time.

Whenever you have a recurring pattern of text in a source module, a macro might simplify your life. For example, if you used the following instruction pair frequently:

```
ASL     %1
ASL     %1
```

you might consider defining a macro called FOURX

```
.MACRO  FOURX
ASL     %1
ASL     %1
.ENDM
```

Then any time you needed to use consecutive pairs of `ASL %1`, you could simply write FOURX in the opcode field. What if you also had many instances of consecutive pairs of `ASL %4` in the same program? Instead of defining a second macro for this situation, you can use another feature supported by the assembler. It is similar to the idea behind arguments used by subroutines.

Macro Arguments

A MACRO-11 macro definition can have as many symbolic arguments as fit on one line. Each of them is represented by a symbolic name subject to the same spelling rules as apply to any label. The argument names are written following the macro's name on the .MACRO line. We can now write one definition to take care of the ASL pairs we were looking at:

```
.MACRO  FX A
ASL     A
ASL     A
.ENDM
```

It does not matter whether the name A is being used anywhere else in this module, even as an argument in another definition. The argument names in a macro definition are "local" to that definition. They can be used elsewhere independently of their use in the body of a macro definition. Instead of invoking the macro FOURX, we can now invoke FX with the argument %1 and generate the code associated with FOURX.

Similarly, `FX %4` will lead to the expansion:

```
ASL     %4
ASL     %4
```

In both cases, all occurrences of the dummy argument A, in FX, are replaced by the value corresponding to A's position in the macro invocation.

It is not unusual to see the same argument names being used again and again:

```
.MACRO  DOUBLE X
ASL     X
.ENDM
.MACRO  ADDONE X
INC     X
.ENDM
```

The name X can also appear elsewhere as a label, with no problems. The names following the macro's name are *dummy argument* names; they have *no* significance outside the scope of the macro definition.

Sometimes macros are used simply to provide better mnemonics. You think of `MOV (%6)+,ABC` as a "pop" operation. Why not have

```
.MACRO  POP ARG
MOV     (%6)+,ARG
.ENDM
```

Then you can write

```
POP     ABC
```

or

```
POP     (%2)
```

instead of

```
MOV     (%6)+,ABC
```

or

```
MOV     (%6)+,(%2)
```

Common uses such as RTS PC and RTS %5 can be abbreviated:

```
.MACRO  RPC
RTS     PC
.ENDM
```

```
.MACRO  RRS
RTS     %5
.ENDM
```

The use of macros is not restricted to generating instructions. Any reasonable text can be included. Suppose you are testing a program and you need several test cases. You could set up a macro to help provide the test input and what you consider to be the correct result. Suppose the program being tested is one that accepts binary words and produces an ASCII representation as a decimal number. Then the following macro will provide the test pairs which follow:

```
            .MACRO      TEST    Y
            .WORD       Y
            .ASCIZ      /Y/
            .EVEN
            .ENDM
TST:        TEST        1
X:          TEST        12
```

The expansion for these last two lines is:

```
TST:        .WORD       1
            .ASCIZ      /1/
            .EVEN
X:          .WORD       12
            .ASCIZ      /12/
            .EVEN
```

This saves you a lot of writing.

Note that a macro invocation can have a label, just as if it were an ordinary opcode.

Nested Macro Use

Suppose we had the following macro definition:

```
            .MACRO  R ARG1
            RTS     ARG1
            .ENDM
```

Then we could rewrite the previous definitions for RPC and RR5 as follows:

```
.MACRO  RPC         .MACRO  RR5
R       PC          R       %5
.ENDM               .ENDM
```

When RPC is used (invoked), the assembler will generate the expansion as follows:

```
Use     →       Expansion
RPC             R   PC
```

But, having replaced the invocation RPC by its definition, the assembler looks at what it has just generated and realizes that R itself is a macro invocation, so a second expansion occurs:

```
Use      →      Expansion
RPC
```

This could conceivably go on for many more levels if RTS were itself another macro invocation rather than an instruction mnemonic.

The use of a previously defined macro to define another macro is analogous to use of subroutines, but with a difference. The nested use of subroutines remains visible and in effect at run time. Nested use of macros is taken into account entirely at assembly time (during macro expansion), and you can't really tell by looking at the resulting machine code whether nested uses of macro definitions were or were not involved.

What Can Go Wrong?

The key rule in using macros is to ensure that the macro definitions are physically seen by the assembler before those macros are used. Why? Play assembler for a minute. If you saw

```
          TST     X
          BLAH
X:        ...
```

how would you assemble this by hand? If you had no definition for BLAH yet, you might guess, "It probably only needs one word, so I now know how to update the location counter." You could then set aside a one-word hole for the BLAH code, update the location counter by 2, assign an address for X, etc. After all, this is exactly how you would proceed in assembling the reference to X in TST X, since you would not have seen its definition yet. If you then see a definition for BLAH and it generates only one word of code, all is well. If not, then the "hole" you left is too large or too small. In that event, *all* the labels defined following the first invocation of BLAH will have had incorrect addresses assigned to them by the assembler. The assembler will flag each such address with a P (Phase error) flag. One misplaced macro definition can trigger dozens of P errors.

What else can go wrong? A simple typographical error can cause the assembler to fail dramatically and leave you without even an assembler listing to look at. Consider

```
.MACRO   RPC
RPC      PC
.ENDM
```

Due to a momentary lapse of attention, you typed "RPC PC" instead of "RTS PC." When you come to use the macro RPC, the following sequence of expansions is triggered:

```
Use → Expansion → Expansion → Expansion → ...
RPC   RPC PC      RPC PC      RPC PC      ...
```

This expansion will go on indefinitely, because we inadvertently created a circular definition. At each level of expansion the assembler will stack its internal return address, while it descends one level deeper in the expansion process. Since this will go on indefinitely, the assembler will stack too many things in the system stack, and the stack overflow will cause the running program to abort. The program that is running at the time is not yours; it is the assembler which is executing. Since it never got to look past the line with the RPC invocation, there is no hope of getting any listing beyond that point—if, in fact, you can get any assembly listing at all.

What if you invoke a macro using more arguments than it expects? For instance,

```
POP     ABC,SUM
```

where POP specified only one argument in its definition, would lead the assembler to ignore the arguments following the first one (it will use ABC, but ignore SUM and anything following it on that line). No warning is issued about extraneous arguments.

If you used a macro and provided fewer arguments than were expected, the positions used by the missing arguments are left empty, and more likely than not the assembler will flag some line in the expansion because it finds a syntax error. Thus, a POP not followed by any arguments has the expansion

```
MOV     (%6)+,
```

which clearly will be objected to by the assembler.

Every .MACRO must have a matching .ENDM. If due to a typographical error you misspell the .ENDM, or forget it, all the text following the unmatched .MACRO will be "swallowed up" as part of that macro definition, up to and including the module's final line, .END. Since even the invocation is thus swallowed up, the assembler has no code to assemble, so your assembly listing won't show any octal being generated following the unmatched .MACRO line.

Summary

Some directives are provided to facilitate documentation of programs. These include .TITLE, .SBTTL, .IDENT, .LIST, and .NLIST. Conditional assembly directives allow you to write statements within an assembly language program, and subsequently inform the assembler to include or exclude these statements, at your request. This is supported by the directives .IF and .ENDC, which are used to enclose the statements to be conditionally assembled. The conditions which can be specified include EQ, NE, GT, LT, GE, and LE. The values tested in conjunction with these must be determinable at assembly time.

Local labels of the form 1$, 2$, . . ., 63$ can each be defined several times within the same module and used unambiguously, provided that consecutive definitions are separated by the occurrence of an ordinary label definition.

The assembler location counter has the symbolic name ".". It can be used just as any other name is used. The directive .ASECT is used to

govern assigning absolute addresses in those cases where you must use specific absolute memory locations.

The syntax 'a and "bc can be used to specify that the ASCII codes corresponding to the single character a or the character pair bc should used. This construction can also be used in conjunction with immediate addressing, as in `MOVB #'H,%1`.

A macro is a set of assembly-language statements which has been assigned a name. The macro may or may not involve the use of dummy arguments. A macro may already have been defined elsewhere and can be made available using the .MCALL directive. Otherwise the .MACRO and .ENDM directives are used to delimit the defining statements and to provide a name.

A macro is invoked by use of its name as if it were an opcode. This causes the assembler to substitute the statements which constitute this macro's definition. If the macro invocation has arguments, these will be substituted in place of the dummy arguments in the definition. When defining a macro, one can use previously defined macros. A macro can be used (invoked) only after its definition has been made available.

Exercises

12.1 Obtain a printout of the expansion of the system macros you are using for input of data. Reconstruct from the expansion what would be a reasonable macro definition. If the system macro has optional arguments, explain how the definition supports the various expansion possibilities.

12.2 Write a macro which generates a sequence of instructions to subtract two signed integers. It *cannot* use the SUB instruction. To what extent is your code's action equivalent or not equivalent to that of a SUB?

12.3 After execution of the following program, what is the content of M3, M4, and T1? In one sentence, what does this program do?

```
S3:    CLR %5
       .MACRO M1  S1,S2
        JSR    %5,M2
       .WORD  S1,S2
       .ENDM
      M1   M3,M4
      M1   M3,M4
     HALT
      M3:  .WORD 26
      M4:  .WORD 28.
      M2:  MOV (%5)+,%1
           MOV (%5)+,%2
           MOV (%1),T1
           MOV (%2),(%1)
           MOV T1, (%2)
           RTS %5
      T1:  .WORD  0
           .END  S3
```

12.4 Does the use of macros speed up program execution? Explain.

12.5 Using an assembly-time switch, write those parts of a program which would allow you to assemble a program for a PDP-11/20 or a PDP-11/34. The former has no MUL instruction; the latter does. When the program is written, what should the programmer write when he needs a multiplication? When the program is assembled, show what the assembler will generate in both cases. Do not use macros.

12.6 Repeat the previous exercise, making good use of macros.

12.7 Write a macro definition for a macro which copies a string. A typical invocation could be

```
COPY STR1,STR2,12. ; source, destination, length
```

12.8 Suppose you have such a long program that you forgot that a macro named AHA was defined on page 1, and you provided another definition on page 3. Does the assembler flag this? Are any other assembler flags triggered by this event? Under what circumstances? What other reasonable choices of action could the assembler's designer have implemented?

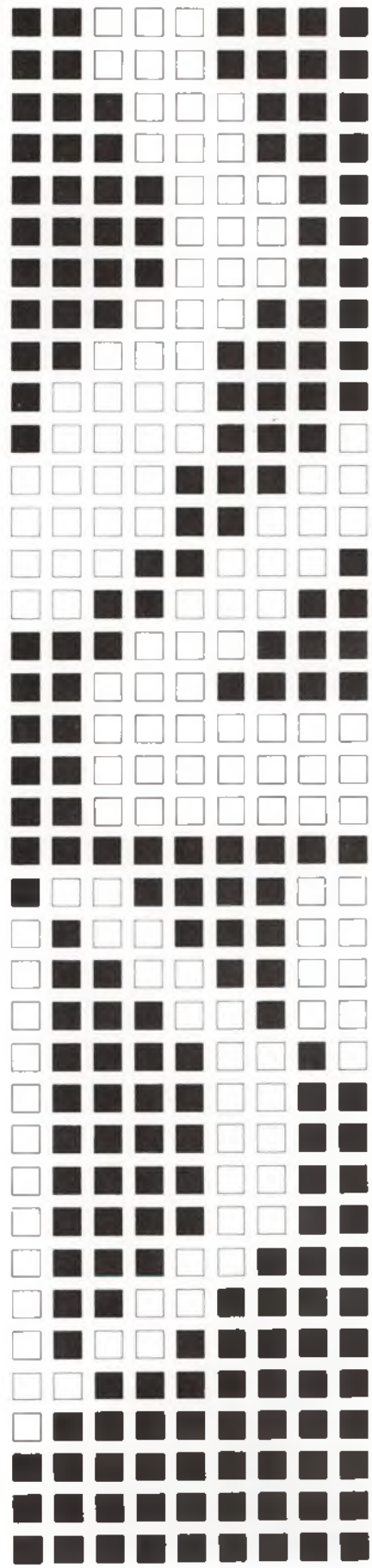
12.9 Write a macro which assigns the names Reg0, Reg1, . . . to the registers %0, %1, etc.

12.10 Write a macro called BLOCK which is to be used just as a .BLKW directive would be used but which does not use a .BLKW in its definition.

12.11 Find out how you can obtain a load map from a linker. Obtain one and interpret it.

13

solving real world problems



People who write compilers for high level languages (or assemblers for lower level languages), or who write operating systems, are quite content with the arithmetic facilities we have been using. You do not need much more than integer arithmetic for most systems-programming uses. However, these people are not the principal users the computers are designed for.

Commercial data processing applications manage nicely with hardware-supported BCD arithmetic. Is there any need for other kinds of number representations or special hardware and instructions to support other kinds of representations? Consider figure 13.1. It shows the range of sizes one must contend with in physics. It is not entirely coincidental that physics is used for this example. Many of the computer characteristics we are about to see are the direct result of requirements to support the work of physicists. Very little of the research and development in the area of nuclear weapons and nuclear energy would have been possible otherwise. This is a grim reminder of the fact that any tool can be used for purposes which some may argue are antisocial; you cannot blame the tool. By the same token, phenomenal advances in medicine (such as the CAT scanner) are directly traceable to what we are about to discuss.

If we take the smallest number in the chart in figure 13.1 to be 10^{-10} cm and the largest to be 10^{25} cm, we have a "spread" of 10^{35} from the smallest to the largest numbers used by physicists. A general-purpose computer, to be of any use to scientists and engineers, would need to support high speed arithmetic with operands of 35 decimal digits. This is not practical. (Using 4 bits per digit, we need 140-bit numbers!) Why not use a word capable of holding binary numbers as large as 10^{35}? That takes about 120 bits— $2^{10} \approx 10^3$, so $(10^3)^{12} \approx 10^{35}$, thus $(2^{10})^{12} \approx 2^{120}$. We are down to words of 120 bits; this is still impractical. The hardware to support high speed arithmetic with 120 bit operands is too expensive.

The solution is based on accepting the fact that few if any numbers scientists deal with are known with absolute accuracy. So a compromise can be worked out. Scientists normally work with approximations; they do not pretend to measure things with absolute accuracy. If we took a 32-bit word and adapted the scientific notation in use by scientists for many centuries, we would be able to represent very very large numbers, and extremely small ones as well, with a reasonable word size, such as 32 bits. This is precisely what most calculator manufacturers have done. With a decimal calculator using 8-digit decimal numbers, you can get an enormous range (from 10^{-99} to 10^{99} is quite common). The scientific notation has the form:

$$\pm\ d_1.d_2d_3 \ldots d_m * 10^{\pm n}$$

where $d_1d_2 \ldots d_m$ are decimal digits and n is a decimal exponent. A scientist would not normally write "100 moles" or "3,000 grams." Neither would a chemistry major. They would write "$1.00*10^2$ moles", or "$3.000 *10^3$ grams." The trailing zeroes in both cases express the confidence they have in these measurements. Saying "1.00" means the result is known to be between 0.995 and 1.004; so the implications of a value "1.00" are different from those of either "1.0" or "1.0000". If a number is smaller, negative exponents come into play. The fraction 13/1,000th of a kilogram is written $1.3*10^{-2}$ kilogram.

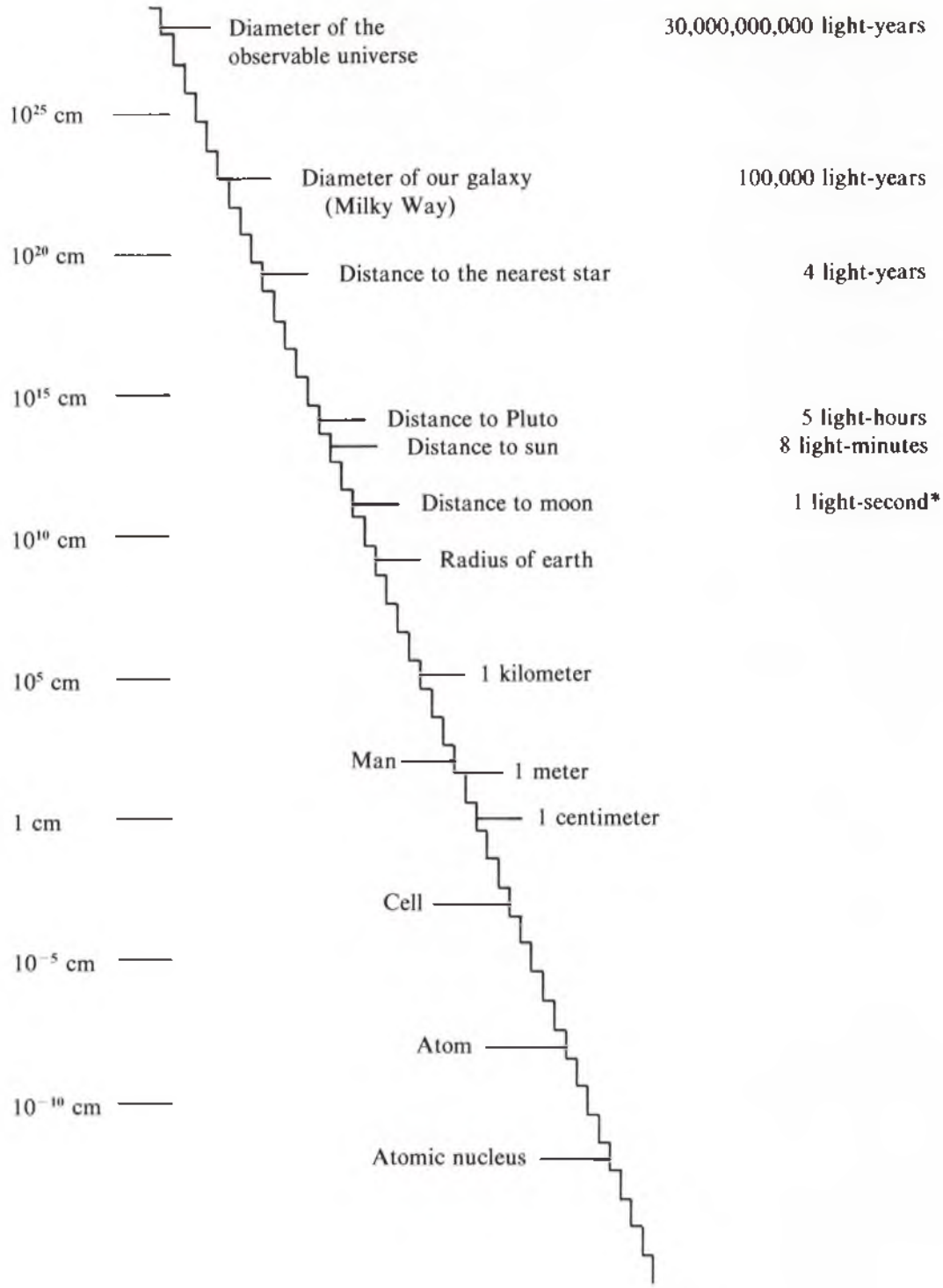


Figure 13.1 Ranges of sizes in physics. Each step corresponds to a factor of 10 increase in size. *1 light-second = distance that light travels in 1 second = 3×10^{10} cm or 186,000 mi.

(Reproduced from *Physics in Perspective, Vol. I.* National Academy Press, Washington, D.C., 1972.)

If the exponent size n is limited to two digits (as on many calculators), the largest number you can express (or compute) with 8-digit numbers is $9.9999999*10^{99}$ (about 10^{100}) and the smallest number is $0.0000001*10^{-99}$ (about 10^{-107}). If you try to enter a number with too many digits (say, 1.234567892) with a calculator limited to 8 digits, you have to compromise. You can enter only an approximation of that number. You have to introduce an unavoidable error. The simple approach is to *chop* or *truncate* the excess digits from the trailing end. So here you would get 1.2345678.

A better approach involves *rounding*. Rounding means examining the tail you are about to chop off. If it is of the form 5xxx or larger, a one is added to the first of the digits of the new tail. Otherwise the tail is simply chopped off as before. In our previous example, dropping the tail 92 from 1.234567892 would have us adding a one to the low digit (the 8) if we are rounding and dropping the tail "92". Had the tail been 43, then the 8 would not be affected. Rounding rounds away from zero. Truncation brings numbers closer to zero. The point of rounding is to reduce the error in the approximation you are being forced to use.

Floating-Point Numbers

The computer representation for numbers in the scientific notation is called the floating-point representation. It has three principal goals:

1. Support a much larger number range with a fixed number of bits.
2. Retain the maximum precision possible (i.e., minimize errors).
3. Provide an unambiguous (unique) representation for each floating-point number.

These goals can be met by viewing the bits used to represent a floating-point number as having two parts : signed exponent and signed fraction. The fraction part holds the numeric value we wish to represent. It is called the fraction because it is assumed that an imaginary binary point appears immediately to its left. So it will always be true that the magnitude (i.e., the unsigned value, designated by using a pair of bars) is less than 1:

$$|\text{fraction}| < 1$$

Given an exponent e and fraction f, the value represented by the pair (e, f) is

$$.f * 2^e,$$

if we choose to assume that the exponent radix is 2. This is the choice made for the PDP-11 and many other computers. But some designers have elected to use a radix value of 16 (notably on the IBM 360 series, since 1964, and continued with the IBM 370 and its successors).

By allowing e to assume large positive or negative values, the range goal is met. Retaining maximum precision can be achieved by reducing the number of leading zeroes to no-leading-zeroes. Of course, if you drop leading zeroes that are significant, you have to adjust the exponent to compensate. In 000.123, the leading zeroes have no significance whatsoever; in fact, writing a number in this way has no scientific meaning (it may be used when printing checks, to reduce the temptation to alter the number). On the other hand, a number such as .0034 has two significant leading zeroes. We can "normalize" it—that is, put it in a standard or normal form—by writing $.34 * 10^{-2}$.

Similarly, 5,600 can be written as $.56 * 10^{4}$. The complete definition for *normalization* is: given a floating-point number (e, f) with a radix r, this number is considered normalized if

$$1/r \leq .f < 1 \text{ , for all f except 0.}$$

This means that with a radix of 2 (r=2), a normalized number always has its leading bit set, except for the fraction with the value zero.

Normalization has the property of also letting us meet the third goal: unique representation. You can imagine how frustrating it is to be given an address "22 Victory Square," and search for hours before you finally find the building. The square was renamed "Great Leader Square" a few years ago and all the street signs were changed, but the local people still know it by the old name. Well, it is just as bad to have two different bit patterns having the same value. The numbers (1, 110) and (2, 011) have the same value (here we are assuming that the fractions are 3 bits wide). The first number's value is $.110*2^{1}$, or 1.1 (binary). The second number's value is $.011*2^{2}$, or 1.1 (binary). Two different bit patterns can have the same value. We can prevent this from occurring by requiring that henceforth and forever only normalized numbers will be stored. Thus the unnormalized number (the one with the leading 0: 2, 011) would be normalized, which would transform it into 1, 110 which is exactly what the other number looks like. Normalization guarantees that each value representable as a normalized floating-point number has a unique representation.

As a practical matter, rather than store a sign bit for the signed exponent, it is convenient to store a *biased exponent*. The ordinary signed exponent has a constant added to it, the so-called bias, transforming the signed exponent into an unsigned biased exponent. So we can now speak of floating-point numbers as consisting of

biased exponent, signed normalized fraction.

Biased exponents are also called the "characteristic," and the fraction is often called the "mantissa." Since we regard positive numbers as larger than negative numbers, it would be convenient to store the fraction's sign sf where signs are usually kept in a computer word. We now have

sf, biased exponent, normalized fraction.

This arrangement has the pleasant consequence of allowing us to use the *same* CMP compare instruction and the same signed conditional branches we were using with signed numbers! So we can avoid having to pay for more hardware to support a new class of compare-and-branch instructions. Given two floating-point numbers a and b, if they have opposite signs, then the one with the zero sign bit is larger. If they have the same sign (let us, for simplicity, assume that both numbers are positive), then the number with the larger biased exponent is obviously the larger number. If the biased exponents are equal, then the one with the larger fraction is the larger floating-point number. Since all of the above is consistent with the way the compare instruction CMP works, all goes well.

If the two floating-point numbers we wish to compare are both negative, we have to be careful, because we are dealing with a strange kind of mixed-sign-magnitude system. We need merely clear the sign bits of both operands and proceed as if both operands were positive, then take the original signs into account after the comparison.

Floating-point numbers are often called "real numbers," in part because the FORTRAN language has a directive REAL. A computer "real number" is an approximation for what mathematicians call "real numbers."

PDP-11 Floating-Point Representation

The PDP-11 supports floating-point numbers of two kinds. The most frequently used kind of PDP-11 floating-point number always will occupy contiguous pairs of words when in memory, as shown in figure 13.2. The biased exponent field is 8 bits wide, and a bias of 200 octal is used. So a true exponent of 1 becomes a biased exponent of 201. The smallest true exponent we can represent is thus -200 and the largest is 177, since biased exponents can range from 000 to 377.

The fraction field is broken into two parts. The most significant part of the fraction occupies the low 7 bits of the first word and all 16 bits of the second word. It is important to note that the second word is treated as an unsigned number. The sign for the whole fraction comes from bit 15 of the first word. The number of fraction bits is 23 (7+16), but the true fraction width is 24 bits. Since we have agreed to work exclusively with normalized floating-point numbers, their leading bits are always set (except for the number 0). It seems foolish to store something which can never change and which is always known, so we agree to *hide* the leading bit of the fraction. Thus, 23 fraction bits in memory are equivalent to 24 bits of fraction.

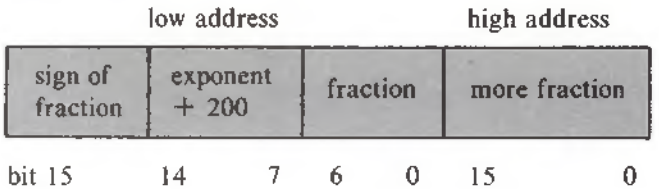


Figure 13.2 PDP-11 floating-point format.

Let us examine a few PDP-11 floating point numbers to see how all this fits. Zero is simple. Since $0*2^n = 0$ for any value of n, by convention we simply set both the fraction and biased exponent fields to 0, as well as the sign bit. Thus, `.WORD 0,0` represents a pair of zero integers or a single floating-point zero.

The number one: $1 = .1 * 2^1$, $= 2^1 * .1$

```
     true exponent = 1                    full fraction 1000...000
     biased exponent = 201                23 bit fraction 000...000
     sign S = 0
S    B-E          23 bit fraction
0    201          000...000                  mixed octal-binary
0  10 000 001     000 0000   00...00         binary
0   4   0   2      0   0      000000         octal only
```

```
.WORD 040200,000000
```

Similarly, −1 becomes 140200, 000000 as a floating-point number. A 2 simply has a larger exponent, by 1:

+2 → 040400, 000000

A biased exponent 404, as extracted from the word, is only 1 larger than 402, since as 8-bit biased exponents the first is 202 and the second is 201. What of .5, or 1/2? 1/2 = .1 (binary) = $2^0 * .1$ This is similar to the case for 1, except that the exponent is just a bit (exactly) smaller.

1/2 → 040000, 000000

How would 10.5 fare? "Mixed" numbers, with both an integer part and a fraction part, should have these two parts converted separately into binary numbers. Then the two binary parts can be combined and normalized, and we are back to the standard case.

```
     Given the number 10.5
     integer part 10.    → 12 octal
                         → 1010 binary (I)
fraction part .5         → .4 octal
                         → .1 binary (F)
Add I+F, → 1010.1
Normalize 1010.1
1010.1 * 2^0
101.01 * 2^1
10.101 * 2^2
1.0101 * 2^3
.10101 * 2^4
true exponent      4   full fraction      1010100...000
biased exponent  204    23 bit fraction   010100...000
204 octal
10 000 100 binary
```

```
s b-e            23 bit fraction
0 10 000 100     0101 00 00...00

0 4    1  0        5    0     00000
So 10.5 —→ 041050,   000000
```

Floating-Point Arithmetic

Given the description of the PDP-11 floating-point number format, you can write a program which can add pairs of such numbers. Such an addition is called a floating-point addition. Given two normalized floating-point operands, you have to ensure that the result is also normalized. What processing does a floating-point addition entail?

Let us illustrate the processing involved by using a simpler representation first. We will use decimal numbers with two-digit fractions and one-digit signed exponents, with a decimal radix. So:

$$10 \rightarrow .10 * 10^{1}$$
$$.1 \rightarrow .10 * 10^{0}$$
$$12.5 \rightarrow .13 * 10^{2}$$

If we were to add 12.5 and 1.0 as normalized floating-point numbers, we have $.13 * 10^{2} + .10 * 10^{1}$. The addition requires that we first align the decimal points by adjusting the decimal exponent of one of the two operands. A little reflection will show that it is better to scale the operand with the smaller exponent. So we pick $.10 * 10^{1}$, and keep shifting the decimal point left, increasing the exponent by 1 each time. We stop doing this when the exponent matches that of the other operand. So $.10* 10^{1}$ becomes $.01 *10^{2}$. The addition can then proceed:

$$\begin{array}{r} .13 * 10^{2} \\ +\underline{.01} * 10^{2} \\ .14 * 10^{2} \end{array}$$

Since the result is in normal form, the operation is complete. What if we had $.95 * 10^{1} + .60 * 10^{1}$? Then the result $1.55 * 10^{1}$ has to be normalized, giving $.16 * 10^{2}$.

What if we had to add 120 and 2 as floating-point numbers?

$$\begin{array}{rl} 120 \rightarrow .12 * 10^{3} \rightarrow & .12 * 10^{3} \\ 2 \rightarrow .20 * 10^{1} \rightarrow & \underline{.002} * 10^{3} \\ & .122 * 10^{3} \end{array}$$

This is already normalized, but it has to be truncated (or rounded) because the assumed precision allows for only two decimal digits. So we have $.12 * 10^{3} + x = .12 * 10^{3}$ with $x \neq 0$. This violates a fundamental law of arithmetic. Only the number 0 is supposed to have this property. This is an unavoidable consequence when using floating-point arithmetic. If you add a large number to a small non-zero number, the result may be identical to the larger number you started with. The operands are then said to have been incommensurate.

This problem shows itself when you are adding a series of floating-point numbers. Given three floating-point numbers a, b, c, the sum (a+b)+c should be the same as the sum a+(b+c). This is what the associative law of arithmetic tells us. It is very easy to find floating-point numbers which violate this law.

You can minimize this problem by first sorting the numbers, from low to high, then adding with the smaller operands first. This reflects the fact that the order of evaluation can affect the result.

Floating-point subtraction is very much like floating-point addition. Multiplication and division turn out to be a little simpler. To multiply the floating-point representations for 12 and 3 ($.12 * 10^2$ and $.30 * 10^1$), you merely multiply the fraction parts ($.12 * .30 \rightarrow .0360$) and add the exponents ($10^2, 10^1 \rightarrow 10^3$). The result ($.036 * 10^3$) can be normalized, making it $.36 * 10^2$.

Comparing Floating-Point Numbers

The shortest floating-point number occupies two words. The most significant word (with the sign, biased exponent, leading fraction bits) has the smaller address. Comparing two floating-point numbers begins with a comparison of the most significant words from each floating-point number. If they turn out to be equal, then the least significant words have to be compared. Since bit 15 of these is not being used as a sign bit but as part of the fraction, the low words are regarded as unsigned numbers, so their comparison should be followed by branches appropriate for unsigned numbers.

What Can Go Wrong?

Many things can go wrong when you are working with floating-point numbers. Every floating-point addition, subtraction, multiplication, or division is capable of producing one of the following two errors:

1. floating-point overflow
2. floating-point underflow

Using the simple decimal format for our floating-point numbers, we can illustrate these errors here. Given $.90 * 10^5$ multiplied by $.80 * 10^5$, we get $.7200 * 10^{10}$. The resulting exponent is too large to be stored in the exponent field. This is a case of *floating-point overflow*. Consider another example: $.25 * 10^9$ added to $.80 * 10^9$ produces $1.05 * 10^9$, which when normalized gives us $.11 * 10^{10}$. This is another overflow situation, triggered not by the immediate arithmetic operation, but afterward, during the process of normalizing the result.

Consider:

$$\begin{array}{rl} & -.12 * 10^{9} \\ + & \underline{.13 * 10^{-9}} \\ & .01 * 10^{9} \rightarrow .10 * 10^{-10} \end{array}$$

Here the exponent is too small to be represented in the exponent field. This is a case of *floating-point underflow*.

One of the subtle problems with floating-point numbers stems from an earlier observation. It is possible for two nonzero floating-point numbers a and b to satisfy the equality a+b=a. In effect, the number b is "too small" to have any effect when added to a. The problem then shows up as follows. You are looking for a floating-point number x which satisfies some relation such as f(x)=0. The function f(x) might be a polynomial such as $f(x) = 3.5*x^2 + 2*x - 5$, and you are trying to find the roots of this quadratic expression. In a high level language, you might write something like:

```
....
if ( f(x) ≠ 0 ) try again
...
```

The correct way to state this, if x and f(x) are floating point-numbers, is:

```
if( f(x) is not small enough ) try again
```

You have to define what "small enough" means. It depends not only on the specifics of the floating-point number representation, but also on the problem being solved. If you make "small enough" too small, your program might iterate forever, looking for the right "x" and never finding it. Writing quality mathematical software is an art as well as a science, as the report in figure 13.3 indicates.

Double-Precision Floating-Point Arithmetic

For some applications, a 24-bit fraction is not adequate. In that case, the PDP-11 supports a double-precision floating-point format in which four consecutive words are involved. The first two words use the same format used in "normal" precision, which we will now call single-precision floating point. The additional two words are devoted entirely to storing an extra 32 bits of fraction. So the name "double-precision" is misleading, since you are now getting much more than double the original precision—24+32=56 bits instead of 24.

The range has not changed, because the biased exponent field is still only 8 bits wide.

Converting Fractions

We discussed how the external representation of integers could be converted into binary numbers some time ago. It involved reducing the ASCII code for each digit into its binary equivalent and then, proceeding from left to right, multiplying each binary equivalent by successive powers of 10. We added the results, and got the desired result. Thus, given the three-digit string 234, we first see the bytes 62, 63, and 64. We reduce these to 2, 3, and 4. Then we compute 2*12*12, plus 3*12, plus 4, in octal. This leaves us with the result 352. We do it in octal for our convenience; the CPU does it in binary. How does this relate to converting a decimal fraction into a binary fraction?

Figure 13.3 IMSL square root problem.

(Reprinted from Numerical Computations Newsletter, November 1978.)

The Computing Center at Purdue University has made the IMSL Library available to its users since 1973 on a CDC 6500 system. The Center has published a series of articles illustrating library features and promoting usage.

The following reprint* illustrates characteristics of ZQADR from the Zeros and Extrema Chapter of the library. ZQADR finds the roots of a quadratic equation having real coefficients. The article has been modified slightly for editorial purposes.

"The object of this article is to illustrate, by way of an example, the quality which has been built into the IMSL Library. We do this by writing a subroutine to solve a rather simple problem, using the algorithm many people would use, and then by comparing the results we obtain with those of the corresponding IMSL routine.

"The problem we choose is to find the roots of a quadratic equation: given real numbers a, b, and c, find x such that $ax^2 + bx + c = 0$. For simplicity, we assume that a, b, and c are such that the solution is also real. The two roots of a quadratic equation may be found by the well-known 'Quadratic Formula,'

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

where one root is obtained by using the "+" of the "±", and the other is obtained with the "−". The assumption that the roots are real means that $b^2 - 4ac \geq 0$. We can solve this problem with the following straight forward subroutine:

```
      SUBROUTINE QUAD
      (A,B,C,X1,X2)
      D = SQRT (B*B−4.0*A*C)
      X1 = 0.5*(−B+D)/A
      X2 = 0.5*(−B−D)/A
      RETURN
      END
```

"When we use this subroutine to solve the rather difficult quadratic $x^2 + 10^7 x + 1 = 0$ we obtain

$X1 = -8.940696716309 \cdot 10^{-8}$,
$X2 = -1.000000000000 \cdot 10^{7}$.

This is not the correct solution, however. The corresponding IMSL routine ZQADR does compute the correct solution which is

$X1 = -1.000000000000 \cdot 10^{-7}$,
$X2 = -1.000000000000 \cdot 10^{7}$

to the number of digits shown.

"Where did QUAD go wrong? The second statement computes D slightly less than 10^7. Then the third statement forms the difference between D and B, losing almost all significance in the process. ZQADR is more careful than QUAD and thus is able to retain full significance.

"Now we touch briefly on several additional problems with QUAD. The first deals with problem scaling. If the quadratic equation above is multiplied by a constant, the solution is not changed mathematically, but it is changed computationally. For example, the quadratic

$$10^{200}x^2 + 10^{207}x + 10^{200} = 0$$

results in a MODE 2 (use of infinite operand) error because 10^{400} cannot be represented by the computer. Similarly, the quadratic

$$10^{-200}x^2 + 10^{-193}x + 10^{-200} = 0$$

yields the solution

$X1 = X2 = -5.000000000000 \cdot 10^{6}$

because 10^{-400} cannot be represented by the computer and is treated as zero. However, ZQADR still computes the same, correct solution.

"Finally, consider QUAD's actions if the coefficient of x^2 in the quadratic is zero. In this case QUAD returns an infinite value for one root and an indefinite value for the other. ZQADR returns the mathematically correct value $-c/b$ for one root and infinity for the other.

"What is the point of this article if you never solve difficult quadratic equations? The point is that if it is hard to solve a simple problem and take all of the various problem areas into consideration, why attempt a really difficult problem yourself when robust state-of-the-art routines are already available? In other words, before writing a routine to solve a particular problem, check the IMSL Library first—you might save yourself a lot of time and trouble."

*NEWSLETTER, Purdue University Computing Center, Vol. X, No. 5, West Lafayette, IN, p. 9–10.

The process of converting a decimal fraction, in its external representation, is the reverse of the above. Instead of using repeated multiplications, we use repeated division. Let us examine the simplest case first. Given the digit string .5, we interpret this to mean 5/10, which we instinctively reduce to 1/2, and then to .1 in binary, since the binary fraction $.b_1b_2b_3b_4$ means $b_1*2^{-1} + b_2*2^{-2} + b_3*^{-3} + b_4*^{-4}$.

A program which reads the two-character string .5 will see the bytes 056, 065. The program can determine how many decimal digits appear to the right of the decimal point (056). In this case, we have only one digit. We convert all the fraction digits as if they formed an integer. Here we simply get the value 5. Computer division of 5 by the implied denominator 10 will get us nowhere, since the numerator is smaller than the denominator. So we *double* our number, getting 2*5, or 10 (decimal). We repeat the division; the quotient we keep as the first fraction bit, and we repeat the process on the remainder, so long as the remainder is not 0. This algorithm works whether you do the actual arithmetic in decimal, binary, octal, or any other base.

Let us work out a second example. Given .25, this implies 25/100.

double 25 → 50

divide by 100, getting quotient q1 =0, remainder 50

Repeat

double remainder of 50 → 100
divide by 100, getting quotient q2 =1, remainder 0

Stop, since remainder is 0

Binary fraction is q1 q2, .01 (=.25)

We can follow exactly the same steps in binary and obtain the same results:

25 → 11 001 binary
double 11 001 → 110 010 (just add an extra 0)
divide by 1 100 100 (=100 decimal)
obtain quotient q1=0, remainder 110 010

Repeat

double remainder 110 010 → 1 100 100
divide by 1 100 100 (=100 decimal)
obtain quotient q2=1, remainder 0

Stop, since remainder is 0.

Binary fraction is q1 q2, .01 (=.25)

It is, of course, possible that the process will not terminate. Just as 1/3 has no finite representation as a decimal fraction, exact decimal fractions such as .1 (one-tenth) do not necessarily have finite representations in other bases. Fractions with nonterminating representations are called repeating fractions. For these, we stop the conversion process when we have generated sufficiently many fraction bits.

Since we are interested in converting mixed numbers (i.e., having both an integer and a fraction part), what do we do next? When we have generated the desired binary fraction, we simply write it next to the binary integer we generated separately, normalize, adjust the biased exponent, throw away the leading fraction bit, attach the sign, and we have the desired floating-point representation.

Software and Hardware Support for Floating Point

MACRO-11 provides two directives for use in defining floating-point constants. These directives are:

```
.FLT2 list of decimal numbers
.FLT4 list of decimal numbers
```

These are similar to the .WORD directive in that several constants may appear in the operand field. Each one of them will be stored in either two words (for .FLT2) or four (for .FLT4). The decimal numbers may be signed, they may have an integer or fraction part or both, and a signed decimal exponent can be introduced using the E notation. The following

```
.FLT2 10,10.,10.0,10E0,1E1,100E-1
```

defines the same single-precision floating-point number six times and places the appropriate bit pattern in six consecutive word pairs.

So much for the software support of floating-point numbers provided by the assembler. The hardware situation is somewhat chaotic, as indicated in figure 13.4.

Figure 13.4 PDP-11 floating-point hardware.

Model	Instructions	Comments	Timing
11/20	none	program a FADD	~50-100 usec
LSI/11	FADD FSUB FMUL FDIV	optional uses %0-%5	42-232 usec
11/40	same as LSI-11	optional	18-35 usec
11/34a	ADDF SUBF MULF, etc	optional	8-35 usec
	ADDD SUBD CMPD, etc	double-precision	
LSI-11/23		As for the 11/34a	
11/45	as for 11/34a		1-6 usec
11/60	as for 11/34a	these use AC0-AC5	
11/70	as for 11/34a		

Using the optional floating-point hardware available for an LSI-11 or a PDP-11/40, you can add two single-precision floating-point numbers A and B as shown in the program in figure 13.5. The FADD uses a general register in its operand field. The register is expected to hold a pointer to a six-word block which holds the two inputs followed by space for the output. In no time at all you would be defining macros such as DOFADD, so you can simply write

```
DOFADD A,B,C
```

instead of all of the above.

The higher-performance extra-cost floating-point hardware processors available for the PDP-11/34a, /44, . . . have a new set of registers just to handle floating-point operations. The floating-point processors used on the PDP-11/60 and 11/70 operate in parallel with the CPU, somewhat like the overlapped I/O operations we discussed earlier. Six 64-bit registers named AC0, AC1, . . ., AC5 are provided, and instructions for moving/storing FLT2/FLT4 type operands are provided, along with a full set of floating-point operations. Even CMPD is provided, to eliminate having to perform as many as four comparisons when comparing two double-precision operands. Using the optional floating-point hardware of the other PDP-11s, you would write the code shown in the program in figure 13.6. In this example, the LDF instruction copies a 32-bit floating-point operand, using the source address provided by (%0), into the 64-bit floating-point register AC0. The incrementation implied by (%0)+ will use an increment of 4, since LDF always copies two consecutive words.

```
A:      .FLT2   12.5E1
B:      .FLT2   1
C:      .BLKW   2 ; stack space
BLK:    .BLKW   6
        ...
        MOV     #BLK,%1 ; set ptr to data
        MOV     B,(%1)+ ; first real
        MOV     B+2,(%1)+
        MOV     A,(%1)+ ; second real
        MOV     A+2,(%1)
        MOV     #BLK,%1 ; reset data pointer
        FADD    %1
        MOV     (%1)+,C ; results
        MOV     (%1),C+2
        ...
```

Figure 13.5 Use of floating-point-instructions.

```
MOV     #A,%0 ; set data pointer
LDF     (%0)+,AC0 ; copy 32 bit operand → AC0
LDF     (%0)+,AC1 ; copy second operand → AC1
ADDF    AC0,AC1 ; perform floating point add
STF     AC1,(%0) ; store result in memory
```

Figure 13.6 Using the ADDF instruction.

In a similar fashion, we could use instructions LDD and STD to load and store double-precision floating-point operands, using the optional floating-point hardware. Any use of auto-increment or auto-decrement in conjunction with these instructions would cause an increment of 8 to be used, since four words are involved.

The LSI-11 and PDP-11/40 use the ordinary registers in performing single-precision operations. There is no hardware support for double-precision operations. The LSI-11/23 has a floating-point option compatible with that of the PDP-11/34a.

Floating-Point Traps

When the result of a floating-point operation is non-0, but too small to represent as a normalized floating-point number, it triggers a *floating-point underflow trap*. A trap is handled like an interrupt. The vector address 244 is associated with all the floating-point problems which might trigger an interrupt; the FPP (floating point processor) status register can be examined to determine which problem is being reported. On an LSI-11 or PDP-11/40, the CC bits would indicate whether overflow, underflow, or divide by 0 was involved. As a general rule, when floating-point underflow is reported to you, you might proceed by forcing the result to be 0.

When the result of a floating-point operation is too large to represent as a normalized floating-point number, the FPP triggers a *floating-point overflow trap*. If you were not expecting such a situation, the only reasonable thing to do is to stop execution and start thinking. If you were expecting it, you could slip into a programmed extended-range floating-point arithmetic. Few people do that, because it tends to be slower by a factor of 50 or so.

In any case, floating-point overflow cannot be lightly dismissed. Some years ago a renowned research institution went through a transition from an older brand X computer to a newer, more powerful brand Y computer. Users began moving their FORTRAN programs from computer X to computer Y. The computing center began getting complaints that "machine Y is messing up our output listings" because it was reporting floating-point overflow errors. Since the programs in question had been running satisfactorily on machine X for many years, clearly the programs could not be at fault. It had to be a problem with the new machine Y. The management yielded to pressure from irate users, and the floating-point trap interrupt-enable bit was reset. The customers went away happy. After all, they were getting the answers they expected—just like the Dr. Feldstein we will see in the next section (he was not involved in the case being discussed).

It seems that brand X used a 64-bit word with a very large exponent field for its single-precision numbers, whereas brand Y used a "more efficient" word of 32 or 36 bits, with a correspondingly smaller exponent field. Obviously, some operations not leading to floating-point overflow on machine X could and would lead to overflow on machine Y.

Who is to blame in this incident? It seems clear to the author that the computer center management, presumably aware of the significance of floating-point overflow, should not have yielded. Any continued execution after a single floating-point overflow is complete nonsense. The researchers should not be let off the hook. Their failure to look into why their time-tested programs were suddenly misbehaving is inexcusable. Pity the unsuspecting users who began using system Y only after the floating-point trap had been disabled. They may never know why their results were so good (or so bad).

Feldstein's Computer Error

In some areas of endeavor you get to bury your mistakes. In Martin Feldstein's case (see the newspaper article in figure 13.7), his mistake received nationwide attention. His error (actually caused by his overconfidence in someone else's programming skill) was accepting the computer's output because it reinforced his expectations.

Arrays

We have been using single-dimensional arrays for a long time. The following

```
A:      .WORD   2,4,-7
```

is a one-dimensional array of three words, and each word is regarded as a signed integer.

```
B:      .BYTE   177,0,307,041
```

is a one-dimensional array of 4 bytes, each of which holds an unsigned number.

```
C:      .ASCIZ  /THIS IS A TEST/
```

is an array of bytes whose length, instead of being specified by an explicit count, is implied by the first 0 byte encountered. Finally,

```
D:      .FLT2   -24.5E34,0,E12,.56E-21
```

is an array of four single-precision floating-point numbers.

Arrays have three distinguishing features:

1. The dimensionality does not change once set.
2. The structure is predetermined (1 by 5, 2 by 3, etc.).
3. Each array element is of the same type.

The last property is often given the name "homogeneity" (as with milk). Arrays are easy to work with because there are no surprises. Each item or element occupies the same amount of space as any other item in the array (otherwise, don't call it an array; call it a table, or something else). The size of the whole array is fixed, as is the number of subscripts.

Figure 13.7 Feldstein's Computer Error.

(© 1979/80 by The New York Times Company. Reprinted by permission.)

Martin Feldstein's computer error

By KAREN W. ARENSON

Do people save less of their income because they know Social Security will provide something to live on in retirement? Yes indeed, concluded an important study six years ago by one of the country's best-known economists, a man who is sometimes mentioned as a likely head of the Council of Economic Advisers in a Reagan Administration.

That study, by Martin S. Feldstein, the 40-year-old head of the National Bureau of Economic Research in Cambridge and a Harvard professor, is now in question. Six weeks ago, two economists in the Social Security Administration, Dean R. Leimer and Selig D. Lesnoy, disclosed a fundamental flaw in the Feldstein paper and their findings have stirred a ruckus in the world of economics.

The computer program that Mr. Feldstein used, it turned out, erred in repeating a calculation over and over again. As a result, the value that consumers place on Social Security was greatly overestimated. That means, said the authors, that the Feldstein conclusion cannot be proven or disproven on the basis of available work.

Do economists often make mistakes of such magnitude? "There are probably untold numbers of errors buried in the economic literature," says Alan Blinder, an economist who has worked on his own Social Security studies, and is spending his sabbatical from Princeton doing research at the National Bureau of Economic Research. "If you had made a small programming mistake," he said, "it would probably not be discovered unless it had produced crazy numbers, and Marty's model did not."

Mr. Feldstein contends that a corrected version of his model still points to his original conclusion. And, he adds, "My sense is that there is a general belief in the profession that Social Security still depresses savings, although the evidence is not finally in on the magnitude of that effect."

He says a Harvard student wrote the program, but that it was reread by people familiar with programming, and he takes responsibility for the error. "It's hard to think of what else one can do short of having all work done independently a second time," he says. "That would cost twice as much."

Mr. Feldstein contends that once some crucial 1972 legislative changes are incorporated into his model, the conclusion is still justified. He criticizes the other authors for leaving those changes out of their model, changes he says he himself did not realize were necessary until they brought the error to his attention. He points out that with the error corrected, his model, which he says still made sense when based on data through 1971, no longer made sense when it incorporated data from after 1972.

"When I simply extrapolated my model through 1974 and the results looked quite consistent, I let it go at that," he said. "But when it no longer worked, I had to think long and hard about what had happened in 1972, and realized that something important had happened that I hadn't realized before."

The two Government economists respond that they did not incorporate the 1972 legislation in their analysis because Mr. Feldstein had not included those changes when he extended his data through 1974 from its earlier end of 1971. They have not seen Dr. Feldstein's new work.

More importantly, the two economists say that they did incorporate the legislative changes in their other approaches, still without showing that Social Security reduces saving. They note that Mr. Feldstein has not yet addressed himself to that work, other than to call the lack of proof of an effect improbable. (He says he will, at a later date.)

Yet, they say, Mr. Feldstein himself has pointed out that Social Security may cause people to retire earlier, thereby increasing the need for retirement saving, so that the net effect on saving is in principle ambiguous. Their views, they say, do not necessarily represent the views of the Social Security Administration.

What They Found...

By DEAN R. LEIMER
and SELIG D. LESNOY

Two opposing theoretical arguments have been advanced concerning the effect of Social Security on saving. One recognizes two potential effects:

First, by promising a stream of future benefits during retirement, Social Security reduces the need for retirement saving. Because Social Security is financed on a pay-as-you-go basis, there is no governmental offset to this reduction in personal saving, so national saving is reduced.

The first argument also recognizes an offsetting effect. Social Security may induce earlier retirement, so that workers save at a higher rate over a shorter working life.

The net effect of Social Security on saving depends on the relative strength of these opposing forces. The competing argument suggests that Social Security transfers may simply be offset by changes in private intergenerational transfers (such as children's support of parents), so that Social Security has little or no impact on saving.

Most of the evidence relating to this issue is based on consumption functions—economic models explaining national consumer spending—estimated by using historical data and incorporating a variable that measures perceived "Social Security wealth." (the estimated value of expected future benefits).

In a pioneering study completed in 1974, Mr. Feldstein concluded that Social Security had reduced personal saving by 50 percent with serious consequences for capital formation and output. Other major studies have yielded smaller estimated effects on saving or concluded that there was no evidence of a significant effect.

Noting that all of the studies used the Social Security wealth series constructed by Mr. Feldstein, we wondered whether the results were sensitive to certain assumptions underlying his construction.

First, we corrected a previously undiscovered error in the Social Security wealth series constructed by Mr. Feldstein, which substantially changed his original results.

Second, we modified his algorithm for computing Social Security wealth by incorporating a wide range of alternative assumptions about the ways in which individuals form their estimates of Social Security wealth and then adapt them to changes in legislation.

Third, we also used a somewhat more elaborate algorithm of our own construction. When we made these changes, we found that the estimated effect of Social Security on saving was sometimes positive and sometimes negative, but, in the main, consistent with the hypothesis that there was no effect. In no case was a significant negative effect on saving indicated.

Where does this leave us? Prior to our study, we accepted the conventional wisdom that Social Security probably depressed saving, although by less than the amount estimated by Mr. Feldstein. But the new evidence, presented in our study, suggests that the introduction of the Social Security system has not significantly reduced personal saving.

Further examination of the historical data may yield different results. It is unlikely, however, that such examination will provide a definitive answer, in part because we simply do not know how individuals perceive their Social Security wealth, and in part because of inherent difficulties of using historical data.

Unlike experimental scientists, economists cannot conduct and easily replicate controlled experiments, and the data generated by history have the unfortunate characteristic of moving together over time. This makes it difficult to sort out the unique effect of a particular variable and accounts for much of the disagreement among economists concerning the effect of Social Security on saving. The evidence at hand, however, does not support policy recommendations designed to offset a negative impact of Social Security on saving.

...And His Defense

BY MARTIN S. FELDSTEIN

When a man who has had average earnings all his life retires at the age of 65, he and his wife receive Social Security benefits equal to two-thirds of his peak earnings point. Since Social Security benefits are not taxed, his monthly Social Security check replaces about 80 percent of his previous take-home pay. And since benefits are now indexed to keep up with the inflation in prices, the purchasing power of these benefits is guaranteed for life.

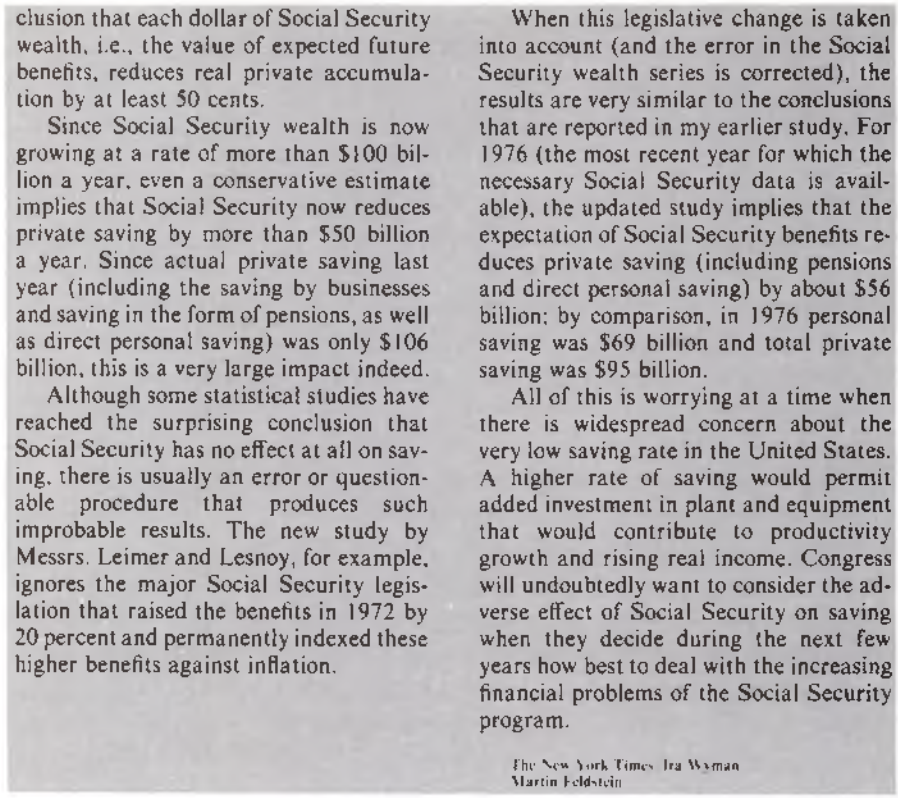
This obviously leaves most employees with little incentive to save during their earlier working years—when their earnings are lower and their family responsibilities are greater—just to add to the benefits that Social Security will provide when they are retired. Saving in the form of private pensions is also kept down for the same reason. As a result, most retirees now depend primarily on Social Security to finance their retirement and most employees save very little for their old age.

Because of the pay-as-you-go character of the Social Security program, this has serious consequences for the nation. The Social Security taxes that we pay are used to finance the benefits of current retirees. The Government doesn't attempt to develop a capital fund as a private pension plan would. As a result, there is no extra government saving to offset the fall in private saving.

Economists have been concerned about the adverse impact of Social Security on private saving and have done a number of studies to try to measure the extent of this effect. As Messrs. Leimer and Lesnoy note, it is difficult to make a very precise estimate. My own analysis of the evidence, including not only the historic data that they referred to but also the studies based on extensive surveys of household assets and the analyses of international differences in saving, lead me to the con-

Figure 13.7 Continued.

Figure 13.7 Continued.



clusion that each dollar of Social Security wealth, i.e., the value of expected future benefits, reduces real private accumulation by at least 50 cents.

Since Social Security wealth is now growing at a rate of more than $100 billion a year, even a conservative estimate implies that Social Security now reduces private saving by more than $50 billion a year. Since actual private saving last year (including the saving by businesses and saving in the form of pensions, as well as direct personal saving) was only $106 billion, this is a very large impact indeed.

Although some statistical studies have reached the surprising conclusion that Social Security has no effect at all on saving, there is usually an error or questionable procedure that produces such improbable results. The new study by Messrs. Leimer and Lesnoy, for example, ignores the major Social Security legislation that raised the benefits in 1972 by 20 percent and permanently indexed these higher benefits against inflation.

When this legislative change is taken into account (and the error in the Social Security wealth series is corrected), the results are very similar to the conclusions that are reported in my earlier study. For 1976 (the most recent year for which the necessary Social Security data is available), the updated study implies that the expectation of Social Security benefits reduces private saving (including pensions and direct personal saving) by about $56 billion; by comparison, in 1976 personal saving was $69 billion and total private saving was $95 billion.

All of this is worrying at a time when there is widespread concern about the very low saving rate in the United States. A higher rate of saving would permit added investment in plant and equipment that would contribute to productivity growth and rising real income. Congress will undoubtedly want to consider the adverse effect of Social Security on saving when they decide during the next few years how best to deal with the increasing financial problems of the Social Security program.

The New York Times Ira Wyman
Martin Feldstein

One-Dimensional Arrays

Since we have been dealing with one-dimensional arrays for so long, what can we add now? Just a little more, regarding the distinction between 0-origin and 1-origin indexing. In all our work in assembly language, the use of 0-origin indexing has been implicit. However, since 1-origin indexing is prevalent in everyday life (we usually think of chapter 1 as the first chapter, not chapter 0), and since it is common in high level languages such as FORTRAN, let us clarify the distinction.

Let A be an array of five words. In some high level languages the five elements would be referred to thus:

```
A(1), A(2), ..., A(5)
```

In assembler we define the array thus:

```
A:          .BLKW   5
```

and its five elements have the following addresses:

```
A+0, A+2, ..., A+8.
```

Pictorially, we have:

0-origin i					
	0	1	2	3	4
FWA	A(1)	A(2)	A(3)	A(4)	A(5)
	1	2	3	4	5
1-origin j					

FWA (first word address) is equivalent to the array's base address A. Given 0-origin subscript variable i,

```
MOV     i,%1
ASL     %1
MOV     A(%1),X
```

will fetch item A(i+1). If we have 1-origin subscript j, then

```
MOV     j,%1
ASL     %1
MOV     A-2(%1),X
```

or

```
MOV     j,%1
DEC     %1
ASL     %1
MOV     A(%1),X
```

will copy item A(j) into X. With this background we can now look into the more interesting problem of mapping a two-dimensional array, or an n-dimensional array, into a one-dimensional computer memory.

Two-Dimensional Arrays

As we prepare to discuss n-dimensional arrays, let us start with a two-dimensional array. Suppose we wanted to have a 2-by-2 array called A. Using 1-origin indexing, its elements have the indicated subscripts:

	Column Number	
Row	1	2
1	1,1	1,2
2	2,1	2,2

By row

```
A(1,1)  A(1,2)  A(2,1)  A(2,2)
Row 1           Row 2
```

By column

```
A(1,1)  A(2,1)  A(1,2)  A(2,2)
Column 1        Column 2
```

Figure 13.8 Storing in memory.

Figure 13.9 Two-by-three array.

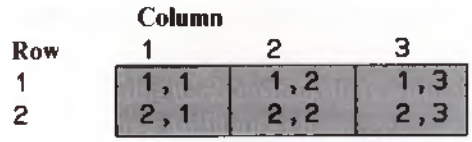


Figure 13.10 Mapping subscript pairs into offsets.

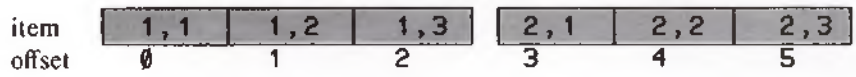
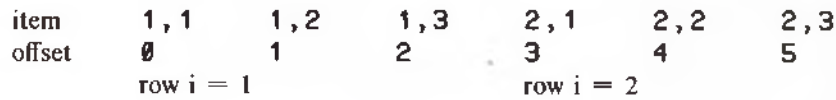


Figure 13.11 Mapping subscript pairs into offsets.



How can we store the four items corresponding to these positions in memory without complicating access to them? There are basically two choices. The choice may seem trivial, but it can have important consequences. The designers of FORTRAN elected to store two-dimensional arrays by column. This is called *column-major ordering.* A few years later the designers of ALGOL, well aware of the FORTRAN precedent, elected to store arrays by rows. This is called *row-major ordering,* as you might expect. As a consequence, if you transfer your arrays from one program to another, or from one computer to another, the linear ordering of your multi-dimensional arrays used by a FORTRAN program are incompatible with those used by ALGOL programs. Computer processing time is required to convert one linear representation into the other, unless you choose to modify the programs involved. Neither approach is desirable.

The designers of FORTRAN presumably elected to use column-ordering of arrays because the existing applications they were aware of, programmed in assembly or machine language, were already using this ordering. The ALGOL choice of row-ordering can be defended on the basis of universal practice, the so-called lexicographic-ordering. Does A(2,1) come before or after A(1,2)? The lexicographer (people who write dictionaries) says "A(1,2) comes first because 1 comes before 2." So, since row-ordering is consistent with the way we file everything else, why deviate?

Consider figure 13.9, which shows a 2-by-3 array of two rows and three columns. The 1-origin subscripts for each array element position are written in the position the element should occupy. If we map this array into a linear memory using lexicographic ordering (i.e., row-major ordering), we get figure 13.10. The relationship between the subscript pairs i,j and the 0-origin offsets is:

$$\text{offset} = (i-1)*3 + (j-1)$$

We can see this more clearly if we underscore the two rows: The relationship between the coordinates (i.e., the subscript pairs i,j) of an n-by-m array and its mapping into a linear array is given by:

offset = (i−1)*m + (j−1), or
offset = (row subscript −1)*number of columns
+ (column subscript −1)

N-Dimensional Arrays

As we go from two-dimensional arrays to three-dimensional, four-dimensional, . . ., N-dimensional arrays, the mappings generalize as follows. Given an array A of dimension N, it has N subscripts i_1 , i_2 , . . ., i_N. If each of them assumes integer values between the limits 1 and S_j—that is, $1 \leq i_j \leq S_j$ for j=1, 2, . . ., N−1, N—then:

$$\begin{aligned}\text{offset} = \quad & (i_1 - 1)*S_2 *S_3 * \ldots *S_{N-1} *S_N \\ & + (i_2 - 1)*S_3 *S_4 * \ldots *S_N \\ & + \ldots \\ & + (i_{N-1} - 1)*S_N \\ & + (i_N - 1)\end{aligned}$$

We can illustrate this with a 1-by-2-by-3 array D. The parameters are:

dimensionality N = number of subscripts = 3
number of rows S_1 = 1
number of columns S_2 = 2
number of layers S_3 = 3
Total size of D in units of one : 1*2*3 = 6

It is entirely coincidental here that the values for S_1, S_2, S_3 are the same as their subscripts. Given the base address D as the first word address (FWA), the symbolic higher level reference to an element of D, given by D(i,j,k), maps as follows:

$$\begin{aligned}\text{FWA} + & (i - 1)*S_2 *S_3 \\ & + (j - 1)*S_3 \\ & + (k - 1)\end{aligned}$$

or

$$\begin{aligned}D + & (i-1)*2*3 \\ & + (j-1)*3 \\ & + (k-1)\end{aligned}$$

which is

$$D + (i-1)*6 + (j-1)*3 + (k-1)$$

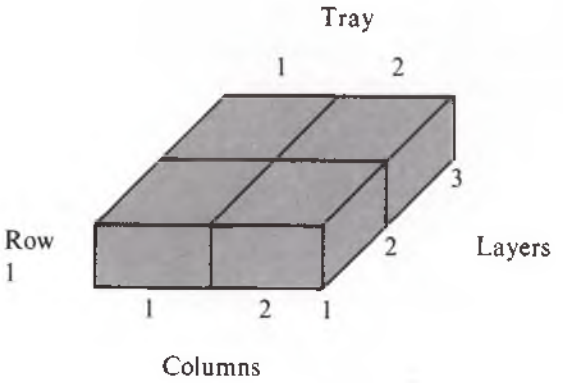


Figure 13.12 Three-dimensional array.

Figure 13.13
Lexicographical order.

We can picture the array D as shown in figure 13.12. The lexicographical order is as in figure 13.13. Clearly the left-front corner element at D(1,1,1) maps into the address D+0 +0 +0 = D. As we step through the array in lexical order, the third subscript varies most rapidly (just as the last letter of the words beginning with A would in a dictionary). The middle subscript varies not quite so fast, and the first subscript most slowly of all.

We can add one important refinement in the array offset formula. If we let w be the element addressing increment, then we can build in the correction for handling arrays which have elements larger than bytes.

$$
\begin{aligned}
\text{Offset} = \quad & w*(i_1-1)*S_2*S_3*\ldots*S_{N-1}*S_N \\
+ \quad & w*(i_2-1)*S_3*S_4*\ldots*S_N \\
+ \quad & \ldots \\
+ \quad & w*(i_{N-1}-1)*S_N \\
+ \quad & w*(i_N-1)
\end{aligned}
$$

where w = 1 for bytes
2 for words
4 for single precision floating point
8 for double precision floating point
w bytes, for each array element

Computing Array Displacements

There are basically three ways of converting a subscript reference into an offset or displacement for the desired array element. These are:

1. in-line code
2. dope vector
3. hybrid

```
...
MOV     I,%1        ; subscr I
DEC     %1          ; reduce to 0-origin
ASL     %1          ; (I-1)*2
ASL     %1          ; (I-1)*4=(I-1)*NCOL
MOV     J,%2
DEC     %2          ; (J-1)
ADD     %2,%1       ; (I-1)*NCOL +(J-1)
ASL     %1          ; word offset
ADD     A(%1),SUM
...
```

Figure 13.14 In-line array reference code.

In-Line Code

In-line code is what you usually write when you want something done here and now. You simply write all the instructions needed where they are needed, as opposed to calling a subroutine. If you are trying to compute

ADD item A(i,j) to SUM

you can write the in-line instructions to fetch item A(i,j). For a 3-by-4 array A of integer words, you can write the code shown in the program in figure 13.14. Note that here it takes eight instructions to generate the array component address we need.

Dope Vectors

A dope vector access to array elements implies that you will use a subroutine call for each array reference, and provide the subroutine with a description of the relevant array parameters and its subscripts. This description of an array is traditionally called a dope vector ("dope" is slang for "information"). There is no standard format for a dope vector, but it could contain the following information:

1. array base address
2. array size (not essential, but useful)
3. array element type (byte, word, ...)
4. number of subscripts
5. subscript range for each subscript
6. addresses or values for current subscripts

Given this information, you can arrange to place it in a dope vector (i.e., in a table). Thus, a 2-by-3 array of words beginning at location ABC could have the following dope vector:

```
DVABC:  .WORD   ABC     ; base address
        .WORD   6       ; size 2*3
        .WORD   2       ; 2= "word"
        .WORD   2       ; 2 subs
        .WORD   2       ; subsc 1 runs from 1 to 2
        .WORD   3       ; subsc 2 runs from 1 to 3
        .WORD   I,J     ; current subsc at addresses I and J

        ...
```

A 3-by-2-by-4 array of bytes starting at Q could have the following dope vector:

```
QDV: .WORD Q,24.,1,3,3,2,4,I1,I2,I3
```

The subroutine which interprets such a descriptor block (which is what a dope vector is) can have the following invocation:

```
JSR     %5,SUBSC
.WORD   DVABC
MOV     ABC(%0),X
...
JSR     %5,SUBSC
.WORD   QDV
MOVB     Q(%0),Y
...
```

Subroutine SUBSC uses its first argument to access the dope vector, then it interprets the dope vector items and leaves the offset it computes in %0. It could have left the array element address in %0, in which case MOV (%0),. . . or MOVB (%0) would suffice.

A Hybrid Approach

The hybrid approach is appropriate for special situations when the array dimensionality is sufficiently small and ultra-high-speed array referencing is crucial.

Suppose you have an n-by-3 array of bytes at location X, and you are using variables I and J as one-origin subscripts. Fast access can be provided by using in-line code and a table of addresses. Consider:

```
        MOV     J,%1
        ASL     %1
        MOV     COL-2(%1),%1 ; column adr
        ADD     I,%1
        MOVB    -1(%1),%0
        ...
COL:    .WORD COL1,COL2,COL3
X:COL1: .BYTE   1,2,3,...,N; note 2 labels being defined
COL2:   .BYTE   ...
COL3:   .BYTE   ...
```

The high speed access to elements of X is provided at the cost of storing a set of column addresses. Obviously it is not practical to access an n-dimensional array this way if n is a very large number.

Array referencing of the form

Array(sub1, sub2, . . .,subn)

generally implies that you are accessing randomly selected elements of an array. That is, you are not stepping through consecutive array elements. If, indeed, you are, then the appropriate manner of doing so is to use auto-increment addressing. Failure to use auto-increment addressing when you could and using instead the direct addressing implied by the subscript evaluation is very expensive. The latter may reduce program-execution time by a factor of 50 when compared to using auto-increment addressing.

Clearing a 10-by-10 array takes only some 200 instruction executions (CLR,SOB) when you use the auto-increment mode. If you use a dope vector, it takes about 50 instructions per array reference, giving a total of 5,000 instructions executed, which makes it about 25 times slower. The speed degradation rises quickly as the dimensionality increases.

Why use a dope vector? It allows you to defer the array binding time until run time, when you can allocate both the array size and the array location. This run-time storage allocation (dynamic storage allocation) allows you to manage your memory use much better. This is a degree of flexibility for which the access overhead costs may well be justified.

What Can Go Wrong?

Subscripts might be out of bounds. For a 2-by-3 array using subscripts i and j, you expect

$$1 \leqslant i \leqslant 2, \text{ and } 1 \leqslant j \leqslant 3.$$

If it happens that i or j is not in the correct range, the offset might still be between 0 and 5, which seems fine, but of course it is wrong. On some computers, the hardware maintains an offset-limit register which can detect that you computed an offset that goes beyond the whole array. This is a useful feature, but it is not a substitute for checking to see that each subscript is in its correct range. If the subscript check passes, then the offset cannot possibly be wrong. The only way you can guarantee that the subscripts are in their correct ranges is to check each one of them, each time an array reference occurs, at run time. This is called run-time subscript checking. It may sound very expensive; but what is the cost of using the wrong array element because of a subscript error? If one is using dope vectors, it is very easy to incorporate run-time subscript checking as part of the dope-vector interpretation subroutine. The additional overhead is not too costly.

Other errors? The wrong "step size" might have been used. Consecutive addresses of an array of bytes differ by one; those for an array of words, by 2; those for an array of reals, by 4; etc. You are less likely to make an error in step size when using a dope vector than when using in-line code, for obvious reasons. In the former case, the step size is incorporated in a single copy of a single table (i.e., the dope vector). In the latter case, the step size is used repeatedly, perhaps several times, for each occurrence of the in-line code. Clearly, if you insisted on using in-line code for array referencing, it would behoove you to make up a set of macros for this purpose.

Other problems? Failure to pay attention to the relatively small amount of memory available to a PDP-11 user can lead to disappointment. Your largest direct address on a single-user system is 56KB; on a multi-user system, it is 64KB. This is true even if your machine has 4MB of physical memory, as is possible on a PDP-11/70. When you deal with floating-point numbers, a 64K array of bytes becomes a 16K array of single-precision floating-point numbers, or a mere 8K array of double-precision floating-point numbers. You could not even squeeze in a 20-by-30-by-40 array of reals, single or double precision! The only way you can handle large arrays on a computer with a small direct address space involves using the disk a lot, and if possible, interacting with the MMU using system service requests, so you could have different rows or columns of an array in different virtual memories.

Special Arrays

Two-dimensional arrays are also called *matrices*. An n-by-n array is also called a square matrix. When a square matrix can be folded in half, over the diagonal entries 1,1 2,2 . . . n,n without loss of information, the matrix is said to be symmetric. Most mileage tables found on maps are symmetric in this way. When most of the values in a matrix are zero, the matrix is said to be sparse. This terminology is used in many other disciplines (e.g., statistics, economics, etc.), and it is good to be familiar with it.

Summary

We have discussed the need for a number representation that provides a greater range than do integers on computers, and which does so without unnecessary loss of accuracy. Normalization of fractions is used to maximize the number of significant bits at all stages. The representation used is called floating point, and it is implemented on the PDP-11 in two forms. The first form, single precision, uses adjacent words to support numbers with an 8-bit biased exponent field, and a 24-bit fraction. The second form, double precision, uses four consecutive words, the first two of which are identical to the single-precision representation for the same number. The next two words contribute 32 more bits to the fraction.

Any arithmetic operation on floating-point numbers can give rise to floating-point overflow or floating-point underflow. These are such significant events that on computers which have floating-point hardware, special interrupts known as floating-point traps are triggered by these conditions.

Multi-dimensional arrays are an important tool in computing. We have seen the mapping between an n-dimensional array and its representation in computer memory. The two principal ways of storing such arrays are the row-major and column-major organizations. Array referencing is generally supported either by in-line code or by subroutine calls which interpret a dope vector. In special cases a hybrid approach may be warranted. This involves storing and using tables of array row or column pointers, in order to speed up access.

Exercises

13.1 For the following numbers, give the components of the PDP-11 floating-point representations, in octal, with the hidden 1 and the bias included:

(a) $-1/2$; (b) 36.75 ; (c) -127 ; (d) .1

13.2 For the following floating-point numbers:

N1 = 152 000 007 216, N2 = 043 200 000 000,
N3 = 036 003 011 157
N4 = 157 452 103 677, N5 = 062 313 107 430,
N6 = 107 657 043 724

(a) Which is the largest positive?
(b) Which is the largest negative (furthest from zero)?
(c) Which is nearest to 0?
(d) Which pair would produce floating-point overflow on multiplication?

13.3 Write the segment of code which places the value of the array element from row i and column j of a 3-row-by-4-column array of bytes in %0. Indicate the changes in the code necessary to:

(a) use it with a 3-by-4 array of words instead of bytes.
(b) use it with a 5-by-8 array of bytes.

13.4 What are the three major fields or components of a floating-point number used for? Give two reasons for always dealing with normalized floating-point numbers.

13.5 (a) What is the general formula for accessing element X(i,j,k) of a 3-by-4-by-5 array of integers?
(b) What advantage does using a dope vector provide?

13.6 Write the code required to fetch item i,j of a 32-row-by-16-column array of integer items stored by row (e.g., item 1,1 if followed by item 1,2, etc.).

13.7 Match the decimal and floating-point representations:

(a)	1.0	(i)	035 603 011 157
(b)	100	(ii)	000 000 000 000
(c)	1E100	(iii)	037 314 146 315
(d)	$-120E-3$	(iv)	000 000 000 000
(e)	.001	(v)	041 312 000 000
(f)	-0	(vi)	040 200 000 000
(g)	10E36	(vii)	040 511 007 732
(h)	3.1415926	(viii)	040 200 000 000
(i)	1024	(ix)	041 710 000 000
(j)	-1.234	(x)	140 235 171 666

(k)	E−5	(xi)	137 365 141 217
(l)	25.25	(xii)	042 600 000 000
(m)	.1	(xiii)	076 760 136 702
(n)	1.000 001	(xiv)	063 222 046 551

13.8 What are the values of the following floating-point numbers, which appear as they would in a dump printout?
(a) 040200 000000 (b) 037314 146315 (c) 063222 046551

13.9 Which of the numbers in question 13.8, if squared, would produce floating-point (a) overflow? (b) underflow?

13.10 What are the largest positive numbers (within a factor of 10) in each of the following categories which can be processed by a PDP-11?

(a) unsigned integer
(b) signed integer
(c) single-precision floating-point number
(d) double-precision floating-point number
(e) signed double-word integer

13.11 Suppose you were to redesign the PDP-11 floating-point number format so that it used a radix of 16 instead of 2 as the exponent radix. Discuss what this does to the range and precision of numbers in the new format. This happens to be the radix used on IBM 370 computers, which use one 32-bit word per single-precision floating-point number.

13.12 Write a program which converts the row-major 3-by-4 array A into its 3-by-4 column-major equivalent.

13.13 Write a macro which facilitates fetching a component of a two-dimensional array.

13.14 Write the code which can compare two single-precision floating-point numbers, using CMPs. Write it as a subroutine.

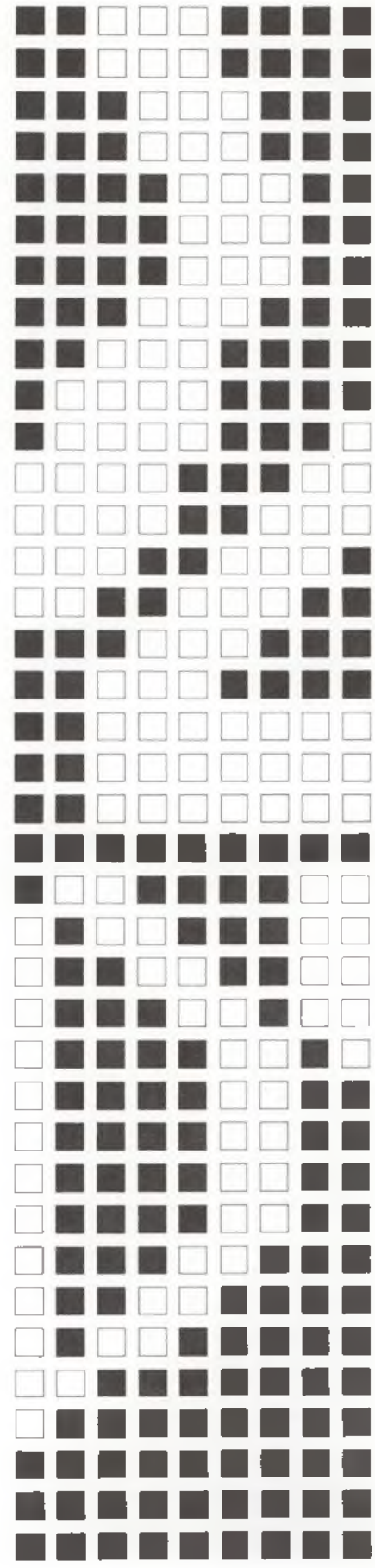
13.15 Write the code which can compare two double-precision numbers, using CMPs.

13.16 Convert the decimal fraction .1 into binary, providing the first ten bits. Is this a repeating fraction?

13.17 Sketch an algorithm which converts binary fractions into decimal fractions.

14

high speed I/O and other topics



The I/O we discussed previously is suitable for handling low- to moderate-speed devices such as CRT terminals, character printers, etc. Even then, the I/O overhead for just a few CRTs running at 9,600 baud may overwhelm a CPU. Consider the processing required for an output interrupt resulting from sending one byte to a CRT:

1. save PC, PS(2)
2. fetch new PC, PS (2)
3. TST, Bxy, Test, MOVB, INC (10)
4. restore PC, PS (3)

The parentheses hold the minimum number of memory references needed to do the work. They total 17. If the CPU's average instruction execution time is 1 μsec/instruction (about right for an 11/44 or 11/70; other models are slower), then a single output interrupt takes at least 17 μsec to process. At that rate, not quite 60,000 interrupts/second could be processed, if the CPU had nothing else to do. How many CRTs being driven at 9600 could be supported? The 9600 baud is equivalent to a maximum data rate of 960 cps (recall that 1 character = 8 information bits + 2 framing bits, at that speed). Almost 62 CRTs could be driven at that rate. Of course, the CPU could do absolutely nothing else; no computing, no other I/O, etc.

We can look at this from another point of view. If the CPU can handle only 60,000 interrupts per second, its I/O bandwidth (a measure of its ability to move data) cannot exceed 60KB/sec, which is ridiculously low. There must be a better way. Why this is too low will be clear after we examine the various I/O device types. Then we will examine a better way to perform I/O for the high speed devices.

Device Types

CRTs and other keyboard-print interactive terminals can be classified as *character-oriented devices.* Other character-oriented devices are:

1. paper tape reader, punch, reader-punch
2. RO character and line printers
3. card reader, card punch

These generally have a low to medium speed data rate. A 1,000 line/minute line printer which uses a 132-character line accepts 2.2 K cps. A 1,200 card/min card reader using 80-character (80-column) cards generates 1.6 K cps. The fastest CRTs using serial I/O run at 9,600, 2 * 9600 or 4 * 9,600 baud, giving data rates of 960, 1.92 K, or 3.84 K cps. In our earlier discussion, if we ran the CRTs at 38,400 baud, we could support no more than 15, instead of 62, if we insisted on getting the best possible response.

Block-Oriented Devices

Mass storage devices of one kind or another constitute a major category of devices directly associated with computers. A paper tape can store 10 characters per inch of paper tape, so that a 300-foot (100-meter) roll can hold 36 KB; a standard magnetic tape can hold millions of bytes; and disk storage systems can hold hundreds of millions of bytes. Not only can these devices hold large quantities of data, some of them can transfer data at rates in excess of one million bytes per second.

These mass storage devices are called *block-oriented devices,* because as a general rule you never send them (or expect to fetch from them) just one byte per operation. If you did so, the interrupt overhead for single-byte I/O transfers would reduce the device's data rate from a high of perhaps 1 MB/sec to something approaching 50 KB/sec, a degradation by a factor of 20. These block-oriented devices are constructed to work most efficiently when transferring blocks of information. The length of a block may be a few dozen bytes or several thousand bytes. We will discuss how I/O is performed with these devices after looking at their main characteristics.

Sequential Access Devices

The block-oriented mass storage devices can be separated into two groups. One involves sequential access devices. The *industry compatible magnetic tape (ICMT)* is the prime example of a sequential-access storage medium. ICMTs come in a standard size (standard by virtue of an ANSI (American National Standards Institute) standard) of 10.5-inch diameter reels which hold almost half a mile (800 m, 2,400 feet) of 1/2 inch-wide (2.5 cm) mylar tape with a ferromagnetic coating on one side (the side used for recording bits). When a tape is purchased (for as little as $15, although the price may increase rapidly, because tape is an oil by-product), you have to glue two silver markers on it to indicate the beginning and end of the usable section of tape. The tape drive (on which tapes are mounted, when they are said to be "on-line") can sense the BOT (beginning of tape) and EOT (end of tape) markers. After a tape has been used frequently, the portion near the BOT gets the most wear. It is interesting to see computer operators salvaging a tape by reeling off the first 30–40 feet, cutting it off, and gluing a new BOT on the remaining section.

Most magnetic tapes are certified by the manufacturer as capable of reliably accepting information to be recorded at 6,250 bpi (bits per inch). This number, *the tape recording density,* means that if you used a 6250 bpi tape drive, and you recorded bytes over a full inch of tape, 6250 bytes would be recorded. The measure "bpi" is a linear density indicating the bit density on a 1-bit-wide tape channel or track of the magnetic tape. To put things in perspective, a paper tape records at 10 bpi. Since it uses 8 channels, each of which records at 10 bpi, and 1 byte of 8 bits is written in one frame which spans all 8 channels, we get 10 cpi out of a 10-bpi paper tape. Similarly, we can obtain 6,250 cpi of storage capacity from a 6,250 bpi magnetic tape. The nomenclature "fpi" is also used in place of "bpi". Fpi stands for "frames per inch", and here it is equivalent to "bits per inch."

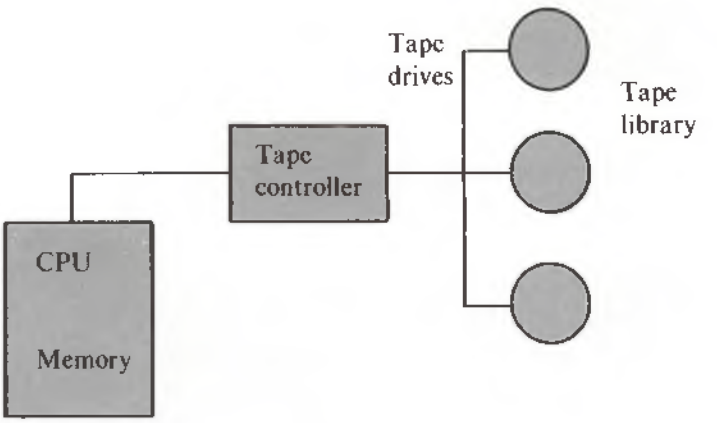


Figure 14.1 A tape-based computing system.

Current ICMT uses 9 tracks (9 channels). Some systems also support the older 7-track tapes, sometimes at densities as low as 200 bpi, and no higher than 556 bpi. Seven- and 9-track tape media are the same, but the tape drives cannot read each other's tapes. An 8-bit byte can be recorded in one 9-bit frame. The tape drive controller attaches its own check bit as the ninth bit and strips it off when the information is read at another time. The relationship of the CPU, memory, and tapes is shown in figure 14.1.

As a general rule, one tape controller (similar to a CRT interface, but much more sophisticated and expensive) can manage several (up to eight) tape drives. It is not unusual to have several tape controllers on a large computer, with each controller managing several tape drives. However, only one tape drive per controller may be transferring data at a time. It is possible to have several tapes spinning simultaneously, but only one per controller can be moving bytes. The other tapes in motion are either being rewound to the BOT mark or performing non-data-transfer skips.

You record information on tape by presenting the tape drive controller with three pieces of information:

1. memory buffer address (BA)
2. word count (WC)
3. I/O command

The command may be a "write," "read," "rewind," etc. If the three items are not more than 16 bits wide, you could execute

```
MOV     #512.,@#WCR ; set WC
MOV     #ABC,@#BAR  ; load BA
BIS     #1,@#CSR    ; select drive
BIS     #10,@#CSR   ; bits 3, 4, 5 = 001 for write
```

to write (record) the 512 words at memory locations ABC, ABC+2, . . . , provided that setting bits 3–5 to 001 in the tape controller's CSR means "write." Of course, the symbols WCR, BAR, CSR would have to be defined (using "=") to correspond to the desired tape controller's I/O register addresses. The interrupt-enable bit can be set in the usual way. Some tape controllers expect a byte count instead of a word count.

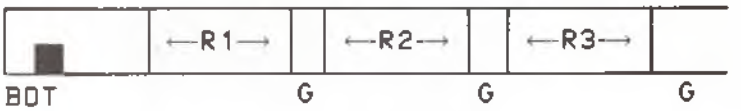


Figure 14.2 Tape record, record-gap format.

Figure 14.3 A tape with four records.

Once the tape controller accepts these three items, it begins accelerating the tape until the recording speed is reached, then it begins moving bytes *directly* from memory, copying them to the tape, *while the CPU goes on executing instructions as usual.* The tape controller keeps decrementing its word count register as it increments its buffer address register, while transferring data. When the word count reaches zero, the controller will write some check bytes on tape, to complete the block it has just written. The controller will then request an interrupt. If you repeat this process several times, the tape will look as shown in figure 14.2 (we will not show the check bytes; they are always implied): Each "write" command generates a record (or block) R on the tape *and* an inter-record gap. Records can vary in length, depending on the word counts used when each record was written. The inter-record gap will never be less than 1/2-inch long; its actual size varies, depending on variations in the tape speed, tape acceleration, etc. The interaction between record size and the presence of record gaps has a significant effect on both tape storage capacity and the tape's effective data-transfer rate, as we shall soon see.

If you have only written records R1, R2, and R3 as shown above, the next record written will follow the gap following record R3. If you attempt to write too many records on a tape, the tape drive will sense the EOT marker and inform the controller, which in turn will both abort the write operation and request an interrupt, having set the appropriate CSR error bits.

If you write too short a record (i.e., fewer than 18 bytes is too few for ANSI), you may not be able to read it later. If you write records exceeding 2048 bytes (2 KB), you may not be able to read them on other computers using ICMT and conforming to ANSI tape standards.

You can position the tape at will, by doing any of the following:

1. rewinding it to the BOT
2. backspace 1 record
3. read over 1 record

You can always add new records to a tape (if it is not full), but you cannot reliably replace a record (unless it is the last record); *update in place* is not supported by ICMT. Consider a tape with four records; figure 14.3. You can add a fifth record R5 following R4. You could have replaced R4 (before adding R5). You can replace R3 if you also replace R4 at the same time, etc. If you attempt to replace (i.e., change) any one of R1, R2, or R3, you cannot expect that the records following the replaced record will remain readable. Tape drives are electromechanical; the tape medium itself is slightly elastic. The gap sizes will vary in length.

Density	Record Size	Capacity (MB)
800	128	6
800	256	9
800	512	13
800	1024	16
800	2048	19
800	4096	21
800	8192	22
800	16384	22
800	32768	23
1600	128	6
1600	256	11
1600	512	18
1600	1024	26
1600	2048	33
1600	4096	38
1600	8192	42
1600	16384	44
1600	32768	45
6250	128	7
6250	256	14
6250	512	25
6250	1024	44
6250	2048	71
6250	4096	101
6250	8192	129
6250	16384	150
6250	32768	163

Figure 14.4 Tape density, record sizes, and capacity.

If you tried replacing R3 and the gap following R3 had been exactly 1/2-inch, but the new gap was slightly longer, R4 might be unreadable because the new gap overwrote the beginning of R4. Replacing the last group of records means reading them into memory, positioning the tape in front of the group of records and performing another set of write operations. A write is automatically preceded by an erase operation.

As a consequence of the inability to update tape records in place, the standard practice is to read and copy the leading records of a tape to a new tape; then read and change the record to be changed and copy its new version to the new tape; then copy the remainder of the old tape to the new tape. This seems like a nuisance, but it has a beneficial aspect. You now have an "old master" tape as well as a "new master" tape. If you maintain a copy of what the change was, you can regenerate the new master if it happens to get destroyed or stolen or misplaced. That is, you can if you kept the old master in a safe place.

Information Density

The three most common recording densities in use are 800, 1600, and 6250 bpi. Tape drives which support the highest density are too expensive for most PDP-11 users. Figure 14.4 illustrates the impact of selecting a given density and a given record size on the tape's storage capacity. We are assuming that full-length tapes are being used, that a 20-foot length

is sacrificed for the tape leader and trailer functions, and that 1/2-inch gaps are used. If you use small records, most of the tape is used for record gaps, so higher densities have almost no effect. If your records are too long, other problems may occur.

Tape Data-Transfer Speeds

Typical tape drive speeds vary from a low of 12.5 ips (inches/second) to a high of 125 ips (on PDP-11s). The most popular speeds are 45 and 75 ips (the cost of a drive climbs rapidly as its speed goes up). The interaction of tape speed, tape recording density, and record size in providing high speed data transfer is shown in figure 14.5.

You can increase both the storage capacity of a tape and its effective transfer rate by using huge records. There are three problems associated with huge records:

1. The probability of losing more information/record increases because the probability of a speck of dust or a tape defect increases with the record length.
2. It is not always possible to find large memory buffers when you need them, either to read or write very large records.
3. Other systems may refuse to read your tapes if your record lengths exceed 2,048 bytes. This is a software-imposed limitation, to promote ease of information transfer via tape.

The most common record (block) size used on PDP-11s is 512 bytes.

Records and Files

Another kind of object which can be recorded on tape, and read from tape, is called a *file mark* (also called a tape mark). The tape controller can distinguish a file mark from a record, or the BOT and EOT marks. The tape controller creates one upon receipt of a "write file mark" request.

Figure 14.5 Data transfer rates (KB/sec).

		Tape drive speeds			
Density (bpi)	**Size B**	**12.5**	**45**	**75**	**125**
		Data transfer rates (KB/sec)			
1600.	**128.**	**3.**	**10.**	**17.**	**28.**
1600.	**256.**	**5.**	**17.**	**29.**	**48.**
1600.	**512.**	**8.**	**28.**	**47.**	**78.**
1600.	**1024.**	**11.**	**40.**	**67.**	**112.**
1600.	**2048.**	**14.**	**52.**	**86.**	**144.**
1600.	**4096.**	**17.**	**60.**	**100.**	**167.**
1600.	**8192.**	**18.**	**66.**	**109.**	**182.**
1600.	**16384.**	**19.**	**69.**	**114.**	**191.**
6250.	**128.**	**3.**	**11.**	**18.**	**31.**
6250.	**256.**	**6.**	**21.**	**35.**	**59.**
6250.	**512.**	**11.**	**40.**	**66.**	**110.**
6250.	**1024.**	**19.**	**69.**	**116.**	**193.**
6250.	**2048.**	**31.**	**111.**	**186.**	**309.**
6250.	**4096.**	**44.**	**160.**	**266.**	**443.**
6250.	**8192.**	**57.**	**204.**	**339.**	**566.**
6250.	**16384.**	**66.**	**236.**	**394.**	**656.**

Figure 14.6 A file mark for a source module of 32 records.

This creates a 3-inch block of tape which is always recognized as a file mark. The file mark allows you to organize things nicely on tape. It is like having a set of filing cabinets—as many as you need—each one being just the right size. Suppose you wanted to record 80-column card images on tape. The first 32 records might be the images of the 32 cards which represent a source module S1. If you write a file mark FM following record 32, the first 32 records can then be thought of as "file 1." If you have too many small files on a tape, the 3-inch file mark rapidly uses up valuable storage space and degrades the data transfer rate.

Industry-Compatible Magnetic Tape

If you use an ICMT and record on it using a nine-track tape drive set at one of the standard densities—say, 800 bpi at 75 ips—you can in principle read that tape on any ICMT which supports the same density you used while recording, at any speed the drive supports. You can, however, run into some problems. Some low speed drives (12.5 ips) will accept only mini tape reels (7-inch diameter) which hold 600 feet of tape. You might also have records which are either too small or too large for the target computer's tape controller or software system. If the target computer is not another PDP-11, and you prepared your tape on a PDP-11 using a controller which writes one word at a time, the target computer's software may have to compensate for the neighboring byte interchange problem.

For instance, if you had consecutive bytes holding the codes for ABCD . . . and wrote them on a tape using a controller which fetches one byte at a time, then the codes for ABCD . . . will be written out on tape in that sequence, and all will be well. If your tape controller happens to fetch a word at a time, it may well write the codes out as if they had been in the sequence BADC . . . . If so, the target machine's software has to compensate for this.

Binary Data

The character-oriented devices people interact with generally accept/generate only the codes of some character set (usually ASCII or EBCDIC). Paper tape deviates from that restriction by allowing you to record arbitrary patterns of 8 bits in each 8-bit frame. In particular, you can record machine-language instructions and binary data words by using consecutive pairs of 8-bit frames on paper tape.

Magnetic tape is similar to paper tape in that respect. It is not *code sensitive*. You can record and read arbitrary bit patterns. This is very important in some situations. For large-scale applications (e.g., processing the U.S. census data), the processing sequence may be:

1. Input of census forms using optical character recognition, producing ASCII codes
2. Display for verification and correction
3. Conversion of all numeric data to binary, which may be integer or floating point, as required
4. Record all data "as-is" on tape

We now have a situation where some of the data which we regard as being in an internal representation is, in fact, recorded that way on tape. You could "dump" memory "as is" on tape, bit for bit. This record would be called a binary record, as distinct from other records which use a standard code. Binary tape records save both space and time. For instance, a decimal number such as "−12.345678E−12" takes 14 bytes if stored as a character string. It takes a mere 4 bytes if converted to the single precision floating point representation.

Tapes such as those used for the census, once written and verified, are never changed. In fact, they constitute a document which by law must be kept intact. We call these *archival tapes*. In order to prevent accidental destruction of such tapes (or any others) when you intend only to be reading them at a given time, a removable ring in the tape hub serves as a write-protect interlock, just in case your program had a bug, or your tape was accidentally mounted in place of another.

Tape Labels

If you file your personal papers in old shoe boxes, you probably label the boxes. With any collection of magnetic tapes, you expect to see labels on each one of them. The label would be read by you or a computer operator to find the correct tape for a given job. There is another kind of label, an *internal label,* which is the first thing written on the tape (i.e., the first record). When a tape is mounted, you are expected to key in its external label identification. Then the operating system reads the first record of the tape and verifies that its internal and external IDs match. The internal label contains other information, such as who owns the tape, when it was first used, etc. There is an ANSI standard for these internal labels, specifying their format and content. Many computing installations insist that all tapes handled by them have an internal label. If you wrote an ordinary tape, with nothing but data on it, and carried it over to a site which insists on reading labeled tapes, you would have to rewrite your tape to bring it into conformity. It may seem like a nuisance, but at sites where thousands of tapes are in use, the chances of using the wrong tape are just too high. Requiring internal labels on all tapes is a sound management practice.

Failure to verify labels, if any, may have been responsible for the near disaster at NORAD, as discussed in figure 14.7.

Interrupts and DMA Transfers

Once you initiate a data-transfer tape operation (read or write a block), the entire record (block) is transferred into/from memory from/to tape without any CPU intervention. This is called *direct memory access* (DMA). If a device has a DMA controller, then it is likely to have a higher data rate than a non-DMA controlled device.

False Alarm on Attack Sends Fighters into Sky

WASHINGTON, Nov. 9 (UPI)—A false alarm in the nation's early warning system sent 10 fighters into the air today before Pentagon officials determined that the nation was not being attacked by hostile missiles.

The alarm was set off by a test tape run through a computer of the North American Air Defense Command, the State Department later explained. The tape simulated a missile attack.

"Through a possible mechanical malfunction the tape transmitted to other commands and agencies," the Pentagon said in a statement.

Officials caught the error within six minutes but in that time 10 fighter aircraft took off, including two F-106's from Kingley Field, two F-106's from Sawyer Air Force Base and six F-101's from Komox, British Columbia.

Neither President Carter, Defense Secretary Harold Brown nor Gen. David Jones, chairman of the Joint Chiefs of Staff, were notified. Middle-level officials determined that the alert was not the real thing. The Pentagon said that the missile alert had nothing to do with the situation in Iran, where American diplomats are being held hostage at the United States Embassy.

New York Times
Nov 10, 1979

False Missile Alert Reported. A mechanical error placed American forces on a false nuclear war alert for six minutes Nov. 9, the Defense Department said Nov. 10.

Pentagon officials said the alert was viewed skeptically and was handled by "middle-level officers." But had the alert lasted any longer, those officials acknowledged, the information would have been passed to President Carter and Defense Secretary Harold Brown.

According to Defense Department officials, a mechanical error sent "war game" information into the highly sensitive early warning system. That system read the information as a "live launch" by a Soviet submarine and initiated a series of checks. Ten jet interceptors from three bases in the U.S. and Canada were scrambled and missile bases went on a low-level alert.

After six minutes of low level alert the mistake was discovered. (A "war game" tape had been loaded into the NORAD computer in Colorado Springs, Colo. as part of a computer test and by mechanical error fed into the early warning system.) The entire incident was under investigation, military officials said.

Tass, the official Soviet press agency, Nov. 10 criticized the incident, and warned that another such error could have "irreparable consequences for the whole world."

Pentagon officials said there had been several false alarms over the years, caused by computer failures and test firings, especially in the late 1950s and early 1960s, when the early warning system was in its infancy.

Facts On File
Nov 16, 1979

Figure 14.7 Wrong tape used at NORAD.

(Left—© 1970/80 by The New York Times Company. Reprinted by permission. Right—From the Facts on File Weekly News Digest. © 1979. Reprinted by permission of Facts on File, Inc.)

If you read or write blocks of data, then you will get only one interrupt per block. So if you were dealing with 512-byte blocks without DMA, you would expect and get 512 interrupts per block. If you are using a DMA controller to read or write these blocks of 512 bytes, you will get only one interrupt for the whole block. Your interrupt overhead has been reduced by a factor of 512.

DMA controllers such as those used by a tape drive are much more complicated and expensive than the simple serial interface we saw earlier. It takes from thirteen to twenty-four pages of fine print to describe the registers and functions for each of the various ICMT DMA controllers available for the PDP-11. A tape controller (e.g., TU45) may have as many as fourteen control, status, buffer, and count registers. Details may be found in DEC's PDP-11 Peripherals Handbook. The full story is told in the reference manuals for each controller.

Since a DMA device controller is considerably more complicated, processing one interrupt for it will take a little longer than was the case with simpler interfaces. One or two hundred instructions executed per interrupt is typical. Nonetheless, even if each interrupt costs you, say, 200 μsec of execution time, you could still sustain an effective data transfer rate of 2,500 KB/sec, if you assume that 100 instructions per DMA interrupt is typical, and we use 2 μsec as an average instruction execution time, while dealing with 512 B blocks. Under the same assumptions, 1024 B blocks could allow data rates as high as 5 MB/sec, which is fine, since no PDP-11 can handle data rates much over 1.5 MB/sec.

Other Tape Devices

ICMT controllers and drives were once prohibitively expensive (i.e., over $50,000 each) and simply out of the question for use on presumably low-cost mini- or microcomputer systems. In the 1960s DEC developed a relatively inexpensive tape drive called DECtape. The objective of low cost and high reliability in a dirty environment (as in a typical physics or engineering laboratory) was met by sacrificing compatibility with ICMT and by also sacrificing capacity and performance. Actually, for many users DECtape offered an increase in capacity and performance, since the only economic alternative at the time was to use paper tape (lots of it, since it is not easily erasable). With the advent of floppy disks (discussed below), DECtape systems were phased out.

A successor to DECtape, called DECtape II, uses a cartridge tape based on one pioneered by the 3M Company specifically for digital recording. DECtape II is not compatible with ICMT. Its limited capacity (approximately 1/4 MB per cartridge) limits its use, as does its low speed. It is limited to the speeds supported by standard serial interfaces (e.g., as slow as 110 baud, as fast as 38.4 Kbaud). The device and its controller are very low in cost because not only is it non-DMA, but it connects to the CPU by the same kind of inexpensive interface we use for CRTs.

DECtape used fixed-size blocks with prerecorded block numbers and had a controller which could find any block, given a block number. Tape records could even be read backward to save time (yet appear correct in memory); read-backward is an infrequent option on ICMT. The use of block numbers for each tape block made it possible for a DECtape to simulate what is done with a disk system. DECtape II also simulates a disk system in that it uses numbered fixed-size blocks. The DECtape family is thus both physically and conceptually different from ICMT.

Cassette Tapes

Audio cassette recorders are analog recording devices. By interconnection with audio-to-digital and digital-to-audio converters (such as an acoustic coupler, discussed under "Telecommunications and Teleprocessing"), they can be used as storage devices for computers. Many personal (hobby) computers use this inexpensive kind of equipment. However, the performance and reliability are not up to the usual computer industry standards, so digital cassette drives have been designed specifically for use as low-cost storage devices.

The DEC entry in this field is the TU60/TU11 tape system, which uses a Philips-type cassette. The maximum tape capacity is 92 KB, with a transfer rate of 562 bits/second. Its main application seems to be as a substitute for paper tape readers and punches. Information on the cassette tape is organized in records and files of sizes of your choice, very much as with ICMT. As you might expect, this cassette tape is not compatible with any of the tape devices we have discussed.

Direct Access Devices; Disks

Tapes are classified as sequential access devices because accessing any record requires traversing all records which separate the current record from the desired record. If you are at the BOT mark and you need the last record on the 2,400-foot tape, and the tape is almost full, with a 45-ips drive, you can expect to wait at least ten minutes to reach that record. Ironically, if the situation were reversed, you could get to the BOT in a mere two minutes, using a high speed rewind request (fast forward is an uncommon option).

The delay in reaching the desired record is called the *access time*. For a magnetic tape, the worst-case access time is measured in minutes. Even the average access time is measured in minutes. Tapes shine in situations where you want to access consecutive records. What if you don't want to access records consecutively as they appear on tape? We see here something like the various addressing modes. Auto-increment and auto-decrement addressing are marvelous for sequential access to consecutive array elements. Indexed addressing is great for direct (random) access to array elements. So we need some kind of direct access or random access device.

Disk Drives

A disk can be thought of as a two-dimensional storage device. Logically, a disk can be viewed as an array of looped magnetic tapes, organized as in figure 14.8. The tracks are actually circular, so for each track item, Sn is immediately followed by item S0. Disk practice is somewhat different from that used with tape. When a byte is recorded on tape, a 9-bit frame cutting across all 9 tape channels is used. When a byte is recorded on disk, the bits are placed on a single track, and instead of providing each byte with a check bit, a larger set of check bits is attached at the end of the block.

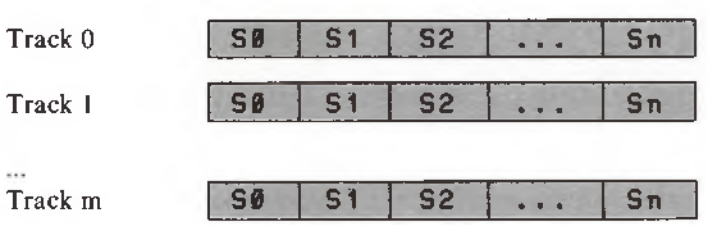


Figure 14.8 A disk viewed as an array of looped magnetic tapes.

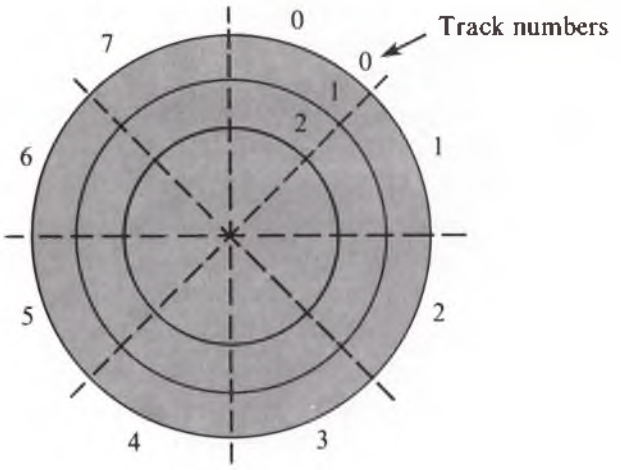


Figure 14.9 Tracks on a disk surface.

Most disks are organized in fixed-size records called *sectors*. The most frequently used sector size on PDP-11 disk systems is 256 KW (512 KB). A typical disk drive would have several recording surfaces. Let us examine a simple drive which uses only one side of its single disk. The tracks are physically arranged as a series of concentric circles, as shown in figure 14.9.

In the diagram, the sector numbers are shown on the outside, the track numbers are shown on each track. A single-surface disk drive uses a single read/write head mounted on a moving arm, somewhat like the arm of a phonograph with its needle, but disks have no grooves.

The operating speed of a disk drive depends on:

1. the movement of the arm so that its read/write head is placed above the desired track;
2. the wait for the desired sector to come under the read/write head as the disk rotates at a uniform rate.

The moving time of the arm is called the *seek time*. You expect that the seek time will vary according to the distance the arm must travel, so the disk seek time is usually given as a set of three numbers:

1. Maximum seek time: move from the outermost track to innermost track, or vice versa.
2. Seek next track.
3. Average seek time.

If it takes x milliseconds to seek an adjacent track (on either side) it takes slightly less then 2*x to seek one that is two tracks away, and much less than y*x to seek one that is y away if y is large. Thus, item 1 is smaller than t*(item 2 time) if the disk has t tracks. This is simply because of the arm acceleration delay which is incurred whether the movement is to a nearby track or one much further away.

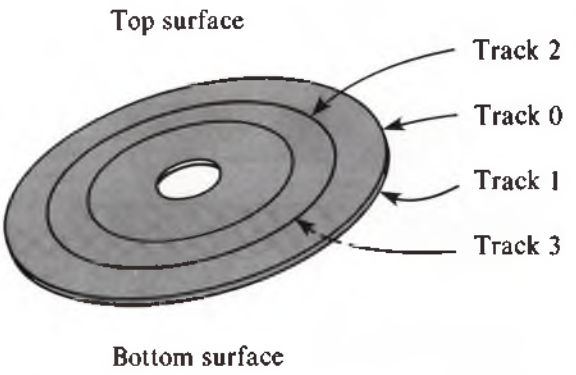


Figure 14.10 Track addressing on a disk.

The waiting time for the desired sector after the arm is positioned over the correct track is called the *latency time* or the *rotational delay*. Latency time is a function of the speed at which the disk rotates, the distance along the track, and separation of the read/write head from the desired sector at the instant the seek is completed. Latency time is usually stated as two numbers:

1. Maximum latency (1/rotation time).
2. Average latency (maximum/2).

The worst-case access delay in using a disk is then:

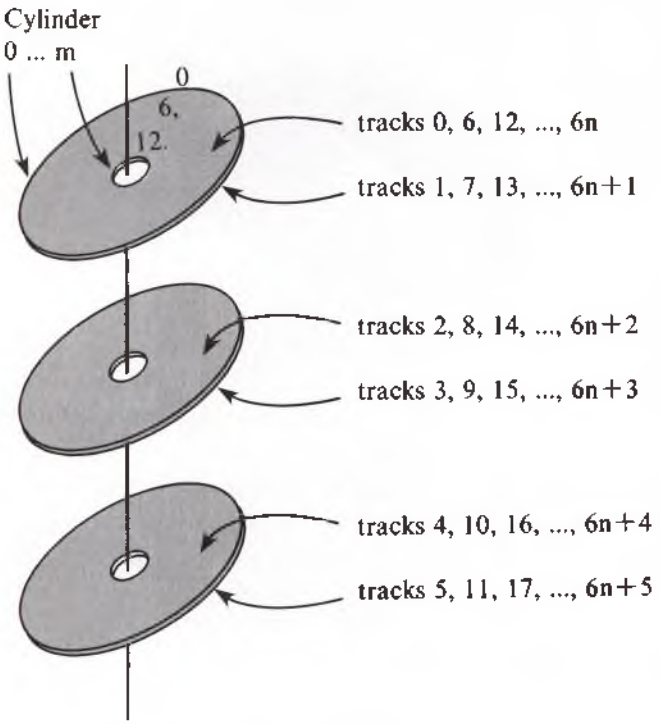
maximum seek time + maximum latency

The slowest disk used on a PDP-11, a floppy disk, has a worst-case access delay of just under 1 second (compared to 10 minutes for a 45 ips tape). Higher-performance, higher-capacity disks have a worst-case access delay well under 75 milliseconds, which is 13 times better.

Since a disk by its nature has two surfaces, it is cost-effective to use both surfaces for storage. In such a case, the track addresses would be assigned so that all even numbered tracks would be on surface 0, and all odd numbered tracks would be on surface 1, as shown in figure 14.10. The "arm" then becomes two arms, each with its read/write head. However, the two arms move as a unit. This means, if the top arm is over track 6, then the bottom arm must necessarily be over track 7. Tracks 0 and 1, 2 and 3, . . ., 2n and 2n+ 1 are seen as belonging to a series of concentric *cylinders*. The outer cylinder is numbered 0; it contains tracks 0 and 1. The next cylinder is numbered 1; it contains tracks 2 and 3. If the last cylinder is numbered x, it contains tracks 2x and 2x+ 1.

The disk controller exploits this numbering as follows. You usually read or write on a disk in units of one sector. You can treat the disk as if it had large blocks by grouping contiguous sectors and treating them as a single block. If you have a disk transfer that goes beyond the last sector of track 6, then the controller will immediately activate the lower read/write head so you may continue over to track 7, which is in the same cylinder. In effect, because those tracks which belong to a cylinder have consecutive addresses, it is as if the two tracks constituted one very long track. The more successful you are in staying in the same cylinder, the fewer seek delays you will incur.

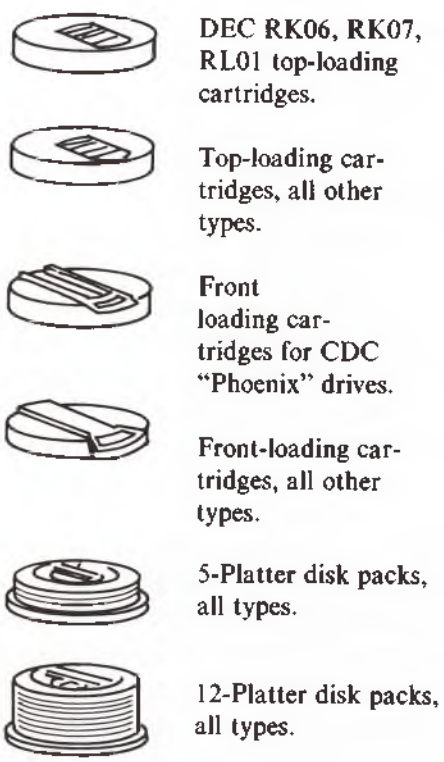
Figure 14.11 Addressing on a multisurface disk.



Track numbering on an n-cylinder disk

Figure 14.12 Removable disk storage media.

(From Inmac Corporation catalog of computer supplies and accessories.)



DEC RK06, RK07, RL01 top-loading cartridges.

Top-loading cartridges, all other types.

Front loading cartridges for CDC "Phoenix" drives.

Front-loading cartridges, all other types.

5-Platter disk packs, all types.

12-Platter disk packs, all types.

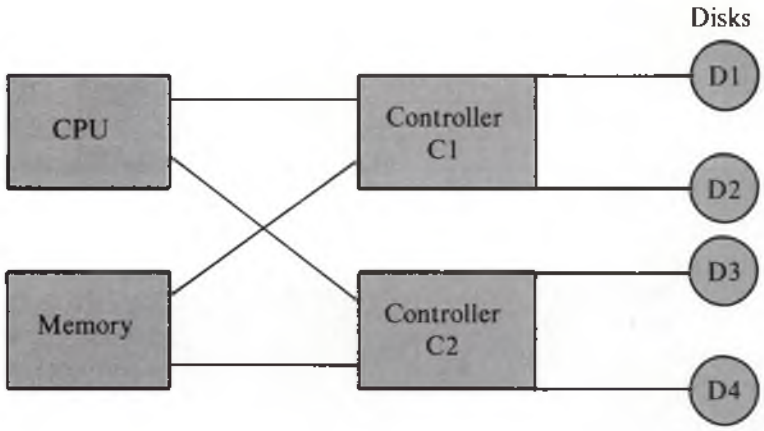


Figure 14.13 Multiple controller, multiple disk system.

If the cost per byte for building a two-surface disk is lower than for a one-surface disk, it is even better for a multidisk drive. Disks with several "platters" are common. The track numbering then extends as indicated in figure 14.11. The track-to-track switching within a cylinder is done at electronic speeds. The worst-case access delay within a cylinder would occur if you just missed the desired sector, necessitating a one-revolution delay.

The excerpt from a computer supplies catalog shown in figure 14.12 illustrates the variety of removable disk storage media presently in use. Note that even though two different computers may be using compatible disk packs or cartridges, differences in the file structures written on the disks by the different operating systems may make transferring files by pack interchange awkward.

Disk I/O

If you are using a disk on a bare machine, it appears to be like any other device. The disk controller has a set of register addresses in the I/O address space; it has an interrupt vector address and a hard-wired priority. It may have as few as five and as many as twenty-two device registers.

With the exception of one low-performance floppy disk, all the PDP-11 disk drive controllers use DMA access to memory. As with tape drives, several disk drives of the same type may share one disk controller, usually up to eight drives per controller. Of course, one controller can support only a single data transfer at any one time.

The principal way in which disk I/O differs from that of other devices is due to its two-dimensional nature and the fact that its sectors have explicit addresses. Furthermore, disks are specifically designed to permit update-in-place. Any sector can be rewritten without damaging the information in adjacent sectors. If we had the configuration shown in figure 14.13, then access to a given sector on disk D3 requires:

1. using the set of device registers for controller C2
2. presenting the following information to C2
   (a) drive number
   (b) cylinder address
   (c) track address
   (d) sector address
   (e) memory buffer address
   (f) word count

Figure 14.14 Disk organization and capacity.

Disk	Bytes per Sector	Sectors per Track	Tracks per Surface	Number of Surfaces	Capacity in MB
RL01	256	40	256	2	5.2
RM03	512	32	823	5	67
RX01	128	26	77	1	.256

Figure 14.15 Seek, latency, and access times.

	Seek Time				Latency		Access		
	Max	Av	T-T	RPM	Max	Av	Max	Av	T-T
RL01	100	55	15	2400	25	12.5	125	80	27.5
RM03	55	30	6	3600	17.3	8.3	72.3	47.3	14.3
RX01	760	380	10	360	173	83	953	483	113

Note: T-T is the track-to-track seek for tracks in adjacent cylinders. For floppy disks such as the RX01, a "head settling" time must be added following every seek. It is 20 msec for the RX01.

If all the bits for items (a) through (f) could fit in one 16-bit word, and you had a sector size of 256 B, then you could address no more than 64 K * 256, or 16 MB. Sixteen million bytes on-line on one controller may seem like a lot of storage, but it would not accommodate even a single 67 MB disk drive. As a general rule, it takes several registers to specify items (a) through (f).

Disk Capacity and Performance

The PDP-11 supports at least ten different models of disk drives. Even a summary of their characteristics would be overwhelming. The following tables summarize the characteristics of three disks, chosen because of the contrasts between them.

The table in figure 14.14 shows how the recording surfaces of each disk are subdivided into sectors and tracks, how many surfaces can be used, etc., culminating in the total usable storage for each disk. You may sometimes hear of an 80 MB disk and wonder why it holds only 67 MB of data. The larger figure is the unformatted capacity (as with magnetic tape, before any gaps or file marks use up some of the usable capacity). Space is lost for the storing of track and sector addresses, as well as for each sector's check bits. Over 15 percent of the raw storage capacity of a disk is eaten up before you even get to use it.

How can we compare one disk with another, or with magnetic tapes? Here the focus is on the worst case, best case and average case, for the following three items: (1) access time; (2) data transfer rate; (3) transactions per second. The access times for the three disks we have selected are shown in figure 14.15. The data transfer rates have to be explained. The recording density on a standard floppy disk such as an RX01 is a maximum of 3200 bpi. An RX01 spins at 360 rpm; one track can hold 26 sectors, each sector holding 128 8-bit bytes. This gives us an apparent data rate of 19,968 bytes per second; let us say that it is 20 KB/sec, or 50 msec/byte. The RX01 manual states that the data transfer rate is 18 μsec/byte. How can we resolve the discrepancy between our 50 msec/byte (or 50,000 μsec/byte) versus their 18 μsec? How can we both be right when we are so far apart?

	Instantaneous usec/B	KB/sec	Per track KB/sec	Track with access
RL01	1.9	512	410	195
RM03	0.82	1,200	947	518
RX01	18	55	19	12

Figure 14.16 Disk data rates.

	Seek av	Rev time	Sector time	Operation time	Ops/sec
RL01	55	25	.625	93.75	10.66
RM03	30	17.3	.541	57.03	17.53
RX01	380	173	6.65	652.8	1.5

Figure 14.17 Disk operations per second.

Note: Operation time here is defined as the time to access a record, read it, then rewrite it. This gives us:
operation time = average seek + average latency + read one sector + full revolution + write one sector.

One byte recorded at a density of 3,200 bpi occupies 8/3200 inches, or 1/400 inch, which is .0025 inches of track space. A standard size floppy disk fits in an 8-by-8-inch jacket, so the diameter of the floppy disk is just under 8 inches. The circumference (the track length) is provided by (pi*diameter), giving us approximately 25 inches as the length of the outermost track. At 360 rpm, we have one full revolution every 1/6 second. Our one byte occupies .0025 inches of the 25-inch track, so it takes

$$(.0025 \text{ in}/25 \text{ in}) * (1/6 \text{ sec}) = .000\ 0166 \text{ seconds}$$
$$17 \text{ microseconds}$$

to traverse one byte on this floppy disk drive. Based on the approximations we have made, our 17 μsec is very close to their 18 μsec. What does it mean? Since the RX01 controller does not use a DMA path into memory, the CPU has to respond to what is, from its point of view, the worst-case data rate: a byte arriving each 17 or 18 μsec. This gives a peak data rate of almost 60 KB. The user, however, usually experiences the data rate averaged over a whole track. This is analogous to the performance degradation that tape gaps introduce.

Comparing the instantaneous data rate of 60 KB/sec for a byte within a sector with the average data rate of 20 KB/sec for the bytes of a track, it is clear that you have to read hardware specifications very carefully lest you be lulled into expecting far more than the hardware can deliver. With this in mind, we can examine in figure 14.16 the various data rates for the three disks we are comparing.

We can compare disks in another way which relates closely to how they are used in some applications. What is the number of operations per second that a given disk can support? The definition of "operations per second" can be made to correspond roughly with the intended use. For instance, if you are building an information retrieval system in which queries are frequent but changes are very infrequent, then you are interested in using disk-reads/second as your definition for operations/second. In another system—say, an airline reservation system—the typical transaction involves reading a record, then rewriting it after it has been updated. In figure 14.17, we will use the read-change-rewrite definition.

Other Disks

Unless otherwise stated, a disk drive uses a moving-head arm (or arms). There are very-high performance disks which get their superior performance by providing one head per track. That being the case, the arms never move so these disks are called fixed-head disks. As a general rule, the cost of providing all these read/write heads leads to limiting the storage capacity. The largest fixed-head disk for the PDP-11 (RS04) is limited to 1 MB. It has zero seek time, and an average access time of 85 msec. Such devices are best used as "swapping devices" in a time-sharing system. If a user's program has to be suspended momentarily because it is waiting for I/O or because it is some other program's turn to run, the suspended program can be quickly "swapped out" to the fixed-head disk, while some waiting program can be quickly "swapped in." Having a zero seek time helps considerably.

With the cost of memory chips dropping so rapidly, a kind of "solid state" disk has been introduced. Since it has no moving parts, its seek time is zero. It uses ordinary memory chips organized to simulate a "track," so it does have a "simulated" latency time, but it is a mere 100 μsec instead of being in the millisecond range. Solid state disks can have almost as many MBs of storage as you are willing to pay for. Some reader may object: "Is it not silly to put memory chips in a box with a significant latency time when you could have simply made your main memory larger?" That is correct, except for the fact that when you have 256 KB of memory on a PDP-11/45, or 4 MB on a PDP-11/70, or 1 MB on a PDP-11/44, you cannot add any more main memory. Those are the physical memory limits for those models.

Recording Media

The illustration in figure 14.18 from a supplies catalog depicts the variety of noncompatible features associated with floppy disks. Each of these requires that you use a matching drive and controller. The soft-sectored disks must be formatted prior to use. At that time you, in effect, write a special track that defines the sector size from then on.

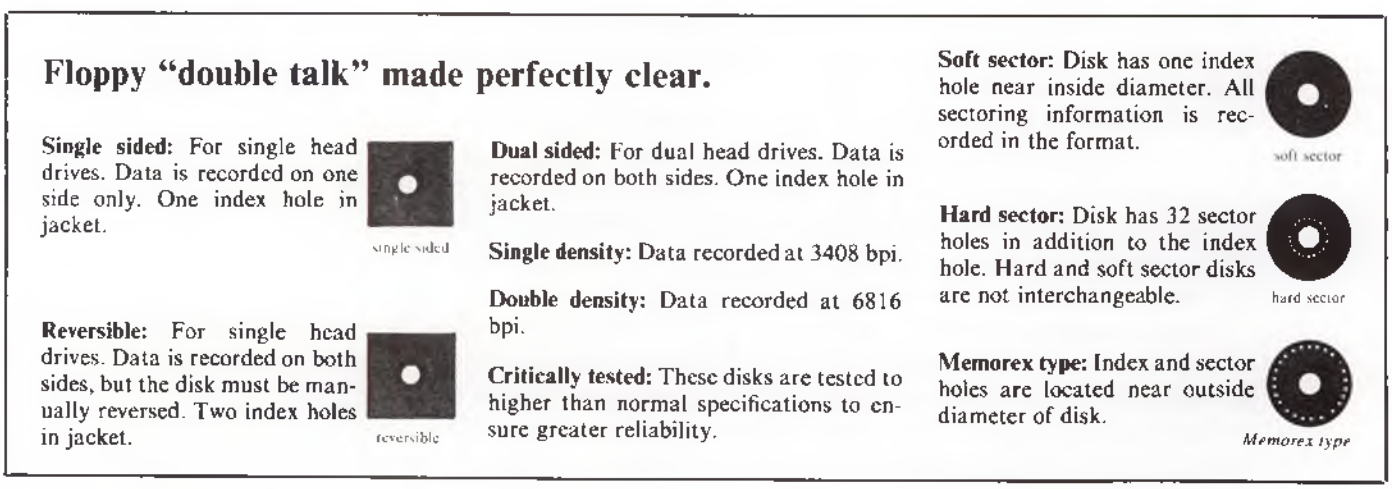


Figure 14.8 Variety of Floppy Disks

(From Inmac Corporation catalog of computer supplies and accessories.)

Winchester Drives

Most disk drives have a removable disk pack or disk cartridge, so you can have a disk library similar to a tape library. A fixed-head disk is a notable exception. Another interesting exception is provided by a special class of disks called Winchester disks. Winchester disks are unusual in that each disk pack has its own set of read/write heads "built in." In principle, this gives you a removable pack, but the cost is too high to make it practical to have a whole bunch of Winchester packs sitting on a shelf (they sell for $3,000 to $20,000 per "pack").

Ordinary removable packs are exposed to air whenever they are mounted or dismounted. Even when in use, some dust particles may sneak through the fine air filters used on ordinary drives. A speck of dust can contaminate a pack, leading to a disk head crash. The loss of a $1,000 pack is negligible when compared with the lost computer production time in repairing the drive, or the cost of reconstructing the information that was on the pack that just got destroyed. A Winchester drive has its own airtight sealed environment. Its recording surfaces are never exposed to air. We can expect to see this type of disk in wide use because of its inherent superior reliability.

Direct Access Storage Devices

Disk storage devices are often called direct access storage devices (DASD). They are also called random access devices. The DASD comes from the ability to access a record on disk more or less directly, far more so than is the case with an SASD (sequential access storage device). The random access nomenclature conveys the notion that access times for randomly located records on a disk are more or less independent of their position, at least far more so than would be the case with a SASD. A disk as a random access memory should not be confused with RAM (random access memory) memory chips.

Spooling

"Spool" is an acronym for simultaneous peripheral (device) operations on-line. As is the custom, the acronym became a noun, and we now have the verb "spooling." Almost any disk-based system allows you to "spool" output to a printer. You simply identify the file you wish printed, and the spooling software proceeds to take care of your print request, while you go on to other computing tasks. This is yet another level at which I/O and other activities (i.e., your own tasks) can be overlapped. The evolution of computing systems to be able to support spooling illustrates the interaction between the right kinds of devices, the right kinds of control mechanisms, and superior user services.

Prior to the introduction of affordable disks in the mid-'60s, many large computing systems operated as follows:

1. Prepare a "job" and/or data on punched cards with a keypunch (offline).
2. Use a card-to-magnetic-tape machine to copy a "batch" of job card decks onto the magnetic tape. This tape then becomes the "job input tape."

3. Mount the job input tape on one of the dozen tape drives attached to the computer.
4. Process the jobs on the input tape sequentially, copying all "print output" to successive files of a tape designated as the "print tape."
5. When all processing in step 4 is complete, mount the print tape on another machine, which reads the tape and drives a printer (a tape-to-print machine).

This mode of computing, in which jobs are processed one at a time, the next job not being started until the current job is completed, and with a strict first-in first-out service maintained, is called *batch processing*. The off-line I/O operations allowed a certain amount of overlapped I/O while the CPU was running, at the cost of having dedicated off-line stations just for the card-to-tape and tape-to-print operations. The need to use punched cards prepared off-line (on keypunches) and the need for the auxiliary machines are both consequences of not having affordable direct access storage devices, such as disks.

When disks became practical, the off-line auxiliary I/O machines were retired, and punched card input slowly began to be displaced by on-line CRTs. It is still the case that batch processing endures, even if no cards are involved. For the sake of simplicity, let us continue to assume that cards are used in preparing batch jobs. The computer configuration would then have:

1. an on-line card reader;
2. an on-line printer;
3. adequate on-line disk storage.

The batch processing job flow would now be, thanks to the on-line devices and the spooling permitted by interrupt control:

1. On-line card reader attempts to keep reading cards and copying card images to memory.
2. Card images in memory are copied to and appended to a job input file on disk.
3. System scheduler attempts to read next available job from the disk's job input file.
4. Any job which is running builds its print file on disk. When a job completes, its disk print file is appended to the print output queue on disk.
5. The printer software driver keeps trying to empty the disk printer queue. So long as it is not empty, the printer will keep running at full speed.

What we see here is the ultimate in fully overlapped input, processing, and output operations, short of going into a multiprogrammed batch operation. The latter essentially involves "time-sharing" the CPU among the batch jobs.

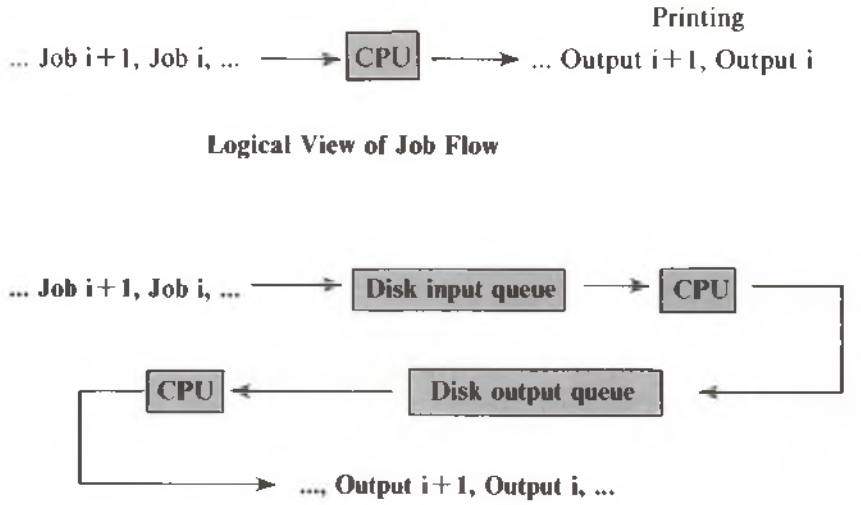


Figure 14.19 Actual job flow in spooling.

Using disk as a job input buffer and as a print output buffer (queues and buffers support the same function), a job's input and its output can be physically disassociated from the job. When you say "Read a card," the card may have been read one hour ago; your program's request causes a card image to be fetched from disk. Similarly, when your program says "Print a line," the actual printing may be done several hours later, depending on the length of the print queue and the speed of the printer. Sometimes the print output will never appear, because the system "crashed" after the time your job ran but before your printing started, and the print queue was not recovered when the system was rebooted. The sketch in figure 14.19 illustrates the logical and physical views of the job flow in a spooled system. Spooling clearly requires the CPU to do more work, and it requires a larger disk capacity than would otherwise be the case. These extra costs are deemed worthwhile in view of the performance increases which spooling provides.

Performance

Performance is in the eyes of the beholder. There are several measures of performance of computer systems. The subject is a vast one, and we will examine only a few useful definitions.

Turnaround Time

In a batch-oriented system, the time interval from the submission of a computer job in machine-readable form (e.g., as a card deck or as a set of commands at a CRT) to the time of its completion (up to and including any printer output, plotter output, etc.) is called the *turnaround time*. In a college computing center, the average turnaround time measured over a whole year might be an hour or so per job. At the end-of-semester peak load time it might become as long as 12 to 24 hours per job. Just as was the case with disk seek-time parameters, a simple average does not provide much information. It has to be qualified regarding worst-case and best-case times at the very least. The computing center management is very much interested in keeping the turnaround time down to a reasonable level, but the management has another measure of performance to monitor.

Throughput

The average number of jobs per hour is a measure of performance which relates to the efficient use of computing resources. This measure is called *throughput*. All other things being equal, if a system X can process 200 "typical" jobs per hour while another comparably priced system Y can process only 150 of the same kind of jobs in an hour, obviously X is a "better" system. Throughput and turnaround can be opposing forces. Consider that the "throughput" of a Boeing 747 is much greater than that of a Piper Cub, but the "turnaround" of the latter may be much better.

Turnaround and throughput are useful measures of performance in a batch-oriented system. In time-sharing systems, two closely related measures of performance are used.

Response time

The time between keying in the last character of a request at an on-line CRT and receiving a response is called the *response time*. For instance, if you are using an on-line editor, how long must you usually wait for

(a) the first prompt from the editor?
(b) the display of a given line?
(c) a response to a search request?
(d) the return to the operating system, after editing?

By actually measuring the system's response time, an average response time and a distribution can be obtained. In a general-purpose time-sharing system, the average may not mean a lot because different users are doing vastly different tasks. In a special-purpose on-line system, the designer of the system (say, for airline reservations) may, in fact, be required to guarantee:

(a) average response time $<$ 2 seconds;
(b) worst case response time $<$ 10 seconds.

Response time is an action-by-action measure of turnaround. The counterpart of throughput in batch processing is the number of concurrent users in a timesharing system. If a time-sharing system physically has ports for say fifty CRTs, and its operating system allows all fifty CRTs to be in use simultaneously, then that system is said to "support fifty concurrent users." All other things being equal, if a fifty-user system X provides average and peak response times of one and five seconds, respectively, while a system Y supports fifty users with average and peak response times of two and twenty seconds, clearly system X is the better system. It may subsequently turn out that, as the user work profile changes over the next year, system X's performance might go downhill rapidly, while system Y's might degrade only a little.

Measures of performance appropriate for batch or time-sharing systems may be of little interest to the owner and user of a single-user system. Provided that the performance is "adequate," other characteristics may be of greater importance. Ease of programming, debugging, operating, reliability, purchase price, upkeep cost, etc. may be far more important in such circumstances.

Blocking

In few instances in life do you find a better match between a container and its contents than with eggs and egg crates. The crates are designed to hold eggs, and they do that well. Computer systems are designed for the widest possible application, and so it is unusual to find such a neat match between the tape and disk storage devices and the information they are to hold.

Disks most efficiently store and transfer records whose sizes are multiples of the disks' sector size. It would clearly be a poor use of disk space to store one 80-character line per 128 byte sector.

With magnetic tape, you can record blocks of varying size. However, doing so makes for poor use of the tape's storage capacity. If you were processing a series of records whose lengths were 100, 36, 57, 24, . . . bytes, the record gaps on an 800-bpi tape would use more space than would the actual data. Using a higher density does not result in using shorter gaps, so the ratio between data recording use of tape and gap use actually worsens then.

The technique developed to alleviate the general problem involves packing several of your records (now called *logical records)* into one larger, more efficiently handled, record (now called a *physical record).* If you had logical records as shown in figure 14.20, they could be stored as shown in figure 14.21. If the last record does not exactly fill out the physical record, then a suitable number of padding characters may be appended. If the last record happened to be too large, then the excess would be stored in the beginning of the next physical record. Some method of distinguishing between logical records must be provided, if they are not of a known, uniform length. This information can be provided either by preceding each logical record with a fixed-size logical-record byte-count field, or by adding an end-of-record character at the end of each logical record, as we did here using "¶".

Consider an application which deals with 80-character records. One thousand such records on an 800-bpi tape occupy 1,000*(80/800 + .5), or 510 inches. If we decided to pack our 80-character logical records into 512-byte blocks, we could reduce the amount of tape used as follows:

80 char. record + 1 end-of-record byte → 1 log. record
1,000 log. records * 81 bytes each → 81,000 bytes needed
Use 81000/512 → 158.2 physical records of 512 bytes
158.2*(512/800 + .5) → 180.3 inches of tape

Figure 14.20 Logical records.

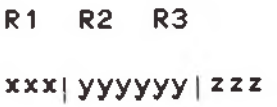


Figure 14.21 One physical record.

This tape length is 2.8 times shorter than we had earlier. It also means, all other things being equal, that this data will be read 2.8 times faster.

The ratio between the lengths of the physical and logical records is called the *blocking factor*. In the preceding example, the blocking factor is 512/80, or 6.4. We often ignore the extra end-of-record character in computing the blocking factor. When the logical records are not all the same length, we can use an approximate average length to compute the blocking factor.

When we use a disk, it is normal to select a physical record size which is no smaller than the disk's sector size; it can be some whole multiple thereof. Most operating systems take care of blocking data for the user when the user is creating a disk or tape file. Conversely, the operating system would take care of the unblocking when you read these blocked files. The operating system may force a particular physical record size to be used with disk files. Some operating systems may give you a choice of blocking factor or physical record size when dealing with magnetic tape files. Blocking is particularly effective when it is used in conjunction with spooling.

RESET

Computer components used to store data, programs, status information, etc. can be classified as either (a) volatile or (b) non-volatile.

Volatile storage devices lose information when their power source is removed. Computer memories constructed with RAM chips and most registers used in the CPU and peripheral devices are volatile.

A nonvolatile storage device does not need a continuous source of electric power to function as a storage device. It can "remember" the bits stored before the power was turned off, and they will be accessible after the power is turned back on. Magnetic tapes, magnetic disks, and ferromagnetic core memory are examples of nonvolatile storage devices.

When power is applied to a CPU, its volatile registers (e.g., the IR, the PC, etc.) and those of its I/O devices (e.g., a disk controller's CSR) contain random bits. The consequences, if nothing were done about this, could be tragic. Several million bytes on a disk might be destroyed, among other things. Special power-on sequencing hardware is provided just to avoid this possibility. This power-on hardware is responsible for forcing a few critical bits in some volatile registers to a known, presumably safe, state. Thus it will reset all the device interrupt-enable bits, and leave the CPU's control unit "run bit" reset, unless it happens to force a safe address into the PC.

What happens if, due to some bug or some transient electrical glitch (static electricity), the CPU gets into a strange state? To restore it to a known state, one could power-down and start over again. Unfortunately, the power-down power-up sequence for large disks takes several minutes.

The RESET instruction is provided for just this purpose. It allows you to force the whole system into the state it had immediately after the power-up sequence. As you would expect, it has no effect on information stored on nonvolatile storage devices, nor does it change the memory. It even leaves the eight registers %0–%7 alone (it might be embarrassing if it cleared %7). A RESET is a very powerful instruction. It stops all in-progress I/O operations. You can execute it on a single-user system. If you try it on a multi-user system, you can expect an M-TRAP, unless you happen to be running in privileged mode.

WAIT

The WAIT instruction is another instruction we should examine. It seems very strange. You would use it in place of a busy-wait loop such as

```
1$:     TSTB    CSR
        BPL     $1
```

WAIT has no operands, so you can replace both of the preceding lines by

```
WAIT; wait here
```

Since presumably the CPU had nothing else to do at the time, what does it matter how the CPU "kills time"?

In some circumstances it makes no difference. If you are waiting for a keyboard input character, and nothing else is happening, WAIT is no better than a busy-wait loop. If, on the other hand, you have initiated a DMA transfer, then WAIT is much better than a busy-wait loop. The busy-wait loop competes with the active I/O device for memory cycles (the CPU keeps fetching first a TSTB, then the data, then a BPL, ad infinitum) while the I/O device is also trying to fetch or store data from or in memory. The WAIT forces the control unit to suspend its fetch-execute cycle! This frees the memory for the exclusive use of I/O devices. So the I/O will now go faster!

How long does a WAIT take? It is a very unusual instruction in that its instruction execution time is "indefinite." It stops the CPU, and the fetch-execute cycle will resume (following the WAIT) if and only if an interrupt occurs. When an interrupt finally occurs, the normal interrupt processing saves the PC, pointing at the location of the WAIT, plus 2. So, following the interrupt handler's return with an RTI, normal processing resumes.

The WAIT is a nice bridge into the next topic, a very important one.

Bus Organized Computers

In the 1960s most computers had their peripheral devices connected to the CPU and memory with hardware channels, as shown in figure 14.22.

A channel is similar to the device controllers we saw earlier, but it also has a direct connection to the memory. This provides for ultra-high speed data transfers, since a system with mutiple channels can have many if not all of them actively transferring data simultaneously. Unfortunately, channels and a memory system capable of accommodating several channels are very expensive. In an attempt to provide significantly lower-cost computing, DEC developed a series of computers culminating in the first PDP-11. They eliminated the multiple channels and provided a single channel to be shared by *all* components of the system. This type of organization is called a *bus-organized computer*. The name *bus* is used to identify a path shared by a number of devices.

In a channel-organized system, the CPU and memory play the dominant role. In a bus-organized system, the bus plays a dominant role. Note that we managed to get this far without talking about a bus before. How does it "dominate"?

In the PDP-11, the bus has the name UNIBUS. A PDP-11 has the structure indicated in figure 14.23. All data traffic flows over the bus.

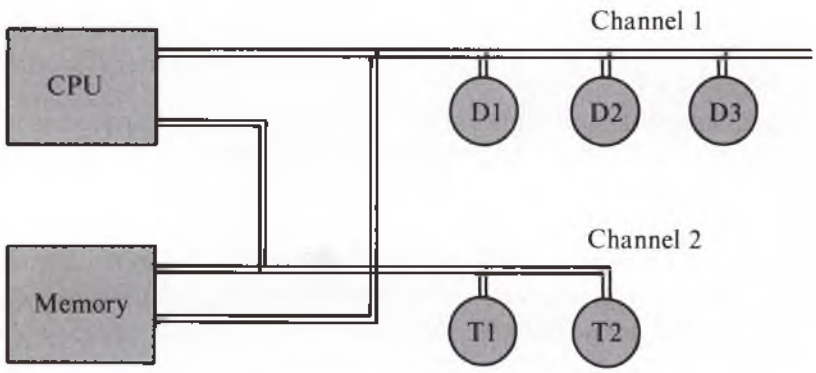


Figure 14.22 A channel-organized computer.

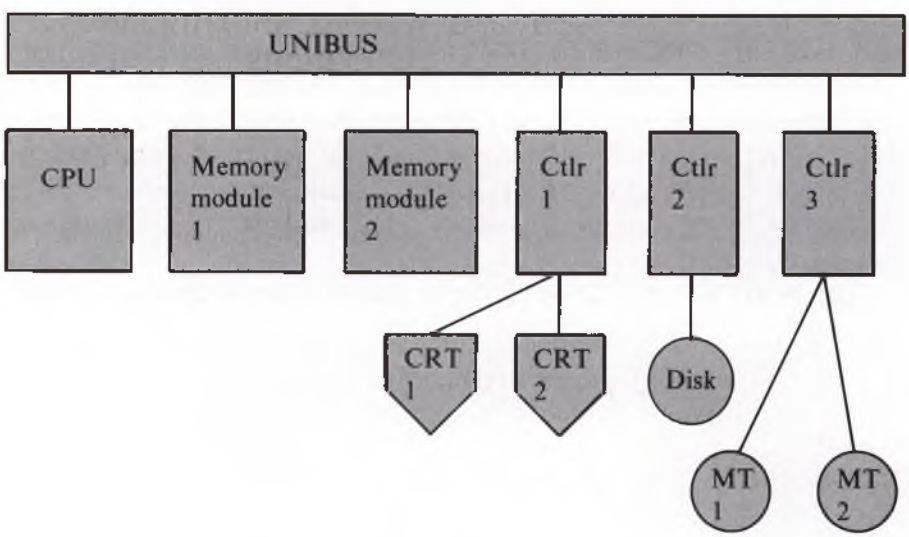


Figure 14.23 A typical PDP-11 system.

Every instruction executed by the CPU must flow from the appropriate memory module to the CPU, over the bus. Every byte sent or received by a CRT flows over the bus. In the case of a CRT attached to a single-port serial interface, the bytes would flow from the CRT to the CPU, over the bus, and vice versa. In the case of a disk using a DMA controller, the bytes go over the UNIBUS from the disk *directly to the memory,* or vice versa. When a unit "talks" to the bus, all the other units on the bus "listen." All addresses placed on the bus by a unit are broadcast to all other units. Each unit "looks" at the addresses broadcast on the bus for addresses it "owns." This explains how a memory-mapped I/O addressing scheme can work.

What are the advantages of having a bus-organized computer?

1. Lower cost.
2. Ease of interfacing (adding more units).
3. Ability to add faster memory modules as the technology improves, without having to remove the slower, older memory modules.
4. Ease of adding read-only memory (ROM).
5. Ability to transfer data directly between compatible devices without using the memory.

We have assumed in point 3 that the system is asynchronous. There are synchronous bus-organized systems; these cannot support memory modules with differing timing characteristics.

Do bus-organized systems have any disadvantages? By design, since all traffic must flow over the bus, sooner or later, some application will surface for which the bus is the bottleneck. Expansion of a bus-organized system is also limited by the bus's ability to electronically support but a relatively small number of controllers (perhaps a few dozen).

Stealing Memory Cycles

Suppose the CPU has just initiated an I/O transfer from disk into memory, the disk having a DMA controller. The CPU then goes on executing whatever program is in progress. While the control unit is fetching instructions and storing results in memory, the disk controller is also sending data to memory. Both units (the CPU and the disk) are making interleaved use of the bus and the memory. In an ordinary memory, only one store or fetch per memory cycle is possible. That is, only one word can be transferred to or from memory in one memory cycle. What if the CPU and the disk both want to use the memory at the same instant?

This conflict is similar in some ways to that involving interrupts. Interrupt conflicts are resolved by priorities. Memory access conflicts are resolved far more simply. The CPU is made to wait while the disk fetches or stores a word in that memory cycle. Then, unless the same device immediately needs the next memory cycle, the CPU can use the next memory cycle, provided that some other device did not present a request to the memory in the interim. This phenomenon is called *stealing memory cycles.* If it happens too often, the CPU will run very slowly, because it will, in effect, be using a "slow" memory.

Why should the CPU be forced to wait while some other device has its need for a memory access satisfied? The alternative, forcing the device to wait, could lead to data loss. Consider that even such a slow device as a floppy disk can deliver a byte every 18 μsec. Much faster DMA devices presumably can present consecutive bytes within just a few microseconds. If a read-from-disk is in progress, and the disk had a word to store in memory but could not gain access to memory fast enough, the next word coming in from disk could overwrite the current word before it had been placed in memory. This case of data overrun requires that we wait for a whole disk revolution to retrieve the lost data. This is over 17,000 μsec for the fastest disk we have seen.

It seems wiser to force the CPU to pause for a few microseconds than to waste several thousand microseconds waiting to reread or rewrite a sector. Missing a byte on tape would take even longer because the tape repositioning operations are considerably slower.

The funneling of all data/instructions/addresses over one shared path, the bus, can lead to the bus being the bottleneck. The PDP-11 UNIBUS is rated as being capable of handling up to 2.6 MB/sec (1.3 MW/sec). Clearly, if you attempted to use several disks with data rates of 1.2 MB/sec, each with its own DMA controller, something would have to give.

The simplicity of connecting a new device to a bus-organized computer and the economies in building the bus-organized computer have made it the dominant way of building minicomputer and microcomputer systems.

Bus organized microcomputers are now quite common. The LSI-11, considered a microcomputer by DEC, is in most respects the same as a PDP-11. Its most significant differences are:

1. cost (lower)
2. size (smaller)
3. speed (slower)
4. bus (Q-bus, not UNIBUS)

The electrically different bus of the LSI-11 is a major reason for the cost, size, and speed being less. Whereas the UNIBUS has a maximum data rate of 2.6 MB/sec, the Q-bus, having fewer wires to carry traffic in parallel, can run at near half (1.6 MB/sec) that data rate. The UNIBUS has a higher data rate partly because it sends addresses on one set of lines and data on a separate set of lines. The Q-bus uses fewer lines, so it shares them between the address and data signals. This sharing takes time, leading to the lower data rate.

Programs can migrate easily from an LSI-11 to a PDP-11 or vice versa. Interfaces and controllers cannot migrate. A Q-bus-compatible serial interface is not compatible with the UNIBUS. UNIBUS-compatible serial interfaces are not compatible with the Q-bus. They may use exactly the same software, but they cannot fit on the same bus.

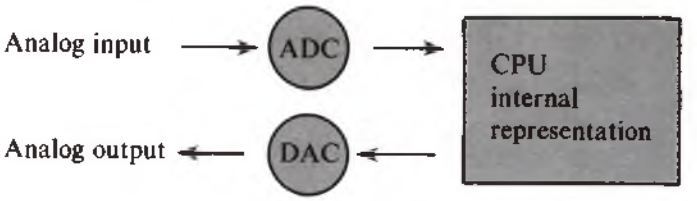


Figure 14.24 Transducer.

Analog Data

All the I/O we have discussed involves digital data. Most naturally occurring phenomena are basically analog in nature. How can a computer use analog inputs or produce analog outputs? How can a computer sense that a temperature is too high, or that a seismic tremor is of a particular magnitude? How can a computer make a subway car go faster or slower?

The answer to these questions comes in the form of another black box which accepts analog inputs and produces a digital equivalent output; it is called a transducer. In computing, if it is unidirectional, it is called an ADC or A/D (analog-digital-converter, analog-to-digital converter). A transducer, which takes digital inputs and produces an equivalent analog output signal, is called a DAC or D/A (digital-to-analog converter). Many transducers use voltage levels for their analog signals. The user then provides a means of sensing some item of interest—say, the temperature—and having the thermometer produce a voltage level proportional to the current temperature. Another temperature sensor might vary its resistance as a function of the temperature. Several different means may be used. Sooner or later they can be converted to a voltage level, which then can be fed into an ADC.

Conceptually, this transformation is similar to the situation we had regarding internal versus external number representations, represented in figure 14.24. Any quantity that can be measured can be converted into a voltage level proportional to the measured value. Temperature, weight, speed, sound level, brightness, vibration, humidity, etc. are just a few of the quantities that can be processed by a computer.

From the CPU's point of view, an analog input data item reaches the CPU as a binary number (usually 12 bits wide) at a given rate, determined by the ADC's sampling rate. Having previously been informed as to the correspondence between the high and low input values and the measured quantity, and just how the measured quantity varies as the voltage varies, the CPU's software driver can interpret the 12-bit data points. The hardware I/O for analog input is otherwise very much like character-oriented input. The difficulty, if any, is to assure consistency between the analog data source and the digital inputs.

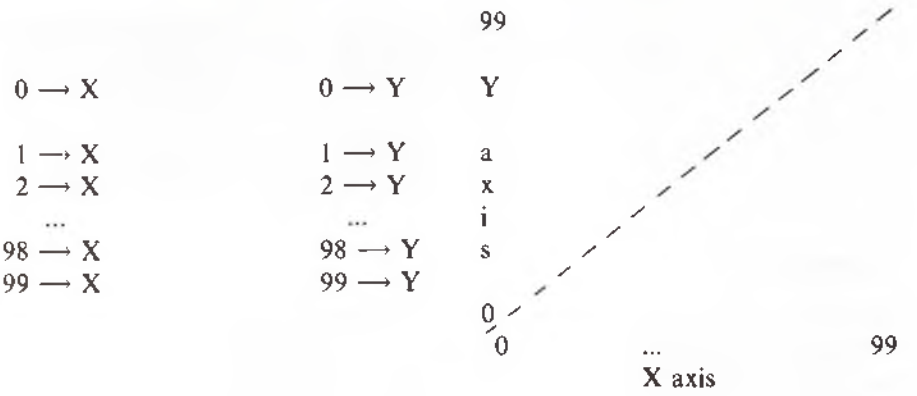


Figure 14.25 Plotting a diagonal line.

Analog outputs are handled by the CPU as binary data items. The DAC can be used to drive devices such as an X-Y display. This could be an X-Y plotter or an X-Y oscilloscope. Either of them can draw two-dimensional figures, as the X and Y voltage outputs from a pair of DACs vary from 0 to some maximum voltage level. The sequence of values shown in figure 14.25 would result in plotting or displaying the diagonal line shown in the same figure. A music synthesizer can be constructed by having a computer use several DACs to drive a number of audio amplifiers.

Digital I/O

It may seem strange to be discussing "digital" I/O on a digital computer, as if it were a special case. What have we been doing all along? The phrase "digital I/O" is used for what we could otherwise call logical I/O. We speak of digital I/O when the data coming in or going out are independent single-bit values. A typical digital input interface may have 16 signal lines connected to it. Suppose each such line is connected to an intrusion detector (i.e., a piece of silver foil glued to a window). If any intrusion detector is triggered (by a burglar?), the signal line connected to it switches its state, and the digital I/O interface senses this and requests an interrupt. The interface's interrupt handler will then fetch the content of the interface's 16-bit buffer to see which of the signal lines went high (or low).

A "counter" watching some experiment can be connected as a digital input source. Whenever the counter reaches a preset value, it sends a signal on its digital input line to the CPU while it resets itself. The CPU updates a counter-variable in its memory. In this way, extremely rapid events that could overwhelm any PDP-11 if it tried responding to each event directly can be "off-loaded" to a special piece of hardware (in this case, a counter), and the CPU need only service the counter periodically.

Digital outputs are usually made to control relays. The digital output signal is too weak to power any useful activity directly, but its weak signal is adequate for controlling a relay which in turn can control as powerful an electric source as you may wish. For instance, a computer-controlled parking-lot gate system uses digital outputs to raise access gates. A sensor in the roadway detects that the vehicle has gone beyond the gate, and it sends a digital input signal so that the CPU can in turn send a digital output signal to lower the gate. From the programming point of view, digital I/O is very much like byte-at-a-time character I/O, except that several bytes may be involved, and no character code is used; the data items are all logical in nature.

Instruction	Vector Addresses
EMT	30-32
TRAP	34-36
BPT	14-16
IOT	20-22

Figure 14.26 Trap interrupt vector addresses.

Traps

The occurrence of I/O interrupts are somewhat unpredictable. You initiate an I/O operation, and some time later, depending on the location of a desired record on a tape or disk, an interrupt occurs following its transfer.

A trap is similar to an interrupt except that it occurs in a very predictable way. In fact, the timing of trap-type instructions is so predictable that on some computers trap instructions are called user-invoked interrupts. When you execute a trap-type instruction, an interrupt is immediately triggered. The PS priority is not even examined; trap-type instructions always have the highest priorities, and no one can disable them.

The PDP-11 supports four trap-type instructions:

```
EMT     x ; emulate
TRAP    x ; trap

BPT       ; breakpoint trap
IOT       ; I/O trap
```

The first two instructions allow you to provide an 8-bit operand as part of the 1-word instruction. Thus you can think you have 256 different EMT instructions:

```
EMT 0, EMT 1, EMT 2, ..., EMT 255.
```

Similarly for the TRAP instruction; IOT and BPT have no operands. All four instructions have similar effects when executed. They cause the PC and PS to be saved in the system stack and a new PC and PS to be fetched from their interrupt vectors. The interrupt vectors for these instructions are shown in figure 14.26.

Just as certain symbolic names should be avoided when you are creating names for use with MACRO-11 (e.g., names beginning with "."), EMTs should be used only to access system-defined services. Many of the system macros are defined with EMT x. Users are free to use TRAP as they see fit.

IOT is slightly different from EMT and TRAP in that it has no operand field. The usual exit from the interrupt handlers for these four instructions is the same RTI we used previously. IOT's principal use is to access system-defined I/O related services. BPT, similarly, has no operand field; it is used in conjunction with some debugging software. A "breakpoint" is the address of a particular instruction of interest to you when you are debugging a machine-language program. Instead of sprinkling .SNAP

requests or similar requests in the program, you select points in the program and identify them as breakpoints. The debugging package you are interacting with copies and saves the instruction or partial instructions at these points and overwrites the saved locations with BPTs. As your program is executed, when it reaches a BPT, the debugging package regains control, and it can allow you to examine memory and registers, change values, etc., and then restore the original instruction and resume running your program, until the next BPT is encountered, and so on.

The EMT and IOT instructions were very important when the cost of disk storage was too high to allow its use on PDP-11s. This made the use of a linking loader very time consuming. So a way of linking independently assembled modules without using a linking program was necessary. This seems like a contradiction.

A linking loader satisfies a request passed to it by the assembler, by virtue of an assembly-time .GLOBL, as in

```
.GLOBL   SIN
...
JSR      %7,SIN
...
```

The JSR will be assembled with a one-word hole left for the unknown 16-bit address of "SIN". When the linker sees this request, it will search the system subroutine library for a module with an entry point name "SIN". Having found such a module, it is attached to the end of the previous module, and the address thus associated with "SIN" is plugged into the JSR. None of this is very rapid without a direct access device such as a disk, although tape-based linkers were used for a long time.

With the trap-type instructions, you simply agree that the SIN subroutine is assigned a code number—say, #137. Then a TRAP 137 is interpreted by the TRAP interrupt service routine as a call to the SIN subroutine. The TRAP interrupt handler would have to have a "transfer vector" equating TRAP argument numbers with subroutine addresses. User-provided subroutine arguments can be provided, if needed, by any of the techniques we saw earlier.

Linkage to service routines using trap-type instructions is simple, but the associated run-time overhead is higher than is the case with standard JSR-RTS linkage. You might say that using trap-type instructions is a method of deferring the binding of subroutine addresses from the usual link-time binding to a run-time binding.

Historically, trap-type instructions came late, as a generalization of the floating-point traps. After all, if we did not have any of EMT/TRAP/IOT/BPT, we could get exactly the same effect by forcing a floating-point overflow and having the overflow handler examine some flag to distinguish between a "legitimate" overflow problem versus a "user-invoked" interrupt. It has since become the practice on many computers to

provide trap-type instructions; in the meantime, instead of the term "floating-point traps," the phrase "floating-point exception" is used. On multi-user systems, the trap-type instructions are the *only* legitimate way to request I/O and other system services. A multi-user operating system can arrange for all interrupt vectors to be "protected" against user modification. Since a trap necessarily swaps the PS and PC, a trap can provide a protected entry into the operating system. On other computers, trap-type instructions may be called "supervisor call" or "executive request" instructions.

Processor Traps

The mechanism used to report "M-TRAP TO 4" errors and "M-TRAP TO 10" errors is similar to that used with the trap-type instructions we just examined.

Whereas a trap-type instruction is executed at the user's request, certain conditions sensed by the CPU trigger the processor traps. The ensuing action is otherwise identical to that initiated by the trap-type instructions. We can think of processor traps as being hardware-initiated traps, as opposed to the user-initiated traps due to trap-type instructions. What triggers a processor trap?

When the CPU detects an attempt to execute an illegal instruction, or use a nonexisting address, or use an odd address when an even address is required, it will force a trap using the interrupt vector at location 4. Similarly, if you used a reserved instruction (i.e., a privileged instruction) such as a HALT when in multi-user mode, the CPU would force a trap using the interrupt vector at location 10.

A power failure leads to using the interrupt vector at location 24, in an attempt to reach an orderly shutdown handler.

Trap Trace

We saw locations 14–16 assigned earlier as the interrupt vector for the BPT trap instruction. The same vector is used in conjunction with T-bit-initiated traps. The T bit is the PS bit which is located between the PS priority field and the four CC bits, as seen in figure 14.27. T is short for trap. Whenever the T bit is set, the control unit will force a trap using the interrupt vector at location 14. So long as the T bit remains set, you get the effect of having executed zero-length BPTs between every one of your instructions. Clearly, you would want the PS word at location 16 to have its T bit clear. The trap-handler that location 14 leads to is presumably a debugging package.

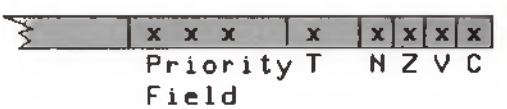


Figure 14.27 Processor status (PS).

Debugging at this level of detail, instruction-by-instruction, is called a *full trace*. The trace-trap-handler has a special problem to contend with. Being an interrupt handler, it should exit with an RTI. However, the RTI will restore the PC and PS, and unless the trap handler made a special effort to clear the T bit in the PS saved in the stack, the restoration of the PS caused by the RTI will immediately force another T-trap! The solution requires using an instruction whose sole purpose is to avoid this problem. The RTT (return from trap) instruction acts like an RTI, but it also inhibits any new T-trap from occurring until one instruction following the RTT's execution can be completed.

A full trace is a very powerful debugging tool, but it is such an expensive one in terms of CPU overhead that its use can rarely be afforded. One should, however, be aware of how hardware can be designed to facilitate debugging.

Summary

We have seen how higher-speed data transfers require reducing the CPU's intervention to a single interrupt per block of data. The data itself are transferred directly to memory from the device, or vice versa, by virtue of a direct memory access (DMA) controller. The two major categories of devices which require DMA controllers are the sequential access and the direct access devices.

The most important of the sequential access devices is the industry compatible magnetic tape. These tapes and tape drives are characterized by the recording density (bpi or fpi), tape speed (ips), and record (block) sizes used. The average access time to a tape record is measured in minutes. Sequential access to tape records is much faster.

The magnetic disk is the principal type of direct-access storage device. Unlike most tapes, it is designed to support update-in-place. As with tapes, disks are not code-sensitive, so binary information can be stored on them. The major physical characteristics of a disk are the number of recording surfaces, the number of tracks per surface, the number of sectors per track, the number of bytes per sector, the rotation speed (rpm), and the track-to-track maximum and average seek times.

Various measures of performance have been discussed. Among these are turnaround, throughput, response time, and the number of concurrent users.

Several special-purpose instructions have been introduced. RESET restores the CPU to a well-defined state. WAIT is used to reduce memory contention, in lieu of a busy-wait loop. Programmer-invoked interrupts EMT, TRAP, BPT, and TRAP were introduced, as was the trace bit T-bit trap.

The processing of analog data is made possible by use of an analog-to-digital converter. Conversely, a computer can produce analog signals using a digital to analog converter. From the programmer's point of view, handling analog I/O is not much different from handling character I/O. Similarly, digital I/O, which refers to handling independent logical signals, is very much like character I/O.

The role of data channels in supporting high speed data transfers was discussed and contrasted with bus-organized computers such as the PDP-11. It uses a UNIBUS, which provides a path linking the CPU, memory, and all other devices. The LSI-11, which supports essentially the same instruction set as the PDP-11, uses a lower-performance, lower-cost bus called the Q-bus. The distinction between hardware compatibility and software compatibility is well illustrated by the contrast between UNIBUS controllers and Q-bus controllers.

Exercises

14.1 An application program processes records of 100 characters each in 50 milliseconds per record, once the record is in memory. Characters can be read from the device at the rate of 1,000 characters per second. How long will it take to read and process 20 records under the following conditions?

(a) Using busy-wait input?
(b) Using direct-memory-access input?

14.2 Exactly what would you have to do if you wanted to assume responsibility for handling all M-TRAP TO 4 errors?

(a) Redefine the system macro definition for M-TRAP TO 4.
(b) Write the code to print an error message.
(c) Initialize memory location 4 so that it points to your error handler.
(d) Enable for error interrupts.
(e) Execute a trap instruction when an error occurs.

14.3 You have been asked to rewrite old PDP-11 programs for a computer which is similar to the PDP-11 except that it has no JSR instruction. Rewrite program segments (a) and (b) so they perform the equivalent operations using the IOT instruction.

```
(a)     MOV     #22,%0      (b)     JSR     %,PQ
        JSR     %7,ABC              .WORD   X
        ...                         ...
ABC:    ASL     %0          PQ:     MOV     @(%5)+,%1
        RTS     %7                  MOV     %1,%2
                                    ASL     %1
                                    ADD     %1,%2
                                    RTS     %S
```

14.4 True or False?

(a) An interrupt handler cannot call a subroutine.
(b) DMA devices have higher data rates than non-DMA devices.
(c) Traps are faster than subroutine calls.
(d) Array descriptors or dope vectors are used to speed up array references.

(e) The average seek time for a fixed-head disc typically lies between 10 and 50 msec.
(f) Interrupts can cause stack overflow.
(g) Latency is negligible on fixed-head discs.
(h) Interrupt priorities are assigned according to interrupt frequencies.
(i) Nested macro definitions usually reduce program memory requirements.
(j) If a program is run again without any changes, the I/O interrupts will occur at exactly the same points in the program.

14.5 At the beginning of this chapter it was stated two memory references are necessary to save the PC and PS and that it takes three memory references to restore them. Explain why this is so.

14.6 It is a common practice to periodically copy all the information from a computer system's disks onto magnetic tapes for secure storage elsewhere. This practice is called creating *backup files*. Suppose you created backup files once a week. If it takes one minute to mount a magnetic tape (place it on a tape drive) and five minutes to rewind and remove a magnetic tape, how long would the backup operation take, given the following configurations, if you used 512-byte records?

(a) One 800-bpi, 45-ips tape drive, two RL01 disks.
(b) Same tape, two RM03 disks.
(c) One 1,600-bpi, 75-ips tape drive, four RM03 disks.

14.7 Redo the previous problems, assuming that 16 KB records are written on tape.

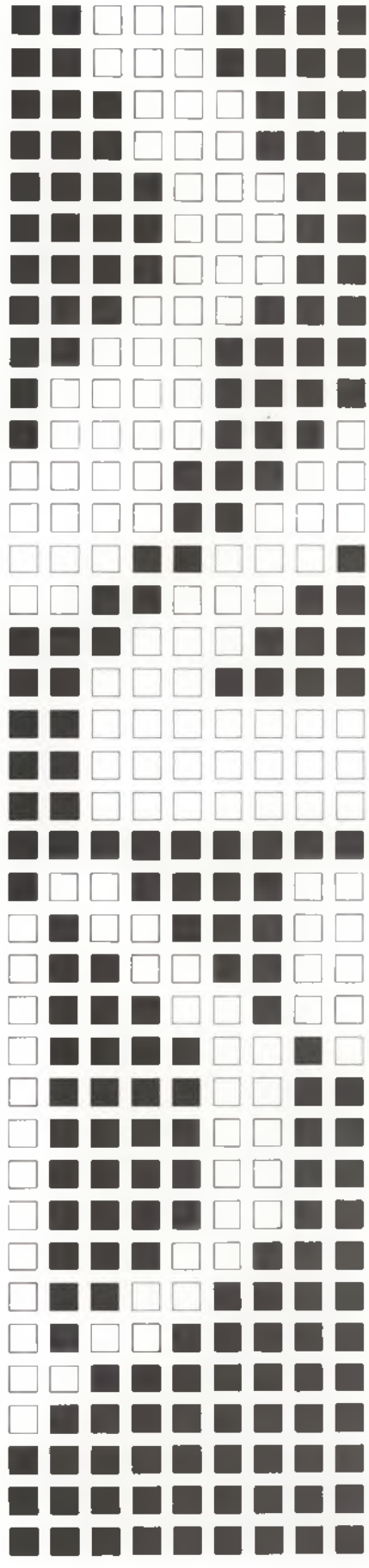
14.8 At some installations, creating backup tapes takes significantly longer than the preceding calculations indicate it should. What operation may be performed in conjunction with the backup that would be worth the extra time?

14.9 Suppose a system supports spooling, and it has copied 10,000 card images into the spool input file, while 4,000 full-width print lines of 132 characters each have accumulated in the spool output file. Spool files are maintained here on a disk with a 256-byte sector size.

(a) Calculate the number of sectors required for the spool files if you are using a blocking factor of 1.
(b) How many sectors are used if you use a 1 KB physical record? In that case, what is the blocking factor?

15

selected topics



We shall discuss a number of topics in this section. Some of them are primarily software oriented, others are hardware oriented. They all play a part in helping us understand computing.

Variable Length Macros

The simple macros we have been writing so far each had a predictable length once they were defined. For instance,

```
.MACRO   SHIFT X
ASL      X
ASL      X
.ENDM
```

will always cause the invocation SHIFT A to expand into two lines. We can say that such simple macros are fixed-length macros. What if you needed the ability to create at assembly time patterns of text with varying numbers of lines? Consider a program in which you have

```
MOV      %1,-(SP)        ; save 1
MOV      %2,-(SP)        ; save 2
...

MOV      %1,-(SP)        ; save 1
MOV      %2,-(SP)        ; save 2
MOV      %3,-(SP)        ; save 3
...
```

You could define a set of SAVE macros

```
.MACRO   SAVE12
MOV      %1,-(SP)
MOV      %2,-(SP)
.ENDM

.MACRO   SAVE123
MOV      %1,-(SP)
MOV      %2,-(SP)
MOV      %3,-(SP)
.ENDM
```

Each of these is a fixed-length macro. Consider, however, the possibility of using an assembly-time conditional directive *within* a macro definition.

```
.MACRO   SAVE      NUM
  .IF EQ NUM-1
         MOV %1,-(SP)
  .ENDC
;
  .IF EQ NUM-2
         MOV       %1,-(SP)
         MOV       %2,-(SP)
  .ENDC
.ENDM
```

Then the two valid invocations and their expansions are

Invocation	**Expansion**
`SAVE 1`	`MOV %1,-(SP)`
`SAVE 2`	`MOV %1,-(SP)`
	`MOV %2,-(SP)`

In general, we could have a sequence of mutually exclusive conditional blocks within a macro definition:

```
.MACRO   M         ARG
  .IF EQ ARG-1
         Block 1
  .ENDC
;
  .IF EQ ARG-2
         Block 2
  .ENDC
;
...
  .IF EQ ARG-n
         Block n
  .ENDC
;
.ENDM
```

Here we are using the dummy argument ARG to select which statements shall be included or excluded when the macro invocation is expanded. The following extended example shows how an idea can evolve and exploit the facilities supported by the assembler. We will be using this as an opportunity to introduce a few more directives.

Consider writing your own debugging macro; call it HELP. At the very least, it will scan its argument list and display (in octal) the address and the value associated with each address in its argument list. It will continue doing so until it finds a zero address. Typical HELP calls are:

```
A:      JSR     R5,HELP
        .WORD   X,0
        ...
B:      JSR     R5,HELP
        .WORD   P,Q,0
```

The first call at A will display X and (X). The second call, at B, will display P, (P), Q, and (Q). You would probably want to control inclusion of these debug statements with an assembly time conditional. Thus, we could have

```
SW = 1 ; include the debug code
...
.IF NE SW
  JSR R5,HELP
  .WORD X,0
.ENDC
...
.IF NE SW
  JSR R5,HELP
  .WORD P,Q,0
.ENDC
...
```

If you find yourself writing these four-line sets of statements frequently, you may be inspired to create a macro H:

```
.MACRO  H       A,B,C,D,E
JSR     R5,HELP
.WORD   A,B,C,D,E
.ENDM
```

Then the previous code sequence becomes:

```
SW =1 ; include the debug code
...
.IF NE SW
  H  X,0
.ENDC

...
.IF NE SW
  H  P,Q,0
.ENDC
...
```

At this time we can introduce the directive .IIF. This is an "immediate .IF" directive for one-line conditional blocks. It has the general form

```
.IIF condition argument,statement
```

The conditions used with .IFF are the same ones used with the old .IF. The conditional block for the .IIF is the "statement" which appears on the same line. The statement may have a label. The statement must not have any leading blanks. No matching .ENDC is expected and none should be provided. The "statement" part may have a label, but the directive .IIF itself may not be labeled. Examples:

```
  (a)                    (b)
ABC:                          .IIF NE XYZ,ABC: HALT
  .IFF NE XYZ,HALT
```

Here the two lines for (a) and the single line for (b) are almost equivalent. They are completely equivalent in terms of including or excluding assembly of the HALT. They differ only in that the label ABC will always have a definition in case (a) but not necessarily so in case (b).

Returning to our extended example, we can rewrite the latest version as

```
SW = 1
...
.IIF NE SW,H    X,0
...
.IIF NE SW,H     P,Q,0
...
```

A perceptive reader may recognize that in each invocation of H we always have a trailing "0". Why not let the macro take care of this? So we can define another macro, call it HH.

```
.MACRO  HH       A,B,C,D,E
JSR     R5,HELP
.WORD   A,B,C,D,E
.WORD   0
.ENDM
```

The code sequence now reads

```
SW = 1
...
.IIF NE SW,HH    X
...
.IIF NE SW,HH     P,Q
...
```

Another reader may suggest, "Why not have a macro H2 so you could reduce unnecessary writing?"

```
.MACRO  H2      A,B,C,D,E
  .IIF NE SW,HH     A,B,C,D,E
.ENDM
```

or

```
.MACRO  H2      A,B,C,D,E
  .IF NE SW
        JSR     R5,HELP
        .WORD A,B,C,D,E,0
  .ENDC
.ENDM
```

The first version of H2 works, but it is somewhat obscure. The second version is easier for us to grasp. They do, however, generate the same code. (What would happen if you accidentally provided two definitions for the same macro? Would that be flagged as an error? No; the first definition would be used until the second definition is seen. Then the second definition would be used from that point on.)

So our code now looks like

```
SW = 1
...
H2  X
...
H2  P,Q
...
```

At this stage we can't really do much more to make the invocations simpler. We can, however, reconsider the implementation of the basic macro. Suppose we had

```
.MACRO  H       A,B,C,D,E,F,G,H
JSR     R5,HELP
.WORD   A,B,C,D,E,F,G,H
.WORD   0
.ENDM
```

There is no conflict in using the dummy argument H in the definition of a macro named H. With this longer argument list, we could provide up to eight parameters in a call on the HELP routine. Suppose we have the invocation

```
H       P,Q
```

Its expansion is

```
 JSR    R5,HELP
.WORD   P,Q,0,0,0,0,0,0
.WORD   0
```

Each missing argument is replaced by a zero. Here, since we provided values for only A and B—namely, P and Q, respectively—the values associated with the other dummy arguments C, D, E, F, G, and H lead to six zeroes in the argument list. Instead of having the HELP subroutine scan its argument list until it finds the first zero, could we supply it with an argument count? Let us have a version of HELP called HELP2, which expects its first argument to be a value that represents how many consecutive non-zero addresses follow the count in its argument list. Thus, we would write

```
JSR     R5,HELP2
.WORD   1,X
...
JSR     R5,HELP2
.WORD   2,P,Q
...
```

The macro processor provides the directive .NARG to help us in this regard. This directive can be used only within macro definitions. Its general form is

```
.NARG   name
```

The dummy name "name" will be assigned an assembly-time value corresponding to the number of nonmissing arguments in the macro invocation currently being expanded. So

```
.MACRO  M       X1,X2,X3
.NARG   A
.WORD   X1,X2,X3,A
.ENDM
```

leads to the following expansions:

Invocation		Expansion	
M	BA	.WORD	BA,0,0,1
M		.WORD	0,0,0,0
M	C,D	.WORD	C,D,0,2

Returning to our extended example, we can use .NARG in conjunction with HELP2 in defining a macro H3

```
.MACRO  H3      A,B,C,D,E,F,G,H
.NARG   CNT
JSR     R5,HELP2
.WORD   CNT,A,B,C,D,E,F,G,H
.ENDM
```

The invocation H3 B,D,Y leads to

```
JSR     R5,HELP2
.WORD   3,B,D,Y,0,0,0,0,0
```

and HELP2 would have to skip over the trailing zero words to effect a proper return with its RTS R5.

One of the deficiencies of this macro processor is shared by many higher level languages. If we think of macro invocations and subroutine calls as being superficially similar in that each has an invocation of the form

macro-name argument-list

subroutine-call argument-list

they both have a similar constraint. There is no natural way of letting the length of the argument list vary from one call to another, such that unnecessary zero words can be avoided. The best we can do in MACRO-11 is clumsy:

```
.MACRO  H4      A,B,C,D,E,F,G,H
.NARG   N
JSR     R5,HELP4
.WORD   N,A
.IIF EQ N-2,.WORD   B
.IIF EQ N-3,.WORD   B,C
.IIF EQ N-4,.WORD   B,C,D
.IIF EQ N-5,.WORD   B,C,D,E
.IIF EQ N-6,.WORD   B,C,D,E,F
.IIF EQ N-7,.WORD   B,C,D,E,F,G
.IIF EQ N-8,.WORD   B,C,D,E,F,G,H
.ENDM
```

Invocation	**Expansion**	
H4 X	JSR	R5,HELP4
	.WORD	1,X
H4 B,D,Y	JSR	R5,HELP4
	.WORD	3,B,D,Y

All along we have not expected much from the various versions of the HELP subroutines. Given the argument list

```
.WORD   X,Y,0
```

HELP merely prints four octal words, displaying X, (X), Y, and (Y). We could easily provide much more useful debugging information. Consider

```
.MACRO   HELPA   A,B,C,D
JSR      R5,HELPA
.NARG    CNT
.WORD    CNT
.ASCIZ   /A/
.EVEN
.WORD    A
.ASCIZ   /B/
.EVEN
.WORD    B
.ASCIZ   /C/
.EVEN
.WORD    C
.ASCIZ   /D/
.EVEN
.WORD    D
.ENDM
```

Then the invocation HELPA P,Q,ABC results in the expansion

```
JSR      R5,HELPA
.WORD    3
.ASCIZ   /P/
.EVEN
.WORD    P
.ASCIZ   /Q/
.EVEN
.WORD    Q
.ASCIZ   /ABC/
.EVEN
.WORD    ABC
```

Then HELPA can print the symbolic name of the variables instead of or in addition to their octal addresses. It could respond to the previous request with output such as

```
    HELPA at location 1242
P= 000001     Q= 177770     ABC= 000123
```

You could arrange for the output to give you a symbolic address of the point at which the subroutine was invoked, such as

```
    HELPA at location 73$
P= 000001     Q= 177770     ABC= 000123
```

by using the following macro:

```
            .MACRO   HELPB   A,B,C,D,?X
            JSR      R5,HELPA
            .NARG    CNT
            .WORD    CNT+1
   .LIST    ME
X:          .ASCIZ   /X/
   .NLIST   ME
            .EVEN
            .WORD    X
            .ASCIZ   /A/
            .EVEN
            .WORD    A
            .ASCIZ   /B/
            .EVEN
            .WORD    B
            .ASCIZ   /C/
            .EVEN
            .WORD    C
            .ASCIZ   /D/
            .EVEN
            .WORD    D
            .ENDM
```

Any invocation of HELPA in which the arguments marked with a "?" are missing will automatically have these missing references filled in by the assembler with a fresh local label in the range 64$ to 127$. We will have more to say about these generated labels later. The purpose of the .LIST ME and .NLIST ME is to force the assembler to print the line with the definition for the label which it generates for you. Otherwise you would have no record of this label or its position within your source module.

Other Assembler Features

Many of the features we will discuss here are heavily used by systems programmers. You may or may not have an opportunity to use these, but at least you will know why they are used and what they are used for.

Generated Labels

It is sometimes useful and sometimes necessary to use labels within a macro definition. Suppose you were using a macro MX while defining a macro SAM

```
            .MACRO   SAM      A,B,C
            MOV      A,%0
            BEQ      HERE
            MX       B
HERE:       MOV      %0,C
            .ENDM
```

It may be the case that the length of MX (i.e., the number of words generated by its expansion) depends on the particular value of the argument B at the time SAM is invoked. If that is the case, you cannot use a label of the form .+n since the value for n is unpredictable. So you have to use a label such as HERE.

Unfortunately, if you invoke SAM two or more times, then HERE will be multiply defined. We could try using a local label—say, 2$—in place of HERE. But then we run the risk of that 2$ conflicting with other uses of 2$ as a local label.

We could solve the problem by making HERE a dummy argument; we could redefine SAM as SAMMY.

```
        .MACRO  SAMMY   A,B,C,HERE
        MOV     A,%0
        BEQ     HERE
        MX      B
HERE:   MOV     %0,C
        .ENDM
```

Then each invocation of SAMMY would require that we provide a fourth argument, which would have to be a name that is not otherwise being used. We might write

```
SAMMY   0,Y,Z,NNN

...
SAMMY   P,Q,R,MMM
...
```

This is a solution which avoids one problem by placing an additional burden on you. MACRO-11 provides a mechanism for making names for you.

If you precede a dummy argument name by the character ? (question mark), nothing special will happen unless you invoke the macro without specifying a value for the ?-qualified dummy argument. If the value is not given, then the assembler will provide one for you. It will choose the next available local label from the list 64$, 65$, . . ., 127$. Each occurrence of the dummy argument in the body of the macro definition will be replaced by the same local label in one given expansion. You can have several different ?-qualified dummy arguments being used in a definition. We can rewrite the preceding definition as follows:

```
        .MACRO  SAMMY   A,B,C,?HERE ; note the ?
        MOV     A,%0
        BEQ     HERE
        MX      B
HERE:   MOV     %0,C
        .ENDM
```

Notice that the ? is used only in the .MACRO line.

Each new invocation of a macro which needs these so-called automatically created symbols will get a fresh value from the list 64$, . . ., 127$. When the list is exhausted, it recycles and starts over again with 64$.

Temporary Radix Control

We discussed the .RADIX directive some time ago. Its use can lead to problems, especially if macros are being used. Suppose you have a macro

```
.MACRO    TIME     A
EMT       123
.WORD     A
.ENDM
```

You expect `TIME TM` to lead to execution of `EMT 123` followed by a `.WORD TM`. This might be a request for the operating system to provide you with the elapsed wall-clock time as a double-precision integer in locations TM and TM+2. What would happen if you used TIME three pages later, after the occurrence of a `.RADIX 10` hidden somewhere on the second page? Then the assembler would interpret the 123 in `EMT 123` as if it had been EMT 123.; 123 decimal! Then at run time you would be invoking the emulate service 173 octal instead of 123 octal. That might be difficult to track down. How can we avoid such problems?

The temporary radix notation is provided for such an eventuality. It is also convenient (i.e., just like the special case .IIF for the .IF). The temporary radix notation is:

```
^D        ; decimal radix
^O        ; octal radix
^B        ; binary radix
^C        ; one's-complement
```

The numeric constant immediately following the ^x notation is processed as a number in the specified radix, or complemented in the case of a ^C. We can now rewrite our TIME macro so that it becomes insensitive to the particular default radix forced by any .RADIX directive.

```
.MACRO    TIME     A
EMT       ^O123
.WORD     A
.ENDM
```

The expansion of this form of the macro definition is now insensitive to the effects of any .RADIX directives.

.IRP

The indefinite repeat directive .IRP allows you to duplicate a block of statements with each duplicate differing in a useful way. Consider how we defined the SAVE macros earlier:

```
.MACRO   SAVE1
MOV      %1,-(SP)
.ENDM
.MACRO   SAVE12
MOV      %1,-(SP)
MOV      %2,-(SP)
.ENDM
```

Using .IRP, we no longer need to define a macro. We can write

```
.IRP     X,<%1,%2>
MOV      X,-(SP)
.ENDM
```

which leads the assembler to generate

```
MOV      %1,-(SP)
MOV      %2,-(SP)
```

Even though .IRP is not a macro, it always has one dummy argument, which should be followed by a list of items enclosed in braces. The general form is

.IRP dummy argument, *argument list*
block of statements, some of which use the dummy argument
.ENDM

The block will be duplicated once for each item in the argument list. The first duplicate will have all occurrences of the dummy argument replaced by the first item in the argument list. The second duplicate will likewise have all occurrences of the dummy argument replaced by the second item from the argument list, and so on. So in the preceding example we get exactly the same code that would have resulted from the invocation of the macro SAVE12.

The statements

```
.IRP     A,<7,XXX+1,-5>
.WORD    A
.ASCII   /A/
.EVEN
.ENDM
```

will expand as follows

```
.WORD     7
.ASCII    /7/
.EVEN
.WORD     XXX+1
.ASCII    /XXX+1/
.EVEN
.WORD     -5
.ASCII    /-5/
.EVEN
```

The .IRP directive can be used in a macro definition, provided that you don't forget to provide its matching .ENDM. Consider

```
.MACRO    SAVE L
  .IRP    Y,<L>
  PUSH Y
  .ENDM
.ENDM
```

Then `SAVE <%2,%4,A>` leads to

```
.IRP Y,<%2,%4,A>
PUSH Y
.ENDM
```

which leads to

```
PUSH      %2
PUSH      %4
PUSH      A
```

Whenever braces are used to provide a macro with an argument, the outermost level of braces is stripped off as the macro expansion descends into inner levels, when macros are defined in terms of other macros or when list-oriented directives such as .IRP are used.

We could have written

```
.MACRO  SAVE L
  .IRP X,L
  PUSH X
  .ENDM
.ENDM
```

in which case, the equivalent invocation is

```
SAVE <<%2,%4,A>>
```

See exercise 15.7 for a slightly different kind of .IRP directive.

.REPT

There is another directive which can simplify life. Misusing it can also be very expensive. The .REPT directive allows you to duplicate a block of statements a specified number of times. The repeat count must be an expression with an assembly-time value. None of the statements in a .REPT block may have a label. Consider that

```
.REPT   3
.WORD   0
.ENDM
```

is a hard way of writing its equivalent

```
.WORD   0,0,0
```

Since any legal statement may appear in a repeat block, we can write

```
.REPT   3
.WORD   N
 N=N+2
.ENDM
```

Presumably, if N had the value 2 prior to reaching this .REPT request (perhaps due to an N=2 statement), then this repeat will expand as

```
.WORD   2 ; N=2 before, =4 after
.WORD   4 ; N=4 before, =6 after
.WORD   6 ; N=6 before, =10 after
```

The repeat count may be an expression. If we had

```
.REPT   M-2
BLAH
.ENDM
```

then BLAH will be duplicated M−2 times. If M−2 has a value below 1, the BLAH will simply be skipped by the assembler.

It is not a good idea to write

```
A:      .REPT   10000.
        .BYTE   0
        .ENDM
```

This places an enormous burden on the assembler, and a correspondingly large burden on the linker. Neither of them perform any compression of repeated data. So doing this also consumes a large amount of disk space for the object modules and executable modules.

We can use .REPT within macros. Our old example

```
.MACRO   DOUBLE X
ASL      X
ASL      X
.ENDM
```

could be rewritten as

```
.MACRO   DBL     N,X
  .REPT  N
  ASL X
  .ENDM
.ENDM
```

and then

`DBL 2, ABC` and `DOUBLE ABC`

are equivalent.

Nested Macro Definitions

The body of a macro may contain almost arbitrary text. It might even contain another macro definition. You might say: "But we discussed this a long time ago." Containing a macro definition and using previously defined macros are two very different things. We have discussed the latter. The former is definitely a new topic.

Suppose you have been commissioned to write macro definitions to facilitate the use of the mathematical subroutine library. It contains subroutines with entry point names such as SIN, COS, TAN, etc. for mathematical functions such as the sinus, cosine, tangent, etc. There might be some 150 entry points. So you begin defining the following macros:

```
.MACRO   SIN     A
JSR      R5,SIN
.WORD    A
.ENDM
.MACRO   COS     B
JSR      R5,COS
.WORD    B
.ENDM
.MACRO   TAN     C
JSR      R5,TAN
.WORD            C
.ENDM
```

Then it dawns on you that there is no need to use different dummy argument names A, B, C, . . . in each definition. So you proceed to write:

```
.MACRO    SIN      X
JSR       R5,SIN
.WORD     X
.ENDM
.MACRO    COS      X
JSR       R5,COS
.WORD     X
.ENDM
.MACRO    TAN      X
JSR       R5,TAN
.WORD     X
.ENDM
```

Then it strikes you that these sets of four-line definitions differ from each other in precisely two places, as shown by the circles below:

```
.MACRO (SIN) X    .MACRO (COS) X    .MACRO (TAN) X
JSR R5,(SIN)      JSR R5,(COS)      JSR R5,(TAN)
.WORD X           .WORD X           .WORD X
.ENDM             .ENDM             .ENDM
```

What do you normally do when you have a set of lines which recur frequently and which differ in some systematic fashion? You consider using a macro to generate these lines. So here we can try

```
.MACRO  DEF      NAME
   .MACRO NAME X
   JSR R5,NAME
   .WORD X
   .ENDM
.ENDM
```

NAME is just a dummy argument. When the assembler first sees the definition for the macro DEF, its body will be stored away, and the name DEF is placed in the macro name table. If subsequently DEF is invoked, as in

```
DEF       SIN
...
DEF       COS
...
```

then the following "code" will result from the expansion of these invocations of the macro DEF:

```
.MACRO    SIN      X
JSR       R5,SIN
.WORD     X
.ENDM
```

```
            ...
.MACRO      COS       X
JSR         R5,COS
.WORD       X
.ENDM
            ...
```

As usual, the assembler immediately tries to assemble what it is generating—namely, macro definitions for SIN and COS. This results in placing the names SIN and COS in the macro name table and saving their definitions. So here we have reduced our writing from (4 lines)* (150 needed macro definitions) to (6 lines)+1*(150 invocations of DEF). You could, if you wished, let the assembler do more for you.

```
.MACRO    DEFX     LIST
  .IRP    NAME,<LIST>
  DEF NAME
  .ENDM
.ENDM
```

permits you to write

```
DEFX      <SIN,COS,TAN>
```

which leads to

```
.IRP      NAME,<SIN,COS,TAN>
DEF       NAME
.ENDM
```

which in turn leads to

```
DEF       SIN
DEF       COS
DEF       TAN
```

leading to the three desired definitions.

Note that a failure to have matching .ENDMs for any use of .MACRO, .IRP, or .REPT can be a disaster. Too many .ENDMs is a minor problem. Too few .ENDMs will force all the text following the missing .ENDM to be included as part of the body of a macro definition or an .IRP/.REPT block. Nothing beyond the missing .ENDM will be assembled. The assembler's view of which .ENDM is missing may not coincide with your view.

Tables, Lists, Queues, and Trees

Recall that all the entries of an array are alike. Because they are homogeneous, it is a relatively simple matter to calculate an array component's offset, given the component's subscripts. A table, on the other hand, has rows and columns with distinctly nonhomogeneous entries. One column of a table might be a list of names, another a set of addresses, etc. A restaurant menu is a typical table, as in figure 15.1. A sales report with monthly and year-to-date data may include tables of the form shown in figure 15.2. Alphabetic entries may vary in length considerably. Consider one way of storing part of this table (we will store only the names and the monthly sales column):

```
.ASCII    /NUTS            /
.WORD     12
.ASCII    /BOLTS           /
.WORD     1
.ASCII    /LOCOMOTIVES     /
.WORD     8
```

At the cost of setting aside a dozen or so bytes for each name, we can ensure that adequate space is provided for the longest name we anticipate and provide uniform-length entries to facilitate rapid access. This may be fine for small tables, but it would be a disaster for a firm with thousands of products, or for a student record system with a student population in the tens of thousands. We should consider packing the names in contiguous bytes, not wasting any space for trailing blanks. Then the quantities could be paired with pointers to the corresponding names, as in figure 15.3.

Figure 15.1 A menu as a table.

Item	Price
TEA	.25
COFFEE	.35
POP	.30

Figure 15.2 A sales report as a table.

Item	M-sales	YTD sales	New YTD	% goal
NUTS	12	102	114	96
BOLTS	1	2	3	5
LOCOMOTIVES	8	98	106	130

Figure 15.3 Using pointers.

```
N1:       .ASCIZ   /NUTS/
N2:       .ASCIZ   /BOLTS/
N3:       .ASCIZ   /LOCOMOTIVES/
;
TABLE:
COL1:     .WORD    N1,N2,N3          ; name pointers
COL2:     .WORD    12,1,8            ; monthly sales
```

Figure 15.4 Using pointers.

If we were to sort this table based on lexical ordering of the names, we would leave the names where they are and simply reorder the name pointers N1, N2, and N3 as they appear in COL1. Of course, the values in COL2 should be reordered to remain consistent. We could avoid moving anything by building yet another vector, one which has its pointers arranged to retrieve items alphabetically, as in

```
VECT:       .WORD    N2,N3,N1
```

Lists

What we have just constructed above, using pointers, is a kind of list. Lists use pointers extensively.

A *one-way linked list* consists of a series of *nodes*, each of which consists of a pair of items (value, pointer). Any list always has a *head* and a *tail*. The head is the beginning of the list and the tail is the end of the list. It is useful to maintain two pointers for each list: a pointer to the list's head and a pointer to the list's tail. We have to have some way of distinguishing between a valid pointer and a null pointer; we will use the number 0 to represent a null pointer. An empty list is designated by having a null value as the pointer to its head.

The most important characteristic of a list is that its nodes need not occupy consecutive memory locations. You do not need a large block of contiguous memory to store a large linked list. Its nodes can be placed wherever any space is available. That is the purpose of having a pointer in each node. Each node points to its successor node, wherever it may be.

A linked list can be used in place of a one-dimensional array. Instead of having the array RR:

```
RR:         .WORD    12,1,8
```

we could construct the one-way linked list OWL:

```
OWL:        .WORD    12,1$     ; (value, pointer)
1$:         .WORD    1,2$
2$:         .WORD    8,0       ;
```

The pointer to the head of this list will have the value OWL; the tail pointer value is 2$. Symbolically, we can depict this as in figure 15.5. Why use a list? You should not if a static array will do just as well. If the number of entries is expected to grow or shrink at execution time, then you should consider using a linked list. An array is similar to a magnetic tape in that inserting or removing information from the middle of an array (or a tape) is awkward. A linked list is analogous to a direct-access storage device; insertions and removals are easy. Consider inserting a node with the value 123, following the first node, at OWL.

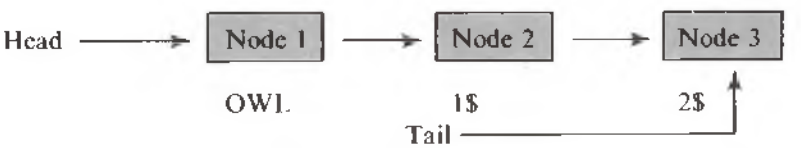


Figure 15.5 A one-way linked list.

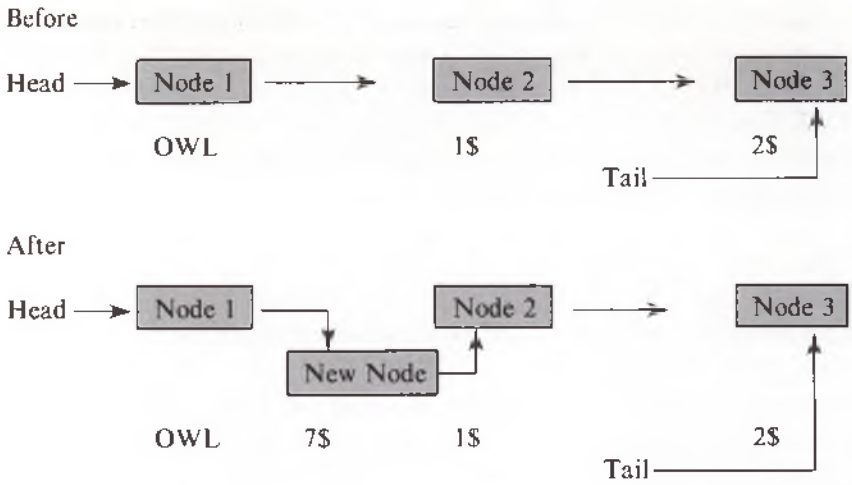


Figure 15.6 Inserting a node.

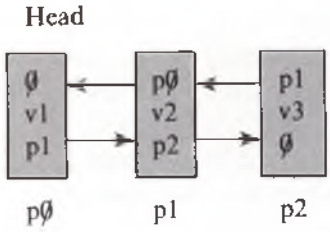


Figure 15.7 A two-way linked list.

Before

```
OWL:      .WORD    12,1$     ; (value, pointer)
1$:       .WORD    1,2$
2$:       .WORD    8,0       ;
```

After

```
OWL:      .WORD    12,7$     ; (value, pointer)
1$:       .WORD    1,2$
2$:       .WORD    8,0       ;
;
7$:       .WORD    123,1$    ; new node follows node 1
```

We can use any available space to store the new node. We then update node 1's pointer to link in the new node, and ensure that the new node's pointer field maintains the desired link. This is depicted in figure 15.6.

Deletion of a node is just as simple. Managing the space to allocate for new nodes can be performed by using another list called the *free list*. It can help you keep track of which space is available for use (free). Whenever you delete a node from a list, you don't just throw it away; it is placed back on the free list.

Finding an item in a one-way linked list requires searching each node, beginning with the head node. If you expected to do a lot of searching, you might prefer to use a *two-way* linked list. Each node in such a list has two pointers, the usual pointer to the successor node (now called a forward pointer), and a second pointer. The second pointer, called a backward pointer, points to the node's predecessor. In our representations, it will be easier to make our nodes vertical, as in figure 15.7. You can traverse a two-way linked list forward or backward; you can reverse your direction of travel at any point.

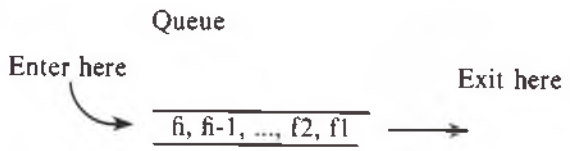


Figure 15.8 A normal queue.

```
QHEAD:   .WORD  x
QTAIL:   .WORD  y
```

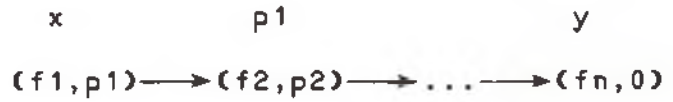


Figure 15.9 Priorities assigned.

The head and tail pointers are usually kept separately, as variables, so they can change at run time. One nice use of a linked list is to support a queue in which priorities apply. In a normal queue, a FIFO discipline applies. For instance, a printer spooler maintains a queue for the print output files f1, f2, . . ., fi, as shown in figure 15.8.

Suppose the queue entries are pointers to the print files (it is easier to move pointers than to move files). Suppose we also assign priorities to print files, based upon the urgency of the task. So instead of printing on a democratic basis of first-in, first-out, the file with the greatest urgency field "u" will go to the head of the queue. In the event of a tie, then FIFO will apply. This one-way linked list simulates a queue; entries (nodes) are always removed from the head, and new entries are always appended to the tail. Now let us add the second "value" field for the urgency factor "u":

$$(f1,u,p1) \rightarrow (f2,u,p2) \rightarrow \ldots \rightarrow (fn,u,0)$$

The queue manager, in this case the print spooler, would ensure that any new entry is placed in the queue as its urgency factor u dictates. It is an interesting problem to develop software that will prevent a nasty user from forging a u-field entry.

If one wishes to impose a fixed size on a queue, rather than let it grow arbitrarily large, it can be supported using an array instead of a linked list. By manipulating queue pointers we can avoid continually shifting array entries each time a new item arrives or an old item leaves. This mechanism supports a *circular queue,* as shown in figure 15.10.

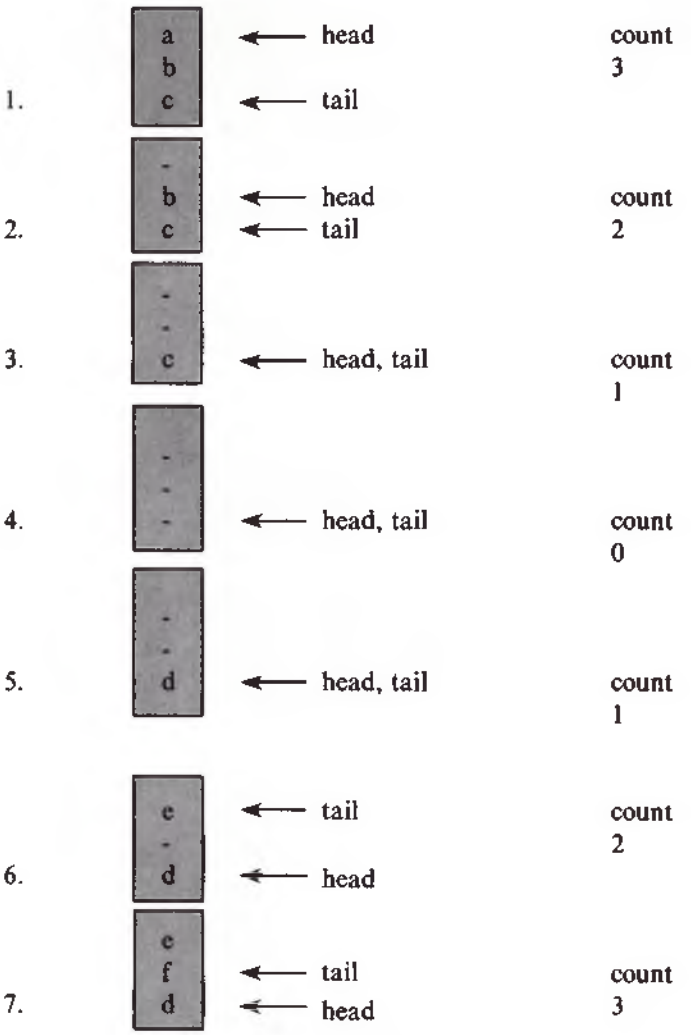


Figure 15.10 Circular queue.

In step 1 of figure 15.10 we have a fixed-size queue which is full. It has three entries, namely, a, b, and c. An attempt to place another item in this queue would be regarded as a queue overflow error. It is designed to hold no more than three items.

In step 2 we assume that the queue manager has removed the item a from the head of the queue. Item b thus automatically becomes the head of the queue (without moving b). We assume in step 3 that b is then removed by the manager (the manager can remove only a head item). That makes c the head item (step 3), until it finally is removed in step 4. At each step the queue manager updates the head and tail pointers, if required, as well as the queue item count.

When a new item arrives, as does d in step 5, it is placed where the head pointer designates, because the queue was empty. When e arrives in step 6, the queue "wraps-around" so that e can occupy the new tail position. So long as the queue manager is consistent about updating the head and tail pointers and the queue item count, this circular queue mechanism works very well.

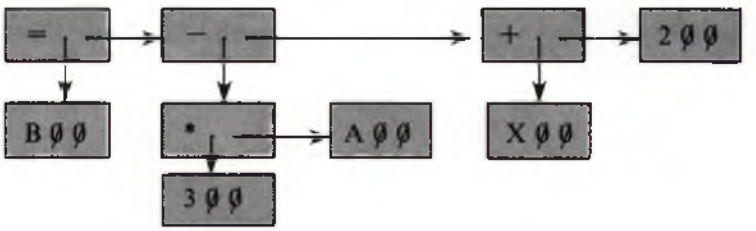


Figure 15.11 A two-dimensional linked list.

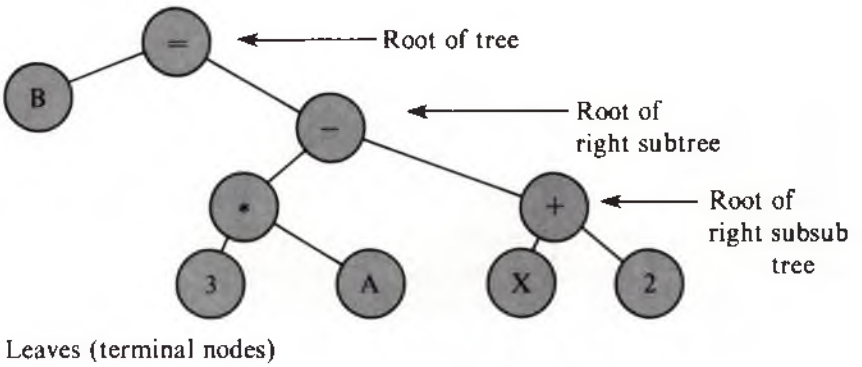


Figure 15.12 A binary tree.

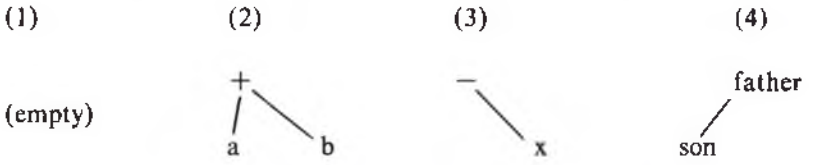


Figure 15.13 Some small binary trees.

Two-Dimensional Linked Lists

A two-way linked list is still a one-dimensional list; each node has at most one successor and one predecessor. A two-dimensional linked list is used to represent two-dimensional structures. When you look at a string such as B=3*A−(X+2), you think of it as a one-dimensional array of bytes. But if you had to interpret this string as an arithmetic statement in some high level language, with the meaning "Evaluate 3*A−(X+2) using the values currently assigned to the variables A and X, then assign this value to the variable B," you could visualize the structure as shown in figure 15.11.

We are using two pointers per node. What we have here is actually a linked-list representation of a data structure that computer scientists are particularly fond of. It is called a *binary tree*. The binary tree diagram for this arithmetic expression is shown in figure 15.12. A binary tree consists of a finite set of nodes. The set may be empty, or may consist of a *root node* and two disjoint binary trees stemming from the root node. Notice that this definition is recursive, since a binary tree is being defined in terms of binary trees. Examples of short binary trees are given in figure 15.13.

Recursion

Recursion plays an important role in mathematics. Its simplest application comes in its use for defining a function which is vital to probability theory and practice. This is the factorial function. Factorial has one argument; call it n. This argument must be a positive integer. The symbol n! is used to represent factorial(n). The definition for n! is

$$n! = n * (n-1)!$$
$$1! = 1$$

You can read this as: given some integer $n \neq 0$, factorial n is found by multiplying n by (factorial of $n-1$), and if n is 1, then the value of factorial n is also 1.

We say that an item is defined recursively if it is defined in terms of itself in a manner which does not lead to infinite regression. For a given integer $n>0$, we can guarantee that the sequence n, $n-1$, . . ., 2, 1 is finite, so factorial n can always be evaluated in a finite number of steps. We can compute n! by building the trace of the partial calculations:

$$n! = n * (n-1)!$$
$$(n-1)! = (n-1) * (n-2)!$$
$$(n-2)! = (n-2) * (n-3)!$$
$$\ldots$$
$$3! = 3 * 2!$$
$$2! = 2 * 1!$$
$$1! = 1 \text{ AHA}$$

Since we know 1! is 1, we can start reversing our path:

$$1! = 1$$
$$2! = 2*1$$
$$3! = 3*2$$
$$\ldots$$
$$n! = n * (n-1)!$$

Recursion appears frequently in computing. It is used

a) in defining some mathematical functions;
b) in writing recursive subroutines;
c) in processing recursively defined data structures;
d) in writing recursive definitions of macros.

Consider something as simple as an arithmetic expression AE, of the type

a+2, or (b−2)*(c+3), etc.

We can build up a formal definition for AE:

1. A *term* is either a variable or a constant or a parenthesized AE.
2. An AE is either a term or a signed term or an AE+term or an AE−term.

Leaving out the English, and using | to indicate alternates, we can write the AE definition as:

1. Term := variable | constant | (AE)
2. AE := term | +term | −term | AE+term | AE−term

This is clearly a recursive definition. You have implicitly been using such a definition each time you have evaluated (by hand) expressions such as

3 * (7 − 2)

As you process this in your mind, you are forced into "3 times whatever the (. . .) evaluates to." This is a recursive process.

Recursively Defined Macros

Recall the caution about the consequences of certain typographical errors when you are defining a macro. If, while defining the macro P, you accidentally wrote

```
.MACRO    P <argument list>
...
P         <operands>
...
.ENDM
```

and it had not been your intention to use P in the definition of P, the macro processor nonetheless imagines that you are defining P recursively, and at the point P is invoked elsewhere in your source module, a presumably infinite regression will be triggered, as the P expansion leads to a P expansion, which leads to yet another P expansion, ad infinitum, until the assembler blows its stack.

Figure 15.14 A recursively defined macro.

```
1                       .LIST ME
2                       .MACRO INT N    ; a recursively
3                         .IIF EQ N,.WORD N     ; defined
4                         .IF NE N              ; macro
5                               .WORD N
6                                INT N-1
7                         .ENDC
                        .ENDM
9 000000                    INT 3       ;      invocation
                        .IIF EQ 3,.WORD 3      ; expansion
                        .IF NE 3               ; begins
000000 000003                .WORD 3           ; first word
000002                        INT 3-1
                        .IIF EQ 3-1,.WORD 3-1
                        .IF NE 3-1
000002 000002                .WORD 3-1         ; second word
000004                        INT 3-1-1
                        .IIF EQ 3-1-1,.WORD 3-1-1
                        .IF NE 3-1-1
000004 000001                .WORD 3-1-1      ; third word
000006                        INT 3-1-1-1      ; last word
000006 000000           .IIF EQ 3-1-1-1,.WORD 3-1-1-1
                        .IF NE 3-1-1-1
                              .WORD 3-1-1-1
                               INT 3-1-1-1-1
                        .ENDC
                        .ENDC
                        .ENDC
                        .ENDC
```

You can write "legitimate" recursively defined macros. The program in figure 15.14 shows a clever way of generating .WORD n,n−1, . . .,1,0

Using Trees

Trees have a multitude of uses. One of them is to maintain items in a way which will facilitate rapid access to them. Normally you would simply use an array, and once all the items were present, you would sort the array. Given a sorted array, say of size 2^n, then one can guarantee that a lookup can be performed in n or fewer probes by using a *binary search*. In a binary search, given a key k, which you wish either to find in the array or establish is not present in the array, you determine which half of the array k could possibly be in by looking at its middle item. You keep repeating that process until either you find a match for k or you reduce the array size down to 1. Given a sorted array of size 2^n, successive halvings can occur only n times before you have nothing left.

However, if you have a situation in which the collection of items is not static, but grows and shrinks dynamically at run time, using an array or a one-way linked list would not be very efficient. Consider the following approach.

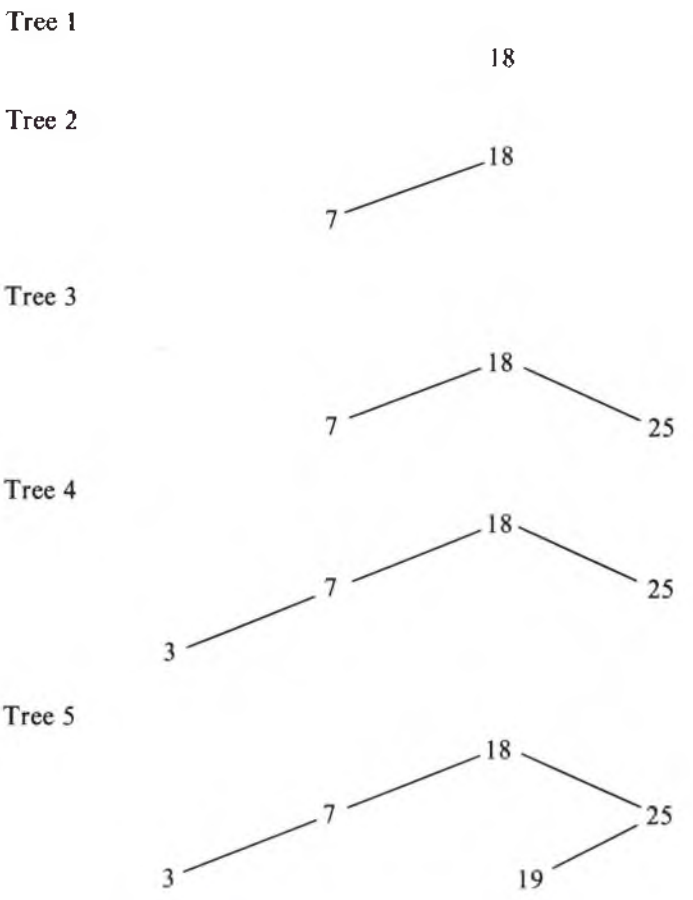


Figure 15.15 Growing A in-order binary tree.

Suppose, as items arrive in a random sequence, we construct a tree such that all nodes in each left-subtree have values smaller than all values in all right-subtrees. Given the sequence 18, 7, 25, 3, and 19, we would want to construct a tree beginning with the first value (18) and add nodes as each of the following items are encountered. The growth of our tree is shown in the series of snapshots in figure 15.15. As usual, our computer trees grow upside down. If we built this tree at assembly time, we would have written

```
ROOT:     .WORD   18,1$,2$ ; value,l-ptr, r-ptr
1$:       .WORD   7,3$,0
3$:       .WORD   3,0,0
2$:       .WORD   25,5$,0
5$:       .WORD   19,0,0
```

Once such a tree is constructed, any new item will be appended to the tree to maintain this "already sorted" property. Any lookup is rapid because of the tree's structure and definition. At best, it compares with a binary search. At worst, it degenerates to a sequential search. A final printout, when a tree stops changing, can produce the sorted sequence

3, 7, 18, 19, 25

by using a recursively defined subroutine based upon the following recursively defined algorithm for symmetric (also known as in-order) tree traversal.

1. Traverse the left subtree and display the node values.
2. Display the root value.
3. Traverse the right subtree and display the node values.

Another Example of Recursion

Horizontal tabs are represented by an ASCII tab code (control I). Some CRTs and printers will respond to the receipt of a tab code by spacing over to the next tab stop. Other CRTs and printers will simply replace the tab code by a space code—in effect, ignoring the tab request. The first group of terminals are said to have hardware-supported tabs. In such a case it is common to have the tab stop settings wired to columns 9, 17, and every eighth column thereafter. Some very smart terminals have soft tabs, which you may set as desired if you do not like the default setting. How does a multi-user system handle a variety of terminals which have many different characteristics? It should accept from each user a description of the user's terminal. This is often as simple as saying, "I have a model 43 Teletype," or "This is a VT100." Otherwise you have to spell out parameters such as your line length, screen capacity, type of tab support, etc. How would an operating system support tabs for terminals which ignore tabs?

The common practice has the terminal driver in the operating system replace an outgoing tab code by an appropriate number of spaces, if the terminal has no hardware-supported tabs. This could be done iteratively. It can also be done recursively, as shown here. Suppose the tab stops are set at columns 9, 17, . . ., $8n+1$, ... If the string THIS>IS>A>TEST were being sent to a CRT, then each ">" (the symbol I chose to make the tab code visible to you) would be replaced by an appropriate number of blanks, leading to

```
1       9       17      25      33  Columns
THIS    IS      A       TEST
```

The first tab will be replaced by four spaces, the next by six, and so on.

We can outline how the software driver sends output to a terminal:

```
...
Get next character.
    None left: take care of end-of-line processing.
Examine character.
    Not a tab, so emit it and
        increment the character count.
        Proceed to get-next-character.
    Found a tab code: invoke the tab procedure.
        Proceed to get-next-character.

Tab procedure. Are we at a tab stop?
        If so, exit.
        If not, emit a space code, up the character count, and
            invoke the tab procedure.
```

Once the nature of the work is clearly understood, and it can be described in a few lines of compressed English, actually writing the code is straightforward:

```
TAB:    CMP     %3,CHARCNT
        BNE     1$
        RTS     PC
;
1$:     MOVB    #' ,(%2)+ ; emit space
        INC     CHARCNT
        JSR     PC,TAB ; recursive call
        RTS     PC
```

The main line routine uses %2 to point to the output buffer string it is creating. It also manages %3, which specifies where the current tab stop column position should be.

Cross Assembly

It is a common practice to prepare, assemble, and test programs on a host computer with the intention of executing the object modules on some other computer, called the target computer, as depicted in figure 15.16.

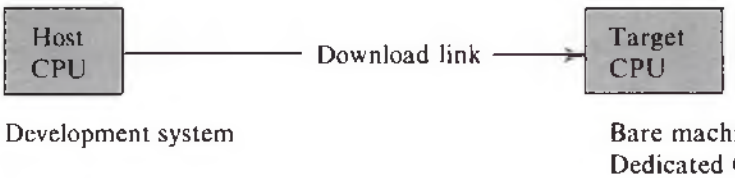


Figure 15.16 Development and target CPUs.

It is a straightforward operation if both CPUs execute the same instruction set. The download link is used to transfer over to the target CPU the bytes of the executable program generated on the host CPU. The link can be as simple as a pair of serial interfaces, with the host's software treating the target CPU as if it were merely an intelligent terminal. Sometimes the object modules are transferred via some common medium of removable storage, such as a floppy disk. In a mass production situation (e.g., the manufacture of hundreds of thousands of microcomputer-controlled carburetors for automobiles, the object modules prepared on a development system are reproduced in read-only-memory ROM chips, which are then plugged into the carburetor circuit board).

What if the target CPU is not compatible with the host CPU? Suppose the host is a PDP-11/34 and the target is an Intel 8080 microcomputer. You could write an 8080 assembler which accepts 8080 symbolic programs and produces 8080 object modules, but which itself (the 8080 assembler) executes on the PDP-11. This kind of program is called a cross-assembler. If the CPUs involved have sufficiently many similarities, you might avoid writing a whole assembler by using the host's own assembler very carefully.

The 8080 is a byte-addressed CPU which uses 8-bit words (here a byte and a word have the same meaning) and 16-bit addresses. An 8080 ADD instruction has the symbolic form

```
ADD REG
```

The particular use

```
ADD R1
```

should generate the octal machine representation 201 as a single-word instruction. The 8080 ADD executes an "add register to accumulator" sequence. Using MACRO-11, we can define macros which "pretend" to be 8080 instructions.

```
.MACRO  ADD REG
.BYTE   200+REG
.ENDM
```

However, since these definitions might be used by other people in other contexts, it is wiser to build in the implied octal numeric radix:

```
.MACRO  ADD REG
.BYTE   ^O200+REG
.ENDM
```

Can we build in any assembly-time syntax checking? MACRO-11 provides a simple assembly-time message capability which can be used for this purpose. The capability is provided by the .ERROR directive, which is used in the following examples.

```
.MACRO   ADD REG
ER       REG ; check ``REG''
.BYTE    ^0200+REG
.ENDM
```

where

```
.MACRO   ER A
2$ = 2
  .IIF GE,A 2$ = 2$-1
  .IIF LE,<A-^07> 2$ = 2$-1
  .IIF  NE,2$,.ERROR; illegal reg spec
.ENDM
```

An invocation ADD 15 leads to

```
2$ = 2
  .IIF GE,15 2$ = 2$-1; so 2$ = 1
  .IIF NE,1,.ERROR ; illegal reg spec
```

The .ERROR is thus effective at assembly time because the .IIF which controls it has the .IIF condition satisfied. When a .ERROR directive is processed, its comment field is printed at that instant, at assembly time. So if you had the default .NLIST ME in effect, using an invalid operand with the ADD would trigger this syntax error message in spite of the .NLIST. It will also raise a P error flag.

A correct use of this ADD means using the symbols R1, R2, . . ., R8, with each of these being defined: R1=1, R2=2, . . ., R8=8. Then the ER expansion would not trigger the .ERROR message. Consider an ADD R2; its expansion is

```
2$ = 2
.IIF GE,2 2$ = 1
.IIF LE,<2-7> 2$ = 0
```

which causes the .ERROR line to be skipped. An 8080 jump-if-zero instruction has the symbolic form

```
JZ       ADR ; 16 bit adr
```

We can define a JZ macro using the assembly-time arithmetic and logical operators "/" and "&":

```
.MACRO   JZ ADR
.BYTE    ^0312      ; opcode
.BYTE    ^0377&ADR ; & means AND
.BYTE    ADR/^0400
.ENDM
```

Given the invocation JZ 2004, the generated code is

```
.BYTE   312
.BYTE   004 ; low address byte
.BYTE   004 ; high address byte
```

We have previously used "+" and "−" in MACRO-11 expressions such as

```
.WORD   A+2,B-1
```

Assembly-time multiplication and division can be requested by "*" and "/". Similarly, the logical AND and OR can be evoked by "&" and "!". All these symbols, called operators, expect two operands. For this reason they are also called binary operators; this has nothing to do with their operands being binary numbers.

In ordinary arithmetic,

$$2 + 3 * 4 \rightarrow 2 + 12 \rightarrow 14 \text{ (decimal)}$$

In MACRO-11 arithmetic, all the binary operators have the same precedence, and the evaluation is strictly from left to right, unless you use braces as parentheses. Thus, MACRO-11 evaluates the preceding expression as

$$2 + 3 * 4 \rightarrow 5 * 4 \rightarrow 20 \text{ (if in decimal)}$$

Thus, we have

```
.WORD   3+4*5    ; 43 octal
.WORD   3+<4*5>  ; 27 octal
```

It turns out that it is possible to define MACRO-11 macros for the 8080 instruction set so that you can assemble 8080 symbolic programs with the PDP-11's assembler. You would say that MACRO-11 and a suitable collection of macro definitions serve as an 8080 cross-assembler. It would be very difficult doing this with the assembler for a computer such as the UNIVAC 1100. It has very little in common with the 8080, and its macro-processing capability does not lend itself to supporting cross-assemblers very easily. Many cross-assemblers are, in fact, written in a portable subset of FORTRAN.

B and NB

There is another pair of conditions used with the .IF and .IIF. The conditions B and NB examine a given macro argument to see if it is blank (B) or not blank (NB) (missing). These tests may be used *only* within a macro definition. Consider the invocations

```
TTYIN         ; input a byte to %0

TTYIN   (%1)  ; input a byte to (%1)
```

We could have the first invocation expand as

```
EMT     ^O340
BCS     .-2
```

and the second expand to

```
EMT     ^O340
BCS     .-2
MOVB    %0,(%1)
```

if we had defined TTYIN using

```
.MACRO  TTYIN ARG
EMT     ^O340
BCS     .-2
    .IF NB<ARG>
      MOVB  %0,ARG
    .ENDC
.ENDM
```

In this case,

```
.IIF NB<ARG>,MOVB %0,ARG
```

could replace the three-line .IF block.

Concatenation

We saw the apostrophe (single quote) ' used earlier to specify a single immediate ASCII character as an operand, as in:

```
MOV     #'A,%2 ; code for ``A'' → %2
```

The same ' character can play a completely different role when used in a special way within a macro definition. When the apostrophe immediately precedes or immediately follows a dummy argument in the body of a macro definition, it functions as a *concatenation operator*. In this role it serves to initially separate (i.e., delimit) its left or right neighbor from the argument it adjoins. When the macro is invoked, the ' is removed during its expansion, and the current value of the adjoining argument is filled in, as if the ' had not been there at all. Suppose we had:

```
        .MACRO  EX A
L'A:    .WORD   A'M
        .ENDM
```

The invocation EX BBB generates the expansion

```
LBBB:   .WORD   BBBM
```

Since any field of a MACRO-11 statement can be a candidate for specification in a macro argument, we can even have a macro which is used to build directives, if we choose, as in:

```
.MACRO  BLK  ARG,CNT
.BLK'ARG     CNT
.ENDM
```

Thus, the invocations

```
X:      BLK     B 5        and Y:  BLK  W 6
```

expand to

```
X:      .BLKB 5            and Y:.BLKW 6
```

respectively.

If an apostrophe is used in a macro, and it is not immediately preceded or followed by a dummy argument, it will merely be copied as the macro is being expanded. So you can still use immediate operands, as in:

```
.MACRO  SPACE,R
MOVB    #' ,%'R
.ENDM
```

which, given the invocation SPACE 1, generates

```
MOVB    #' ,%1
```

Here the first apostrophe was used to introduce an immediate ASCII code as desired, and the second apostrophe was removed as the argument value 1 was concatenated to the %, yielding %1.

Threaded Code

In some applications such heavy use is made of subroutines that the main program appears to be nothing but one subroutine call after another. This is the case, for instance, when you are doing all your arithmetic in multiple-precision. You might have long sequences of code such as:

```
JSR R5,TADD ; triple-precision ADD
.WORD A,B,C ; (A,A+2,A+4)+(B,B+2,B+4) →
              C, C+2, C+4
JSR R5,TSUB ; triple-precision SUB
.WORD B,X,Y
JSR R5,TMUL ; triple-precision MUL
.WORD Y,C,M
...
```

In an attempt to reduce the memory requirements for such code, you might consider revising each of the subroutines slightly and rewriting the invocations so that the JSR R5 is implied. Then the previous sequence becomes:

```
P1:     .WORD TADDX ; triple-precision ADD
        .WORD A,B,C
        .WORD TSUBX ; triple-precision SUB
        .WORD B,X,Y
        .WORD TMULX ; triple-precision MUL
        .WORD Y,C,M
```

The subroutine names have been changed slightly to remind you that this technique does require some changes to the subroutines. The "code" at P1 does not have any executable instructions. You cause it to perform the desired function by using a small initiating program, such as:

```
SUPER:  MOV     #P1,%0 ; use %0 as pointer
        JMP     @(%0)+
```

This brief segment manages to invoke TADDX while leaving %0 set at P1+2, where TADDX's first argument can be found. If we can arrange to have the subroutines use %0 as their argument pointers, then the only extra feature in them involves using JMP @(%0)+ instead of the usual RTS to return control. In effect, each subroutine finishes its work by calling the subroutine you wanted called. The compact representation of the code beginning at P1 is called a *thread*. The main parts of the subroutines which work with this thread are shown here:

```
SUPER:  MOV     #P1,%0 ; set thread pointer
        JMP     @(%0)+
;
TADDX:  MOV     @(%0)+,%1 ; get 1st arg
        MOV     @(%0)+,%2 ; get 2nd arg
        ...perform triple-precision add...
        MOV     @(%0)+,%3 ; get 3rd arg
        ...use %3 to store result...
        JMP     @(%0)+ ; return
;
SUBX:   MOV     @(%0)+,%1 ; get 1st arg
        ...etc...
        JMP     @(%0)+ ; return
;
        ...similarly for other subroutines...
```

Register 0 is initialized so that it points to the thread we intend to interpret. The first entry in the thread is a pointer to the first subroutine we plan to invoke. From then on %0 is either pointing to an argument or to an entry point. The only care we need take is to ensure that our thread ends with a pointer to return control to the main program. This is done very simply:

```
; main program
        ...perform I/O etc....
SUPER:  MOV     #THREAD,%0
        JMP     @(%0)+
;
RESUME:
        ...more I/O etc...
        .EXIT
;
THREAD: .WORD TADDX
        .WORD A,B,C
        ...
        .WORD RESUME ; leave the thread
```

The threads can be made a little more presentable if the .WORD directive is not written. When MACRO-11 scans a statement which does not have an instruction mnemonic or a directive in it, it assumes that a .WORD was desired. So we can now rewrite the preceding thread as:

```
P1:     TADDX
        A,B,C
        ...
        TMUL
        Y,C,M
        RESUME
```

or in the following more readable form:

```
P1:     TADDX
                A,B,C
        ...
        TMUL
                Y,C,M
        RESUME
```

The basic idea behind threaded code can be extended a little further. It leads to the notion of nothing less than knotted code; see exercise 15.30.

The original motivation for threaded code was to reduce the amount of memory used. Savings, if any, on subroutine linkage overhead were probably an unexpected benefit. Due to the way some computers are now constructed (including some models of the PDP-11), significant speed advantages can be obtained by use of the denser code (more effective-instructions per unit of memory) which threaded code typifies. The particular hardware feature which provides this speedup is called cache memory, which is discussed later in this chapter.

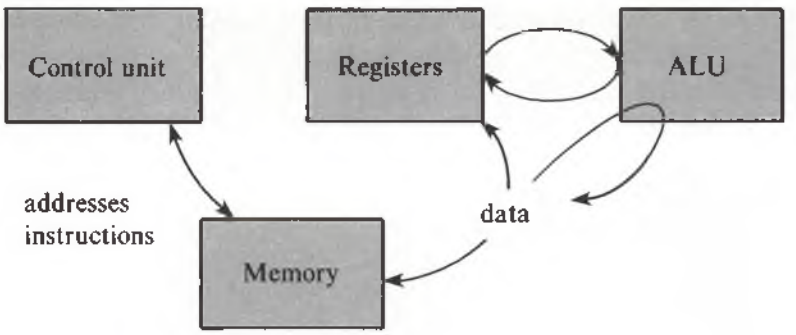


Figure 15.17 Typical computer organization.

Speeding Up Programs

The typical relationship of the ALU, the registers, the control unit, and the memory is shown in figure 15.17.

It is relatively easy to predict what the relative execution times of two versions of a segment of code will be. Prediction gets more complicated if a memory management unit is involved, and if you have to take the I/O into consideration. As a general rule, you can tell what should be done to speed up a program segment. Consider the loop

```
        MOV     #3,%0 ; count
        MOV     #A,%1 ; base address
1$:     CLR     (%1)+
        SOB     %0,1$
```

If we write out each instruction which is executed, we get

```
MOV     #3,%0
MOV     #A,%1
CLR     (%1)+   ; 0 → A
SOB     %0,1$
CLR     (%1)+   ; 0 → A+2
SOB     %0,1$
CLR     (%1)+   ; 0 → A+4
SOB     %0,1$
```

Obviously it would have been better to have written

```
CLR     A
CLR     A+2
CLR     A+4
```

Only three instructions, taking up only six words, is faster than a set of four instructions that take up six words which lead to executing eight instructions.

By the same argument, which of the following is it better to write?

```
INCB    B
INCB    B+1
...
INCB    B+998.
INCB    B+999.
```

instead of

```
        MOV     #1000.,%0
        MOV     #B,%1
2$:     INCB    (%1)+
        SOB     %0,2$
```

The first set of 1,000 instructions uses up 2,000 words of memory (we could do it with only 1,001 words). The second technique uses only seven words but causes 2,002 instructions to be executed. If you count the number of memory references each technique requires, they come out even at some 4,000 memory references. So what might appear to be faster may not be; and in view of its enormously larger memory requirements, the "written-out" form here is not recommended. Careful analysis is required, for this may not always be the case. If the loop had used an indexed address—say, INCB B(%3)—then the execution time advantage would fall to the written-out version.

What about the overhead introduced by subroutine linkage? Is it better to write slightly longer programs than to pay the price in extra execution time when using subroutines? Consider

```
        MOV     #500.,R4
10$:    JSR     R5,SUB1
        .WORD   A,B,C
        JSR     R5,SUB2
        .WORD   C,P,Q
        JSR     R5,SUB1
        .WORD   Q,B,A
        SOB     R4,10$
```

Would it be better to write out the body of the code for each of subroutines SUB1 and SUB2 as in-line code, to avoid the 500*(3 JSR + 3 RTS execution times)? The saving can be even greater, as the argument passing itself involves some sizable overhead in memory references. Whether it might be wise to eliminate some subroutine calls and use in-line code instead would depend on the memory that is available and the response-time requirements of the given application. (This, by the way, is greatly facilitated if you define macros for the in-line code; then you at least get the cosmetic effect of having almost used subroutines.) You should also consider whether using threaded code would be profitable.

However, there is a way of building computers that invalidates much of what we have just discussed! Which brings us to how people design computers to go faster.

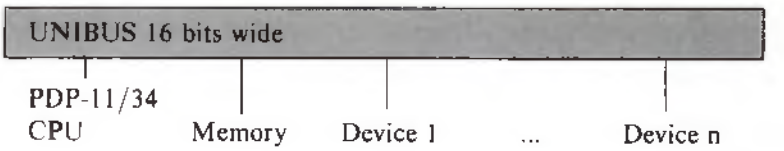


Figure 15.18 PDP-11/34 bus organization.

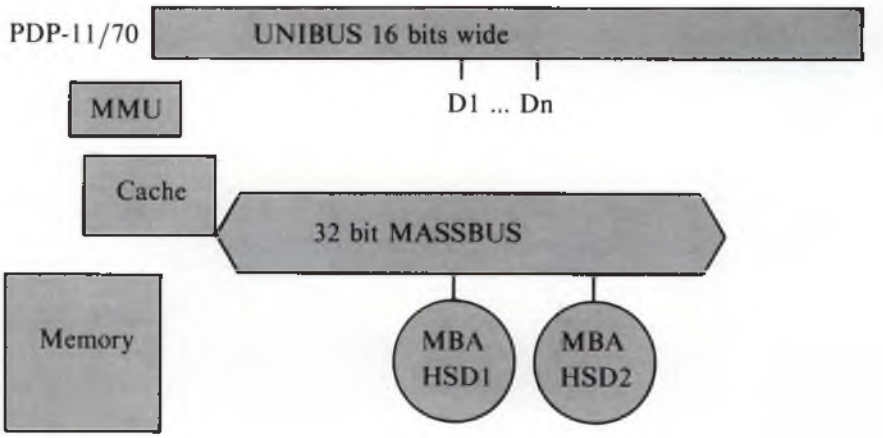


Figure 15.19 PDP-11/70 multiple bus organization.

Faster Buses and Cache Memory

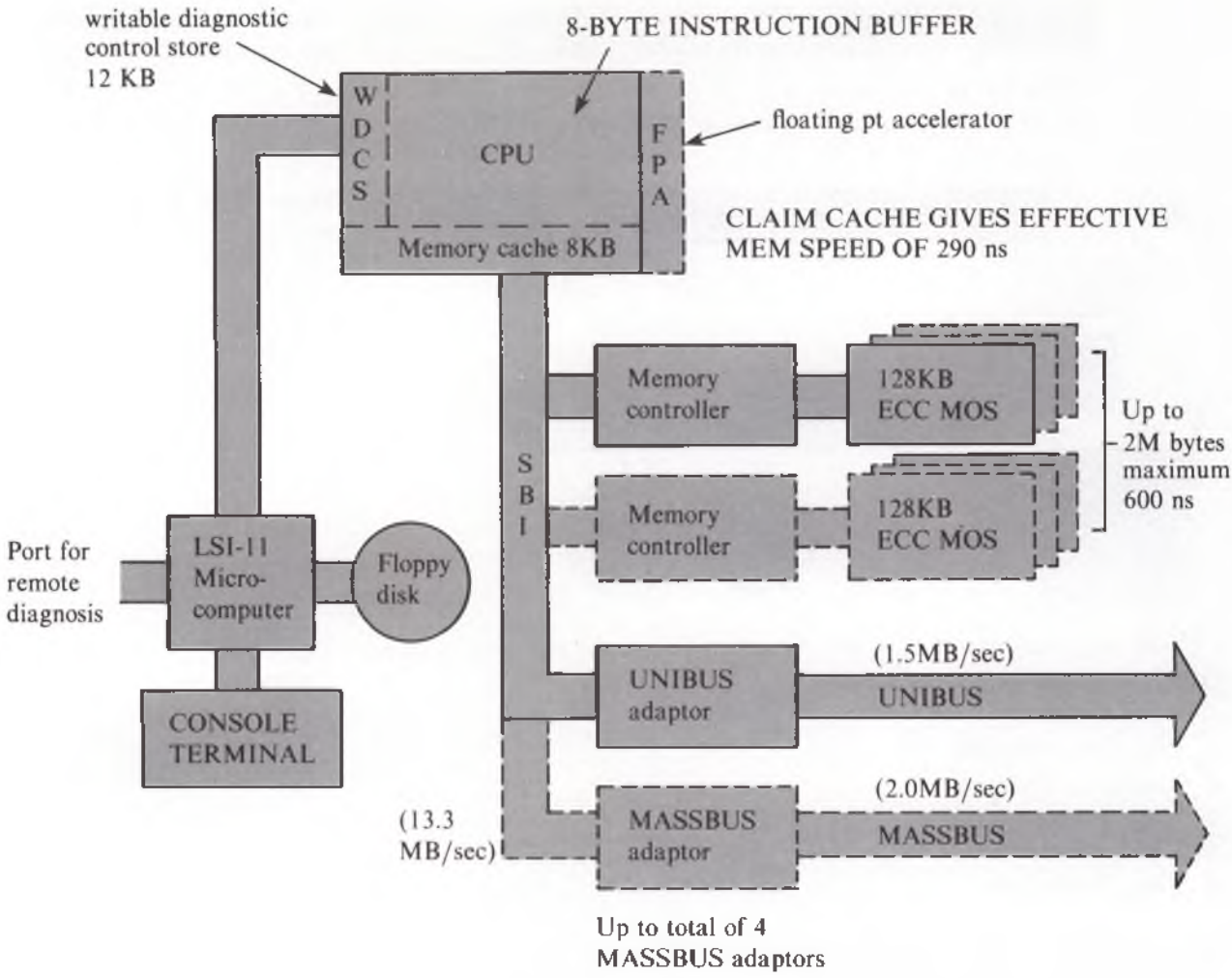
If you are a computer designer, how can you make the computer go faster? There are many possibilities.

1. Use more registers.
2. Use faster memory.
3. Use a larger word size.
4. Use a wider bus.
5. Use a faster bus.

What if you are told, "Make the computer go faster, but do not force users to make any changes to their existing programs in order to get the benefit of this speedup." In other words, your hardware enhancement has to be *transparent* to the user. Users will continue running their old programs, unchanged, but they will execute more rapidly. The transparency requirement eliminates adding more registers or using a larger word size. Hardware enhancements or new computer designs which allow you to continue running your old programs, but with some changes to get the speedup benefit, are said to be *upward compatible*. This is the sense in which the DEC VAX-11/780 and the PDP-11s are related; the one is upward compatible with the other, to a large extent.

Using a wider bus or a faster bus does not violate the transparency constraint. That, in fact, is one of the differences between the slower Q-bus-based LSI-11 and the faster UNIBUS-based PDP-11. The former uses a bus which is both narrower and slower. The bus differences are, however, transparent to the user.

The various models of the PDP-11 differ in having some additional faster buses. The PDP-11/34 uses the UNIBUS as its only bus, as shown in figure 15.18. The PDP-11/70, designed for higher performance, at a higher cost, introduces a faster bus called the MASSBUS, in addition to still using the UNIBUS, as shown in figure 15.19. The UNIBUS is still used by the relatively slow UNIBUS devices D1, . . ., Dn. The MASSBUS is a high speed bus that can transfer 32 bits at a time (as opposed to 16 bits at a time for the UNIBUS) at the rate of 4 MB/sec. High speed devices HSD1, HSD2 connect through MASSBUS adaptors (MBA) onto the MASSBUS.



FPA = floating point accelerator
WDCS = writable diagnostic control store

Figure 15.20 VAX-11/780 multiple bus organization.

What we see happening to the PDP-11 as it has evolved over a ten-year period is quite interesting. The simple single-bus concept has evolved into a multiple-bus organization, with specialization of the bus functions (e.g., a high speed MASSBUS, a medium speed UNIBUS). The idea is carried one step further with the VAX-11/780, which has three kinds of buses. The VAX-11/780 continues to use the UNIBUS and the MASSBUS, those found on the PDP-11/70; in addition (actually, as its starting point), it has a third very high speed (13.3 MB/sec) bus called the system backplane interconnect (SBI), as shown in figure 15.20.

We seem to be coming full circle back to the classical channel-organized computers that were dominant at the beginning of the PDP-11's history.

Cache Memory

There is one little box in the diagram for the PDP-11/70 (and also for the VAX) which merits attention. The *cache* is a very high speed memory through which all CPU stores and fetches to main memory must flow. It seems as though we have just introduced a bottleneck. How can a cache help speed CPU execution?

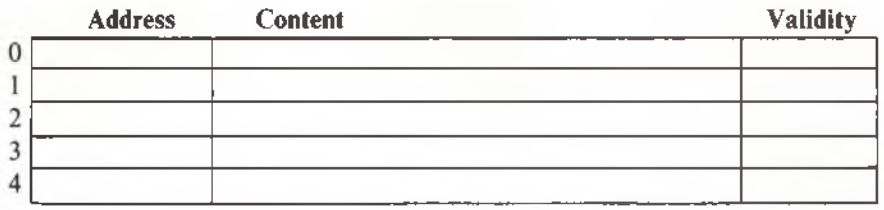


Figure 15.21 Five-word cache memory.

Suppose the cache is a mere five-position memory (five "long" words), as shown in figure 15.21. The cache can be regarded as having three fields in each of its words. When the CPU is powered-on (or a RESET is executed in privileged mode), the validity bit in each row is cleared. This means that the information in the content field is to be disregarded. It also means that, in effect, the row is unoccupied. Whenever the CPU initiates a memory reference, the memory address will be sent to the cache controller. The address will then be compared with all the addresses in the cache's address field which have their row's validity bit set. If no match is found, this is called a cache *miss*, and the address is sent on to the memory unit. When the memory unit responds—say, to a fetch request—the data and the address will be copied into an available row of the cache, and the corresponding validity bit will be set. Of course, the fetched data will also proceed onto the CPU.

A subsequent use by the CPU of that address should find it in the cache, in which case the fetch will copy the content field from the cache row with the matching address field, provided the validity bit is set. When the CPU finds what it needs in the cache, that is called a cache *hit*. The hit ratio measures the percentage of the time the CPU expects to find what it needs sitting in the cache. Every hit means that the CPU can avoid waiting for the slower memory.

When the CPU is storing something in memory, it is only copied into the cache if a hit occurred. A store operation forces a cache "write-through" in order to keep the memory and the cache consistent. So stores force memory references to take place even in the event of a cache hit, whereas memory fetches will force memory references only if a cache miss occurs. This provides a great deal of incentive for you to write non-self-modifying read-only code.

Sooner or later the cache will fill up. The cache controller has built into it a *cache replacement policy* algorithm. It decides which item in the cache should be overwritten to make way for the latest incoming item. The item which is being discarded has previously either come from memory, or already been written into memory, thanks to the write-through policy.

The PDP-11/70 cache has a 1 KW capacity (i.e., 2 KB of user data). When data are copied into the cache, two words are fetched, in the hope that if the CPU wants the word at location n now, then it will very likely want the word at location n+2 next.

On the PDP-11/70, a cache miss costs you 1.02 usec in extra delay because of the memory reference. A cache hit takes only 0.30 usec. So, effective use of a cache gives you the appearance of using a main memory that is almost four times faster (4.4 times, to be exact).

The PDP-11/70 designers state that a typical program's execution involves 90 percent read references (fetches) and only 10 percent write (store) references. For such programs, they have observed hit ratios with averages in the 80–95 percent range. So a little fast memory can make a big difference, when it is used as a cache. Why not make the cache even larger and get better hit ratios? One of the factors that makes the cache so fast is that it performs an address match operation in a single cache-memory cycle. The desired address is simultaneously compared with all the cache addresses! That takes a lot of hardware—very expensive hardware. Memories with this property are called *associative* or content-addressable memories. These high costs preclude making cache memories too large. In any event, doubling the cache size is not likely to double the hit ratio.

The use of a cache is transparent to the user. No program changes are required. In fact, most caches have a cache-disable bit or switch, which in effect removes the cache from the CPU-memory data flow path. Programs will continue to execute with or without the cache. Presumably, you would notice the difference because the CPU would run faster one way than the other.

Cache is a standard feature on some PDP-11s (e.g., PDP-11/70), an option on others (e.g., PDP-11/34), and not available for some models (e.g., LSI-11). As you would expect, the higher-performance models have it, the in-between models can have it, and the single-user systems don't need it.

On-Line, Real-Time Computing

We used the phrase "on-line" when we were discussing spooling. The card reader and the printer were attached to the CPU, making them on-line devices. The phrase on-line is also used to describe what are otherwise known as multi-user, or time-sharing, systems. In data processing circles, the phrase "on-line system" is used to describe what we call interactive systems as opposed to batch systems. The key distinction is based on where the data preparation takes place: on-line, or off-line (e.g., perhaps on a key-to-disk system).

Real-time computing has two very different meanings. In scientific and engineering applications, when an on-line device requests service, it may require real-time response from the CPU. A computer used in an airborne navigation system, one used to control a carburetor, one used to monitor the valves in a nuclear power plant, etc. are all being used for on-line real-time computing.

In data processing, an on-line real-time system has a quite different connotation. No special equipment is involved, and the real-time response requirements tend to be specified in seconds rather than in micro- or milliseconds. Consider a banking system. On-line terminals may be used to record customer transactions as they occur. That makes this system an on-line system. The key question is: "Are all relevant files and balances immediately updated?" Until a few years ago it was the practice not to compute new account balances until the close of the banking day (which is why banks used to close at 3 p.m.). The transactions recorded during the day were totaled in the late afternoon, in a batch, so that all files and totals would be current for the opening of the next banking day.

In contrast, a bank considers its computer system to be "real-time" if all the implications of a given transaction are computed when the transaction is recorded. In this sense, an airline reservation system is inherently a real-time system.

An interesting anomaly occurs in the case of scientific real-time systems. You might believe that the fastest computer you can find will be better for such real-time work, when a millisecond delay can be critical. Suppose four scientists, each having his own laboratory, were to consider pooling their resources to begin using computers in their laboratories. They presumably had been using some computing service to assist in the analysis of their experimental results. They now want to take the next step, which means involving the computer in the actual data acquisition process while their experiments are taking place. Would it be better for these four scientists to go off and each buy one smaller computer, so that each laboratory would have its own computer, or should they pool their money and buy a bigger, faster computer which they can share? Presumably the faster computer will be better for real-time data acquisition and experimental control.

Let us conjecture that each scientist could buy a modestly configured PDP-11/34, or they could jointly buy and share a much more powerful PDP-11/70. Which is the better choice? We know that the 11/70 is much faster than an 11/34. For the sake of argument, let us assume that it is four times faster. Which will give the better real-time response: a dedicated 11/34 or a shared 11/70? The 11/70 derives much of its performance from the effective use of its cache memory. If the cache memory hit ratio cannot be maintained at a high level, its performance can degrade by a factor of four. Because an 11/70 is shared, when a real-time device suddenly needs service, the entire content of the cache may be irrelevant because some other task was in progress. Having to update the whole cache, because of a 100 percent miss rate, means the CPU is operating at main memory speed, not at cache memory speed.

Bigger or faster is not necessarily better. Wiring laboratories several floors apart so that they may share one CPU may not be as cost-effective as providing each laboratory with its dedicated computer. Connecting high speed devices to a computer becomes costly if distances much over ten meters are involved. There are many issues involved in the choice between multiple dedicated CPUs and a single shared CPU. Fortunately, it is now possible to have the best of both worlds by setting up a small network of computers, as depicted in figure 15.22.

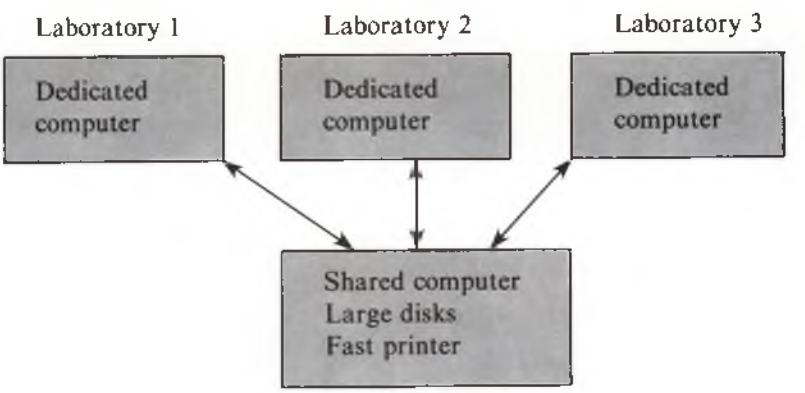


Figure 15.22 A small network of computers.

It is not unusual to find several dedicated computers in a single laboratory, each one devoted to the real-time monitoring of some apparatus. Adequate disk or tape storage is provided on each one of these. They, in turn, can rely on the larger shared system for non-time-critical transfer of experimental data into the shared system's much larger file system. This mode of resource sharing is now fairly common.

Reentrant Code

Reentrant code is the description attached to programs which may be time-shared by many users who are sharing a single executing copy of the code. The simplest way to write a reentrant program is to ensure that each independent user of it provides his own data work space (uninitialized memory), so, in consequence, no part of the reentrant program is ever modified. The instruction part of a reentrant program is thus "pure code," a nice name for non-self-modifying programs.

Reentrancy is one of those wonderful ideas that are extremely important to two classes of users with almost nothing in common. Reentrancy is very important on large systems, and just as important on very small systems for quite different reasons.

Reentrancy on Large Systems

It almost goes without saying that a large system, by virtue of its high cost, must be a multi-user system. Most programs which run on a multi-user system are (or should be) reentrant. The editing program, the FORTRAN compiler, the BASIC interpreter, etc. should all be reentrant. This was not the case a few years ago, and it had interesting and costly consequences. When you configure a computer system, you specify how much memory you will install, how many disks of what type, and what other options and peripheral devices you wish to purchase. In order to do this you have to have some idea of how the computer will be used. So you describe a hypothetical day in the life of the proposed computer. The scenario you outline represents your best guess as to what constitutes a typical work mix under reasonably heavy load conditions. It might be as shown in figure 15.23. Here a typical snapshot finds ten users involved in the indicated activities. We assume that using the editor Edit takes 25 KB of memory, including the user's data space. Similarly, running the FORTRAN and COBOL compilers uses more memory. Finally, we expect some user-written programs (written in FORTRAN or COBOL) will sometimes be executed. So it takes 360 KB to support this job mix.

Figure 15.23 Memory use without reentrant code.

User	Activity	Memory (KB)
1.	Edit	25
2.	Edit	25
3.	Edit	25
4.	Edit	25
5.	FORTRAN	40
6.	FORTRAN	40
7.	COBOL	50
8.	COBOL	50
9.	User	40
10.	User	40
	Total	360

The operating system itself needs between 10–20 percent more memory, so you may end up buying 400–500 KB of memory.

If the editor and the compilers were reentrant, then you could either get by with much less memory or support many more concurrent users. Consider what reentrancy could do, as depicted in figure 15.24.

The first user of a reentrant program (e.g., user 1, using Edit) must provide memory for one copy of the edit program (here, 25 KB). Additional users of the same program need provide only their own workspaces, which for the Editor we assumed was 5 KB per user. The first user's 5 KB was included in the 25 KB. In this example we reduced our memory needs by 36 percent (not counting operating system needs). A few years ago memory was very expensive, and this would have been a significant cost reduction. Nowadays we would use the reduced per/user memory requirements due to reentrancy as an opportunity to support more users.

Reentrancy should not be confused with recursion. The two concepts are independent of each other. Consequently, they can be used in the same program, provided that each user supplies the reentrant program with its own private stack. Whereas it is hard to imagine supporting recursion without a stack, reentrancy is definitely not stack-oriented. Suppose we had three users sharing a reentrant editor, as in figure 15.25. Here each user provides an adequate workspace (WS) so they can share the reentrant editor. The Editor maintains a table of pointers to the user workspaces. The processing sequence among users 1, 2, and 3 is entirely user-driven. It is not FIFO, nor is it LIFO. So it would not be appropriate for the reentrant program to store user-related pointers in either a queue or a stack. A direct-access structure is needed: here an array would be entirely suitable.

User	Activity	Memory (KB)
1.	Edit	25
2.	Edit	5
3.	Edit	5
4.	Edit	5
5.	FORTRAN	40
6.	FORTRAN	10
7.	COBOL	50
8.	COBOL	10
9.	User	40
10.	User	40
	Total	230

Figure 15.24 Memory use with reentrant code.

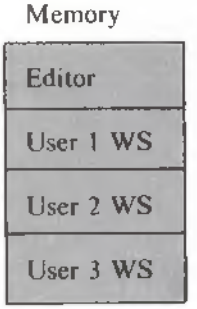


Figure 15.25 Users sharing a reentrant editor.

Reentrancy on a Dedicated Computer

A dedicated computer is a single-user computer which always has the same user, and the user tends to be a thing rather than a person. The microcomputer in a microwave oven is a very dedicated computer. How can a concept such as reentrancy, which is clearly advantageous on multi-user systems, have any value on very small single-user systems? Since there will never be any sharing of programs, what is the point of making them reentrant on a dedicated computer?

A dedicated computer might be used in a very hostile environment. The microcomputer controlling a carburetor is subject to temperature extremes, vibration, and dirt that would destroy delicate computer peripherals such as magnetic tape or disk within a few minutes. A microwave oven may be in a clean, comfortable kitchen, but cost precludes using peripheral devices. So the one and only program the dedicated computer runs has to be in some nonvolatile memory. Why not use volatile RAM with battery backup? Imagine your car breaking down somewhere in the great heartland of America, and you discover that the backup battery failed. You can call a mechanic and have the battery replaced. Will the mechanic be able to reload the program into your car's RAM? Which program? Which version?

Granted that some dedicated computers must use ROM (read only memory) so that their program will never be destroyed (unless the whole computer goes up in flames), what does this have to do with reentrancy? By virtue of the fact that reentrant programs are pure programs, they can reside in ROM. The dedicated computer uses its registers and RAM as a workspace. In the event of a loss of power for the computer or the RAM, a power-up sequence would trigger execution of a reinitialization sequence which is part of the reentrant program in the ROM. You probably would have used a development system (host system) for the program preparation and testing. When it is deemed correct, the bits destined for the ROM are written into the ROM with a ROM-programmer device, if the ROM is user-programmable. Programming a ROM simply means copying into it a desired bit pattern; a ROM-programmer is a machine, not a person, which lets you do this.

Reentrancy is so important that it is supported directly by the hardware in some PDP-11s. PDP-11 models 44, 45, and 70 use a memory management unit that distinguishes between instructions and data. It maintains two sets of memory address tables, one for the I (instruction) space, and one for the D (data space). How does the memory management unit know which bits represent instruction addresses and which represent operand addresses? It assumes that if an item is directed into the IR, it must be an instruction, and if the item is sent to the ALU, it must be data. That being the case, you cannot have a program in I space modify itself, because the MMU won't let you. So you may as well make the program reentrant. This support of split I and D space has an important consequence. Programs can now be twice as large! The instruction part of the program can use 32 KW of instruction space, and the data part of the program can also use 32 KW of data space.

Even though two models of the PDP-11 may support the same instruction sets, it may be very difficult to move a program which runs on a PDP-11 with split I and D space into one which supports only a single physical or virtual memory.

Coroutines

You can imagine a high level language in which input and output statements have the symbolic names of variables interspersed with descriptions of the expected data representations, as in:

Read an Integer of 5 digits, and assign the value to A, then Read a Floating-Point item of 10 characters and assign the value to B, and then Read a Boolean (Logical) item of one character and assign the value to C.

The high level language's compiler could translate this statement into a straightforward series of subroutine calls:

```
JSR     PC,READ   ; read data
JSR     R5,CONVI  ; convert an integer
.WORD   5,A       ; length 5, assign to A
JSR     R5,CONVF  ; convert fl pt #
.WORD   10.,B     ; length 10., assign to B
JSR     R5,CONVB  ; convert a Boolean
.WORD   1,C       ; length 1, assign to C
```

You can also visualize a high level language in which the input variable list and the input representation descriptors are maintained separately. Thus, the preceding high level language input (read) statement might now have the following rendition:

Read variables A,B,C using format descriptor 13.
many lines later
Format descriptor 13: integer 5, floating point 10, boolean 1.

In such a case, a high level language compiler would prefer not to have to merge the two sets of instructions corresponding to this pair of statements. Besides being difficult in itself, some other "read" statements might also be referring to the same format descriptor. What we would like to do here is facilitated by the construct called the *coroutine*. Whereas a subroutine is viewed as being in a subservient relationship with respect to its caller, a coroutine is co-equal with its invoker, which is also a coroutine. With a subroutine, each invocation of the subroutine causes it to be executed from its beginning. With a coroutine, you can arrange for it and its caller, the other coroutine, to take turns executing segments of each other. Suppose you have coroutines I and J. As instruction sequences, we could have

I: $I_1\ I_2\ I_3\ \ldots\ I_n$
J: $J_1\ J_2\ J_3\ \ldots\ J_n$

The execution sequence could be

$I_1\ I_2 \ldots I_a$	$J_1\ J_2 \ldots J_b$	$I_{a+1} \ldots I_c$	$J_{b+1} \ldots$
Start I	Enter J	Resume I	Resume J ...

The instruction I_a is a coroutine-call, as are the ones at J_b, I_c, and so on. The act of "resuming" is the same as a coroutine call, as we shall see momentarily.

The PDP-11 supports the concept of coroutines by using the same instruction for both coroutine calls and coroutine returns; the RTS is not used. The system stack is used to specify the entry point and return (resume) addresses. In terms of our previous example with an input statement and its matching format descriptor, we could write two coroutines C1 and C2 and invoke them as follows:

```
        MOV     #C1,-(SP)       ; set entry point
        JSR     PC,@(SP)+       ; swap PC and top-of-stack
C2:     ...
```

This JSR exchanges the top item in the system stack with the content of the PC. So this sequence will invoke the coroutine whose entry point is C1, and expect the return address C2 to be found on top of the stack. We can now finish writing coroutines C1 and C2. C1 follows:

```
C1:     JSR     PC,READ
C11:    JSR     PC,@(SP)+ ; resume C2
C12:    JSR     R5,STORE
        .WORD   A
C13:    JSR     PC,@(SP)+ ; resume C2
C14:    JSR     R5,STORE
        .WORD   B
C15:    JSR     PC,@(SP)+ ; resume C2
C16:    JSR     R5,STORE
        .WORD   C
C17:    JSR     PC,@(SP)+ ; resume C2
```

The C2 coroutine can be isolated (as opposed to being in-line) from the main flow of control, if desired, or in-line, as we have it below:

```
        MOV     #C1,-(SP)       ; set C1 entry point
C0:     JSR     PC,@(SP)+       ; resume C1
C2:     JSR     R5,READI
        .WORD   5
C21:    JSR     PC,@(SP)+       ; resume C1
C22:    JSR     R5,READF
        .WORD   10.
C23:    JSR     PC,@(SP)+       ; resume C1
C24:    JSR     R5,READB
        .WORD   1
C25:    JSR     PC,@(SP)+       ; resume C1
C2END:
```

The flow of control, starting at C0, will be:

```
C0 C1,C11 C2,C21 C12,C13 C22,C23 ...
    in C1   in C2    in C1     in C2 ...
```

Character	Octal	Binary code
A	101	1 000 001
C	103	1 000 011
B	102	1 000 010
C	103	1 000 011

Figure 15.26 Some ASCII codes.

Coroutines are a natural control mechanism to use in a provider-consumer situation such as we have here. Without coroutines it may be necessary to make two complete passes over a set of data. With coroutines, it may be possible to collapse the processing into a single pass over the data.

Error Detection and Correction

We have been using the concept of a parity bit as a check bit for some time. We will now see what a parity bit can do for us, if properly used.

Given a code set, say ASCII, we can write down all the character code pairs which differ in only one bit. A few of these pairs are shown here in figure 15.26. A single bit error in the correct position can transform an "A" into a "C", or a "C" into a "B", or vice versa.

When we add an extra bit, the parity bit, to a code word, we choose it so that it increases the separation (the distance) between that code and all other members of the code set. We can define the *distance* between binary codes x and y as

$$d(x,y) = \text{sum}(x \neq y)$$

So

$$\begin{array}{lll} d(\text{"A"},\text{"B"}) \rightarrow & 1\ 000\ 001 & \\ \neq & \underline{1\ 000\ 010} & \\ & 0\ 000\ 011 \rightarrow \text{sum } 2 & \end{array}$$

If the distance between x and y is n, then it will take n simultaneous appropriately situated single-bit errors to transform x into y, or vice versa.

If we do not use a parity bit, the minimum distance between the codes in ASCII is 1. So some single-bit errors will not be detected. If we attach a parity bit to each ASCII code, then we can arrange to increase the minimum distance of the code set to 2. Suppose we choose the parity so that the number of 1 bits is even. Clearly, attaching an even parity check bit to a code cannot decrease its distance from all other codes. Then if we consider the effect of attaching an even parity check bit to two codes that were distance 1 apart, it is easy to show that the parity bit will force them to be distance 2 apart. This is true for all such "close" codes, so the ASCII code set's minimum distance is increased to 2, and this guarantees that all single-bit errors can be *detected*. Double-bit errors cannot be detected, but triple bit errors can, and so on.

The notion of the distance in a code set is credited to Richard Hamming, who took it one large step further. When I first became acquainted with his idea, it seemed like magic. It still has a brilliant sparkle to it. How can you arrange things so that an error in transmitting a code will be *self-correcting?* Suppose the probability of single-bit errors is one in one-hundred-million (10^{-8}). If bit errors are independent of each other, then the probability of a double-bit error is the square of the previous probability. That is so small (10^{-16}) that we can almost ignore it. Even if the likelihood of a double-bit error is greater than the product of the individual probabilities, it is still much less likely than a single-bit error. So how can we be in a position to conquer single-bit errors? We want to enhance the code set so that all single-bit errors will be self-correcting—and, as a bonus, be able to detect all double-bit errors. Suppose we had a very small code set consisting of the single-bit codes 0 and 1. Then d(0,1)=1. So we cannot even detect single-bit errors. Let us add an even parity check bit to each code, writing it on the left of the code bit. This transforms the 0 and 1 codes as follows:

$0 \rightarrow 0\ 0$
$1 \rightarrow 1\ 1$
1, 2 bit positions

Now d(00,11)=2, which is better. Suppose we add a third check bit, chosen so that it and the original code information bits (bit, in this case) will produce another even parity check. Adding the new check bit on the left, we now have

$0 \rightarrow 00 \rightarrow 000$

$1 \rightarrow 11 \rightarrow 111$

Now d(000,111)=3. Symbolically we have a 3-bit word (c1,c2,i) with

parity(c1,i) = even

parity(c2,i) = even

Suppose we induce a single-bit error in one of these 3-bit code words:

$111 \rightarrow 110$

Then:

p(c1,i) = p(1,0) = 1, $\rightarrow$ p1
p(c2,i) = p(1,0) = 1, $\rightarrow$ p2

We have written p(. . .) as a parity generator and assigned the values it found to variables p1 and p2. If no error had occurred, then clearly p1 and p2 would both have been 0. If a single-bit error does occur, then one or both of p1 and p2 will be non-0. If that is the case, the binary number (p2,p1) is used to locate the "bad" bit, and it can be corrected simply by complementing it. Since (p2,p1) is 11, the third bit must be bad. Correcting the third bit gives us 111, which was the correct code word.

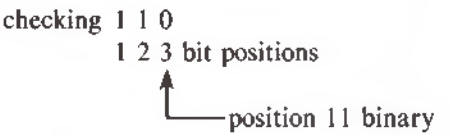


Figure 15.27

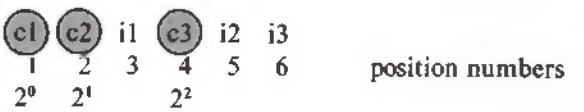


Figure 15.28

Suppose we wanted to apply this technique to a code set that has 3-bit codes. It takes 2 bits to point to each position of a 3-bit byte, to select the potentially bad bit. But you now have the original 3 information bits *and* the 2 check bits to protect. It takes another check bit to have a sufficiently large pointer.

This technique generalizes and applies to any code word size. For a 3-bit-wide code set, we would assign the code information bits i1, i2, and i3 and the check bits c1, c2, and c3 the following positions:

c1	c2	i1	c3	i2	i3	
1	2	3	4	5	6	position numbers
001	010	011	100	101	110	

The relationship between the check bits c and the information bits i are

p(c1,c2,i1) = p1
p(c1,c3,i2) = p2
p(c2,c3,i3) = p3

Notice how bit i1's position 3 just happens to be covered by check bits c1 and c2; similarly bit i2's position 5 is covered by check bits c1 and c3. Note also that each check bit is placed in a bit position whose position number is a power of 2. Whenever the enhanced 6-bit code is received, the values p1, p2, and p3 are computed. If they are all 0, the received code is assumed to be correct, and the check bits can be discarded if desired. Otherwise the three bits p1, p2, and p3 are used as a pointer to identify the presumably "bad" bit. Complementing this bit yields a correct 6-bit code, since it must now by construction pass all three parity checks.

The algorithm we have used is the Hamming single-bit error correction algorithm. It is implemented in the hardware of the ECC (error correcting code) memory controllers used on some PDP-11s and many other computers. When core memory technology was the dominant form of memory used in computers, the likelihood of having single-bit errors was very low, and the cost for storing the Hamming check bits would have been too high. So core memory systems either had no check bits, or they provided one parity bit per word (sometimes one per byte).

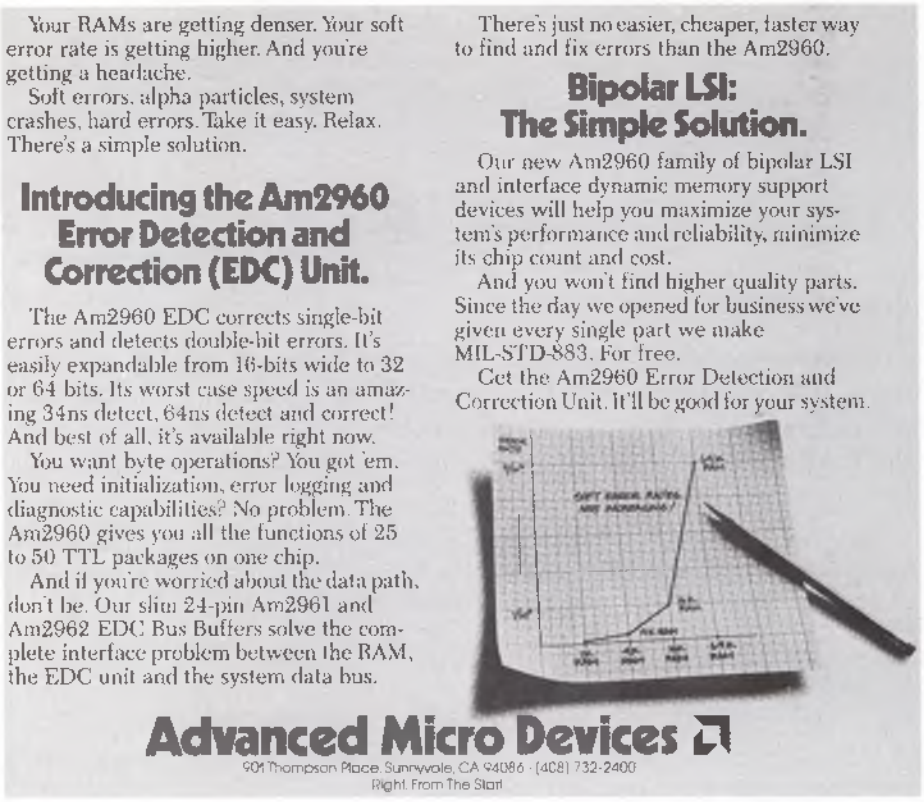


Figure 15.29 AMD ECC chip.

(Reproduced with permission of Advanced Micro Devices, Inc. All rights reserved.)

The solid-state memories which have now to a large extent displaced core memory use RAM chips. It is expected that single-bit errors in a RAM chip may not be uncommon. So one-bit ECC is implemented as an on-board service which is transparent to the user, as indicated in the add copy in figure 15.29. It has the effect of taking a "faulty" RAM memory and making it as fault-free as a core memory, if not better. This is an example of fault-tolerant design.

An ECC check for an 8-bit byte requires 4 check bits. A total of 12 bits are used for each byte, since 4 is the smallest number of bits which can span twelve positions. Each memory fetch or store in an ECC memory triggers an ECC computation in the memory controller. If an error is detected, it will be corrected. All of this is transparent to the user. Normally the ECC hardware will log its activity so that particularly persistent error patterns can lead to replacement of some presumably defective RAM chip.

Summary

We have seen many new ideas in this chapter, and we have revisited some old ideas, to see their power enhanced. As we have seen, macro definitions can take advantage of conditional assembly directives. This makes it possible to tailor macros to meet almost any need.

The directives .IRP and .REPT, which can be used independently as well as within macro definitions, facilitate generating iterative and count-controlled constructs. Each .IRP and each .REPT must have a matching .ENDM.

The support of data structures such as lists, two-dimensional lists, and trees has been discussed. The relation of data structures to recursion and how recursion can be used within subroutines has been discussed. We have examined several techniques for speeding up program execution. These include replacing subroutine calls by in-line code where appropriate. However, the use of certain computer design features such as cache memory favor nontraditional coding techniques such as threaded code.

Ways in which computers may be constructed for faster operation have been investigated. In the PDP-11 family we see the use of two or more specialized buses, in addition to the use of cache memory. All of these techniques have the advantage of being transparent to the user.

Reentrant code, which is sharable, is used on large multi-user systems as well as on dedicated microcomputers. It should not be confused with recursive code; the two ideas are independent of each other. The idea of co-equal subroutines, known as coroutines, was discussed. These are easily supported on the PDP-11, due to its generalized addressing modes.

Finally, we reexamined the use of the parity bit as a 1-bit error-detection technique and generalized it to provide for a 1-bit error-correction code (ECC), using the Hamming technique. ECC hardware makes it possible for RAM memory to function as reliably as core memory.

Exercises

15.1 Consider the following, and assume that all items are properly defined:

```
MOV   #EX,XROOT          F:     MOV TEMP,-(SP)
JSR   PC,F                      MOV XROOT,-(SP)
HALT                            MOV R0,-(SP)
 EX:  .WORD  '-,L,R             MOV XROOT, R0
                                MOV 4(R0),XROOT
 L:   .WORD  '+,A,B             BEQ NUM
 R:   .WORD  '+,D,E     OP:     JSR PC,F
 A:   .WORD  3,0,0              MOV VAL,TEMP
 B:   .WORD  4,0,0              MOV 2(R0),XROOT
 C:   .WORD  2,0,0              JSR PC,F
 D:   .WORD  0,0,0              CMP (R0),#'+
 E:   .WORD  1,0,0              BNE MIN
                                ADD TEMP, VAL
                                BR  DONE
                        MIN:    SUB TEMP, VAL
                                BR  DONE
                        NUM:    MOV (R0), VAL
                        DONE:   MOV (SP)+,R0
                                MOV (SP)+, XROOT
                                MOV (SP)+, TEMP
                                RTS PC
```

(a) Diagram the structure that F operates on.
(b) What does F compute in this case?
(c) In general, what could F be used for?

15.2 (a) Write a macro to find the max and min of a list of 16-bit words. It should be called as in:

```
EXTREM <list of things>,MAX,MIN
```

where MAX is the place to put the maximum and MIN is the place to put the minimum. The macro should generate code to compute the extreme at run time.

(b) What code is generated for the following invocation?

```
EXTREM <A,B,(%6),16(%2)>,%0,%1
```

15.3 Act as an assembler: write the machine code which the assembler produces for the following code. Assume that START is at location 0000 (i.e., START = 0000).

```
            .MCALL .PRINT
            .MACRO PUSH ,WORD
             MOV WORK,-(%6)
            .ENDM
            .MACRO POP,WORD
             MOV (%6)+,WORD
            .ENDM
            .MACRO WRITE,ARG,?B,?C
             PUSH %0
            .PRINT #B
             JMP C
B:          .ASCII /ARG /<12>
            .EVEN
C:           POP %0
            .ENDM WRITE
START:       WRITE <THIS IS AN EXAMPLE>
             HALT
            .END START
```

Assume that the system macro definition of .PRINT is:

```
            .MACRO .PRINT,ARG
            .GLOBL $PRINT
            .IIF NB <ARG>,MOV ARG,%0
            JSR %7,$PRINT
            .ENDM
```

15.4 (a) Given the following data, draw the binary tree that would result from entering this data into a tree-building program:

20, 25, 30, 10, 35, 15, 40

(b) What initial order of this data would have resulted in a more balanced tree? Write down that order and draw the new tree.

15.5 Write the assembler source code that would have the items 20, 25, 30, and 10 as the values in the nodes of:

(a) linked list
(b) a two-way linked list

15.6 Suppose we want to write a macro to multiply two integers together and store the product. Explain briefly how the macro definition and call below will do this; or, if you don't think it will work, explain why not.

```
        .MACRO MULT X,Y,RESULT,?LOOP
        CLR RESULT
LOOP:   .REPT X
        ADD Y, RESULT
        .ENDM
        .ENDM
          ...
        MULT %1,%2,%3       ; STORE PRODUCT
                              ; OF %1 AND %2 IN %3
```

15.7 The .IRPC directive is similar to the .IRP directive we saw in the section ".IRP"; it differs in that the .IRPC repeats the block of code once for each *character* in its argument list. For instance:

```
.IRPC  ARG,<324>
  .BYTE ARG
.ENDM
```

will expand as

```
.BYTE 3
.BYTE 2
.BYTE 4
```

.IRPC can be used inside a macro definition as well as independently of macros. Use .IRPC to define a macro which saves registers 0–5.

15.8 You are given R records, each having 2 fields; part number PN, part description PD. The PN's are between 0 and 50,000. The PD's are character strings of 10 to 60 characters. You are to sort these R records, by PN, using a binary tree.

(a) Describe what a typical node should contain; clearly specify the field sizes and purposes.
(b) What restriction on the number R could lead to an appreciable reduction in the node sizes?
(c) When you execute a recursive tree traversal, and consider all possible trees that could be built, what is the maximum stack depth required to sort these records? What would be in the stack, and what kind of trees does this correspond to?
(d) For R = 31, what is the minimum stack depth?

15.9 Given the list of numbers 25, 6, 29, 76, 24, 26, write the assembly-language statements needed to represent them as the following structures at assembly time (not run time):

(a) One-way linked list.
(b) Two-way linked list.
(c) A binary tree in which the values in the leaves of each left subtree are less than those of right subtrees.

15.10 If macros were not allowed, give the code which you would have to write to replace:

```
.MACRO SAM, A,B,?C
.IIF NB <A>, MOV A, %0
   JSR %7, JOAN
.IF NB <B>
     BCC C
     JMP B
C:
 .ENDC
 .ENDM

    SAM ABC
    SAM #PQ,ED
```

15.11 True or False?

(a) Inserting an item into the middle of an ordered array of items is usually faster than inserting an item into the middle of a linked list of items.
(b) Arrays generally require more space than do linked lists to store the same amount of information.
(c) A queue operates on a "FIFO" principle.
(d) One part of an interrupt vector supplies the priority at which the interrupt handler routine will begin executing.
(e) Reentrant code should use a stack located in read-only memory.

15.12 Given the 4-bit BCD codes 000, 0001, . . ., 1001 for the digits 0, 1, . . ., 9:

(a) How many Hamming bits must be attached to each code word to make it into an ECC?
(b) Write the ECC version of the codes for digits 1, 7, and 8, using even parity checks.
(c) Write the ECC version of the codes for digits 1, 7, and 8, using odd parity checks.
(d) Show how an error in the third bit of the code for 7 obtained in part (b) is corrected.
(e) Show how an error in the third and fifth bits of the code for 7 obtained in part (b) is processed.

15.13 Rewrite the string-copy macro definition of exercise 12.7 so that it does some optimization. Both strings should start on a word boundary, so your definition should use MOV to copy all the whole words involved, and use a final MOVB only if it is required. The best possible code must be generated at assembly time.

15.14 True or False?

(a) A queue is a linear list with a last-in, first-out property.
(b) A queue is a linear list with a first-in, last-out property.
(c) A circular queue cannot overflow.
(d) Random references to components of multi-dimensional arrays are faster than sequential references.
(e) Programs written with macros usually assemble and run faster.
(f) An interrupt handler must not be interrupted.
(g) Floating-point operations are heavily used in scientific problem solving because they are much more accurate.
(h) All the information in a device's control and status register can be examined under program control.
(i) Interrupt handlers must not use subroutine calls because they both use the system stack.
(j) If the CPU sets the overflow bit V in the PS, then an arithmetic error has just occurred.

15.15 Assume that each instruction fetch and each memory reference takes 1 microsecond per word and that nothing else takes any time.

(a) What code does the assembler produce given the following statements?
(b) How long does the code produced by the assembler take to execute?

```
FISH = 0
.REPT 3
.IF EQ FISH
CLR FISH
FISH = FISH + 1
.ENDC
.ENDM
```

15.16 Suppose that in the PDP-11/model x the action of "JSR reg, loc" is to store the current PC in reg and then jump to "loc", and that an RTS reg is equivalent to a JMP @reg. Could recursion still be implemented on this machine? Support your answer.

15.17 What can the following macro be used for? Discuss cases arising from differing invocations.

```
.MACRO  DB  A,B
.IF  NB <A> ; TEST FOR NOT BLANK
     JSR R5,DUMP
     .WORD A,B
.ENDC
.IF B <A> ; TEST FOR BLANK
      .SNAP
.ENDC
.ENDM
```

15.18 Examine the following statements:

```
        .MACRO ARYCLR ABC, COUNT, ?LOOP
        MOV #ABC, %0
LOOP:   CLR (%0)+
        DEC COUNT
        BNE LOOP
        .ENDM
```

(a) Assume that ARYCLR is the only macro in your program. Show the assembly language that the macro process would generate for the following invocation:

```
ARYCLR      FOO, XYZ
```

(b) Now suppose you invoked ARYCLA a second time in the same program, using the same invocation as in (a). Would the assembly language generated be the same? If not, explain any differences.

(c) Briefly explain the difference between a macro and a subroutine.

15.19 What does the following macro do? Be sure to discuss cases which arise from differing invocation.

```
        .MACRO SOLVE PROB ?X
        .IIF NB <PROB> ,JSR PC,PROB'IT
        BR X
        .SNAP
X:
        .ENDM
```

What is wrong here, and how can you fix it?

15.20 Briefly describe what a reentrant code is. What special considerations apply to the use of stacks when a reentrant code is used?

15.21 True or False?

(a) All high level languages (HLLs) use the same standard format for storing multi-dimensional arrays in memory.
(b) Good programming practice: For every PUSH onto a stack, there is a corresponding POP.
(c) The use of conditional assembly statements can greatly affect how long a program will be after it is assembled.
(d) For each .MACRO definition, there must be at least one invocation of that macro.
(e) Array element sizes are important when accessing multi-dimensional arrays.
(f) The system stack is the only legal way to pass parameters to subroutines.
(g) Arrays accessed by assembly-language programs cannot have more than seven dimensions.
(h) A subroutine can reuse label names used by the calling program.
(i) It is illegal for a subroutine to call itself.
(j) It is illegal for a macro to call itself.

15.22 What is the following macro being used for, and why is this regarded as poor practice?

```
.MACRO ZERO ARRAY N
MOV #ARRAY,%0
.REPT N
CLR (%0)+
.ENDM
ENDM

        ...
        ZERO ABC,100
        ...
        ZERO PQR,50
        ...
ABC:    .BLKW 100
PQR:    BLKW 50
```

15.23 The word "overflow" is used in conjunction with many different kinds of operations on a variety of data types or data structures. For which of the following can overflow result from a valid operation?

(a) Arrays and lists.
(b) Integers and stacks.
(c) Queues and floating point numbers.

15.24 The following macro is intended to replace the RTS %5 instruction.

(a) Show the expansion resulting from the invocation RET ;.
(b) Rewrite the definition with fewer instructions, so that it has the same effect.

```
.MACRO   RET ?A
MOV      %5,A       ; save ret addr
MOV      (%6)+,%5           ; rest 5
MOV      A,-(%6)
RTS      %7
.WORD    0
.ENDM
```

15.25 Given two programs that do the same task and are identical in all respects except that one uses macros and the other uses subroutines:

(a) Which program (in general) takes up the most space in memory when it is running, and why?
(b) Which program (in general) has the fastest execution time, and why?

15.26 Explain how stack overflow can occur at assembly time. (Hint: consider macros.)

15.27 You wish to test your tree-traversal program with a sorted tree known to be correct, because you are creating it at assembly time.

(a) Write the assembler source statements which would represent the tree whose nodes have the octal values 10,15,6,22, and 7.
(b) Draw a picture of this tree.
(c) In what sequence would these numbers have to be so that the sorted tree would have no left-subtrees?

15.28 A reentrant program is usually somewhat slower than a non-reentrant version would be.

(a) Why is this so?
(b) In spite of this, there are situations in which you will get computing done more rapidly with reentrant programs. Explain why this can happen, and resolve this apparent contradiction.

15.29 In what sense can the simple SHIFT macro of the section "Variable Length Macros" be said to be of variable length?

15.30 In some applications you may frequently have many consecutive invocations of the same subroutine. Show how the idea behind threaded code can be extended to avoid unnecessarily repeating the consecutive subroutine invocations. For instance, instead of:

```
SUB1
  X,Y,Z
SUB1
  P,Q,R
SUB2
  A,B,C
SUB2
  M,N,OH
```

you could write

```
SUB1
  X,Y,Z
  P,Q,R,0
SUB2
  A,B,C
  M,N,OH,0
```

if the SUB1 and SUB2 subroutines were suitably modified. Sketch the modifications which should suffice: what assumptions must be made regarding the arguments? Threads of this type, where multiple consecutive subroutine invocations are implied, are known as *knotted code.*

Case Study

The following program illustrates the use of recursion, both in storing incoming data in the nodes of a binary tree, and in traversing this tree to generate an alphabetized printout. The few statements that appear in lowercase were used in debugging the program. The "dump" shows the tree when all the data have been read in. The "snap" allows you to trace the recursive calls made by TRAVSL. The only complication occurring in preparing this program was due to a change in conventions. The program was initially prepared for a computer in which the .PRINT routine accepted as its argument the address of a string which was terminated by a line feed code, octal 12. The program was then moved over to a computer in which .PRINT expected a zero-terminated string. Thus, all of the previous .ASCII statements used with .PRINT were changed to .ASCIZ. However, I forgot about the way the data was being stored. Thus, in processing the lines:

```
AL
CHUCK
MARY
BOB
ANNE
```

the data were properly echoed as expected, but after the message SORTED DATA was generated, the program seemed to fail after it had printed

```
AL
ANNE
BOB
CHUCK
```

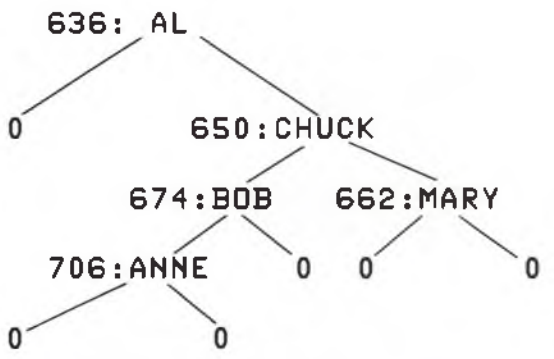
The last item, MARY, was not being printed. What was the problem? Consider that when each item is read in, it is placed in the next five-word block available in the array "TREE". If that block and the following words are initially zero, then the echoing would find a 0-terminated string. However, after we have filled in the left and right pointers, then the byte immediately following the 12 stored to terminate CHUCK is no longer 0, since it is part of the whole word that is used as a pointer. Thus, the attempt to print CHUCK leads the print routine to try to print the pointers following CHUCK as if they were ASCII codes. Since in general they are not ASCII codes, the result is such a strange printout that I concluded that the program had failed for some mysterious reason. Simply changing the `MOVB #LF,(R0)` in the CLEAR subroutine so that it stores a 0 instead of a 12 takes care of this problem. Of course, changing the instruction to a `CLRB (R0)` would be better.

The octal dump produced for the data we have here is displayed; note that the address for "TREE" was 0664.

```
UNSORTED DATA
AL
CHUCK
MARY
BOB
ANNE

SORTED DATA
000620: 000416 012600 016404 000010 004767 177720 000207
046101
000640: 000012 005000 000000 000650 044103 041525 005113
000674
000660: 000662 040515 054522 005012 000000 000000 047502
005102
000700: 005000 000706 000000 047101 042516 005012 000000
000000
000720: 000000 000000 000000 000000 000000 000000 000000
000000
000740: 000000 000000 000000 000000 000000 000000 000000
000000

R4,SP,PC: 000636 177610 000572
```

```
R4,SP,PC: 000000 177604 000572
AL
R4,SP,PC: 000650 177606 000572
R4,SP,PC: 000674 177602 000572
R4,SP,PC: 000706 177576 000572
R4,SP,PC: 000000 177572 000572
ANNE

R4,SP,PC: 000000 177574 000572
BOB
R4,SP,PC: 000000 177600 000572
CHUCK
```

The abbreviated snapshot trace following the octal dump shows the state of register 4, the stack pointer, and the program counter immediately after entering TRAVSL. The tree structure this corresponds to is drawn below:



The program we have been discussing follows:

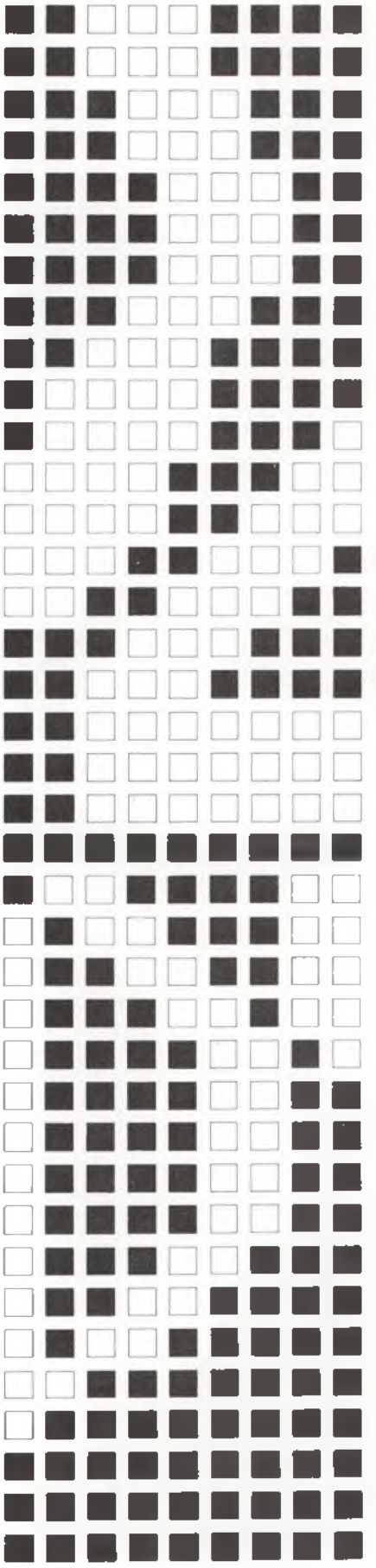
```
; English words of 5 characters or less are read, one word
; per line, or record, or data card. They are assumed to be
; left justified. Each word is echoed, then it is stored
; as the leaf of a subtree, such that all left subtrees are
; alphabetically less than right subtree values.
; When the data are exhausted, a recursive tree traversal
; procedure is called, which prints the desired sorted data.
;
        .MCALL.  PMD,.PRINT,.READ ; fetch macro definitions
        .MCALL  .REGDEF,.SNAP,.EXIT
        .GLOBL  DUMP
        .NLIST  BEX
        LF = 12; line feed
```

```
;
START:  .PMD    START,FIN       ; force dump of (START)-(FIN)
                                ; on M-TRAP
        ;
        .PRINT  #TITLE          ; ''unsorted data''
        .PRINT  #SPACE
        MOV     #TREE,R3        ; put root adr in R3
        .READ   R3,#NODATA      ; get root value or quit
        .PRINT  R3              ; echo
        JSR     PC,CLEAR        ; clear pointer locations
;
NEXT:   ADD     #12,R3          ; R3 holds ptr to next
                                ; available node
        .READ   R3,#NOMORE      ; get next node value
        .PRINT  R3              ; echo raw data
        JSR     PC,CLEAR
        MOV     #TREE,R4        ; R4 points to tree or
                                ; subtree node
; for examining tree, before and after
        JSR     PC,SORT         ; adjust tree nodes to keep
                                ; tree sorted
        BR      NEXT
;
NOMORE: .PRINT  #SPACE          ; data all read now
        .PRINT  #TITLE2         ; ''sorted data''
        jsr     r5,dump
        .word   tree,tree+95.
        MOV     #TREE,R4        ; put tree root adr in R4
        JSR     PC,TRAVSL       ; traverse binary tree
        .EXIT
;
;       subroutines
;
CLEAR:  MOV     R3,R0
        ADD     #5,R0           ; points to end of word
        MOVB    #LF,(R0)        ; LF defines end of word
        CLR     6(R3)           ; clear left pointer
        CLR     10(R3)          ; clear right pointer
        RTS     PC
;
NODATA: .PRINT  #ERROR          ; no data
        .EXIT
;
; recursive routine to find right place for new
; node so tree remains sorted.
```

```
SORT:   MOV     R3,R1    ; save reg3
        MOV     R4,R2    ; save node ptr
CONT:   CMPB    (R1),#LF         ; check for word end
        BEQ     RIGHT
        CMPB    (R1)+,(R2)+      ; letter by letter comparison
        BEQ     CONT
        BGT     RIGHT
        ADD     #6,R4            ; create left ptr
ZCHECK: TST     (R4)
        BEQ     INSERT
        MOV     (R4),R4          ; get subtree node into R4
        JSR     PC,SORT          ; recursive call
PLACE:  RTS     PC
RIGHT:  ADD     #10,R4           ; get right ptr
        BR      ZCHECK
INSERT: MOV     R3,(R4)          ; insert item using R4
        BR      PLACE
;
; this recursive subroutine traverses a binary tree
; given a root pointer in R4, printing end nodes.
;
TRAVSL: TST     R4       ; is node a leaf?
    .snap
        BEQ     RETURN
        MOV     R4,-(SP)         ; save node adr
        MOV     6(R4),R4         ; get left node
        JSR     PC,TRAVSL        ; recurse
        MOV     (SP)+,R4         ; pop node
        .PRINT  R4               ; display value
        MOV     10(R4),R4        ; get right node
        JSR     PC,TRAVSL        ; recurse
RETURN: RTS     PC
;
;       arrays, data, constants
;
TREE:   .BLKW   96.
SPACE:  .ASCIZ  / /<12>
TITLE:  .ASCIZ  /UNSORTED DATA/<12>
TITLE2: .ASCIZ  /SORTED DATA/<12>
ERROR:  .ASCIZ  /ERROR- NO DATA/<12>
        .EVEN
FIN:    .END    START
```

16

micros, minis, maxis



In the process of trying to understand computing, we have to avoid the trap of not seeing the forest for the trees. The PDP-11 is just one of a multitude of different computers. The PDP-11 is a multitude in its own right, because the size of the PDP-11 family ranges from the tiny LSI-11/2 to the twin-refrigerator-sized PDP-11/70 and its VAX cousins. What about other computers?

Common Features

The internal representations for instructions and data which are used in all modern computers are based on binary notation. The most commonly used byte size is 8 bits wide. The two most frequently used code sets are ASCII and EBCDIC. Every CPU has a control unit, an instruction register, a program counter (sometimes called an instruction counter), and an ALU. The differences of performance and capacity derive from the word size, ALU implementation, instruction set, and addressing characteristics, bus or channel organization, as well as the component technology used.

Distinguishing Features

We can arbitrarily classify computers in three categories: (1) micro, (2) mini, (3) maxi (or mainframe). There is no universally accepted definition for these terms which can be applied consistently. At one time the U.S. Government General Services Administration, the government's major buyer of computers, ruled that a computing system costing less than $50,000 was therefore a minicomputer system. As we attempt to describe each class of computer in turn, the distinguishing features will be evident.

Micros

The first microcomputer, the Intel 8008, used an 8-bit word. Most of its successors and competitors use an 8-bit word. Some of these are listed in figure 16.1. We can distinguish between a *microcomputer-on-a-chip* (e.g., Intel 8048, which is a self-contained computer-on-a-chip, with its control unit, ALU, registers, *and* memory all on a single chip about the size of a postage stamp) and a *microprocessor-on-a-chip*, with which some other chip or chips must be used to provide a missing function (e.g., there is no on-chip memory in some single-chip processors).

If a single-chip microcomputer can do the job for a dedicated application, then you will get the utmost in reliability, since you have reduced the chip count to one. Cost as well as reliability is greatly influenced by the chip count (the number of integrated-circuit chips required to implement a system).

Intel 8080, 8080A, 8084, 8085, ...
Zilog Z80, Z8, Z80A, ...
Motorola 6800
RCA Cosmac
Fairchild F8
AMD 6502
...

Figure 16.1 Some microcomputers and microprocessors.

Microprocessors need additional chips to form a whole computer. The full set of chips are usually assembled together on a circuit board. If this results in a self-contained computer (except for a power supply and peripheral devices), it is called a single-board-computer (SBC). An LSI-11 is an SBC. An LSI-11/2 is almost an SBC; it is actually a single-board processor, since it has no on-board memory (the LSI-11 does have on-board memory).

Many micros support a bus similar to the LSI-11's Q-bus. Intel has its Multibus; Prolog uses its STD bus. The best known bus among micros is the S-100 bus. This bus is used on many personal computers (e.g., Cromenco). It is being considered for adoption as an IEEE-EIA standard. If adopted, it would be one step removed from being an ANSI standard.

The 8-bit micro will continue to dominate the world of micros (the mere fact that almost every automobile now being built uses, or will be using three or four 8-bit micros ensures that many millions of them will continue to be built each year for many years to come). However, the next generation of micros is already in existence. Intel has its 8086, Zilog its Z8000, Motorola its M68000, etc. These single-chip microprocessors approach the speed of a PDP-11/34. They use 8-bit bytes, 16-bit words, and some support very large direct addresses so they can address over one million bytes of memory. Some even have an optional memory-management chip.

The uses of microprocessors and microcomputers are analogous to the way we use small electric motors. It is common to find an electric motor in a vacuum cleaner, a blender, a clock, a dishwasher, a fan, a drill, a power saw, etc. In a similar fashion, the low cost of micros makes it possible to use them in dedicated applications, ranging from the control of a sewing machine to the control of a carburetor.

Minis

DEC popularized the minicomputer with its PDP-8 and PDP-12. Over 50,000 of these were sold. The PDP-8 is still being manufactured as the core of a word-processing system. These computers use a 12-bit word. The more recent PDP-11 family is now DEC's principal minicomputer product. In the meantime, many other firms compete in the minicomputer field.

Data General (the Nova family is a DG product), Computer Automation, Perkin-Elmer (formerly Interdata), Prime, General Automation, Hewlett-Packard, Texas Instruments, IBM, Univac-Varian—these are just some of the firms which manufacture minis (although IBM prefers to speak of "small computers" rather than minis).

The 16-bit word size is the most common word size for minicomputers. The Harris Corporation (formerly Datacraft) uses a 24-bit word. The older DEC PDP-15 uses an 18-bit word. There are some 21-bit minis also.

chip sets
SBC
board sets
in a chassis with a power supply
rack mounted with interfaces
fully configured as a working system

Figure 16.2 Levels of physical organization.

Burroughs 6700	48
CDC Cyber	64
DEC PDP-10	36
IBM 370	32
IBM 4300	32
Univac 1100	36

Figure 16.3 Some mainframes and their word sizes.

As a general rule, when you buy a mini, you usually have to buy a whole system, which includes the CPU, memory, power supply, chassis, and some peripherals. It would be very hard for you to buy just the PDP-11/34 CPU boards. The manufacturer (DEC) will sell board sets, but only in large quantities (usually in multiples of 100). About the only way you could obtain the board set would be as part of a $30,000 spare parts kit. In contrast, micros can be purchased (even a quantity of one) at any one of several levels of organization, as shown in figure 16.2. OEM companies (original equipment manufacturer firms) buy hardware from computer manufacturers, package the hardware, add some software, and resell the packaged system. Ordinary users generally do not have as many choices as an OEM has. Note that an OEM is a buyer of hardware, not a manufacturer of hardware.

Many minis are used as dedicated processors within systems which require more computing power than a typical micro can provide. For instance, a CAT scanner (computerized axial tomography) used in medical diagnosis usually has a mini within it. Many other minis are used as general-purpose computing systems or in the support of data processing applications (e.g., a banking system).

Maxis

The standard computer (before minis and micros) was often called a mainframe. It still is, but the term "maxi" is also used to describe the same computer. Maxis generally have a larger word size than micros or minis. Figure 16.3 illustrates the word sizes on some maxis. Maxis are always sold as complete systems, and they generally require a special computer-room environment: special power, cooling, humidity control, dust control, fire-abatement equipment, etc. Minis and micros are generally less sensitive. Of course, you expect more from a maxi in terms of performance. Expectations are not always fulfilled. Some models classified by their makers as minis (e.g., DEC's VAX) have mainframe characteristics (such as using a 32-bit word) and can match or exceed the performance of some of the so-called maxis. This simply emphasizes the arbitrariness of these classifications.

The mainframe systems are found in almost every large firm, government agency, or university. They typically support large data base systems, large transaction processing systems, and statistical or numerically oriented processing uses.

Virtual Memory Revisited

Many minis and maxis differ in the meaning assigned to virtual memory. As we saw earlier, some PDP-11s can have more than 64KB of physical memory. When they do, a memory management unit maps each user's 32KW virtual memory into the larger physical memory. This does not enlarge a user's address space. It does allow the operating system to keep more tasks in memory. On maxis, the use of virtual memory is intended to support the opposite situation.

Suppose that a maxi has an instruction set which allows the use of 24-bit addresses. In principal, you could directly address 2^{24}, or approximately 16MB of memory. Even today few can afford 16MB of physical memory. However, why not let the users pretend they have such a large virtual memory? With a memory management unit, it is possible for the operating system to keep track of which user's virtual memory has been assigned a section of physical memory and which has not. If the location you are referencing is, in fact, mapped into physical memory (by the memory management unit), then your program continues to execute quickly. Whenever you reference a location that has not been mapped into physical memory, the management unit can force an interrupt, let the operating system find the things you need (they must be on a disk) and allocate some physical memory to these. Having initiated the input activity from disk, the operating system can let some other user's task proceed, while you are temporarily held up.

To sum up: the virtual memory supported on most minis does not let the user directly address as much physical memory as could be attached. The virtual memory supported on most maxis allows the user to address significantly more memory than can physically be attached. A further indication of how arbitrary is the mini-maxi classification can be found in the VAX-11/780. DEC classifies it as a mini, in spite of the fact that it uses a 32-bit word. Furthermore, it allows direct addressing to 32MB, even though it is limited to 8MB of physical memory. Thus the VAX, classified as a mini, uses the powerful virtual memory mapping associated with maxis.

What Do Computers Cost?

We have discussed some of the choices you can make when you are putting a program together. You can solve a given problem in many ways. Even after you have selected the best algorithm to solve a problem, there are still many choices you can make while implementing the algorithm as a program. Space-time trade-offs must always be considered. Writing good programs involves making many choices. What choices do you get to make when buying a computer?

Having defined your computing needs (an exhaustive exercise in its own right, sometimes taking many people many months, even years, of effort), you start making choices on the basis of cost. Of course, the meaning of "cost" is itself subject to debate. A sensible approach to cost is to consider the proposed system's life cycle. What will it cost to purchase, install, staff, operate, maintain, and expand this system, say, for the next five years? This establishes the system's life cycle cost.

Description	Cost	BMC
PDP-11/70 TBA package including CPU		
512 KB memory, RM03 disk, TU77 tape	139,000	748
Added 512 KB memory	21,000	160
Second RM03 disk	20,000	140
Second TU77 tape drive	21,000	175
FP11-C	6,000	32
DH11-AD 16 lines	8,100	61
Second DH11-AD	8,100	61
28 VT100 CRTs at 2,100	58,800	476
Four LA38 terminals	6,800	64
LP11-DA 660 lpm printer	28,000	185
Total	316,800	2,102
Software: RSTS/E included		
COBOL	8,900	
FORTRAN	2,100	
Subtotal	11,000	
System Total, for 32 Users	327,800	
Per User Cost	10,000	

Figure 16.4 A large PDP-11/70.

In your mind you keep repeating "All other things being equal, buy the least-cost system." It sounds simple, but as you might suspect, there is no algorithm for verifying that "all other things are equal." So we will simply illustrate what one vendor sold computing equipment for, at one point in time. This is a snapshot. If you have access to a price list, you may wish to see how the prices have changed, and especially to note the different rates at which various component prices have changed.

A Large PDP-11/70 System

PDP-11/70 systems are sold as packaged systems. The packages provide a CPU, memory, one or more disks, and magnetic tape drives with appropriate controllers. A console terminal is provided (usually a keyboard-printer) and operating system software is included (with MACRO-11). The customer can then enhance the configuration according to his needs and budget. As a general rule, the cost of an enhancement is much lower if it is ordered with the packaged system. Field installation of enhancements is more expensive.

PDP-11/70 package prices range from a low of $111,000 to a high of $183,000 (we will deal only in increments of 1,000). These prices do not include terminals and their interfaces. Let us pick a midrange 11/70 and see what it includes and what it costs to enhance it. This is shown in figure 16.4. The CPU includes the cache memory we have discussed and the memory management unit. The RM03 is the 80 MB disk described earlier. The TU77 is a dual-density (800/1600 bpi) magnetic tape drive for 9-channel tapes which operates at 125 ips. The manufacturer counts on using the tape drive to run diagnostic programs as part of the maintenance contract. The BMC column gives the basic monthly charge for a hardware-maintenance agreement for each item. You can elect to have some or all of the components cared for at the indicated monthly charge if you have the CPU under contract. This is a basic service agreement, with coverage provided during normal working hours. It could be extended into around-the-clock coverage at an increase in cost of some 50 percent.

Figure 16.5 Cost breakdown (in K$).

CPU	Peripherals	Terminals
CPU 76	Disk 40	Mux 16
1 MB 42	Tape 42	Terms 66
FPP 6		Prtr 28
Softw 11		
135	82	110

Total System Cost $327,000

The FP11-C is the floating-point hardware unit. Without it, you would have to use software simulation for all the floating-point operations. The terminals include VT100 CRTs and keyboard printer LA38s. Rather than provide each terminal with its own serial interface (at $820 each), it is more economical to purchase serial interfaces as implemented by a DH-11AD multiplexor. This provides sixteen serial interfaces in one package. This particular package comes with the DEC operating system RSTS/E, which includes both a BASIC and a MACRO-11 at no extra-charge. Other software components are extra charge items, with the indicated one-time license fee for use on a single processor. A monthly maintenance contract can be had for each software product.

Figure 16.5 shows a functional organization of costs. The CPU price is an estimate. It is on the high side because it includes the large chassis the CPU came in. You may be surprised at the cost of such a mundane item as a rack or chassis (about $3,000).

The price of memory is dropping rapidly, and the amounts in figure 16.5 should probably be revised every three months. It is possible to configure a PDP-11/70 to support 64 terminals, depending on the work to be done and the kind of response desired. You would add memory (up to 4 MB), disk (four drives per MBA, adding more MBAs as needed), multiplexors, and CRTs.

Figure 16.6 shows the prices for a modest single-user system based on an LSI-11. The RX02 is a newer floppy disk drive. The RX01 we described earlier was a single-density drive which used a non-DMA controller. The RX02 is a dual-density drive (so it has twice the storage capacity), and it uses a DMA controller. A dual RX02 has two drives for a total of 1.0 MB of on-line storage. The cartridge disk is a higher-performance hard disk, and the removable cartridge allows you to easily switch large data files.

For a $3,600 increase in cost, we could substitute an LSI-11/23 for the LSI-11 in this package and have a processor with 128KB that is almost as fast as a PDP-11/34.

Item		Cost	BMC
PDP-11/03L with dual RX02, LA120			
32 KB memory, FIS chip		11,900	111
Added 33 KB memory		1,450	15
Second dual RX02 (1 MB)		4,150	45
RLV11-AK 5 MB cartidge disk		5,500	58
DLV11-J 4 serial lines unit		500	9
	Subtotal	23,550	238
Software:			
RT-11 included (with MACRO-11)			
FORTRAN/RT11		1,000	
BASIC/RT11		950	
	Subtotal	1,950	
Total System Price		$25,500	

Figure 16.6 An LSI-11-based system.

Pricing Trends

Component costs in computer technology have been dropping ever since the introduction of the transistor in the 1950s. This is a truly remarkable phenomenon. Costs in all other sectors of manufacturing have been rising, yet not in the computer electronics area, in spite of inflation. **IBM** takes pride in illustrating their record over twenty-four years, as shown in figure 16.7.

The fact that component costs go down does not necessarily mean that system prices go down. Vendors prefer to deliver better, faster, larger-capacity systems while holding prices constant. DEC has introduced many models of the PDP-11 since 1970. The relationship of price to capability is depicted in figure 16.8. DEC refers to capability as "functionality."

While hardware component costs are dropping, software costs are increasing. The production of software is labor-intensive (as you well know). So the software "manufacturing" costs have gone up in line with salaries. The customer will see fewer no-charge software tools provided with the computer.

The key to lower cost is in volume production. The first hand-held programmable calculator cost Hewlett-Packard well over $1 million to produce. Their manufacturing cost today must be low indeed if they can profit from selling a better calculator for under $50. This reduction from over $1 million to under $50 is attributable to automated mass production. The effect is most striking in the microprocessor field. An M6800 processor chip could be bought for less than $10 a few years ago. The M6800 reference manual, on the other hand, cost $25. The same chip today is much cheaper, but the manual still costs $25. Volume production of books does not reduce costs by any factor approaching that common in the chip fabrication business.

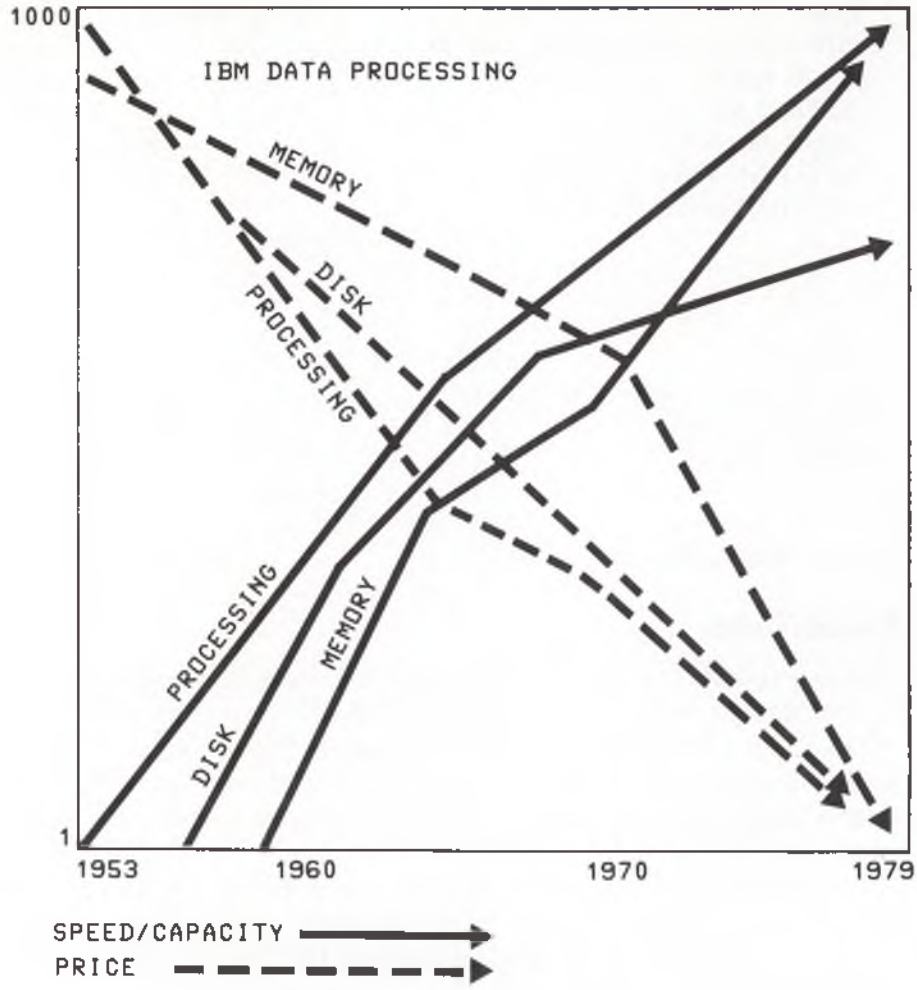


Figure 16.7 IBM component cost decreases.

(Courtesy of International Business Machines Corporation)

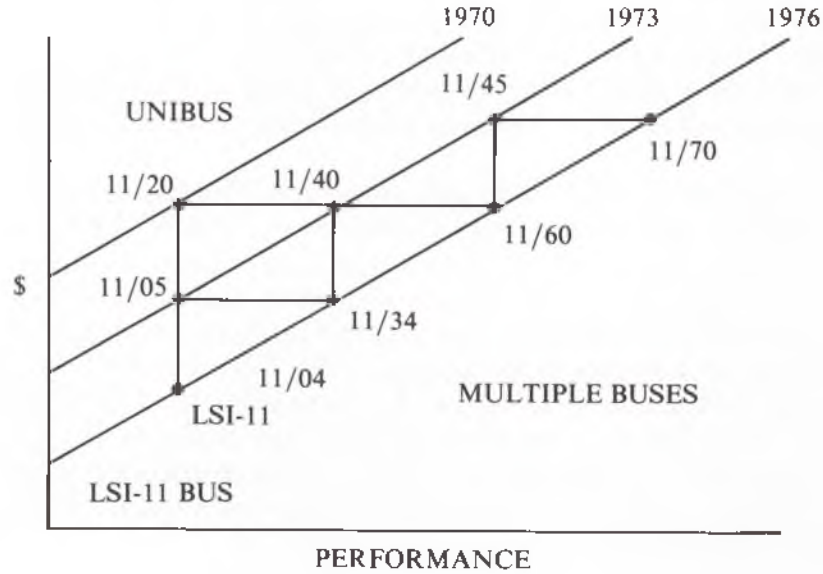


Figure 16.8 DEC functionality chart.

(© Copyright 1980 Digital Equipment Corporation)

It was inconceivable ten years ago that an ordinary person could have a whole computer system at home, for personal use. The cost of automobiles (which individuals take for granted as affordable) has increased over the last ten years from approximately $2,000 to the $6,000–$8,000 range and up. Over the same period we have seen cost reductions in personal computer systems which leave them costing much less than an automobile.

Summary

We have tried to identify the differences from micro-, to mini-, to maxi-computers. The distinctions are somewhat arbitrary, and they change over time. At this time, the clearest differentiation is based on word size and cost. Most micros use an 8-bit word, most minis use a 16-bit word, and most mainframes use a word size of 32-bits or larger. As a general rule, the performance and the cost increase as one goes from a micro to a mini, or from a mini to a maxi.

The micros offer the greatest flexibility in terms of packaging. They also make the greatest demands on a user's technical skills. The larger minis and most maxis come prepackaged as whole systems, ready to use.

Costs for a typical large mini and for a typical small mini were examined. It makes one appreciate the extent to which peripheral devices and terminals contribute a large share of the cost. Pricing trends in computing go counter to those found in almost every other field of endeavor. This has been true for very large as well as very small computers. Hardware component costs will probably continue to decline as new technology evolves. While system costs may not come down, more capability at the same price level will probably be offered. Software costs, on the other hand, are expected to increase.

Exercises

16.1 What percentages of the total system cost are accounted for by the peripheral devices and terminals compared to the software for:

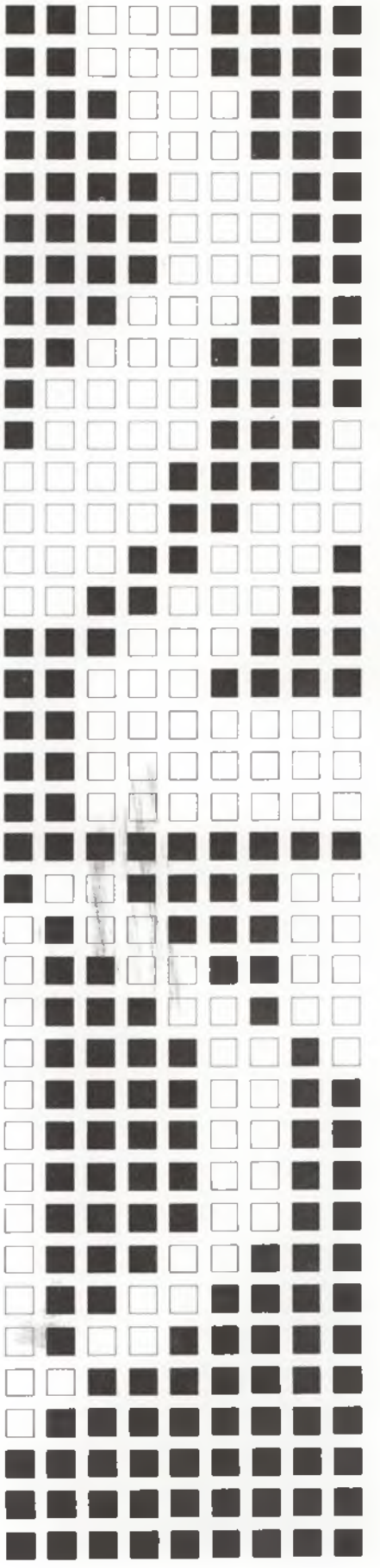
(a) the large PDP-11/70?
(b) the LSI-11?

16.2 The one-time per-user cost for the large PDP-11/70 is $10,000. This is considerably higher than the cost of some personal computers. Identify a specific personal computer and its purchase price for a well-rounded configuration. You can find the necessary information in a Radio Shack, Sears, or Penneys catalog; ads in Byte, the Heath-Zenith catalog, at your local computer store; etc. Discuss the pros and cons of having a single multiple-user system, such as a PDP-11/70, versus having multiple single-user systems. Compare maximum program sizes, file space, performance range, etc.

16.3 Suggest and discuss some approach which combines the best of both worlds, mentioned in problem 16.2. What would it take to implement your suggestion? What new problems would your approach introduced?

17

how does the hardware work



If you wish to understand computing, sooner or later you have to ask yourself, "How does the hardware work?" This is an intriguing question. If you pursue it far enough in one direction, you can make a career of it, in computer engineering. Pursue it further, and you may find yourself studying solid-state physics, or theoretical chemistry. The property of materials, their molecular structure, their behavior under different temperature conditions and in varying electrical fields are at the heart of what makes a modern computer function. Obviously, in one chapter we will barely scratch the surface, but it is surprising how quickly one can get some insight into this subject. Some of the key ideas are very simple and elegant. Do not let the simplicity delude you into thinking that hooking up the components we will describe so they make a harmonious and reliable system is trivial.

Historical Evolution

The earliest machines that we would call "computing machines" were entirely mechanical, and we generally credit Blaise Pascal for having built the first mechanical calculator, in the 1600s. The next major leap forward came in the late 1800s, when electricity and the electromechanical relay came into use. Calculators could now be built using electromechanical relay logic. IBM began building accounting and tabulating machines with relays even before it was called IBM.

Vacuum tubes became commercially available in the 1910s, and by the 1940s vacuum tubes began to replace relays in the building of computers. Figure 17.1 depicts the progress made in electronic components. The transition to vacuum tube computers is credited to the designers of the ENIAC, which was the world's first electronic computer. Machines that used relays were still computers, but they were electromechanical rather than electronic. What difference did that make? Speed: a relay is a gate which can be opened or shut so many times a second. A vacuum tube can simulate a relay; it can implement a gate. Since the vacuum tube has no moving parts, whereas a relay does, the former can open or shut many more times per second. The gate activation frequency (or gate delay) is the fundamental determinant of the computer's operating speed. Hence vacuum tube computers were much faster than relay logic computers.

In the late 1940s a group of scientists at Bell Laboratories invented the transistor. They found that a few transistors could replace a vacuum tube. So they began to implement gates by means of transistors. Shortly thereafter the transistorized computer came into being. Among other properties, transistors are much smaller than vacuum tubes. A transistor is smaller than a pea, while a small vacuum tube is the size of a carrot. Smaller usually means faster, in electronic circuits. Furthermore, the transistor uses almost no power (which is why battery-powered transistor radios became possible), while tubes use much more power, and necessarily get very hot. Heat is the enemy of reliable operation. Going to transistors made computers faster, less expensive, and more reliable.

Transistorized computers use discrete components. You can identify each part. You can see each transistor, each resistor, each capacitor, etc. This means they have to be individually mounted by hand to form the desired circuit. This is an expensive and error-prone, labor-intensive operation. The challenge was to reduce the number of discrete components.

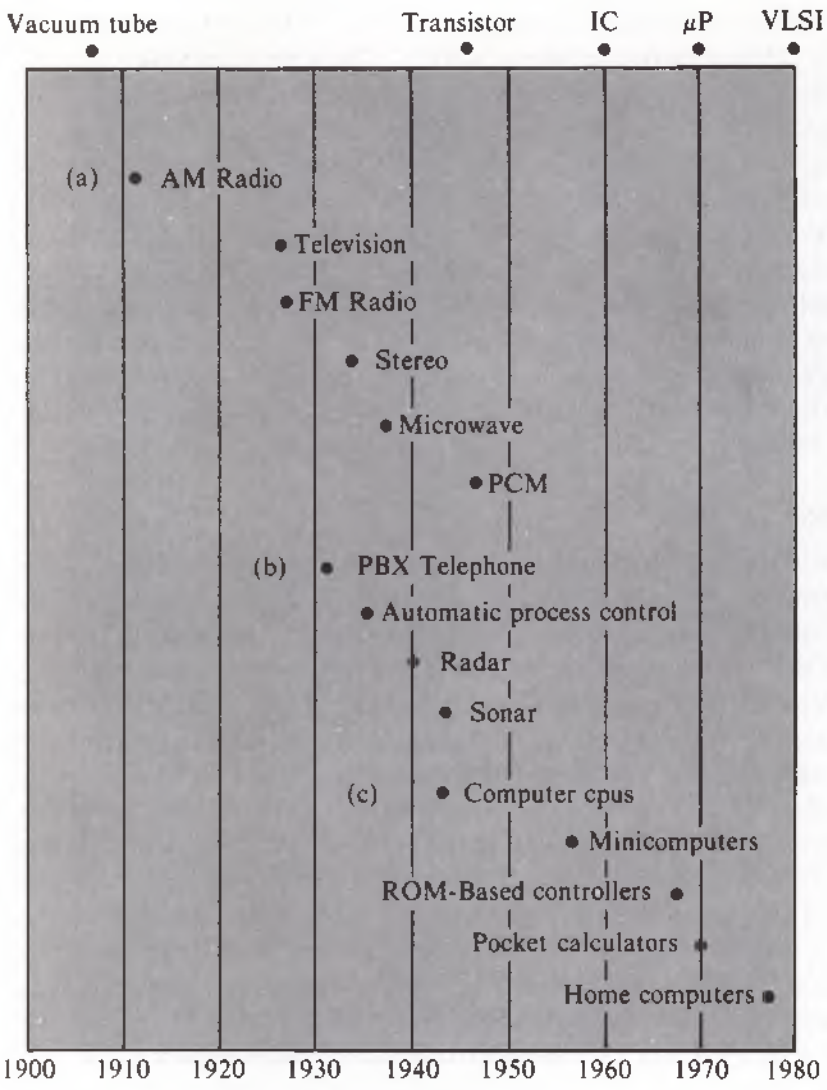


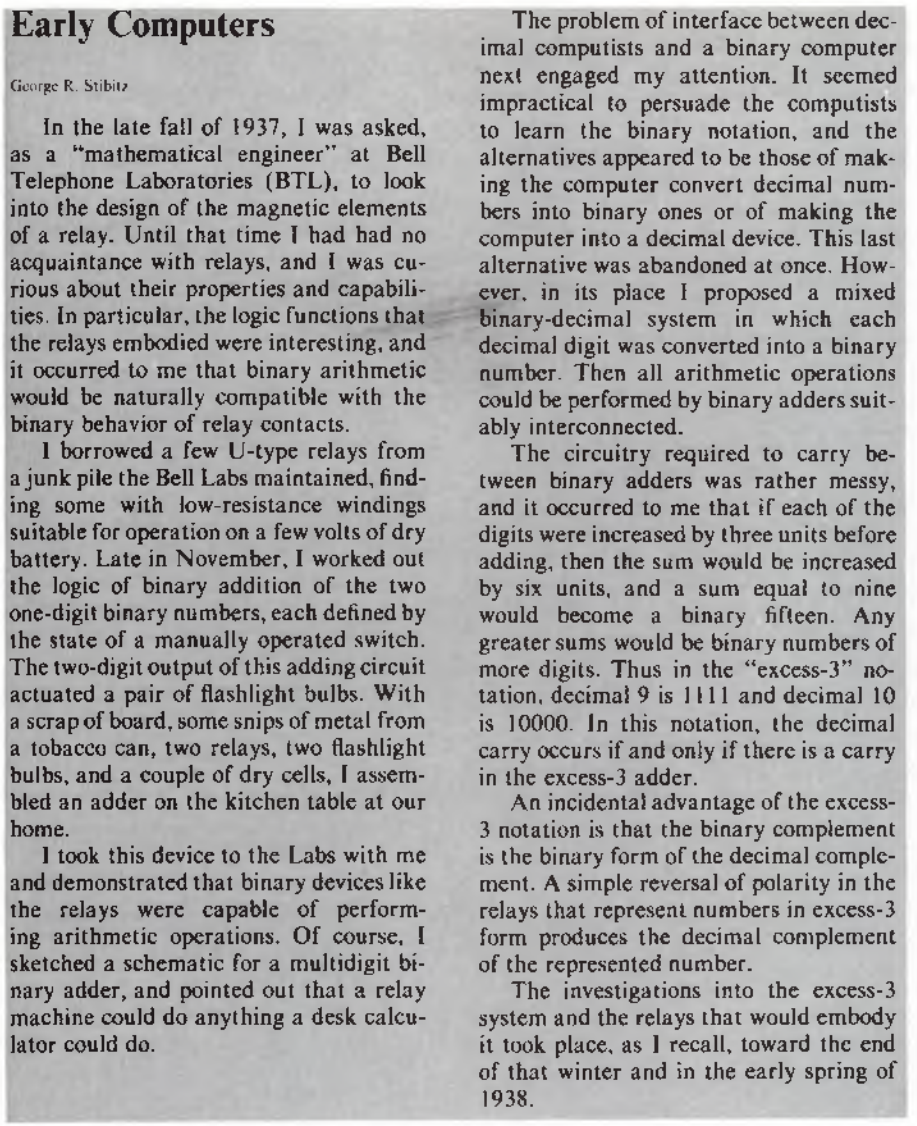
Figure 17.1 The component revolution—from vacuum tube to VLSI—atop the continuum line providing historical scale: (a) represents the *communications* revolution, (b) illustrates the *control* revolution, and (c) depicts the *computation* revolution.

(Reprinted from *Micro-Electronics* by Millman, 1979, McGraw-Hill)

The response to the challenge came in the early 1960s with the advent of the first integrated circuit (IC) chips. A single IC replaces hundreds or thousands of discrete components such as transistors and resistors. Since then the technology has progressed remarkably, from the initial small scale integration (SSI) to MSI (medium scale integration) to LSI (large scale integration), and we are now on the threshold of VLSI (very large scale integration). Computers such as an LSI-11 or a PDP-11/24 are built with some LSI chips and many MSI chips, plus some discrete components.

Speed, reliability, and cost have improved at every stage in this evolution because of a number of interrelated factors: (1) smaller components; (2) fewer components; (3) fewer interconnections; (4) less power use; (5) less heat dissipation.

We could try to describe how the hardware in a PDP-11 works by describing how each chip works and then describing how each chip interacts with all other parts of the hardware. We would not get much insight from this approach. It makes more sense to describe the functions a computer must perform and to describe some hardware that can implement each function.



Early Computers

George R. Stibitz

In the late fall of 1937, I was asked, as a "mathematical engineer" at Bell Telephone Laboratories (BTL), to look into the design of the magnetic elements of a relay. Until that time I had had no acquaintance with relays, and I was curious about their properties and capabilities. In particular, the logic functions that the relays embodied were interesting, and it occurred to me that binary arithmetic would be naturally compatible with the binary behavior of relay contacts.

I borrowed a few U-type relays from a junk pile the Bell Labs maintained, finding some with low-resistance windings suitable for operation on a few volts of dry battery. Late in November, I worked out the logic of binary addition of the two one-digit binary numbers, each defined by the state of a manually operated switch. The two-digit output of this adding circuit actuated a pair of flashlight bulbs. With a scrap of board, some snips of metal from a tobacco can, two relays, two flashlight bulbs, and a couple of dry cells, I assembled an adder on the kitchen table at our home.

I took this device to the Labs with me and demonstrated that binary devices like the relays were capable of performing arithmetic operations. Of course, I sketched a schematic for a multidigit binary adder, and pointed out that a relay machine could do anything a desk calculator could do.

The problem of interface between decimal computists and a binary computer next engaged my attention. It seemed impractical to persuade the computists to learn the binary notation, and the alternatives appeared to be those of making the computer convert decimal numbers into binary ones or of making the computer into a decimal device. This last alternative was abandoned at once. However, in its place I proposed a mixed binary-decimal system in which each decimal digit was converted into a binary number. Then all arithmetic operations could be performed by binary adders suitably interconnected.

The circuitry required to carry between binary adders was rather messy, and it occurred to me that if each of the digits were increased by three units before adding, then the sum would be increased by six units, and a sum equal to nine would become a binary fifteen. Any greater sums would be binary numbers of more digits. Thus in the "excess-3" notation, decimal 9 is 1111 and decimal 10 is 10000. In this notation, the decimal carry occurs if and only if there is a carry in the excess-3 adder.

An incidental advantage of the excess-3 notation is that the binary complement is the binary form of the decimal complement. A simple reversal of polarity in the relays that represent numbers in excess-3 form produces the decimal complement of the represented number.

The investigations into the excess-3 system and the relays that would embody it took place, as I recall, toward the end of that winter and in the early spring of 1938.

Figure 17.2 Reprint from Stibitz.

(Reprinted from *A History of Computing in the Twentieth Century*, by George R. Stibitz, 1980, Academic Press.)

The simplest imaginable computer would use words of 1 bit. Its ALU would process 1-bit operands, store 1-bit values, etc. If you were given such a computer, you could program it to perform multiple-precision arithmetic; in fact, it could be programmed to do anything any other computer can do. So let us see how we could store, transmit, process, and compare single-bit words. Then you can visualize how newer technology could be used to simulate the older technology we will describe. Speeding up our design would also be done by using more parallelism (i.e., 2-bit words instead of 1-bit words, etc.). Some of this flavor is found in the work described by Stibitz in figure 17.2.

Functional Elements of a Computer

What functions must be supported in a computer?

1. Store a bit.
2. Transmit a bit.
3. Combine two bits to produce a result.
4. Compare two bits.

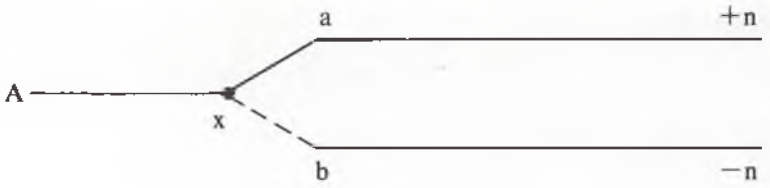


Figure 17.3 Storing a bit.

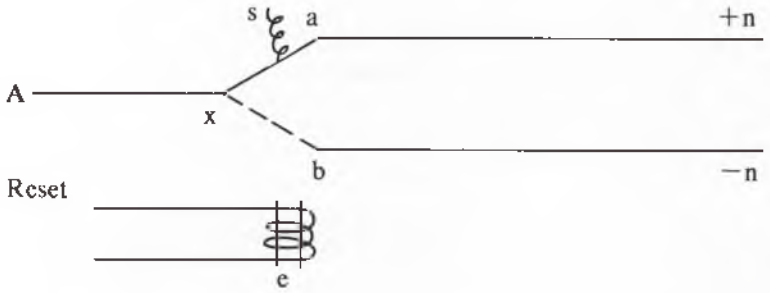


Figure 17.4 Using a relay to store a bit.

We discussed how you can transmit bits when we examined serial communications. In the case of a computer, you can eliminate the framing bits. Since all the parts of our computer will fit into one box, we can use a common clock for the sender and the receiver of the bit being transmitted. The only purpose of the framing bits was to provide for self-clocking in the absence of a common clock.

Storing a Bit

We can consider storing a bit in two ways: one volatile, one nonvolatile: Volatile storage—by means of vacuum tubes, transistors, or integrated circuits—can be simulated by an electromechanical relay. The basic idea is to associate voltage levels in a particular range with a logic "1" or a logic "0". For instance, if a voltage V exceeds a particular threshold of +n volts, we will interpret that as a "1" bit. If the voltage V is below a threshold value of −n volts, we will take that to be a "0". Any other voltage has an undefined logic value.

We can depict storing a bit by using a gate as shown in figure 17.3. The arm which pivots at point x can be either in the position which connects A to a, corresponding to a "1", or in the position which connects A to b, corresponding to a "0". Some control mechanism at point x will determine which position the arm is set at.

A relay implements this function by means of a spring (attached to the arm, pulling it toward point s) and an electromagnet e (see figure 17.4). When an electric current flows through e, a magnetic field is induced. This magnetic field pulls the moving arm toward e and it holds it in this position so long as the current flows through e. Loss of current causes the arm to spring toward s, possibly "forgetting" the bit it was storing by virtue of its position.

Sending a current through the "Reset" input lines will move the arm down and hold it in the "0" bit position. When no current flows through e, the arm position defaults to a "1" value. What we have here is a volatile 1-bit storage device. We can transform it into a 1-bit nonvolatile storage device as shown in figure 17.5. Our previous 1-bit storage device has no difficulty "remembering" a "1" by virtue of the spring s. We need some way of holding the moving arm in the "0" position, even if we suffer a loss of power. Part 1 is a "latch" which will hold the arm in the down

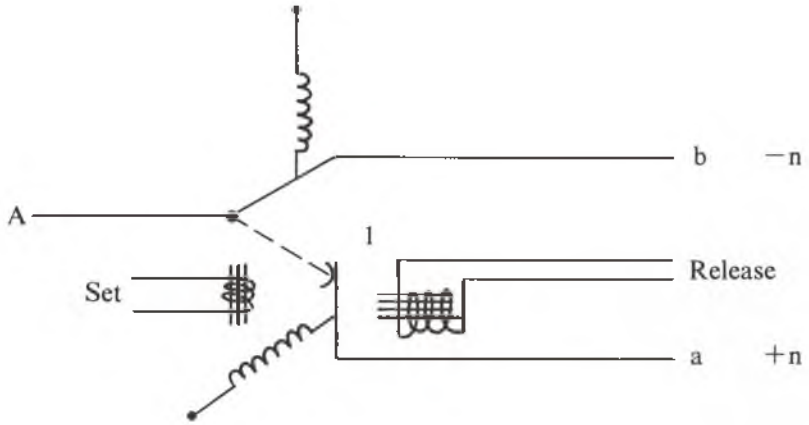


Figure 17.5 One-bit nonvolatile storage device.

Figure 17.6 Sequence of bar magnets.

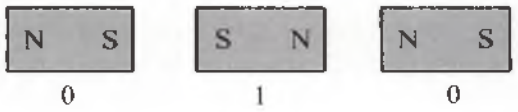


Figure 17.7 Magnets arranged to "store" the three bits "010."

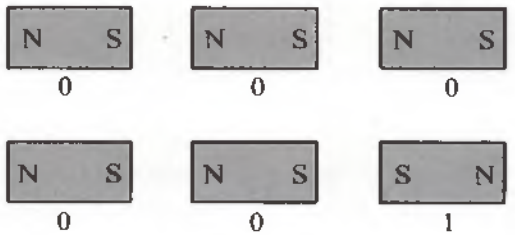


Figure 17.8 Magnets arranged to "store" bits "000" and "001."

position whenever the electromagnet with "Set" inputs is pulsed to lower the moving arm. The electromagnet with the Release inputs will pull the latch-arm back when it is pulsed, releasing the arm so it can be pulled back into its default "1" setting. Clearly what is "0" or "1", and what the default settings are is an arbitrary decision. Once you make it, you must abide by your choice consistently.

Most large-capacity nonvolatile storage devices use ferromagnetic recording media. The basic idea is that the kind of coating material used on one surface of a magnetic tape or a magnetic disk allows you to create and orient miniature magnets in the material's surface as desired. If you had a sequence of bar magnets, as in figure 17.6, you could by arranging them in one pattern or another, cause them to store one bit pattern or another. With three magnets we can "store" the three bits "010" by arranging the magnets as in figure 17.7. The patterns in figure 17.8 store the indicated bits; and so on. The key idea is: when a read-head containing a coil of wire moves along a line of magnets, it intercepts the magnetic field emanating from each magnet. This causes a current to be induced in the read-head coil. The direction of the current reflects the orientation of the magnet. Thus we can "read" the bits we have "stored." It is understood that the spacing of the magnets is well defined; it is directly related to the recording density.

Electricity and magnetism have a marvelous symmetry. You can convert an electric current into a magnetic field; that is what makes a relay work. That is also the basis for "writing" on magnetic media. A sequence of electric currents flowing through the device's write-head, which is a small electromagnet, determines what pattern of magnetic spots shall be written on the recording medium.

The converse operation, converting a magnetic field into an electric current, is the basis for all our heavy-duty electric power generators. It is also the basis for reading the patterns on the tape or disk. When a magnetic field passes through a coil of wire, or vice versa, the field induces an electric current in the coil. The direction of the current is related to the orientation of the magnetic field. With this instant background in electricity and magnetism, we can see that magnetic-media-based recording and playback devices have the following attributes:

1. The write-head uses an electromagnet to create magnetic patterns on the recording medium.
2. The read-head uses a coil to sense the magnetic patterns.
3. The recorded information is not volatile.
4. The recorded information can be erased and rewritten.
5. The recording medium and the read/write-heads must be in relative motion in order to transfer information (i.e., the tape must move, the disk must spin).

The magnetic recording technique described is very close to the one used when recording at 800 bpi; it is called NRZI (non-return-to-zero-invert). At 1,600 bpi, another recording technique is used; it is called phase modulation (PM). Yet another technique is used at 6,250 bpi; it is called GCR (group code recording). Each successive technique has better reliability characteristics but requires more sophisticated and costly hardware support. The basis for all these techniques is found in the writing and sensing of magnetic patterns.

Building an ALU

Let us consider building a simple ALU. It will be able to compute the logical AND, the OR, and the arithmetic sum of pairs of single-bit inputs as well as compare two operands. If we have a supply of relays, we can use them as gates, as depicted in figure 17.9.

We have simplified the diagram by not showing the electromagnet controlled by the signal x. The x control signal can close the moving arm. When x is not signaling for the arm to be pulled down, the arm will by default be pulled up (thanks to a little spring). An input at point x will "close" the gate, allowing a signal to flow from A to B. By using two gates— the first controlled by x, the second by y— we can build an "x AND y" circuit, as in figure 17.10. Given inputs x and y, the moving arms will both be pulled down if and only if x and y are both set. This circuit thus implements the logical AND. A logical OR is just as simple.

If either or both of x and y are set, one or both of the paths from A to B will be established. The circuit in figure 17.11 therefore implements "x OR y".

Figure 17.9 Relays used as gates.

Figure 17.10 An "x AND y" circuit.

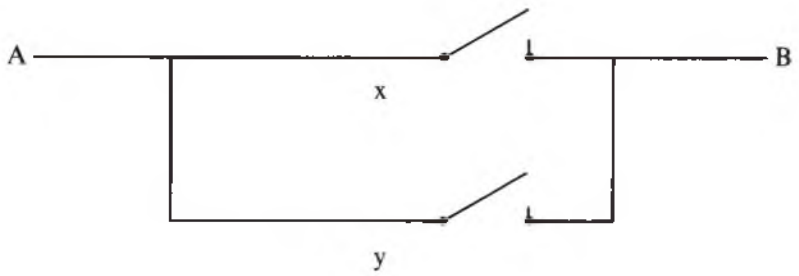


Figure 17.11 An "x OR y" circuit.

x	y	Comparison	
0	0	Same	
1	1	Same	
1	0	x high	Not-equal
0	1	x low	Not-equal

Figure 17.12 Basis of a comparator.

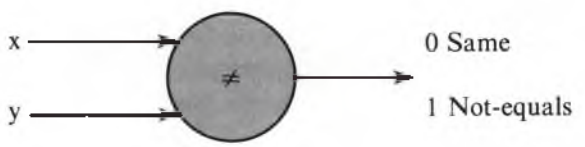


Figure 17.13 Exclusive-OR.

x	y	x∧y	x∨y	x≠y	x+y
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	10

Figure 17.14

A Comparator

What is the basis for a CMP? Given inputs x and y, we have figure 17.12. A comparator has two inputs and three possible outcomes: same, x high, x low.

If x and y are the same, then an exclusive-or would produce a "0", which we can interpret as a "same" signal. So we have figure 17.13. If we get a "1" output, we can use x itself to be the outcome of the x-high or x-low test. We leave it to you to see how you can build a ≠ using relays.

Performing Binary Arithmetic

Recall the definitions for some of the logical functions. Let us write them side by side, next to the definition for the arithmetic "+", as in figure 17.14.

Figure 17.15

x	y	x + y
0	0	0 0
0	1	0 1
1	0	0 1
1	1	1 0

Figure 17.16 A half-adder circuit.

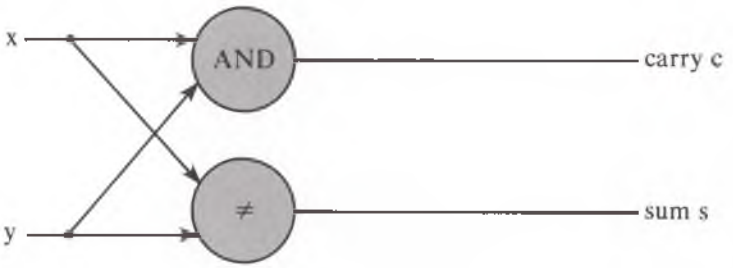
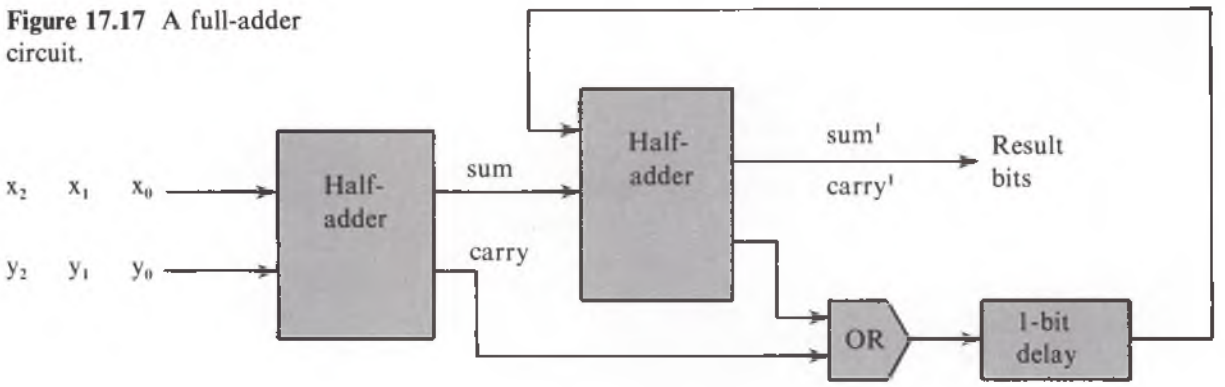


Figure 17.17 A full-adder circuit.



If we rewrite the "+" definition, we can see how to proceed. The right column of the "+" output in figure 17.15 is obtained using a $\neq$; the left column is obtained using an AND. We can build our circuit as in figure 17.16.

Such a circuit is called a half-adder. For our 1-bit word computer it is a complete adder. If we were building a larger, faster computer, we could "cascade" two half-adders to build a full-adder, as shown in figure 17.17.

The design of high speed parallel adders is still under active investigation. There are amazing techniques to reduce the carry ripple delays. But we must move on to other matters.

Sequencing

The harmonious interaction of the hardware elements we have examined requires that their activations be sequenced. The simplest mechanism used for sequencing is found in common household machines such as washing machines and dishwashers. The user selects a desired operation (e.g., heavy-duty wash cycle). This starts a master clock ticking; as time marches on, a timing disk rotates very slowly. At each instant, this timing disk is determining which of the components of the machine shall be active.

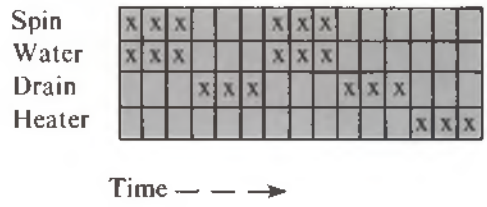


Figure 17.18 Timing chart.

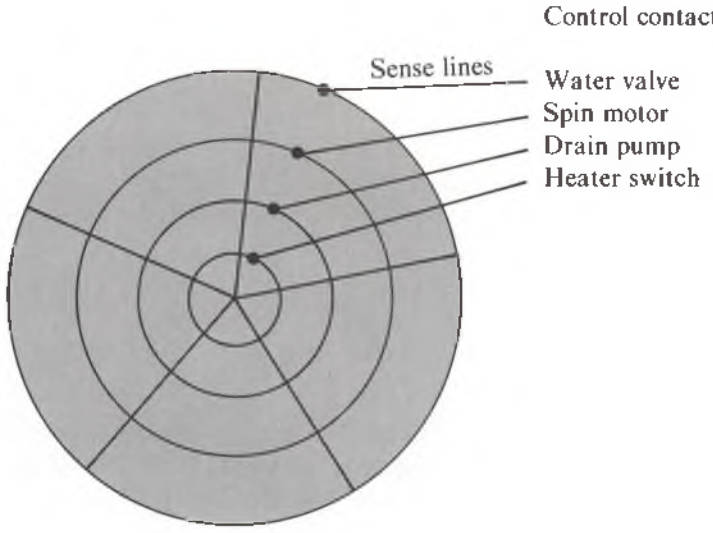


Figure 17.19 A timing disk.

Consider the sequence of steps as depicted in the timing chart in figure 17.18. The timing depicted in this chart can be implemented physically by a timing disk with four concentric circles, each circle associated with a particular physical machine component that is eligible to be activated independently of other components. Along each circle, conducting metal foil corresponds to each x'd part of the chart. This allows a control signal to flow to the desired component when and only when its portion of the timing disk corresponding to the x'd part of the timing chart is under the control sensor contacts. As time progresses, the timing disk shown in figure 17.19 rotates, and the control sensor contacts pass on the desired control signals in accordance with the timing chart.

Each sequence of basic operations in a computer has associated with it a timing chart which indicates the sequence of gate activations required to implement the desired operation. In principle, each of these sequencing activities could be implemented as a timing disk. Of course, present-day computers don't use timing disks, but conceptually their operations can be described in these terms.

The sequencing control signal can open or close gates to block or pass along the desired component activation signals, merely by having the sequenced activities both go through an AND gate.

From Relays to Tubes

A relay can implement a gate, and with the right combinations of gates, you can build ANDs, ORs, etc. From these you can build everything else needed in a computer's ALU. It follows that if a vacuum tube could simulate a relay, it could implement a gate, and the rest follows as before, except that now we have an electronic ALU instead of a slower electromechanical ALU.

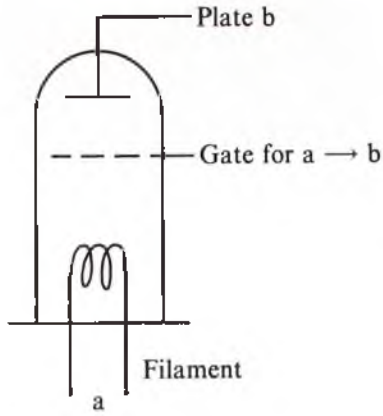


Figure 17.20 A vacuum tube used as a gate.

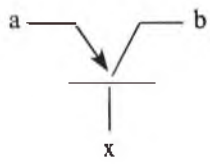


Figure 17.21 A transistor.

A vacuum tube, depicted in figure 17.20, can simulate a gate as follows. A current will flow through the gas in the vacuum tube from the glowing hot filament to the plate, provided that the signal current at the gate allows it to do so. A small gate current can control a much larger filament-to-plate current.

Tubes to Transistors

A transistor simulates a vacuum tube. Transistors are depicted in figure 17.21. A small current at x can open or shut the gate, controlling the flow of current from a to b.

This is admittedly a very sketchy glimpse into how the hardware works. The bibliography provides good references on this topic, many of which do not require that you first get a degree in electrical engineering or physics. Do not by any means imagine that the key concepts in electronics are beyond your grasp. On the other hand, do not trick yourself into thinking that designing and building a reliable computer is child's play. Engineers who knew their electronics inside and out have built computer hardware that self-destructed because they failed to pay attention to minor details such as air flow for heat dissipation. Building a whole computer system requires expertise in almost every area of engineering, not just in electrical engineering and computer science.

Microprogramming

Some people confuse microprogramming with the programming of microcomputers. The two concepts are independent. What is microprogramming, and why are we concerned with it here? The idea of microprogramming originated in the 1960s when Maurice Wilkes proposed it as a way of designing computers. The traditional way of designing computers had been to directly implement in hardware all the necessary functions. You defined your instruction set, sketched out the circuits to implement each instruction, and proceeded to build the hardware.

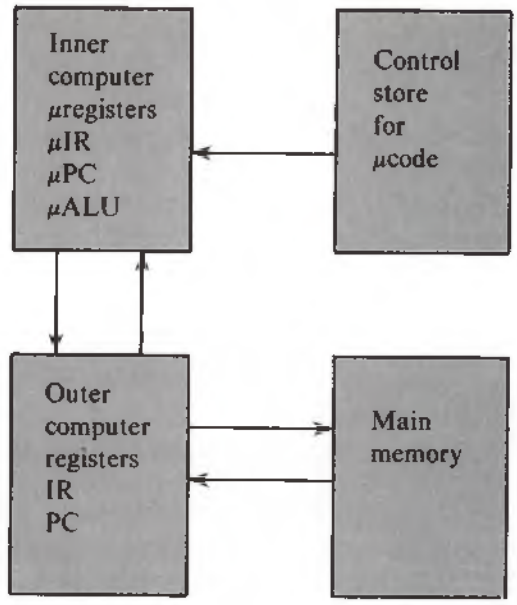


Figure 17.22 Organization in a microprogrammed computer.

Wilkes suggested that it would be better to design and build hardware for relatively simple computers, and then write special control programs which would implement the instruction set of the computer you had set out to build. The end-user of the computer would not even have to be told that the computer was only two-thirds hardware—that the remainder was really software. Since that special software should be protected from accidental destruction, it was usually stored in ROM (read only memory). That made it inevitable that the name *firmware* would be associated with the microcoded software that defines a computer built by this technique. So you now have the organization in a microprogrammed computer as in figure 17.22.

Except for the PDP-11/20 (the oldest PDP-11), *all* members of the PDP-11 family (including the VAX) have been built with an inner computer which interprets the firmware in the control store to implement the PDP-11 instructions. In most PDP-11s, the firmware is kept in ROM. In the PDP-11/60, you can purchase a writable control store (WCS). This allows a privileged user to "define" new instructions or "enhance" old ones by writing microcode sequences and placing them in the WCS. The VAX also has a WCS option. With the LSI-11, the firmware for the floating point instruction set resides in its own ROM chip, called the FIS/EIS chip.

A microprogrammable computer can in principle be made to mimic or emulate any other CPU. You might object, "But I can do the same thing without writing any microcode; I need only write ordinary code." Correct, but if you have a WCS, its access time is considerably faster than that of an ordinary main memory, even when you take the cache speedup into effect. That being the case, a microcoded emulation is much faster. In some cases, programs running in emulation mode run faster than they did on their original hard-wired computers.

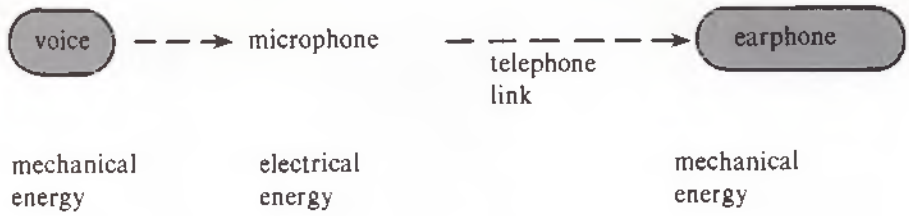


Figure 17.23 Operation of an ordinary telephone.

Figure 17.24 Conversion functions in a telephone–computer-terminal hookups.

Telecommunications, Teleprocessing

We take it for granted that a computer can be used at a distance, thanks to having it connected to a telecommunications network. Teleprocessing is the use of a computer accessed over a telecommunications link. On a personal level, many of us have or will use our home telephones as our link into the largest telecommunications network in the world, the one provided by our telephone companies.

The operation of an ordinary telephone is depicted in figure 17.23. How can something designed to transmit the spoken word be made to transmit bits? The telephone system most of us use is still an analog system. It carries voices by converting them into continuously varying voltage levels. The process involves capturing the mechanical energy generated by our vocal cords and converting this energy into its electrical equivalent by means of a microphone. The telephone lines carry this analog electrical signal to an earphone, which reconstructs the original audio signal. Microphones and earphones are analog-to-analog signal converters which transform mechanical energy in the form of vibrating air into electrical energy, and vice versa, respectively.

Telephone Dial-Up Access

If you wish to have a computer or a computer terminal communicate over the telephone system with another computer or computer terminal, you need a pair of converters at each end. Each converter itself implements two conversion functions, as shown in figure 17.24. When such a converter is connected to the telephone by virtue of an audio signal, it is called an acoustic coupler. These are very convenient for people to use, since no wiring into the telephone system is required. Some converters skip the audio conversion entirely. They generate the electrical signals acceptable to the telephone systems directly, and know how to convert incoming telephone electrical signals back into bits. Such converters are called modems, and they must necessarily be electrically connected to the telephone system. In telephone terminology, a modem is called a "data set," and the modem connection to your CRT or CPU is called its "business machine interface." The modem's other connector, linking it to the telephone system, is called its "switched network interface." In the meantime, the telephone book yellow pages no longer carry a "business machine" listing.

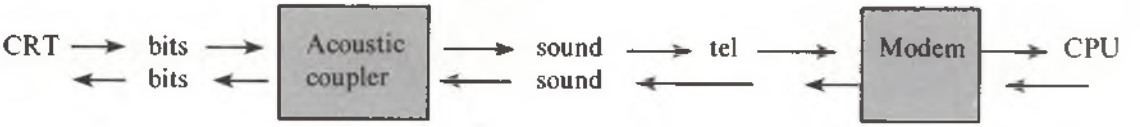


Figure 17.25 A modem to access a computer.

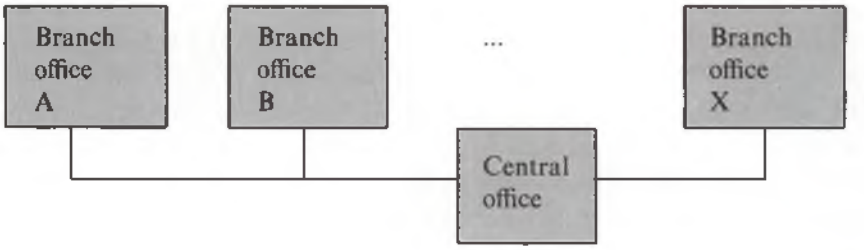


Figure 17.26 A multibranch company.

This direct connection used to be a nuisance. Now that many telephones are equipped with modular jacks, it is easy to hook up a direct-connect modem. The typical use of an acoustic coupler to access a computer with a modem is depicted in figure 17.25. The most popular acoustic couplers are limited to operating at baud rates between 110 and 300 baud. A few acoustic couplers can operate at 1,200 baud. Direct-connect modems can operate at 1,200 and 2,400 baud over ordinary dial-up telephone lines. As you might expect, the low speed and high speed modems are incompatible. One newer type of modem, the 212A, can operate at either 300 or 1,200 baud, and if it happens to be answering an incoming call, it will switch itself to the correct speed.

The occasional user of remote access computer services can use an ordinary telephone to connect a CRT to the computer, provided that:

(a) the user has an originate-modem or coupler;
(b) the computer has an auto-answer modem;
(c) the modems are compatible.

Modem compatibility means they both operate at the same speed and they are both asynchronous (or both synchronous). An auto-answer modem is normally permanently connected into the telephone system. The telephone subscriber is the computer's serial port to which that modem is connected. When an auto-answer modem receives a telephone ring signal, its serial interface interrupts the CPU, and the software driver is activated so that full-duplex communication may take place.

When you are using a hard-wired CRT, the fact that you turn it off may have no significance for the CPU. You probably have a dedicated port, and you can use it as much or as little as you like. With telephone access, particularly on a dial-up basis, there may be a great deal of contention for the telephone line. For instance, you might have a company with many branch offices, as shown in figure 17.26. Each branch office may have a minicomputer equipped with an auto-dialing modem. Each night they automatically call the central office so they can transmit a summary of the day's transactions into the central office's large file system. At the same time, they can pick up messages, requests, etc. that have been waiting for them on the central office CPU. Each branch mini

could be programmed to call into the main CPU at a given time, and to keep trying each fifteen minutes thereafter if it gets a busy signal. In this way, a single telephone line into the main CPU and only one modem would suffice. As a general rule, you would put two lines and two modems in, and program the minis to know what the backup number was.

It is important for a CPU to know that a telephone line is still in use, particularly if there happens to be no traffic on it at a particular time. How does the CPU know the caller is still on the line? This information is provided by the telephone carrier signal. As soon as two modems are communicating, a high-pitched whistle sound is established and maintained until either side disconnects. If at any time loss of the carrier signal is detected by a modem, it will wait a short time (say, 5–10 msec). If the carrier is not reestablished by then, the modem at the CPU end will trigger an on-hook condition, terminating the telephone call. This simultaneously frees the telephone line and interrupts the CPU, so it can either initiate a "logout-the-previous-user-of-this-telephone-line" procedure or try to reestablish the call by using its auto-dial modem to make a call.

Automatic Speed Detection

When you dial up a computer which has a multiple-speed modem, how do you inform it that you wish to operate at a particular speed? Suppose the computer is using a 212A modem which can operate at either 300 or 1200 baud—what can be done to select the speed you want?

There are two widely used conventions. One of them has the computer assume that you are calling in at 1200 baud, and so it sends you a login message at 1200 baud. If you are in fact operating at 1200 baud, then you will be able to read the message and you will do the computing you wish to do. If you are not operating at 1200 baud, then the message you just received will be badly garbled—you won't recognize any part of it. That is your cue to push the BREAK key. As you recall, the BREAK key does not send an ASCII code; it sends a long pulse—that makes it speed insensitive. So the CPU will always recognize a BREAK signal, even when the modem speeds do not match. The CPU uses the BREAK signal when you are in the initial stage of establishing communication as a request to try the next available speed. And so the CPU will repeat the previous message, at the other modem speed. With a two-speed modem, you should now both be running at the same speed. This technique can be used with more speed selections, simply by using successive BREAK signals to cycle through the possible speed selections.

The other technique requires that the user take the initiative in sending to the CPU. Some agreed-upon character or character-string, say, UVWX, must be sent by you to establish the communications link. The CPU by default operates its modem at its top speed. If you sent UVWX while running at the same top speed, then CPU will receive exactly the UVWX you sent, and proceed to the next stage in letting you compute. If you sent UVWX at a lower speed, the CPU would get quite a different bit pattern, but one which is predictable. It will look up the pattern it received in a table from which it will retrieve the corresponding speed, then condition its modem to run at this speed, and send you a message to proceed with your computing.

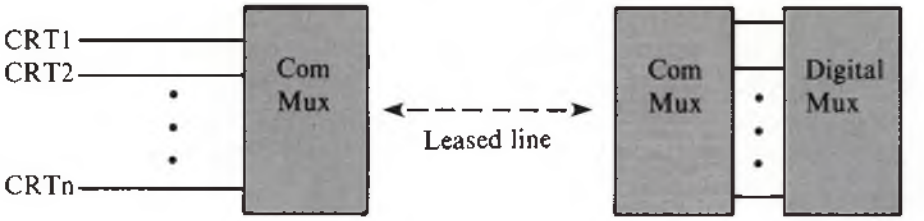


Figure 17.27 Use of a communications multiplexor.

Leased Lines

Heavy users of remote access services will most likely want a dedicated communications link. These can be leased from the telephone company and other firms licensed as common carriers. As a general rule, you will not only get a dedicated line, you will get a higher baud rate and fewer transmission errors. Since the cost of leased lines is not negligible, it makes sense for several CRTs at one site to share the leased line going to some remote CPU. This can be done by using a communications multiplexor. Such a mutiplexor differs from multiplexors which connect directly to a CPU (e.g., the DH-11AD we saw on the PDP-11/70 configuration). The latter implements a set of serial interfaces. The other allows many CRTs to share one leased telephone line. You would use a communications multiplexor as shown in figure 17.27. The CPU is not directly aware of the presence of the pair of communication multiplexors. So far as the CPU is concerned, it interacts with each remote CRT as if each had its own modem and dedicated communication line. Once again we see a service being provided in a transparent fashion.

The "start-stop" asynchronous communication we saw earlier is fine for relatively low speed traffic on either hard-wired or dedicated lines. But as soon as you begin sharing a leased line, you want to get the most out of it, and you notice that the start-stop bits are wasting your time and money. So you proceed to eliminate them in favor of having a common clock for the sender and the receiver, and you no longer just send a character at a time. It makes sense to send groups of characters with a synchronized communications technique. In *synchronous* communications you always transmit a group of characters. This usually corresponds to either a whole line of text or a whole page (screenful). The message format is previously agreed upon, and it looks like:

SOM <header> message bytes <trailer> EOM

No framing bits are used. Instead, what you might call "framing bytes" are used, with one set at the start of the message and another at the end. Once message transmission starts, it proceeds at a uniform rate until all the bits have been sent; it is transmitted synchronously. Many more information-bits per second can be transmitted now that we have eliminated the pair of framing bits we used to attach to each byte. ASCII uses the SOM and EOM codes to designate start-of-message and end-of-message, respectively. The header can include identification of the originator, while the trailer can include a message check-sum.

Data processing systems tend to operate in a one-screen-at-a-time fashion. Filling out the blank fields on the screen corresponds to filling out a form, and this is called a transaction. After any local (off-line) editing to correct keyboarding errors has been performed, the "enter" key on the CRT can be pressed. This raises a flag telling the CPU that the given CRT has a transaction ready to send. If the CPU gives it the signal, the CRT will transmit its entire screen in a synchronous fashion. More often than not the CRT can distinguish between prerecorded information, which defines the "form" on the screen, and those items that have been keyed in. Only the keyed items need be transmitted. This saves a fair amount of line time.

Line-sharing can take place at another level. When we were discussing vectored interrupts, we dedicated an interrupt vector to each device. Upon receipt of an interrupt signal, the CPU knows immediately which device wants service. In data processing you have to balance line cost and interface costs with adequate response time. If you are using synchronous CRTs, they are by definition buffered (they have adequate memory to store a full screen of text). So it makes sense to have many synchronous CRTs sharing one line into the CPU. Upon receipt of an interrupt, the CPU can poll the line to find out which CRT needs service. The interrupt response time is clearly longer, but it is usually adequate (unless you have too many heavily used CRTs sharing one line).

Historically, asynchronous communication has been popular on scientific and engineering computer systems, while synchronous communication has been popular on data-processing-oriented systems.

Communication Interfaces

We are accustomed to seeing two kinds of electrical power plugs in North America. The older type accommodates a two-prong plug. The newer type accommodates a three-prong plug (power plus ground). Fortunately, the newer one is upward compatible with the older one. In almost every respect, the ANSI standard for the communications interface it designates as RS-232C is analogous to the power plug standards. The standard specifies how many prongs shall be provided for (in this case they are so delicate they are called pins). A meaning is assigned to each pin (all 25 of them) and the physical layout (pin-to-pin spacing) is specified, as are the voltage levels. The RS-232C standard prescribes mechanical, electrical, and logical requirements. What does it have to do with computing? That standard is the most widely supported interface standard you will find on CRTs, serial printers, serial computer interfaces, modems, couplers, etc.

The RS-232C standard is slowly being supplanted by the backward compatible RS-422 and RS-423 interconnect standards. The newer standards provide for much longer cable lengths than had been anticipated when the older standard was set up. With the old standard, you cannot count on having much over a few hundred meters of cable (twisted pairs) between your CRT and the serial interface. The new standard allows this distance to be much longer. In principle, RS-232C supports only distances of 50 feet at 20 Kbaud, and less at higher speeds. The RS-422 and RS-423 interface standards will support distances as long as 4,000 feet, or data rates as high as 10 Mbaud. The backward compatibility allows existing systems to provide for a graceful upgrading over a period of time.

The only other common asynchronous serial interface standard is the old one established by the model 33 Teletype. It uses a 20 milliampere current loop interface. Just like a carrier signal, loss of current for whatever reason is detected by the sender and receiver immediately. Your TTY will begin to chatter until you reestablish the broken communication link or turn off your TTY. This was a good way of detecting breaks in communication lines due to storms, acts of war, etc. When an interface is simply called an EIA interface, this is an incomplete specification (EIA stands for Electronics Industries Association). More likely than not, the RS-232C interface is being referred to.

Modem Control

When we discussed character I/O we looked at a simple serial interface. Its control-status-register (CSR) had just a few status bits, because we were dealing with a hard-wired terminal (i.e., one which is directly connected to the interface).

When your serial interface is connected to a modem which is connected to the telephone system, you need an enhanced CSR, at extra cost. If you install a serial interface with an RS-232C interface, you must still specify if you want the modem controls included. If you do, then the CSR will have a number of extra status bits, such as ring-detect and carrier-loss, so you can detect what caused an interrupt on that serial interface.

Networks

We briefly discussed a small laboratory network in chapter 15, without describing how the several computers were physically linked together. One simple way involves point-to-point connections between each pair of computers you wish to link. If ultra-high speed communication is not required, then you could use a pair of asynchronous serial interfaces for each such point-to-point link. If the distances are just a little too much for RS-232C interfaces, then you could use a pair of private-line modems (or line drivers) to meet your distance requirements. You could get a higher baud rate, if needed, by using synchronous interfaces instead. Finally, you could consider putting in sufficient wire to handle 8- or 16-bit transmission in parallel (16 or 32 wires) plus the parallel interfaces. This is an expensive proposition (about $5,000) and you would be limited to distances of about one kilometer. What do people do when setting up nationwide or international computer networks?

There are many large-scale computer networks in use today. Many large corporations have their own computer networks. The communications network appropriate for in-building use is not appropriate for a geographically dispersed network. The former is called a local network. We will briefly discuss particular communication schemes appropriate for each of these kinds of networks.

Packet Switching

When you use a telephone, you are requesting the services of a circuit-switched network. Each time you make a long distance call, a unique circuit with many links is switched into being for your exclusive use for the duration of your call. This is deemed a very poor use of expensive communication circuits when it is being used for computer-to-computer communication, where the digital traffic tends to be very unevenly distributed. It comes in bursts, with many long pauses. Given this bursty traffic, it was natural to begin looking for a better way.

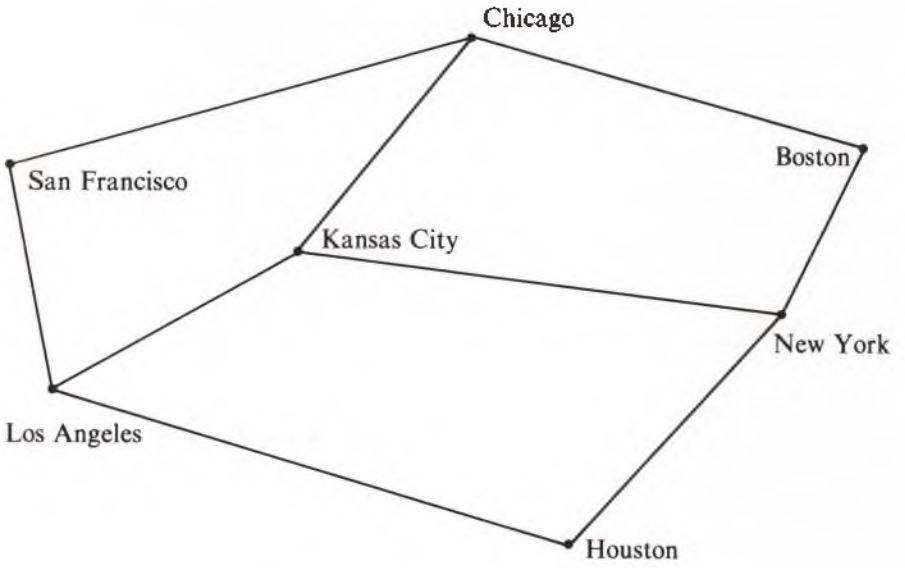


Figure 17.28 A packet-switching network.

The ARPANET, sponsored by the Department of Defense's Advanced Research Projects Agency, pioneered the use of a communications scheme now known as *packet switching*. The computers on such a packet-switching communications network do not use point-to-point links to reach each other. Nor do they use a switched network. Either of these would be too expensive. Instead, the computers share a communications network in which every computer can reach any other over at least two different paths. When CPU A sends a message, or a whole file, to CPU B, it passes the message to its network interface controller, which is called an IMP (interface message processor). The IMP breaks up the message into a series of packets of approximately 1 Kbits each. These packets are routed in whatever way is deemed best at that instant. If a message is broken up into several packets, each packet could conceivably travel over a different path. It is the responsibility of the IMP at B to reassemble the message originally sent from A, or to find out why part of it was lost, negotiate with IMP-A to retrieve the missing part, etc. in such a way that the "path" from A to B appears to be error free.

The common speed used on packet switching systems is 50 Kbaud. An IMP is an active device, usually a minicomputer of the power of a PDP-11/34 or better. If an IMP fails, the network is broken at that point. This is the reason for having alternate paths.

There are commercially available public-packet-switched networks. One of the largest is called TELENET, which operates as a subsidiary of GTE (General Telephone and Electronics). It has over 400 CPUs of all makes connected to it, located in over 250 cities in North America and abroad. The computers do not belong to TELENET, but to the customers who pay TELENET for the privilege of being on the network.

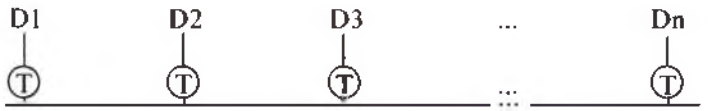


Figure 17.29 A local network.

Local Networks, Ethernet

If you wish to link together all the word processors, digital copiers, telex terminals, laser printers, minis, micros, and mainframes in your building, you should look into using something other than a point-to-point system, for obvious reasons. A switched-circuit system probably does not make much sense either. Is packet-switching appropriate? Not if you need a PDP-11/34 acting as an IMP for each machine you want to connect. You should look into using the same kind of cable the cable television companies use. This is a coaxial cable (coax for short); it has a thick conductor in the middle and is shielded on the outside. Whereas an ordinary twisted pair cannot reliably be used to send bits much faster than 50 Kbaud at distances of 1–2 kilometers, a coax cable can support 10,000 Kbaud at such distances.

If you call each device you wish to connect into your local network D1, D2, . . . , Dn and use an interface T (called a tap) to connect each device to the coax cable, you have something like figure 17.29. The device side of each tap must be compatible with the device's communications interface. The taps here play almost the same role as did the IMPs. The difference is important, however . The IMPs had to keep track of which links were operative or inoperative; which links were heavily used or underused, instant by instant. Here we assume the coax cable is indestructible, so we don't have to worry about the message routing problem. A tap is passive. If a tap fails, only the device connected to it is cut off from the network. A message to be sent from device i to device j is broken up by the device's software driver into packets. The packets are broadcast by the tap, and all the other taps receive it. Tap j should recognize that it is being sent a packet. Tap j will copy it and hand it over to device j. Presumably, every device is "smart enough" to support the packet assembly/disassembly process.

The Ethernet system developed at Xerox works along these lines. It has stirred such interest that DEC, Intel, and Xerox are working together to design and manufacture Ethernet tap interfaces. Connecting to an Ethernet-like local network will then be as simple as plugging in a toaster (you hope). The subject of computer networks, local and distributed, is a vast one. Some knowledge of how they work is essential to understanding computing.

Trends

In many countries the telephone service is provided directly by the government. In the U.S.A. the government provides postal service but not telephone service. In France and England the government provides postal, telephone, telegraph, radio, and television services. In the U.S.A., telephone service is provided as a regulated industry, with each provider having a monopoly in its service area. Each state in the U.S.A. has a public service commission which oversees the quality and cost of telephone services.

Figure 17.30 Xerox Ethernet.

(Courtesy of Xerox Corporation)

THE LEADING EDGE

#1 in a series of reports on new technology from Xerox

About a year ago, Xerox introduced the Ethernet network—a pioneering new development that makes it possible to link different office machines into a single network that's reliable, flexible and easily expandable.

The following are some notes explaining the technological underpinnings of this development. They are contributed by Xerox research scientist David Boggs.

The Ethernet system was designed to meet several rather ambitious objectives.

First, it had to allow many users within a given organization to access the same data. Next, it had to allow the organization the economies that come from resource sharing: that is, if several people could share the same information processing equipment, it would cut down on the amount and expense of hardware needed. In addition, the resulting network had to be flexible; users had to be able to change components easily so the network could grow smoothly as new capability was needed. Finally, it had to have maximum reliability—a system based on the notion of shared information would look pretty silly if users couldn't get at the information because the network was broken.

Collision Detection

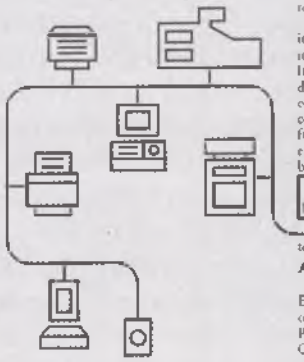
The Ethernet network uses a coaxial cable to connect various pieces of information equipment. Information travels over the cable in packets which are sent from one machine to another.

A key problem in any system of this type is how to control access to the cable: what are the rules determining when a piece of equipment can talk? Ethernet's method resembles the unwritten rules used by people at a party to decide who gets to tell the next story.

While someone is speaking, everyone else waits. When the current speaker stops, those who want to say something pause, and then launch into their speeches. If they *collide* with each other (hear someone else talking, too), they all stop and wait to start up again. Eventually one pauses the shortest time and starts talking so soon that everyone else hears him and waits.

When a piece of equipment wants to use the Ethernet cable, it listens first to hear if any other station is talking. When it hears silence on the cable, the station starts talking, but it also listens. If it hears other stations sending too, it stops, as do the other stations. Then it waits a random amount of time, on the order of microseconds, and tries again. The more times a station collides, the longer, on the average, it waits before trying again.

In the technical literature, this technique is called carrier-sense multiple-access with collision detection. It is a modification of a method developed by researchers at the University of Hawaii and further refined by my colleague Dr. Robert Metcalfe. As long as the interval during which stations elbow each other for control of the cable is short relative to the interval during which the winner uses the cable, it is very efficient. Just as important, it requires no central control—there is no distinguished station to break or become overloaded.



The System

With the foregoing problems solved, Ethernet was ready for introduction. It consists of a few relatively simple components:

Ether. This is the cable referred to earlier. Since it consists of just copper and plastic, its reliability is high and its cost is low.

Transceivers. These are small boxes that insert and extract bits of information as they pass by on the cable.

Controllers. These are large scale integrated circuit chips which enable all sorts of equipment, from communicating typewriters to mainframe computers, regardless of the manufacturer, to connect to the Ethernet.

The resulting system is not only fast (transmitting millions of bits of information per second), it's essentially modular in design. It's largely because of this modularity that Ethernet succeeds in meeting its objectives of economy, reliability and expandability.

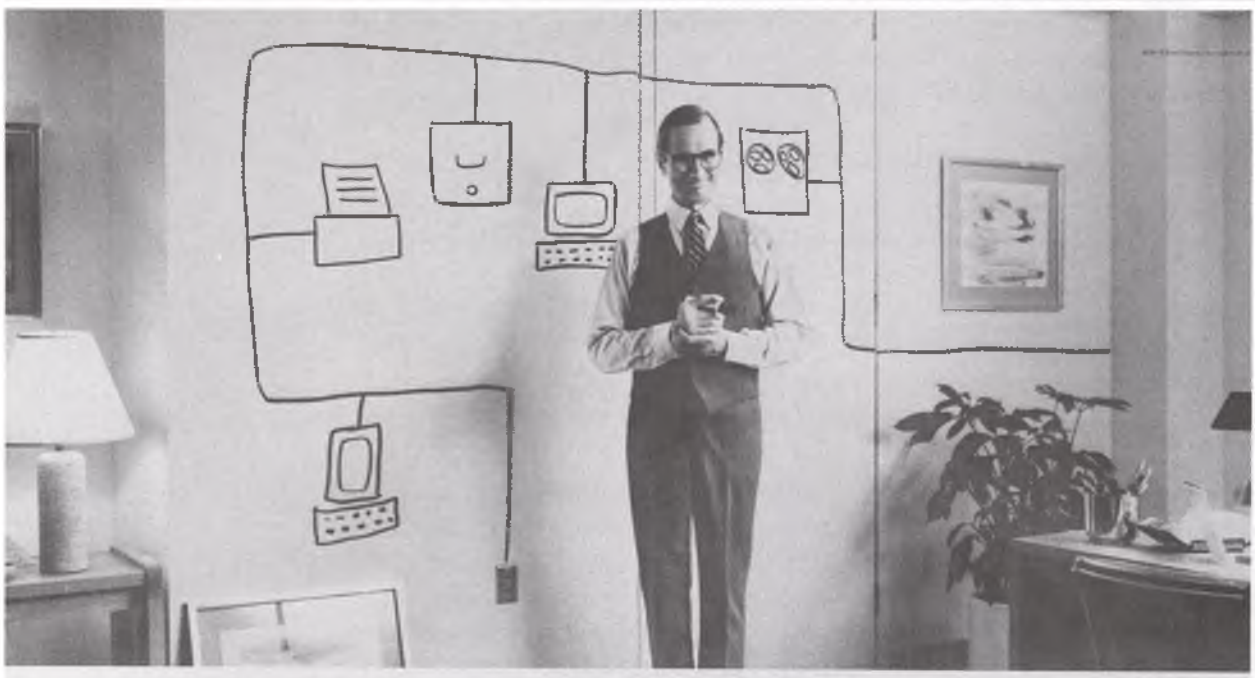
The system is economical simply because it enables users to share both equipment and information, cutting down on hardware costs. It is reliable because control of the system is distributed over many pieces of communicating equipment, instead of being vested in a single central controller where a single piece of malfunctioning equipment can immobilize an entire system. And Ethernet is expandable because it readily accepts new pieces of information processing equipment. This enables an organization to plug in new machines gradually, as its needs dictate, or as technology develops new and better ones.

About The Author

David Boggs is one of the inventors of Ethernet. He is a member of the research staff of the Computer Science Laboratory at Xerox's Palo Alto Research Center.

He holds a Bachelor's degree in Electrical Engineering from Princeton University and a Master's degree from Stanford University, where he is currently pursuing a Ph.D.

XEROX



Xerox introduces the Information Outlet.

If you're wondering how business will handle information in the 80's, the handwriting is clearly on the wall.

We call it the Information Outlet—a new way for you to custom design an information management system that will give you maximum flexibility with minimum expense.

Here's how it works:

The Information Outlet gives you access to a special Xerox Ethernet cable that can link a variety of office machines, including information processors like the Xerox 8010, various electronic printers and files, and, of course, computers.

The Xerox Ethernet network will enable people throughout your company to create, store, retrieve, print and send information to other people in other places—instantaneously.

This network wasn't designed to work exclusively with our equipment. Other companies' products can be connected as well.

As your needs change, so can your network. You'll simply plug in new machines as you need them—or as technology develops better ones.

So, through the Xerox Information Outlet, you'll get to the future the way the future itself will get here.

One step at a time.

XEROX

If you'd like more information on the Information Outlet, write us and we'll send you a booklet: Xerox Corporation, P.O. Box 470165, Dallas, Texas 75247.

At the national level, the Federal Communications Commission (FCC) of the U.S. Government regulates all forms of electrical or electronic communication:

1. telephone
2. telegraph
3. radio
4. television-broadcast
5. television-cable
6. microwave communications
7. satellite communications

You may wonder what all of this has to do with computing? The FCC has come to have more day-to-day influence on computing than any other federal agency (outside of the Department of Defense, because of its huge needs). The interaction among federal agencies, private enterprise, and court rulings has led the world of computing to the threshold of a new era. For many years the FCC held that the telephone companies, especially the American Telephone and Telegraph Company (ATT) were limited to providing communications services exclusively. Similarly, while the computer companies (notably IBM) could provide computing equipment and services, they could not provide communication services. The FCC has now reversed itself on the separation of computing and communication services, with consequences that are hard to foresee. The news articles in figure 17.31 describe some of the implications, as seen from various points of view (e.g., that of a major newspaper and publisher). Keep in mind that the leading newspapers and publishers are already heavily computerized, and national papers like the *New York Times* use satellite communications to publish. Readers in the Midwest have their copies printed in Chicago, thanks to a satellite ground station linking the Chicago printing press to the New York main office.

Remote Diagnosis

The practicality of remote access to computers is now so well established that the computer manufacturers are using it to alleviate what is from their point of view a critical problem.

Most medium-to-large computer systems are maintained in good working order by contracting for maintenance services, usually but not necessarily provided by the computer's manufacturer. Most such contracts call for service during the business day. Seeing that each customer gets prompt service—without a large maintenance staff—is a real challenge. This is particularly true because people skilled in computer maintenance are in short supply.

Some computer service firms have set up telephone hot lines to allow their customers to contact diagnostic centers which are staffed twenty-four hours a day. The customer with an ailing computer system may call the hot-line number and advise the staff of the nature of the problem.

Figure 17.31 New York Times editorial and Time article.

(Ma Bell giving birth—© 1979/80 by The New York Times Company. Reprinted by permission. New Baby Bell—Copyright 1980 Time Inc. All rights reserved. Reprinted by permission from *Time*.)

Ma Bell giving birth

Machines that communicate can no longer be clearly distinguished from those that compute and calculate, and therein lies a fascinating political problem. Just as I.B.M. and other computer companies are rapidly moving into communications, so the mammoth American Telephone and Telegraph Company is straining to get into the computer and information business. To a degree, that's good for both A.T.&T., which would otherwise be damaged by new competitors, and the nation, which can benefit from its technology.

The trouble is that A.T.&T.'s efficient telephone services are best provided by a monopoly, requiring state and Federal regulation, whereas the computer business is best managed by a free market. So how can A.T.&T. be both bound and free? Only if its protected arm is kept clearly distinct from the competitive arm. Easily said, very hard to achieve.

While rivals began challenging its monopoly in the business of voice transmission, A.T.&T.'s expansion into new markets was long kept in check. When it signed a consent decree in 1956 to settle an antitrust suit with the Department of Justice, the company agreed to confine its data-processing activities to the needs of its conventional operations; hence it could not sell exotic computer terminals to private homes or compete for corporate computer and data-processing business.

But now the Federal Communications Commission, sensing the need for change, has lifted its controls and left it to Congress to establish a new order. And while Congress stalls, A.T.&T. is rushing to create not so tiny "Baby Bell" for computers and information-processing, hoping to give birth before Congress can control the offspring's dimensions.

Unless constrained, Ma & Baby Bell would quickly gain unfair advantages, even over large competitors like I.B.M. They would also move into different businesses, like selling information to the home in competition with newspapers, and into bill-collecting in competition with banks. Competition sounds American, but a competitor armed with the capital and technology of a protected monopoly could easily run rivals off the road.

With its huge financing and research powers, an uncontrolled Ma Bell could subsidize Baby Bell and massively undersell competitors in computers and information. It could also give its subsidiary special access to the A.T.&T. system, while treating other companies as hostile aliens.

With a newspaper's bias, we are particularly concerned that A.T.&T. not become a full-fledged information enterprise. Static ads in the Yellow Pages are one thing; up-to-the-minute computer listings of goods and prices would directly challenge other media in many markets. With computers that talk to each other, A.T.&T. would give the nation a valuable new communications system, beneficial to all purveyors of information. But if A.T.&T. were allowed also to own information that moves on the system, it could hardly be trusted to welcome competing traffic. Only if clearly confined to *distributing* information will A.T.&T. have the incentive to service the largest possible number of information peddlers.

Congress has not so far dealt thoughtfully with these questions. The public interest lies in fair competition, in truly free access to any A.T.&T. computer network and in a multitude of information suppliers. The bill prepared by the House Commerce Committee would bar "Baby Bell" from becoming a "mass medium," but not from "data retrieval." It is a fuzzy distinction. And while Congress dawdles, A.T.&T.'s state-by-state experiments with new devices make future regulation ever more difficult.

The nation would benefit from a healthy telecommunications industry. A.T.&T. belongs in that business, but only if it isn't allowed to become a predator.

New Baby Bell

The phone company shake-up

Ma Bell is expecting. The announcement came last week as A T & T, the world's largest corporation, shuffled its management and company organization to prepare for the birth of an independent subsidiary. The new offspring, nicknamed Baby Bell, will battle IBM, Xerox and GTE's Telenet for supremacy in the burgeoning computer communications market. It will begin life as a giant with about 15% of Bell's $122 billion in assets, and by 1985 Baby Bell may be doing $10 billion worth of business.

The reorganization of A T & T is part of the Federal Communications Commission's plan to loosen Government control over the communications industry. Last April the FCC ruled that the phone company could enter the field of computer communications, from which it has been

previously barred. But at the same time, Washington required that A T & T separate its new computer operations from its traditional telephone services. The new corporate structure will result in two almost separate companies, one to handle the regulated phone business and Baby Bell to sell the rapidly expanding array of equipment, from picture phones to computers, that tie in with the telephone lines. The FCC contended that without the separation, Bell could have used profits from its regulated phone service to subsidize new computer products and thus unfairly undercut its competitors.

In order to simplify splitting itself in two, Bell is planning to gain more central control over its empire. It will spend $1 billion to buy out minority shareholders in four Bell System companies not already entirely controlled, including New England Telephone and Pacific Northwest Bell. The company will also set up a new wholly owned subsidiary, A T & T International, which will control all foreign operations.

The upheavals at A T & T, though, are hardly over. The FCC has ordered the company broken down into two parts by March 1982, but the giant firm claimed that such a deadline was virtually impossible. Said the company in a petition to the commission: "This will be one of the largest corporate reorganizations ever carried out." A T & T's restructuring plan will not be totally carried out for about eight years. The FCC did not say last week whether it would give the company more time. Meanwhile, a computer industry trade group representing more than 50 companies that are fearful of the market power A T & T could have in their business has sued the FCC in an effort to block the birth of a bouncing Baby Bell.

The technician at the diagnostic center will ask the customer to load a designated diagnostic program, and then, by flipping a switch on the computer, enable the remote diagnosis module. The technician then dials up your computer's diagnostic module, feeds it appropriate commands, and examines its responses. Should it be necessary for you to do something, such as load some other diagnostic program, the technician can communicate with you via your CPU's console teleprinter.

On the VAX-11/780, an LSI-11 is used as the remote diagnosis module. The LSI-11 comes as part of the VAX. The diagnostic programs are read by means of the floppy disk drive which comes with this LSI-11. The same floppy disk drive is also used to read in the microcode which defines the VAX's instruction set.

The ability to perform remote diagnosis on an ailing computer in this way is predicated on the reliability of the remote diagnosis hardware. In the case of the VAX, the LSI-11 is much simpler than the VAX and therefore far less likely to fail. The PDP-11/70's remote diagnosis module uses an Intel 8080 microcomputer.

The usual outcome of a remote diagnosis session is a final message such as, "The problem seems to be connected with circuit board such-and-such; have your local service office replace it."

Summary

Computing with the help of machines has been going on for over four hundred years. In the last fifty years we have seen many different technologies used to build calculating machines: relays, vacuum tubes, transistors, and now integrated circuit chips. All of these ways of building computers are based on a few simple ideas. Numbers can be stored as a sequence of bits. Each bit can be stored by means of a circuit which can be constructed in many ways. We chose to store bits by means of relay logic, just as Dr. Stibitz did over forty years ago, because this technique is easier to visualize than is the case with more recent devices.

The basic components of a computer add bits, transmit bits, and compare bits. We have seen simple circuits which implement each of these functions. Relays can be replaced by tubes, tubes by transistors, and so on. At each stage in this substitution the speed and reliability improve. Modern digital computers are based on the simple ideas we have seen.

Microprogramming is a technique which lets computer designers build simpler hardware and provide for sophisticated instruction sets by microcoding them either in writable control store (WCS) or read only memory (ROM).

Remote access to computing service is afforded by use of telephone lines. These services may be accessed on a dial-up basis or by means of leased lines. Computer interfaces to remote terminals need more status bits to indicate the state of the communication process. These are called modem control and status bits.

Computer networks are a generalization of remote access to computing services. We have differentiated between circuit-switched and packet-switched networks. We also distinguished between local and geographically distributed networks.

The interplay between communications and computing is intensifying, as business and governments realize. The control of information is being recognized as a vital resource.

Exercises

17.1 Sketch the schematic for building a NOT gate with relays.

17.2 Sketch the schematic for building an EXCLUSIVE-OR gate with relays.

17.3 How would you implement the 1-bit delay needed by the full-adder?

17.4 Providing dial-up access introduces new kinds of security problems. Briefly describe what these problems are and what measures can be taken to control them.

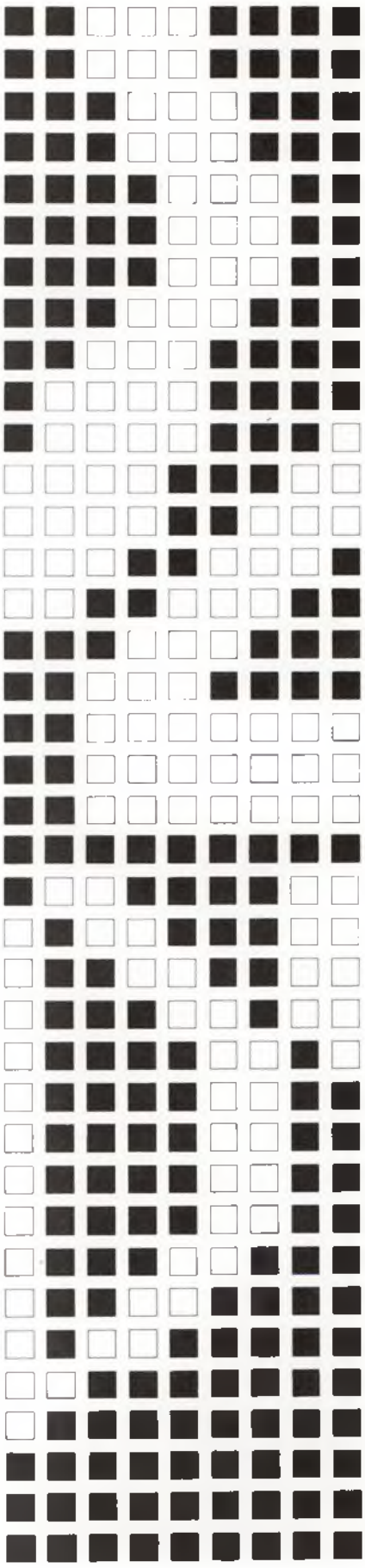
17.5 How efficiently are you using a 1200-baud line when using asynchronous communication with ASCII? How does the efficiency improve (by how much?) if you use synchronous communication with the header and trailer using 20 bytes in 1-KB message blocks? Efficiency here is defined as the ratio of information bits to bits transmitted.

17.6 In some areas of the country dial-up lines tend to be noisy almost all the time. Such noise may cause transmission errors. Suppose the detection of an error requires retransmission. If you are using asynchronous communication, then if a bad character is received, it must be retransmitted. If you are using synchronous communication, a bad character would not be isolated; the whole block has to be retransmitted. At what error rate (1-bit error every n bits) would it be just as efficient to be using asynchronous as synchronous communication, assuming the synchronous format of problem 17.5 is being used?

17.7 Besides introducing security problems, providing dial-up access introduces other operational problems. What if a user is disconnected through no fault of his own? What are the implications? What options can the operating system provide to help? Which specific hardware feature assists in this situation?

18

beyond machine and assembly language



Being able to write assembly-language programs gives one the feeling of great power. Being able to interpret an octal memory dump is an amazing feat in the eyes of the uninitiated. It is clear that any program that could conceivably be written in any high level language can also be written in assembly language. Then what is the point of having and using high level languages?

Looking back at the programs we have written or examined, you could compile a catalog listing all the ways things can go wrong when you use assembly language. The most innocuous typographical error can have literally earthshaking consequences. Why is this so?

The assembler has no concept of what it is you are trying to do. It cannot tell the difference between an array, a tree, a list, or a buffer. You might object and say, "I could enhance the assembler so that it would have built-in data types, and the directives to support them."

Perhaps, but what about the lack of power of expression? It takes almost a dozen lines of assembly code to express A = B + C * (D − 3) / SIN(X) with many opportunities for coding errors. You could respond, "The macro facility could be enhanced to allow infix macros." What we have used with macros has always had the macro invocation *followed* by its operand list. This notation is called *prefix form*. If we could use an *infix form*, then instead of writing

```
BLAH A,B
```

we could define the macro BLAH to be an infix operator, so it could have its two arguments on either side when it was invoked, as in

```
A BLAH B
```

In fact, you should be able to write A=B+C and have that interpreted as a call on the "=" and "+" macros, which would lead to the expansion of the nested macro-invocation "A=B+C" into the machine code

```
MOV B,A
ADD C,A
```

provided that we are dealing with 16-bit integers. There exists portable software which lets you do just this; see the W. Waite entry in the bibliography.

We could carry on this dialog for a long time. For every objection regarding the use of assembly language, there is some technique, some preprocessor, which could be used to respond to the objection. But, in the process, we will have transformed what was a simple (and simple-minded) assembler into a very powerful program which is practically indistinguishable from a good high level language compiler. Why not, then, use the best available tools for the job? For these and many other reasons, high level languages have to a large extent displaced the use of assembly language. Nonetheless, the study of assembly language is still one of the best ways to develop an understanding of computing.

You have to admit that it is easier to learn and use a high level language than it is to learn and use assembly language. The latter requires constant attention to many details. It should be easier to learn a high level language, since a high level language is designed to be used by people. Because a high level language is easier to learn, more people will be trained in its use. That makes it easier for an employer to find someone who knows a high level language than it is to find someone who knows a given assembly language. (By the same token, someone who knows an assembly language is very valuable, should his or her services be required.)

You can be far more productive when using an appropriate high level language. Furthermore, it is much easier to understand someone else's high level language program than to understand someone else's assembly language program. This makes program debugging, maintenance, and enhancement easier, and therefore less costly. I do not intend to review all the reasons why you should use high level languages. The most important reason, from my point of view, is to prevent incorrect programs from being written. When you have to use a computer to get something done, you should use the best available software tools you have access to, and that will almost certainly not be assembly language. A well-designed high level language, with a compiler which has good compile-time error checking and a suitable run-time support package, will help avoid construction and use of programs which are in error.

When writing assembly language, you can write anything that comes to mind, no matter how foolish it may seem to someone else. In a well-designed high level language (e.g., PASCAL), you cannot get away with writing nonsense. You can compare apples to apples; the compiler won't let you compare apples to bananas.

Software has to be correct the first time it is used. Any error which can conceivably be detected at compile time (e.g., adding an integer to a floating point number) should be flagged as an error at compile time. Anything else that cannot be checked until run time should be monitored at run time. If an array subscript cannot be guaranteed to be in bounds, then run-time checking should be enforced. You will object, "Won't this run-time checking slow things down?" Certainly, but what is the point of having a program run a little faster if it is going to generate nonsense?

When a scientist discovers a new fact, other scientists rush to independently repeat the experiment which led to the discovery to see if they too get the same result. If they don't get the same result, the originator is viewed either as a fraud or an incompetent. As you saw in the case of Dr. Feldstein, software is simply too expensive to rewrite. The thought that someone else should write an independent version of a program in order to confirm the correctness of the program in question staggers the mind. (I have in mind the real-time programs which implement the North American Air Defense system at NORAD.) Programs have to be written so they are correct the first time they are used. (You don't get a second chance with the NORAD program.)

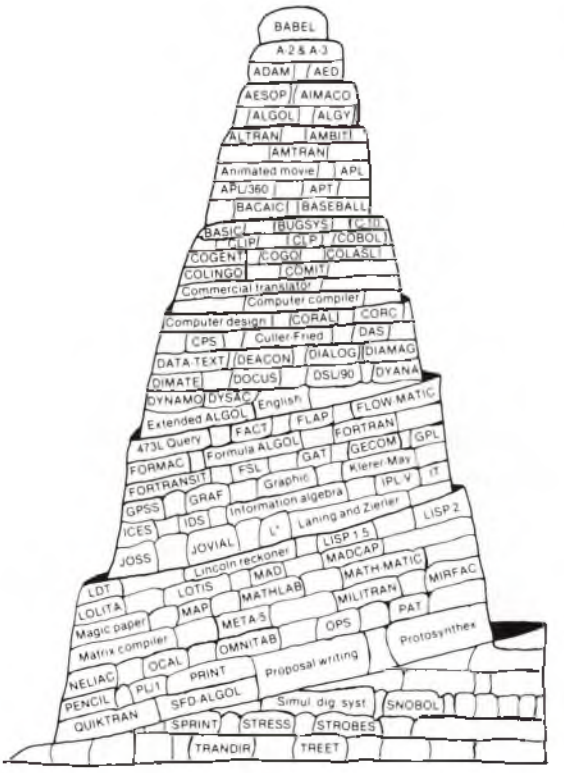


Figure 18.1 The babel of programming languages.

(Concept of the Tower of Babel to represent a larger set of programming languages is due to the communications of the ACM, Vol. 4, no. 1, January 1961 cover, a publication of the Association for Computing Machinery.)

The cost of software is so high that it is far more sensible to link computers together in networks, so that the software which runs on each computer can be shared by all users of the network. This is the kind of computer-network-access that facilitated the research done by Dr. Lawrence R. Klein, who was awarded the 1980 Nobel Prize in economics. He shared his computer program econometric model running at the University of Pennsylvania in return for access to programs running on computers elsewhere in the U.S.A. and abroad.

Other Computer Languages

You have probably used one or more of the following high level languages: APL, BASIC, COBOL, FORTRAN, PASCAL, PL/1. They exemplify a class of computer language known as general-purpose languages. They can be used to implement algorithms to solve almost any kind of problem. COBOL is predisposed to supporting data processing, while FORTRAN's forte is the scientific-engineering kind of problem. Nonetheless, each of the above languages can be used as a general-purpose tool. Figure 18.1 depicts the profusion of programming languages in use in the U.S.A in 1961. Computer languages seem to be created faster than they vanish.

There are other computer languages that are designed to facilitate problem solving in a specific area. Some of these are listed here along with their principal use:

LISP	List processing
SNOBOL4	String processing
FORMAC	Formula manipulation
APT	Numerical tool control

A roster of high level languages in use in the U.S.A. was published in 1969. It listed 120 computer languages in 17 categories. By 1973 the roster had 171 computer languages in 26 categories, after deletion of many languages from the older list. In a single area, defense, the situation is almost chaotic, as evidenced by the news item in figure 18.2.

DBMS

In an ordinary general purpose high level language you have to explicitly describe the algorithm you wish to implement. With an applications-oriented high level language, you are given built-in operators and data structures suitable for that application area; so you merely have to provide the input data in the prescribed format and evoke the predefined operations. This is how most statistical program packages are used.

In all the preceding cases there is an intimate relationship between each program and the data files it uses. The file data format must match the format expected by the program exactly. After a few years a computer facility may have a collection of several hundred programs, each of which use a slightly different file format. That is fine if these programs are used independently of each other, for unrelated problems. What of a computing facility set up to provide service to a single firm? Would it not make sense to allow programs to share data files?

The idea of making data files sharable has matured to the point where data base management systems (DBMS) provide the program interface to data files. Each data item in the data base has a machine-readable description, which is stored on-line in a data dictionary. All programs which use the data do so via DBMS service requests. No user program has any knowledge of the format in which the data are stored other than that provided through the data dictionary. Therefore, all such programs are data-format independent.

As you can imagine, the computing overhead introduced by use of a data dictionary is much higher because of the forced indirection. However, the time necessary to construct a new program that uses data in the data base is reduced by a significant amount. Furthermore, the possibility of easily responding to unanticipated queries is greatly enhanced. This leads us to take one final look at a computing system so we can put the hardware and software in perspective.

The Pentagon's National Military Command Center; Augusta Ada Byron (inset).

Pentagon pins its hopes on Ada; just ask any computer

By RICHARD HALLORAN

WASHINGTON—Ada lives. Not Augusta Ada Byron, the legitimate daughter of the English poet Lord Byron, Countess of Lovelace and, in the mid-1800's, the world's first computer programmer, but the lady's namesake. Some 128 years after her death, she whom Byron poeticized in "Childe Harold" as the "sole daughter of (his) house and heart" has "turned to battle's magnificent stern array," coming alive as a highly sophisticated computer language developed under the auspices of the Defense Department. Testing is scheduled to end today.

Just as human language conveys information and ideas from one mind to another, so a computer language communicates between the human mind and a computer's memory. The language allows a person to put questions to the computer, order it to perform various functions and extract answers from it. In computer jargon, a language is part of the software as compared with the machinery, the hardware.

Ada is one language the Defense Department needs, and needs so badly that though working the bugs out of a computer language usually takes years, Ada's project managers put it through an intensified testing program in 14 months. The completed language was delivered in July. The Defense Department has been refining it ever since, making Ada what department experts consider a remarkable "high-order language," one that closely approximates human language by using recognizable words and phrases in its programming and printed answers. Computations using a binary number code are left to languages of a lower level. In languages like Ada, a single command will initiate a series of operations, much as the general command "walk" will lead a child to perform a series of muscle movements. These languages are easier to learn and use than their lower-level cousins. The New York Times computerized information bank, for instance, has such a high-level language that even the most obtuse reporter can master it in a few hours.

Computer specialists say designing a language is not a mechanical matter, but one of imagination and taste. Ada, they explain, is an advance over earlier languages because its structure—the manner in which questions are put, answers are returned, programs are designed—incorporates the most modern concepts of logic. It takes a highly skilled programmer to write a program in Ada. More important, however, it does not take so highly skilled an individual to use it. And this language can be used in a variety of computers, with minor adjustments. Older languages are keyed to specific machines.

Five years ago, the Pentagon decided to standardize the more than 1,000 languages used by the Defense Department. The annual cost of this software proliferation ran to more than $3 billion last year, not including the cost of training people to use it all.

Figure 18.2 The hopes for Ada.

Figure 18.2 Continued.

For tasks ranging from payroll processing to inventory control, the department had languages. Claiming the largest piece of the software pricetag, however, were the computers that, built into weapons, functioned as integral parts of systems that watch for Soviet intercontinental ballistic missiles, guide patrolling submarines, provide navigation for bombers or relay practical information to commanders on the battlefield as events are taking place. And the department's software problem was getting worse. Beginning with only a handful of computers in the 1950's, the Army is expected to have 13,000 computers in operation sometime during this decade, the Navy 33,000 and the Air Force 40,000.

In 1976, the Pentagon took a big step toward razing the computer Tower of Babel by ordering that only eight languages should survive. A study further determined that settling on one language could save $24 billion between 1983 and 1999. Meanwhile, the military services felt the need for a better language that would incorporate the latest innovations. (Progress in computer hardware almost always outruns developments in software for programming.) The Pentagon also wanted a language that would be accepted by universities, whose faculty members do much of the thinking about military matters, by companies that make military equipment and by America's allies, to improve military coordination.

Instead of forming a committee, the standard Pentagon procedure, the project managers, headed by William E. Carlson of the Defense Advanced Research Projects Agency, held an international competition. Seventeen companies submitted proposals; four were selected as semifinalists. Two companies competed in the final round, the winner being Ada, designed by Jean Ichbiah of Honeywell Bull in Paris, in cooperation with Honeywell Systems and Research Center in Minneapolis.

The military advantage to be gained from Ada is clear. The Soviet Union has built a military machine that relies on numbers and mass in weaponry for its power. To counter that, the United States must rely on speed, mobility and precision firepower—all of which can be improved by the use of the most advanced computer hardware and software, like Ada. As Robert R. Fossum, director of the agency that supervised Ada's development, recently observed, "Military systems of the future will incorporate much more flexible and intelligent control logic."

New weaponry, he said, will have sophisticated devices to detect and identify targets. "Command and control systems will be able to pass target information to fire units very rapidly," he said. "We will have more capable precision-guided weapons than we have today. We also will be upgrading existing tanks, aircraft and other weapons with intelligent digital control systems to extend their useful life."

When the development of the new language began, some dubbed it DoD-1, a name project managers rejected, believing that in the aftermath of Vietnam, the Defense Department initials might have inhibited its use by universities and in the commercial marketplace. J. D. Cooper of the Navy Materiel Command, a project adviser, suggested that it be named for computing's own Countess of Lovelace. Permission was granted by an heir in England.

Lord Byron might have been pleased that his beloved daughter has been memorialized, and so aptly. Augusta Ada Byron once worked for Charles Babbage, a mathematician and inventor of a calculating machine, for which she played computer programmer by preparing operating instructions. Seeing the new Ada, the lady herself may be smiling.

Inner computer	Microcoded CPU
Bare machine	Above + control store + memory
Minimal system	Above + peripherals
Minimal usable system	Above + I/O software
Effective system	Above + operating system
Useful system	Above + compilers + library
Flexible system	Above + DBMS
Application-oriented system	Above + application packages

The next chapter provides readings so that you may examine any of the above views of computing systems, and all the preceding topics, as far as you wish to pursue them. The world of computing is a rapidly changing one, and you would be well advised to find out how you can go about keeping up with developments.

Summary

Assembly language programming provides access to every feature a computing system has. Using assembly-language programming is an excellent way to learn what computers can do and how they do it, but it is too easy to make mistakes with such a low level language. High level languages with error-preventing characteristics should be used whenever possible. There are many application-oriented languages that should be used in preference to assembly language.

There is a class of applications that has led to the development of what could almost be regarded as a new operating system. We are referring to data base management systems. They go a long way toward reducing the opportunities for GIGO (garbage in, garbage out).

Exercises

18.1 For the high level language of your choice, prepare a list of five different programming errors which the language's compiler or run-time system would detect but which would go undetected if you had been using assembly language. For example, the following is incorrect. Why?

```
MOV #100.,%1
ADD X,A(%1)
  ...
A:    .BLKW 10.
```

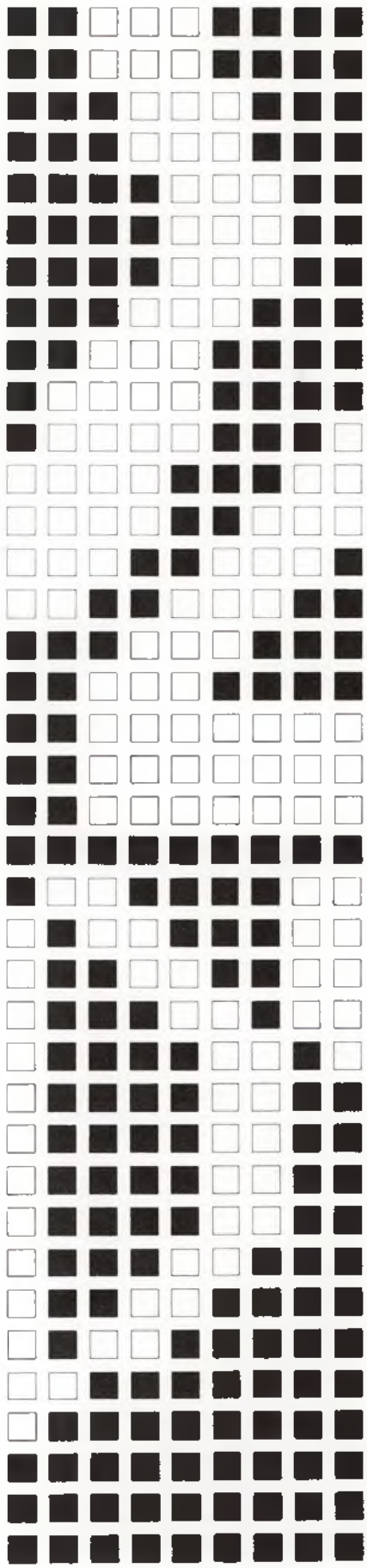
Its counterpart in PASCAL or FORTRAN would be detected as an error at compile time.

18.2 Are there situations which lead MACRO-11 to detect an error, whereas the counterpart in the high level language would be an undetected error?

18.3 Some programs have been in use such a long time that the source code for them can no longer be located. All that remains is the executable binary. It may not even be known if the binary was generated by a compiler or by an assembler. Sketch how you would implement a disassembler. Its input is a binary program image, and its output is a symbolic assembly-language program. What severe problems are you likely to encounter?

18.4 We can generalize problem 18.3 involving the disassembler. Why not have a decompiler? It accepts binary programs and produces a source module in a high level language which, if compiled, would regenerate the original binary code. Sketch an approach to doing this. What severe problems might be encountered?

19



selected readings and annotations to bibliography

Where have we been and where are we heading in our efforts to better understand computing? I hope you have some idea of where we are now, at least in one small segment of computing. This chapter should help you find the literature that explores in depth the history of computing in general and of the PDP-11 in particular. This chapter should also lead you to those journals, periodicals, newspapers, and other sources that will help you keep up with such a rapidly developing field.

Vendor Publications

The public does not realize the extent to which computer manufacturers are involved in the publishing business. IBM, the world's largest computer manufacturer as measured by gross annual sales, is also the world's largest publisher. Each new computer system requires an impressive amount of documentation at many levels (some of which will never leave the factory). The same is true for software or hardware products.

Suppose the product is a new computer system, System X. The documentation support for it can be depicted by a tree, as in figure 19.1.

You should not dismiss the marketing documentation out of hand. Time after time I have found that, were it not for some item in a marketing brochure, I would have failed to notice or appreciate an important characteristic deeply buried in a reference manual. Good marketing publications serve an important educational function.

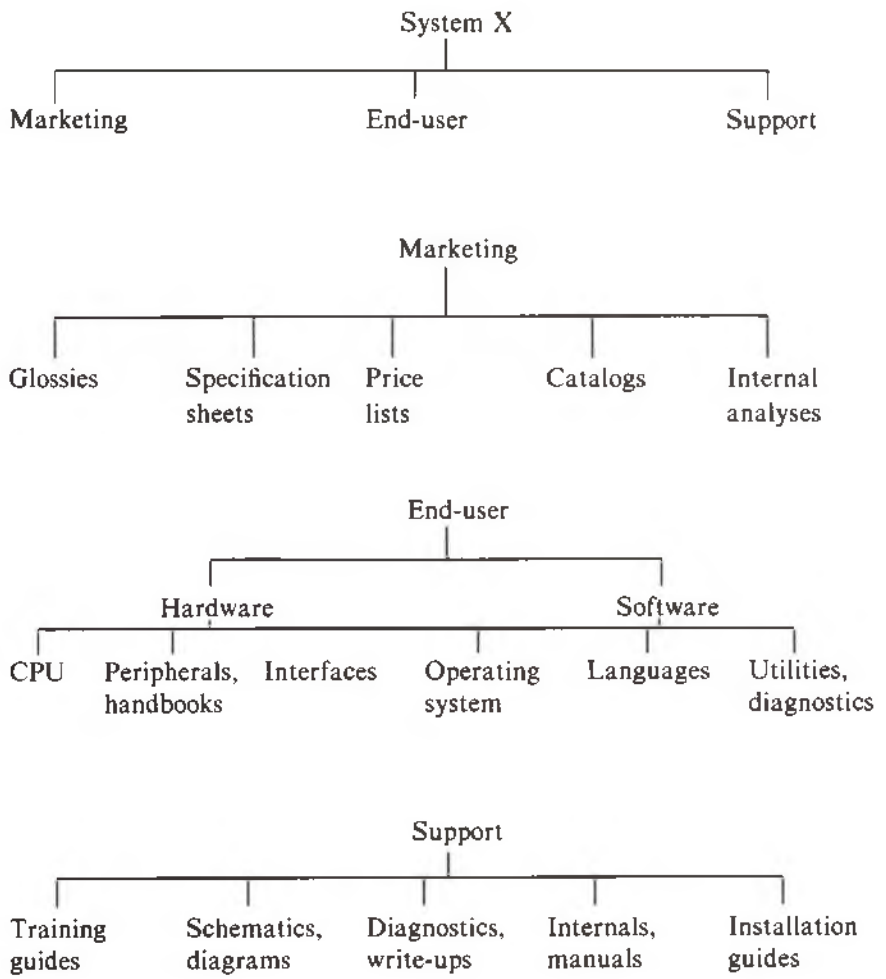


Figure 19.1 A support tree.

Figure 19.2 Documents for one computer system.

RT-11		
	Operating System	$108
	BASIC	37
	FORTRAN	33
	MU-BASIC	41
	Lab Applications	40
	DECNET	22
RSTS/E		
	Operating System	$281
	Programmer documentation	146
	System Manager	40
	System User	62
	Language documents (4)	222
IAS		
	Operating System	$342
	Language documents (6)	256
RSX-11M		
	Operating System	$274
	Language documents (7)	344
	System Logic	295

When the PDP-11 first came out you could hold all the relevant publications in one hand. It is a sign that a product has matured when a complete collection of just the software documentation is as large as we see in figure 19.2. The prices are included for your edification. Note that these are prices for documents alone; the software is not included. Presumably, when you buy the software you get one copy of the relevant publications. You will find that the use of color and shadings to make the manuals easier to read and use makes duplicating them to avoid buying additional copies quite unsatisfactory. In any event, all these publications are protected by copyright laws.

The naive user who buys an entire set of these documents may be shocked to discover a year later that many of the documents are suddenly made obsolete by release of a new version of the operating system. Furthermore, new releases themselves are extra-cost items. A software license applies to a particular release only. The remedy, of course, is to subscribe to a documentation maintenance service for some $200–$300 per year.

The PDP-11 handbooks that you would be most likely to use if you were to be involved in maintaining a PDP-11 system are:

1. Processor Handbook
2. Peripherals Handbook
3. Terminals and Communications Handbook
4. MACRO-11 Reference Manual
5. Relevant operating system guides

The internal logic (i.e., flowchart equivalent) of RSX-11M is described in its systems logic manual. Such is not the case for the other operating systems.

History and Evolution: the PDP-11

The most comprehensive source of information on the origin of the PDP-11 is found in bibliographic item BE 78a. The authors are the actual designers and implementors of the PDP-11. They present the reasons they included the features that were then unique to the PDP-11 and make some attempt to analyze how well their choices turned out. It is a fascinating book, but clearly not for a beginner. Definitely easier to read is [LE 80], organized as a textbook. You should look at it to see how the VAX-11/780 is related to the PDP-11. It was written by the designers of the VAX—for ordinary people, not for other designers. Some of it will repeat what we have been discussing.

History and Evolution: Computing

The oldest professional society devoted exclusively to computing is the Association for Computing Machinery (ACM). In spite of its name, it is concerned primarily with algorithms and software rather than hardware. As an indication of how young the computing field is (i.e., the era of the electronic computer), ACM published the 25th anniversary issue of its monthly journal, *Communications of the ACM (CACM)*, in July 1972. It is well worth looking up. It contains a series of survey articles which look back to the early days of computer hardware and software and tries to assess what real progress has been made and where the major advances are required.

The special commemorative issue for the 50th anniversary of the bimonthly magazine *Electronics* (April 17, 1980) is also well worth looking up. This issue reviews half a century of progress in electronics, of which computing is a significant part. Whereas *CACM* is a scholarly journal, *Electronics* is a trade magazine. You would probably throw away any ordinary copy that is more than a year old. Some of its contents are merely embellished press releases from manufacturers touting their latest products. On the other hand, it is an excellent source of information for impending new hardware products some of which will never reach the marketplace. Each year, *Electronics* reviews what it considers to be the major new products in areas such as computers, communications, microelectronics, etc. From time to time related articles from *Electronics* are pulled together as a book. These books tend to be dated soon after they are published.

The history of computing is well worth looking into. Many ideas which originated in the 1800s and early 1900s still influence the design of the latest computers.

Keeping Up

ACM, described above, produces many periodicals (over twenty), some monographs, conference proceedings, and *Computing Surveys*, a journal you would find particularly interesting. It publishes authoritative tutorial articles on all aspects of computing. Thus, if you wanted to know more about macros, especially how they are implemented on IBM systems, you could read Kent's article entitled "Assembler-language Macroprogramming" [KE 69]. The Presser and White article describes the design of linkers and loaders [PR 72]. Similarly, if you were intrigued by networks, the Kimbleton-Schneider [KI 75] article would bring you up to date as

of the time it was published. Two *Computing Surveys* articles which have become classics are: Denning's "Virtual Memory" paper [DE 70] and Wilkes's survey of microprogramming [WI 69]. Because these tutorials will be used for many years, they provide comprehensive bibliographies, so you could find out which specialized journals could bring you further up to date.

The other major professional society that deals directly with computing is the Institute for Electrical and Electronic Engineers (IEEE). Its major scholarly journal in computing is entitled *Transactions on Electronic Computers.* It has another journal, *Proceedings of the IEEE,* which often devotes an entire issue to computing. The IEEE publication *Spectrum* is a magazine which often has well-illustrated articles on computing or major computer applications (e.g., the national air traffic control system).

The IEEE sponsors the Computer Society, which publishes three monthly journals: *Computer; IEEE Micro;* and *IEEE Computer Graphics and Applications.* As you might expect, the Computer Society is somewhat more hardware-oriented than the ACM. Nonetheless, you can learn a lot about software as well as hardware by reading these periodicals.

Other Periodicals

There are an enormous number of publications relating to computing. Well over 200 books about computing are published every year, and over 50 periodicals deal entirely with computing. There must, as well, be a computer conference every other day which publishes its conference proceedings. You should look at the ACM's *Computing Reviews* to see the abstracts or reviews of thousands of articles and books that appear each month. Once or twice a year *Computing Reviews* publishes the list of periodicals which it covers regularly. The March 1980 issue (pages 135–44) of *Computing Reviews* had some 450 periodicals listed.

Among many less scholarly magazines devoted to computing is *Datamation,* which has the longest history and largest readership. It has been published monthly since 1955 and it is by far the most entertaining publication in computing, and informative as well. Numerous magazines are devoted to special areas of computing. One area you are no doubt being exposed to is personal computers. Some of the magazines devoted to the subject of personal computers are *Byte, Creative Computing,* and *Dr. Dobb's Journal.* Each one is different (e.g., *Byte* seems to cover hardware more than does *Creative Computing*).

Two widely read weekly newspapers that deal only with computing are *Computerworld,* which covers all of computing, and *Minicomputerworld,* which concentrates on part of it.

Scientific American is a magazine that deserves special mention for it should interest almost everyone. It often has authoritative articles on some aspect of computing. Many of its articles are on topics for which computing plays a major role (e.g., compliance with a nuclear test ban treaty, cruise missile navigation, etc.). It occasionally has a special issue on computer-related topics (e.g., the microelectronics issue of September 1977).

A field of human endeavor can be said to have matured when a respectable body of literature in that field has been produced. The appearance of a good encyclopedia is also a sign of maturity. The encyclopedia edited by Ralston [RA 76] has some 1523 pages you should browse through.

User Groups

Every widely used computer has at least one user group associated with it. These are "mutual aid" societies that spring up out of necessity. Anyone responsible for managing a medium-to-large computing installation has something to learn from other people with similar responsibilities as well as some knowledge to share with them. These user groups are also a very useful source of ideas for the computer manufacturers. In some cases user groups have even written original software. SOS is the name of one of the earliest operating systems ever written. It grew out of the efforts of the user group called SHARE, an organization of users of large-scale IBM computers. SOS is the acronym for the SHARE operating system. Likewise, the specifications for the PL/1 language were a joint effort between SHARE and IBM.

The DEC counterpart of SHARE is called DECUS (DEC User Society). It sponsors annual national meetings and occasional regional meetings. There are even local user groups (LUGs) in some cities with an active calendar of activities. DECUS publishes several newsletters (e.g., one for RT-11 users, another for PDP-10 users, yet another for RSX-11 users, etc.). DECUS also maintains a large library of user-contributed software.

Much word-of-mouth information about how to tune your system, how to get around a system bug until it is repaired, and possible new products (hardware and software) is gathered at user group meetings. Likewise, the most useful information one gets at a conference is in the hallways or during social encounters.

Annotations to Bibliography

The annotations that follow mention the most relevant books and papers, classified by subject matter, with some cross-referencing. In a few instances a more recent book or paper has been omitted in favor of an earlier presentation that is worthy of your attention. My intention is not to present an encyclopedic listing. I hope that you will be stimulated to begin exploring on your own.

A great deal of reinventing of the wheel has gone on in computing. The "not invented here" syndrome is common. People stubbornly refuse to consider what others have done or are doing. Much folly could be avoided if people took the time to examine what has already been done. We have enough urgent new problems to solve without trying to justify re-solving old problems for which good solutions already exist.

The categories listed below are used to organize discussion; they are not entirely mutually exclusive:

Algorithms
Codes
Computer Organization and Hardware
History of Computing
Magazines and Newspapers
Networks and Telecommunications
Other Computers
Programming Languages
PDP-11 References
PDP-11 Textbooks
System Software
Thought, Food for

Algorithms

The most authoritative single source of algorithms is found in Donald Knuth's "Art of Computer Programming" series [KN 68, 73, and 81]. A fourth volume is imminent. The origin of each algorithm is traced and its performance and limitations are analyzed. Knuth ends each chapter with some of the most challenging problems you will ever find.

ACM used to publish user-contributed algorithms monthly in *Communications of the ACM*. Those algorithms plus more recent ones are now found in ACM's CALGO [AC 80a], which is updated quarterly. The October 1980 supplement ended with algorithm number 562, on heap programs for efficient table maintenance.

Much effort goes into transforming an algorithm into a robust software product. This is especially true in the production and support of mathematical software, as you recollect from the pitfalls discussed in the reprint of the IMSL article in chapter 13. The problems of mathematical software deserve a whole journal, and there is one. A recent article [CH 80a] describes the effort which goes into managing a good mathematical subroutine library.

Codes

A brief article by Heath [HE 72] discusses the origins of the binary code (it has a fascinating history). Kahn has written a best-seller entitled *The Code-Breakers* [KA 67], which focuses on encryption and how to "break" secret codes. Three books which read like novels [BR 75, LE 78, WI 74] describe the role of codes, code-breaking, and code-breaking machines (ENIGMA, COLOSSUS, ULTRA) in World War II. Greater detail on how these efforts advanced the state of the art in computing is found in ME 80.

Computer Hardware and Computer Organization

You can immerse yourself in the classical engineering view of computer hardware by reading the standard textbook by Bartree [BA 81] which is now in its fifth edition. If you prefer a short overview of computer systems, then the landmark article by Rosen [RO 69] may be more helpful.

Making comparisons between computers is aided by having a common descriptive notation. One such "hardware description language" called PMS is introduced in BE 71 and then used to describe and compare many of the computers in use in the period 1960–1970. This notation was used to formally describe the PDP-11 instruction set and was included in the early editions of the *PDP-11 Processor Handbook* [DI 71]. Gordon Bell, coauthor of this book, was involved in the design of many DEC computers. He and the codesigners describe the technological environment and motivation for building the PDP-10s and PDP-11s in BE 78a.

Hellerman [HE 73] uses the programming language APL as a hardware description language and gives computer components (e.g., an ALU, a memory) and computer organization a thorough treatment with it. He culminates with a formal description of the IBM 360 and 370 computers.

Hammacher [HA 78] focuses almost exclusively on computer organization and uses the PDP-11 throughout to illustrate the topics. Gear [GE 80], with a similar title, gives more emphasis to software topics than does the Hammacher text, at the expense of omitting some detail on the hardware side. Gear illustrates the concepts by using four computers in his examples (PDP-11, IBM 370, CDC CYBER 170, INTEL 8080). Gear also provides a good introduction to the topics of microprogramming organization and the design of an assembler. Tanebaum [TA 76] provides a very modern view of computer organization which distinguishes clearly between the various levels of abstraction possible. A different approach is taken by Stone [ST 75a] in that he places more emphasis on data structures than do most of the other texts mentioned. Stone also uses the PDP-11 to illustrate these concepts. Gorsline [GO 80] uses the PMS notation in discussing several recent systems (e.g., HP 3000); he includes a chapter on networks.

History

The Eames book [EA 73] is a marvelous collection of several hundred photographs showing all kinds of computing machines and the settings in which they were used during the period 1890–1950. It might lead you to investigate Randell [RA 73], which contains reprints of hard-to-find articles, such as those written by Charles Babbage, Herman Hollerith, von Neumann, etc. Goldstine [GO 72] worked with von Neumann, and his book describes the evolution of computing with particular emphasis on the EDVAC and ENIAC computers. The influence of World War II on computing cannot be overestimated, and Goldstine explains why this was so.

Hammer [HA 57] is a collection of papers by scientists who discuss the impact of computing on their disciplines, as they saw it during the mid-'50s. The Metropolis book [ME 80] will become a classic; it is a collection of personal accounts written for this book by individuals who were directly involved in making "computing history."

Richards's book [RI 66] is a gold-mine of information about commercially available computing hardware in the 1950s and early 1960s.

The 50th anniversary issue of *Electronics* mentioned earlier, has been published as a book [EL 81]. It is very well illustrated and helps place the development of computing in the broader field of developments in electronics.

Three special issues of the monthly *Scientific American* can now be categorized as "history." The issue entitled "Information"[SC 66] described computing in the mid-1960s. A 1972 issue was devoted to communications [SC 72]. A 1977 issue on microelectronics [SC 77] provides insight into the design and manufacture of integrated circuits (IC) chips and how quickly the technology may improve.

Magazines and Newspapers

Datamation [DA b] was mentioned earlier. It was by reading *Datamation* recently that the name DEC Datasystem 780 was brought to my attention, in an advertisement. It was the first inkling I had of the metamorphosis of the VAX-11/780 into a DEC Datasystem; even the local DEC sales representative was unaware of the new name.

Computer Design is a far more approachable magazine than its title suggests. You should be able to read many of its articles, and you will find that a fair number of them deal with practical software concepts (e.g., see CH 80b).

The major weekly newspaper devoted exclusively to computing is *Computerworld* [CO b]. Much of it is advertising, as you would expect, but in a 116 page-issue (volume 14, number 49) there is room for informative news items. *Computer Times* [CO a] (formerly *Minicomputer News*) is a monthly newspaper which focuses on smaller computers.

Networks and Telecommunications

There are a number of books, many journals, and a few magazines devoted to this area. McNamara [MC 78] is a reference corresponding exactly to the book's title Technical Aspects of Data Communications. The Fitzgerald book [FI 78] is organized as a textbook.

Data Communications [DA a] is a monthly magazine. It is surprising at first glance how much of it is devoted to the activities of the U.S. Congress, the Supreme Court, and the Department of Justice, as well as the Federal Communications Commission. This reflects the interplay among big business, multinational corporations, world politics, and rapidly changing technology.

Scholarly journals tend to publish the latest research results, but from time to time they include refreshingly readable articles for nonexperts. For instance, an article in *Computer Communications,* "Communication Protocols for Dumb Terminals" [EV 80], fits in very nicely with our earlier discussions. Similarly, *Computer Networks* has an article entitled "Economic Comparisons of Data Communication Services" [RA 80]. Don't let the content of some issues discourage you from looking at other issues.

Digital Equipment Corporation's support for networks is embodied in a software product package called DECNET. It is described briefly in DI 78c, and DI 78a.

PDP-11 References

All the following books are published by DEC. The *PDP-11 Processor Handbook* [DI 79c] describes the instruction set, the registers, and the timings for the models of the PDP-11 which are current products at the time of publication. The 1979–80 edition describes models PDP-11/04, PDP-11/34a, PDP-11/44, PDP-11/60, and PDP-11/70.

The *PDP-11 Peripherals Handbook* [DI 79b] describes all the DEC devices which can be attached to a PDP-11 except for terminals. The device controllers are described in sufficient detail so you know what each bit of each control and status register is used for, what the recommended register addresses are, etc. Only controllers which attach to the UNIBUS or the MASSBUS are described here. Controllers which attach to the LSI-11's Q-bus are described in the *Microcomputer Interfaces Handbook* [DI 79a].

The *Terminals and Communications Handbook* [DI 78b] describes the DEC terminals which may be connected to a PDP-11. The serial interfaces, terminal multiplexors, modem control options, and synchronous interfaces are described here.

The *Microcomputer Handbook* [DI 77] describes the LSI-11 and all the DEC devices and controllers for it before the advent of the LSI-11/23. Since then two books are needed to describe the LSI-11 hardware.

They are the *Microcomputer Processor Handbook* [DI 79d], which describes the LSI-11, the LSI-11/2, and the LSI-11/23. This is where you would look for descriptions of the PDP-11/03 and the PDP-11/23, since these "PDP-11's" are really "LSI-11's," with a Q-bus, not a UNIBUS. The *Microcomputer Interfaces Handbook* [DI 79a] describes all the DEC devices and their controllers and interfaces which can be attached to an LSI-11 of any model.

The *PDP-11 Software Handbook* [DI 78a] is a survey of the DEC-supported software for the PDP-11 family. It gives 20- to 40-page descriptions of the various operating systems (RT-11, RSTS/E, RSX-11, IAS, MUMPS-11, TRAX). The various supported programming languages are described briefly (e.g., MACRO-11 is described in 22 pages, FORTRAN in 30 pages, etc.). The first three chapters give an overview of DEC's approach to computing systems, file naming conventions, command language syntax, etc. and would be useful reading for anyone who will be making extensive use of any PDP-11.

The *Distributed Systems Handbook* [DI 78c] is the least technical of the DEC handbooks. It provides an executive-level tutorial on distributed computing systems. It is useful in introducing many potential applications, and problems, of computing as seen from a management perspective.

PDP-11 Textbooks

Some of the following books could have been discussed under the computer organization category. However, since they emphasize the PDP-11 more strongly from both the hardware and software point of view, I thought it would be better to group them here.

Cooper [CO 77] is a chemistry professor. Like many chemists and other experimental scientists, he had to struggle to master his first computer and make it do his bidding in his laboratory. The first part of this book is an introduction to the PDP-11; the remainder is devoted to laboratory applications. It discusses analog-to-digital conversion (ADC), digital-to-analog conversion (DAC), signal averaging, plotting, and Fourier transforms. Gill's book [GI 78] is not oriented toward any specific application. Little is said in it regarding peripheral equipment.

Eckhouse [EC 79] is more comprehensive than the other books in this section. Two chapters are particularly interesting, one on how the PDP-11/60 is microprogrammed, and another on real-time interactive computing.

Programming Languages

Sammet's history of programming languages [SA 69] is encyclopedic in its scope. It provides short program segments for several hundred higher level computer languages and conveys the "flavor" of these languages. Those programming languages still in active use in the period 1976-77, as well as those introduced since Sammet's 1969 book, are listed in [SA 78].

We listed SNOBOL4 as a special-purpose high level language for string processing. Griswold [GR 72] shows how macros are used to implement the SNOBOL4 interpreter so that it can easily be transported to many different kinds of computers. You will find many interesting examples of macro definitions in this book.

Ada was mentioned as an up-and-coming language in chapter 18. Barnes [BA 80] gives a brief introduction to Ada. Another programming language you should look at, simply called C, is described in [KE 78]. The C language design was directly influenced by the PDP-11 architecture. C is the higher level language in which the UNIX operating system is implemented. The C compiler is implemented in C.

System Software

You might begin by looking at the first computer program ever written for an actual (as opposed to hypothetical) stored-program computer. This program, written by von Neumann, is described by Knuth in [KN 70]. Then you should meditate on the injunctions in the classic book *The Elements of Programming Style* [KE 74]. Move on to *Software Tools* [KE 76], which reads like a good mystery. It shows you with a complete step-by-step case study, using code that you could actually compile and execute, how to design and implement a real system.

Madnick and Donovan [MA 74] is a textbook which provides a good introduction to systems programming. Freeman [FR 75], more advanced, includes reprints of classical papers (e.g., Denning's "Virtual Memory" paper [DE 70]) along with much more supporting material.

A particularly useful journal devoted to the practical aspects of software that lives up to its name is *Software Practice and Experience*. For instance, you could look at [CO 80] in it to see how someone approached writing a software driver for handling a variety of CRT displays. How one driver can be generalized to handle incompatible devices is a very practical problem many of us face.

Very little has been written about the origin of and motivation for the operating systems software provided by DEC for the PDP-11. This contrasts sharply with the situation regarding UNIX, which was originally written for the PDP-11 but now runs on many other computers. The designers and users of UNIX provide a great deal of insight into this system in the collection of twenty papers which form part 2 of BE 78b.

Other Computers

You can always resort to reading the manufacturer's reference manuals if you wish to learn something about a given computer. Unfortunately, these are often not readily available, and when they are, they tend to obscure rather than illuminate the subject. A reference manual is like a dictionary; you may need it to learn a new language, but you certainly need more than that. I have listed here some books that shed light on specific computers. Some of them are concerned with the hardware design and do not emphasize the programming aspects. Others serve primarily as guides to programming a given machine in its assembly language.

The VAX is DEC's response to the performance limitations of the PDP-11 family, in the light of improved technology. The Levy textbook [LE 80] provides a gentle introduction to the VAX-11/780. Since this book's publication, the VAX-11/750 was introduced. Refer to DI 80b for its description. In one sentence: it is a lower-cost UNIBUS-only VAX, with optional MASSBUS adapters.

Burroughs Corporation has traditionally used advanced architectures, and one of them is described in OR 78. The considerations in designing what was once one of the world's fastest computers, the Control Data Corporation CDC 6600, are described in TH 70. This is the parent of the current CDC Cyber family. Programming the UNIVAC 1108 in assembly language is described in CR 72. The UNIVAC 1110 and the 1100/n are descendants of the 1108.

There are many books on assembly-language programming for the IBM 360 and 370 computers. Two of the better ones are by Kudlick [KU 80] and Struble [ST 75b].

Thought, Food for

We tend to react to crises, then promptly forget them—until they recur. In the late 1960s, the U.S. Congress was debating the wisdom of deploying an antiballistic missile system called ABM. Many people joined in the debate and at least one book resulted [CH 69]. Its many papers discussing the ABM issue will give you some insight into the considerations one has to keep in mind in designing what is, after all, a real-time computer-controlled system with unusual peripheral devices—such as ballistic missile ICBMs, over-the-horizon radar, etc. Of particular interest is the chapter by J.C.R. Licklider entitled "Understatements and Overexpectations." It addresses the question: "Can we design computers and their attendant programs with a reasonable expectation that they will function?" He cites several cases with very negative conclusions.

In this era of reindustrialization we are hearing a lot about robotics and artificial intelligence (AI). One of the pioneers in AI, Joseph Weizenbaum, has written a thoughtful and very sobering exposition of the limits to artificial intelligence [WE 76].

The centennial issue of the periodical *Science* (4 July 1980) provides an interesting update to the views of 23 years earlier [HA 57] about the impact of computing on science and society. Of particular interest is the article by Oettinger (a former president of the ACM) entitled "Information Resources: Knowledge and Power in the 21st Century" [OE 80]. The topic is so important that Anthony Smith devoted a whole book to it: *The Geopolitics of Information* [SM 80]. The raison-d'être for computers is information, and people who control information access, transmission, and processing have a great deal of power.

If reading this book has left you with the impression that the computing world is dry and devoid of drama you will find the Kidder book [KI 81] a revelation. When the Digital Equipment Corporation announced its "supermini" VAX-11/780, one of its major rivals, the Data General Corporation (DG), felt it had to respond with a better product. Kidder tells the story of those who designed the DG "Eagle" computer and the human emotions involved. The trials and tribulations of a humanist's attempts to use a computer are described by Schneider in *Travels in Computerland, or Incompatibilities & Interfaces* [SC 74].

Finally, I will mention a book that has such depth, wit, and originality that it won a Pulitzer Prize. Hofstadter [HO 80] discusses the mind and machines, especially computers, in a manner that should interest any educated person.

Figure 19.3 Terminal education.

Terminal education

By Russell Baker

Ever since reading about Clarkson College's plan to replace its library with a computer I have been worrying about what college students will do in the spring. I mean, you can't just haul a computer out on the campus and plunk it down under a budding elm and lie there with the thing on your chest while watching the birds at work, can you?

You can do that with a book, and it is one of the better things about going to college. With a computer, though, you've got to have a video terminal, which is basically a television set that rolls little, green, arthritic-looking letters and number across a dark screen.

It's not much fun reading a television screen, since, for one thing, the print has a terribly tortured look, as if it had spent four months in a Savak cellar, and since, for another, you always expect it to be interrupted by a commercial. Which is neither here nor there, of course, since this kind of reading is not supposed to convey pleasure, but information.

The difficulty is that you can't take your television screen out under the elm tree and plug it into the computer—the information bank or the information center or the information conveyor, or whatever they choose to call it—since (1) television screens are expensive and fragile and no college president in his right mind is going to let students expose them to ants, dew and tree sap, and since (2) colleges aren't going to shortchange the football team to pay for installing electrical outlets in the tree trunks.

What this means for college students of the future—and Clarkson's electronic library is the library of the future, make no mistake—what it means is that students are going to be spending their springs sitting alone in stale air staring at television screens.

Give them a six-pack of beer or a glass of bourbon, you might say, and you have the ideal training program for American adult home life, which, one supposes, they will still be expected to undertake once they leave college stuffed with information. All I can say is: What does this have to do with education?

The answer comes from Dr. Walter Grattidge, director of Clarkson's new Educational Resources Center—Clarkson's term, not mine. "Education," he told a New York Times reporter, "is basically an information-transfer process." At the risk of sounding somewhat snappish, I say, "Fie, Dr. Grattidge! Fie!"

"Information-transfer process" indeed. Education is not like a decal, to be slipped off a piece of stiff paper and pasted on the back of the skull. The point of education is to waken innocent minds to a suspicion of information.

An educated person is one who has learned that information almost always turns out to be at best incomplete and very often false, misleading, fictitious, mendacious—just dead wrong. Ask any seasoned cop or newspaper reporter. Ask anybody who has ever been the defendant in a misdemeanor trial or the subject of a story in a newspaper.

Well, lets grant that Dr. Grattidge's opinion about being "basically an information-transfer process" is only 80 percent baloney. If you're going to learn the importance of mistrusting information, somebody first has to give you some information, and college is a place where people try to do this, if only so the professors can find out how gullible you are.

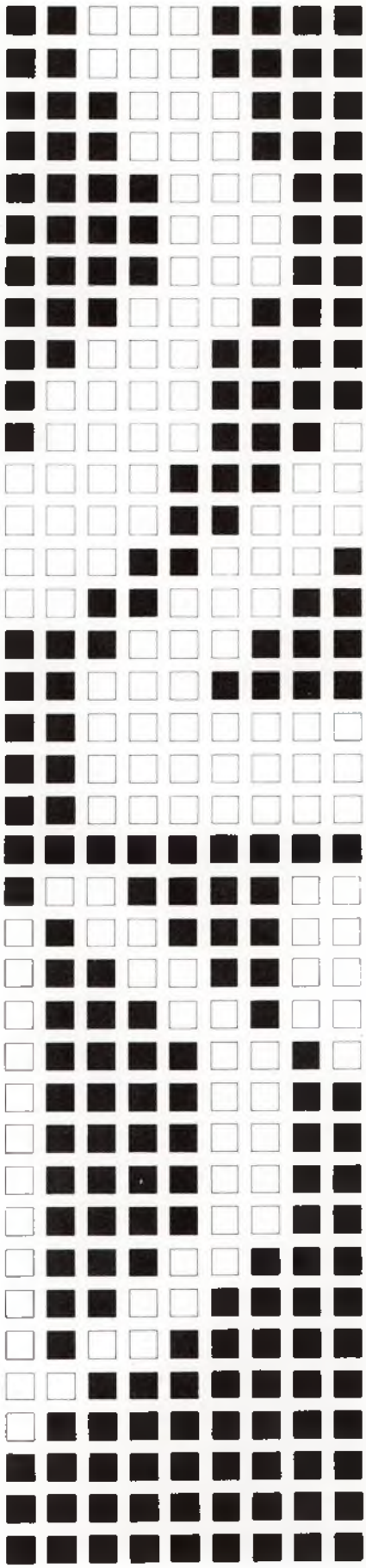
Knowing that, they can then begin to try to teach you to ask a few questions before buying the Brooklyn Bridge or the newest theory about the wherefore of the universe. I'm talking about the good professors now, not the ones who spend all their time compiling fresh information to be transferred to the book-buying public. Even the good professors, however, rarely have enough time to teach the whole student body the art of doubting, which leads to the astonishing act of thinking.

This is why so much of whatever educating happens at college happens in places like the grass under the elm where somebody has gone to read a book, just because it seems like a nicer place to read than the library, and has become distracted by the shape of the clouds, or an ant on the elbow, or an impulse to say to the guy or the girl crossing the quadrangle, "Let's chuck the books for a while and get a beer."

If the time is autumn, and the campus has an apple tree, who knows? Maybe somebody half asleep in an information-transference volume will look up, see an apple fall and revolutionize science. Not much chance of that happening if you're sitting in a room staring at a TV screen plugged into the Educational Resources Center, is there?

In there you are just terribly alone, blotting up information from a machine which, while very, very smart in some ways, has never had an original thought in its life. And no trees grow, and no apples fall.

appendixes



Appendix 1: PDP-11 Instruction Set[1]

ABSOLUTE LOADER

Starting Address: — 500
Memory Size: —
4K 017
8K 037
12K 057
16K 077
20K 117
24K 137
28K 157
(or larger)

BOOTSTRAP LOADER

Address	Contents	Address	Contents
— 744	016 701	— 764	000 002
— 746	000 026	— 766	— 400
— 750	012 702	— 770	005 267
— 752	000 352	— 772	177 756
— 754	005 211	— 774	000 765
— 756	105 711	— 776	177 560 (TTY)
— 760	100 376		or 177 550 (PC11)
— 762	116 162		

773 000 Paper Tape Bootstrap
773 100 Disk/DECtape Bootstrap
773 200 Card Reader Bootstrap
773 300 Cassette Bootstrap

7-BIT ASCII CODE

Octal Code	Char	Octal Code	Char	Octal Code	Char	Octal Code	Char
000	NUL	040	SP	100	@	140	`
001	SOH	041	!	101	A	141	a
002	STX	042	"	102	B	142	b
003	ETX	043	#	103	C	143	c
004	EOT	044	$	104	D	144	d
005	ENQ	045	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BEL	047	'	107	G	147	g
010	BS	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	*	112	J	152	j
013	VT	053	+	113	K	153	k
014	FF	054	,	114	L	154	l
015	CR	055	-	115	M	155	m
016	SO	056	.	116	N	156	n
017	SI	057	/	117	O	157	o
020	DLE	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	CAN	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB	072	:	132	Z	172	z
033	ESC	073	;	133	[	173	{
034	FS	074	<	134	\	174	|
035	GS	075	=	135	]	175	}
036	RS	076	>	136	^	176	~
037	US	077	?	137	—	177	DEL

WORD FORMAT

Mode	Name	Symbolic	Description
0	register	R	(R) is operand [ex. R2=%2]
1	register deferred	(R)	(R) is address
2	auto-increment	(R)+	(R) is adrs; (R) + (1 or 2)
3	auto-incr deferred	@(R)+	(R) is adrs of adrs; (R) + 2
4	auto-decrement	-(R)	(R) - (1 or 2); (R) is adrs
5	auto-decr deferred	@-(R)	(R) - 2; (R) is adrs of adrs
6	index	X(R)	(R) + X is adrs
7	index deferred	@X(R)	(R) + X is adrs of adrs

PROGRAM COUNTER ADDRESSING Reg = 7

2	immediate	#n	operand n follows instr
3	absolute	@#A	address A follows instr
6	relative	A	instr adrs + 4 + X is adrs
7	relative deferred	@A	instr adrs + 4 + X is adrs of adrs

LEGEND

Op Codes

b = 0 for word/1 for byte
SS = source field (6 bits)
DD = destination field (6 bits)
R = gen register (3 bits), 0 to 7
XXX = offset (8 bits), +127 to -128
N = number (3 bits)
NN = number (6 bits)

Operations

( ) = contents of
s = contents of source
d = contents of destination
r = contents of register
← = becomes
X = relative address
% = register definition

Boolean

∧ = AND
∨ = inclusive OR
⊻ = exclusive OR
~ = NOT

Condition Codes

* = conditionally set/cleared
- = not affected
0 = cleared
1 = set

NOTE:

▲ = Applies to the 11/34, 40, 44, 45 & 70 computers
● = Applies to the 11/45 computer

NUMERICAL OP CODE LIST

OP Code	Mnemonic
00 00 00	HALT
00 00 01	WAIT
00 00 02	RTI
00 00 03	BPT
00 00 04	IOT
00 00 05	RESET
00 00 06	RTT
00 00 07 – 00 00 77	(unused)
00 01 DD	JMP
00 02 0R	RTS
00 02 10 – 00 02 27	(unused)
00 02 3N	SPL
00 02 40	NOP
00 02 41 – 00 02 77	cond codes
00 03 DD	SWAB
00 04 XXX	BR
00 10 XXX	BNE
00 14 XXX	BEQ
00 20 XXX	BGE
00 24 XXX	BLT
00 30 XXX	BGT
00 34 XXX	BLE
00 4R DD	JSR
00 50 DD	CLR
00 51 DD	COM
00 52 DD	INC
00 53 DD	DEC
00 54 DD	NEG
00 55 DD	ADC
00 56 DD	SBC
00 57 DD	TST
00 60 DD	ROR
00 61 DD	ROL
00 62 DD	ASR
00 63 DD	ASL
00 64 NN	MARK
00 65 SS	MFPI
00 66 DD	MTPI
00 67 DD	SXT
00 70 00 – 00 77 77	(unused)
01 SS DD	MOV
02 SS DD	CMP
03 SS DD	BIT
04 SS DD	BIC
05 SS DD	BIS
06 SS DD	ADD
07 0R SS	MUL
07 1R SS	DIV
07 2R SS	ASH
07 3R SS	ASHC
07 4R DD	XOR
07 50 0R	FADD
07 50 1R	FSUB
07 50 2R	FMUL
07 50 3R	FDIV
07 50 40 – 07 67 77	(unused)
07 7R NN	SOB
10 00 XXX	BPL
10 04 XXX	BMI
10 10 XXX	BHI
10 14 XXX	BLOS
10 20 XXX	BVC
10 24 XXX	BVS
10 30 XXX	BCC, BHIS
10 34 XXX	BCS, BLO
10 40 00 – 10 43 77	EMT
10 44 00 – 10 47 77	TRAP
10 50 DD	CLRB
10 51 DD	COMB
10 52 DD	INCB
10 53 DD	DECB
10 54 DD	NEGB
10 55 DD	ADCB
10 56 DD	SBCB
10 57 DD	TSTB
10 60 DD	RORB
10 61 DD	ROLB
10 62 DD	ASRB
10 63 DD	ASLB
10 64 00 – 10 64 77	(unused)
10 65 SS	MFPD
10 66 DD	MTPD
10 67 00 – 10 77 77	(unused)
11 SS DD	MOVB
12 SS DD	CMPB
13 SS DD	BITB
14 SS DD	BICB
15 SS DD	BISB
16 SS DD	SUB
17 00 00 – 17 77 77	floating point

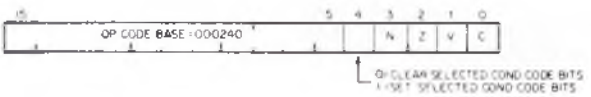
TRAP VECTORS

000	(reserved)	114	Memory Parity
004	Time Out & other errors	240	PIRQ, prog int req
010	illegal & reserved instr	244	Floating Point
014	BPT instruction	250	Memory Management
020	IOT instruction		
024	Power Fail		
030	EMT instruction		
034	TRAP instruction		

MISCELLANEOUS:

Mnemonic	Op Code	Instruction
HALT	000000	halt
WAIT	000001	wait for interrupt
RESET	000005	reset external bus
NOP	000240	(no operation)
●SPL	00023N	set priority level (to N)
▲MFPI	0065SS	move from previous instr space
▲MTPI	0066DD	move to previous instr space
●MFPD	1065SS	move from previous data space
●MTPD	1066DD	move to previous data space

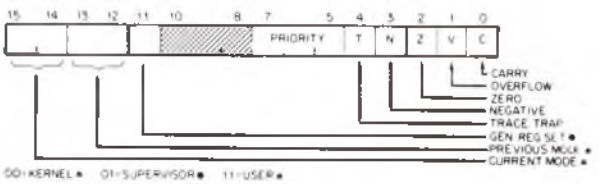
CONDITION CODE OPERATORS:



Mnemonic	Op Code	Instruction	N	Z	V	C
CLC	000241	clear C	–	–	–	0
CLV	000242	clear V	–	–	0	–
CLZ	000244	clear Z	–	0	–	–
CLN	000250	clear N	0	–	–	–
CCC	000257	clear all cc bits	0	0	0	0
SEC	000261	set C	–	–	–	1
SEV	000262	set V	–	–	1	–
SEZ	000264	set Z	–	1	–	–
SEN	000270	set N	1	–	–	–
SCC	000277	set all cc bits	1	1	1	1

PROCESSOR REGISTER ADDRESSES

Processor Status Word
PS - 777 776



▲Stack Limit Register — 777 774

●Program Interrupt Request —777 772

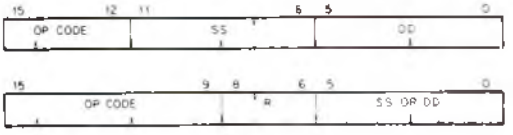
General Registers (console use only)	R0 — 777 700	R4 — 777 704
	R1 — 777 701	R5 — 777 705
	R2 — 777 702	R6 — 777 706
(not for 11/45)	R3 — 777 703	R7 — 777 707

Console Switches & Display Register — 777 570

SINGLE OPERAND: OPR dst

Mnemonic	Op Code	Instruction	dst Result	N	Z	V	C
General							
CLR(B)	■ 050DD	clear	0	0	1	0	0
COM(B)	■ 051DD	complement (1's)	~d	*	*	0	1
INC(B)	■ 052DD	increment	d + 1	*	*	*	–
DEC(B)	■ 053DD	decrement	d − 1	*	*	*	–
NEG(B)	■ 054DD	negate (2's compl)	−d	*	*	*	*
TST(B)	■ 057DD	test	d	*	*	0	0
Rotate & Shift							
ROR(B)	■ 060DD	rotate right	→C, d	*	*	*	*
ROL(B)	■ 061DD	rotate left	C, d←	*	*	*	*
ASR(B)	■ 062DD	arith shift right	d/2	*	*	*	*
ASL(B)	■ 063DD	arith shift left	2d	*	*	*	*
SWAB	0003DD	swap bytes		*	*	*	0
Multiple Precision							
ADC(B)	■ 055DD	add carry	d + C	*	*	*	*
SBC(B)	■ 056DD	subtract carry	d − C	*	*	*	*
▲SXT	0067DD	sign extend	0 or −1	–	*	0	–

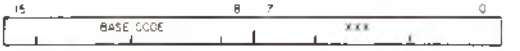
DOUBLE OPERAND: OPR src, dst OPR src, R or OPR R, dst



Mnemonic	Op Code	Instruction	Operation	N	Z	V	C
General							
MOV(B)	■ 1SSDD	move	d ← s	*	*	0	–
CMP(B)	■ 2SSDD	compare	s − d	*	*	*	*
ADD	06SSDD	add	d ← s + d	*	*	*	*
SUB	16SSDD	subtract	d ← d − s	*	*	*	*
Logical							
BIT(B)	■ 3SSDD	bit test (AND)	s ∧ d	*	*	0	–
BIC(B)	■ 4SSDD	bit clear	d ← (~s) ∧ d	*	*	0	–
BIS(B)	■ 5SSDD	bit set (OR)	d ← s ∨ d	*	*	0	–
▲Register							
MUL	070RSS	multiply	r ← r x s	*	*	0	*
DIV	071RSS	divide	r ← r/s	*	*	*	*
ASH	072RSS	shift arithmetically		*	*	*	*
ASHC	073RSS	arith shift combined		*	*	*	*
XOR	074RDD	exclusive OR	d ← r⊻d	*	*	0	–

BRANCH: B – – location

If condition is satisfied:
Branch to location,
New PC ← Updated PC + (2 x offset)
adrs of br instr + 2



Op Code = Base Code + XXX

Mnemonic	Base Code	Instruction	Branch Condition	
Branches				
BR	000400	branch (unconditional)	(always)	
BNE	001000	br if not equal (to 0)	≠ 0	Z = 0
BEQ	001400	br if equal (to 0)	= 0	Z = 1
BPL	100000	branch if plus	+	N = 0
BMI	100400	branch if minus	–	N = 1
BVC	102000	br if overflow is clear		V = 0
BVS	102400	br if overflow is set		V = 1
BCC	103000	br if carry is clear		C = 0
BCS	103400	br if carry is set		C = 1
Signed Conditional Branches				
BGE	002000	br if greater or eq (to 0)	≥0	N⊻V = 0
BLT	002400	br if less than (0)	<0	N⊻V = 1
BGT	003000	br if greater than (0)	>0	Z v (N⊻V) = 0
BLE	003400	br if less or equal (to 0)	≤0	Z v (N⊻V) = 1
Unsigned Conditional Branches				
BHI	101000	branch if higher	>	C v Z = 0
BLOS	101400	branch if lower or same	≤	C v Z = 1
BHIS	103000	branch if higher or same	≥	C = 0
BLO	103400	branch if lower	<	C = 1

JUMP & SUBROUTINE:

Mnemonic	Op Code	Instruction	Notes
JMP	0001DD	jump	PC ← dst
JSR	004RDD	jump to subroutine	use same R
RTS	00020R	return from subroutine	use same R
▲MARK	0064NN	mark	aid in subr return
▲SOB	077RNN	subtract 1 & br (if ≠ 0)	(R) – 1, then if (R) ≠ 0: PC ← Updated PC – (2 x NN)

TRAP & INTERRUPT:

Mnemonic	Op Code	Instruction	Notes
EMT	104000 to 104377	emulator trap (not for general use)	PC at 30, PS at 32
TRAP	104400 to 104777	trap	PC at 34, PS at 36
BPT	000003	breakpoint trap	PC at 14, PS at 16
IOT	000004	input/output trap	PC at 20, PS at 22
RTI	000002	return from interrupt	
▲RTT	000006	return from interrupt	inhibit T bit trap

Representation of PDP-11 Instruction Set and Operands and Operand Address Forms Acceptable to MACRO-11

Special Characters

Character	Function
form feed	Source line terminator
line feed	Source line terminator
carriage return	Formatting character
vertical tab	Source line terminator
:	Label terminator
=	Direct assignment indicator
%	Register term indicator
tab	Item terminator Field terminator
space	Item terminator Field terminator
#	Immediate expression indicator
@	Deferred addressing indicator
(	Initial register indicator
)	Terminal register indicator
, (comma)	Operand field separator
;	Comment field indicator
+	Arithmetic addition operator or auto increment indicator
−	Arithmetic subtraction operator or auto decrement indicator
*	Arithmetic multiplication operator
/	Arithmetic division operator
&	Logical AND operator
!	Logical OR operator
"	Double ASCII character indicator
' (apostrophe)	Single ASCII character indicator
.	Assembly location counter
<	Initial argument indicator
>	Terminal argument indicator
↑ or ^	Universal unary operator Argument indicator
\	MACRO numeric argument indicator

Address Mode Syntax

n is an integer between 0 and 7 representing a register. R is a register expression, E is an expression, ER is either a register expression or an expression in the range 0 to 7.

Format	Address Mode Name	Address Mode Number	Meaning
R	Register	0n	Register R contains the operand. R is a register expression.
@R or (ER)	Deferred Register	1n	Register R contains the operand address.
(ER)+	Autoincrement	2n	The contents of the register specified by ER are incremented *after* being used as the address of the operand.
@(ER)+	Deferred Autoincrement	3n	ER contains the pointer to the address of the operand. ER is incremented *after* use.
−(ER)	Autodecrement	4n	The contents of register ER are decremented *before* being used as the address of the operand.
@−(ER)	Deferred Auto-decrement	5n	The contents of register ER are decremented before being used as the pointer to the address of the operand.
E(ER)	Index	6n	E plus the contents of the register specified, ER, is the address of the operand.
@E(ER)	Deferred Index	7n	E added to ER gives the pointer to the address of the operand.
#E	Immediate	27	E is the operand.
@#E	Absolute	37	E is the address of the operand.
E	Relative	67	E is the address of the operand.
@E	Deferred Relative	77	E is the pointer to the address of the operand.

Instructions

The instructions which follow are grouped according to the operands they take and the bit patterns of their op-codes.

In the instruction type format specification, the following symbols are used:

OP	Instruction mnemonic
R	Register expression
E	Expression

ER	Register expression or expression $\emptyset \leq ER \leq 7$
AC	Floating point register expression
A	General address specification

In the representation of op-codes, the following symbols are used:

SS	Source operand specified by a 6-bit address mode.
DD	Destination operand specified by a 6-bit address mode.
XX	8-bit offset to a location (branch instructions).
R	Integer between 0 and 7 representing a general register.

Symbols used in the description of instruction operations are:

SE	Source Effective Address
FSE	Floating Source Effective Address
DE	Destination Effective Address
FDE	Floating Destination Effective Address
| |	Absolute value of
()	Contents of
→	Becomes

The condition codes in the processor status word (PS) are affected by the instructions. These condition codes are represented as follows:

N	Negative bit:	set if the result is negative
Z	Zero bit:	set if the result is zero
V	oVerflow bit:	set if the operation caused an overflow
C	Carry bit:	set if the operation caused a carry

In the representation of the instruction's effect on the condition codes, the following symbols are used:

*	Conditionally set
–	Not affected
0	Cleared
1	Set

To set conditionally means to use the instruction's result to determine the state of the code (see the *PDP-11 Processor Handbook*).

Logical operations are represented by the following symbols:

!	Inclusive OR
⊕	Exclusive OR
&	AND
–	(used *over* a symbol), "NOT," (i.e., 1's complement)

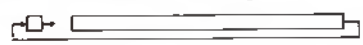
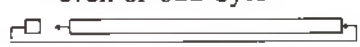
Double-Operand Instructions

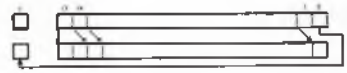
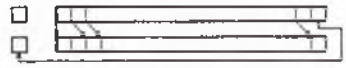
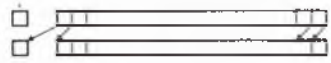
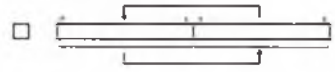
Instruction type format: Op A,A

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
01SSDD	MOV	MOVe	(SE) DE	*	*	0	–
11SSDD	MOVB	MOVe Byte					
02SSDD	CMP	CoMPare	(SE)−(DE)	*	*	*	*
12SSDD	CMPB	CoMPare Byte					
03SSDD	BIT	BIt Test	(SE)&(DE)	*	*	0	–
13SSDD	BITB	BIt Test Byte					
04SSDD	BIC	BIt Clear	(SE)&(DE)→DE	*	*	0	–
14SSDD	BICB	BIt Clear Byte					
05SSDD	BIS	BIt Set	(SE)!(DE)→DE	*	*	0	–
15SSDD	BISB	BIt Set Byte					
06SSDD	ADD	ADD	(SE)+(DE)→DE	*	*	*	*
16SSDD	SUB	SUBtract	(DE)−(SE)→DE	*	*	*	*

Single-Operand Instructions

Instruction type format: Op A

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
0050DD	CLR	CLeaR	∅→DE	0	1	0	0
1050DD	CLRB	CLeaR Byte					
0051DD	COM	COMplement	$(\overline{DE})$→DE	*	*	0	1
1051DD	COMB	COMplement Byte					
0052DD	INC	INCrement	(DE)+1→DE	*	*	*	–
1052DD	INCB	INCrement Byte					
0053DD	DEC	DECrement	(DE)−1→DE	*	*	*	–
1053DD	DECB	DECrement Byte					
0054DD	NEG	NEGate	$(\overline{DE})$+1→DE	*	*	*	*
1054DD	NEGB	NEGate Byte					
0055DD	ADC	ADd Carry	(DE)+(C)→DE	*	*	*	*
1055DD	ADCB	ADd Carry Byte					
0056DD	SBC	SuBtract Carry	(DE)−(C)→DE	*	*	*	*
1056DD	SBCB	SuBtract Carry Byte					
0057DD	TST	TeST	(DE)−∅→DE	*	*	0	0
1057DD	TSTB	TeST Byte					
0060DD	ROR	ROtate Right		*	*	*	*
1060DD	RORB	ROtate Right Byte	even or odd byte 	*	*	*	*
0061DD	ROL	ROtate Left		*	*	*	*
1061DD	ROLB	ROtate Left Byte	even or odd byte 	*	*	*	*

0062DD	ASR	Arithmetic Shift Right		*	*	*	*
1062DD	ASRB	Arithmetic Shift Right Byte	even or odd byte 	*	*	*	*
0063DD	ASL	Arithmetic Shift Left		*	*	*	*
1063DD	ASLB	Arithmetic Shift Left Byte	even or odd byte 	*	*	*	*
0001DD	JMP	JuMP	DE→PC	–	–	–	–
0003DD	SWAB	SWAp Bytes		*	*	0	0

Operate Instructions

Instruction type format: Op

Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
000000	HALT	HALT	The computer stops all functions.	–	–	–	–
0000001	WAIT	WAIT	The computer stops and waits for an interrupt.	–	–	–	–
0000002	RTI	ReTurn from Interrupt	The PC and PS are popped off the SP stack: ((SP))→PC (SP)+2→SP ((SP))→PS (SP)+2→SP	*	*	*	*
000006	RTT	ReTurn from inTerrupt	Same as RTI instruction but inhibits trace trap	*	*	*	*
000005	RESET	RESET	Returns all I/O devices to power-on status.	–	–	–	–
000241	CLC	CLear Carry bit	Ø→C	–	–	–	0
000261	SEC	SEt Carry bit	1→C	–	–	–	1
000242	CLV	CLear oVerflow bit	Ø→V	–	–	0	–
000262	SEV	SEt oVerflow bit	1→V	–	–	1	–
000244	CLZ	CLear Zero bit	Ø→Z	–	0	–	–
000264	SEZ	SEt Zero bit	1→Z	–	1	–	–
000250	CLN	CLear Negative bit	Ø→N	0	–	–	–
000270	SEN	SEt Negative bit	1→N	1	–	–	–
000243	COVC	Clear OVerflow and Carry bits	Ø→V Ø→C	–	–	0	0
000254	CNZ	Clear Negative and Zero bits	Ø→N Ø→Z	0	0	–	–

000257	CCC	Clear all Condition Codes	Ø→N Ø→Z Ø→V Ø→C	0	0	0	0
000277	SCC	Set all Condition Codes	1→N 1→Z 1→V 1→C	1	1	1	1
000240	NOP	No OPeration		–	–	–	–

Trap Instructions

Instruction type format: Op or Op E where $0 \leq E \leq 377_8$
*OP (only)

				Status Word Condition Codes			
Op-Code	**Mnemonic**	**Stands for**	**Operation**	**N**	**Z**	**V**	**C**
*000003	BPT	BreakPoint Trap	Trap to location 14. This is used to call ODT.	*	*	*	*
*000004	IOT	Input/Output Trap	Trap to location 20. This is used to call IOX.	*	*	*	*
104000–104377	EMT	EMulator Trap	Trap to location 30. This is used to call system programs.	*	*	*	*
104400–104777	TRAP	TRAP	Trap to location 34. This is used to call any routine desired by the programmer.	*	*	*	*

Branch Instructions

Instruction type format: Op E where $-128_{10} \leq (E-.-2)/2 \leq 127_{10}$

Op-Code	**Mnemonic**	**Stands for**	**Condition to be met if branch is to occur**
0004XX	BR	BRanch always	
0010XX	BNE	Branch if Not Equal (to zero)	Z=0
0014XX	BEQ	Branch if EQual (to zero)	Z=1
0020XX	BGE	Branch if Greater than or Equal (to zero)	N ⊕ V=0
0024XX	BLT	Branch if Less Than (zero)	N ⊕ V=1
0030XX	BGT	Branch if Greater Than (zero)	Z! (N ⊕ V)=0
0034XX	BLE	Branch if Less than or Equal (to zero)	Z! (N ⊕ V)=1
1000XX	BPL	Branch if PLus	N=0
1004XX	BMI	Branch if MInus	N=1
1010XX	BHI	Branch if HIgher	C ! Z=0
1014XX	BLOS	Branch if LOwer or Same	C ! Z=1
1020XX	BVC	Branch if oVerflow Clear	V=0
1024XX	BVS	Branch if oVerflow Set	V=1

Op-Code	Mnemonic	Stands for	Condition to be Met if Branch is to Occur
1030XX	BCC (or BHIS)	Branch if Carry Clear (or Branch if Higher or Same)	C=0
1034XX	BCS (or BLO)	Branch if Carry Set (or Branch if LOwer)	C=1

Register Destination

Instruction type format: OP ER ,A

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
004RDD	JSR	Jump to SubRoutine	Push register on the SP stack, put the PC in the register: DE→ TEMP (TEMP= temporary storage register internal to processor.) (SP)−2→ SP (REG)→ (SP) (PC)→ REG (TEMP)→ PC	—	—	—	—
074RDD	XOR	eXclusive OR	(R) ! DE→ DE	*	*	0	—

Register-Offset

Instruction type format: OP R,E

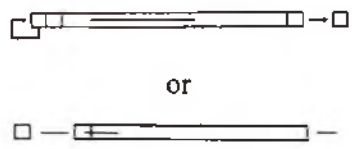
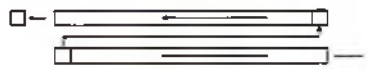
Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
077RDD	SOB	Subtract One and Branch	(R)−1→R PC−(2*DE) → PC	—	—	—	—

Subroutine Return

Instruction type format: Op ER

Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
00020R	RTS	ReTurn from Subroutine	Put register in PC and pop old contents from SP stack into register	—	—	—	—

Instruction type format: Op A,R

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
071RSS	DIV	DIVide	R,Rvl/(SRC)→R,Rvl	*	*	*	*
070RSS	MUL	MULtiply	R*(SRC)→R,Rvl	*	*	0	*
072RSS	ASH	Arithmetic SHift	R is shifted according to low-order 6 bits of source  or	*	*	*	*
073RSS	ASHC	Arithmetic SHift Combined	R,Rvl are shifted according to low-order 6 bits of source R:  R+1 or R:  R+1	*	*	*	*

Appendix 2: MACRO-11 Directives[2]

This is the complete set of MACRO-11 directives. Refer to the MACRO-11 reference manual for those not discussed in the text.

Form	Operation
'	A single quote character (apostrophe) followed by one ASCII character generates a word containing the 7-bit ASCII representation of the character in the low-order byte and zero in the high-order byte.
"	A double quote character followed by two ASCII characters generates a word containing the 7-bit ASCII representation of the two characters.
∧Bn	Temporary radix control; causes the number n to be treated as a binary number.
∧Cn	Creates a word containing the one's complement of n.

2. © Copyright 1980 Digital Equipment Corporation.

∧Dn	Temporary radix control; causes the number n to be treated as a decimal number.
∧Fn	Creates a one-word floating point quantity to represent n.
∧On	Temporary radix control; causes the number n to be treated as an octal number.
.ASCII string	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters) one character per byte.
.ASCIZ string	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters) one character per byte with a zero byte following the specified string.
.ASECT	Begin or resume absolute section.
.BLKB exp	Reserves a block of storage space exp bytes long.
.BLKW exp	Reserves a block of storage space exp words long.
.BYTE exp1,exp2,...	Generates successive bytes of data containing the octal equivalent of the expression(s) specified.
.CSECT symbol .CSECT	Begin or resume named or unnamed relocatable section.
.DSABL arg	Disables the assembler function specified by the argument.
.ENABL arg	Provides the assembler function specified by the argument.
.END .END exp	Indicates the physical end of source program. An optional argument specifies the transfer address.
.ENDC	Indicates the end of a condition block.
.ENDM .ENDM symbol	Indicates the end of the current repeat block, indefinite repeat block, or macro. The optional symbol, if used, must be identical to the macro name.
.EOT	Ignored. Indicates End-of-Tape which is detected automatically by the hardware.
.ERROR exp,string	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.

.EVEN	Ensures that the assembly location counter contains an even address by adding 1 if it is odd.
.FLT2 arg1,arg2,...	Generates successive two-word floating-point equivalents for the floating-point numbers specified as arguments.
.FLT4 arg1,arg2,...	Generates successive four-word floating-point equivalents for the floating-point numbers specified as arguments.
.GLOBL sym1,sym2,...	Defines the symbol(s) specified as global symbol(s).
.IDENT symbol	Provides a means of labeling the object module with the program version number. The symbol is the version number between paired delimiting characters.
.IF cond,arg1,arg2,...	Begins a conditional block of source code which is included in the assembly only if the stated condition is met with respect to the argument(s) specified.
.IFF	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested false.
.IFT	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested true.
.IFTF	Appears only within a conditional block and indicates the beginning of a section of code to be unconditionally assembled.
.IIF cond,arg,statement	Acts as a one-line conditional block where the condition is tested for the argument specified. The statement is assembled only if the condition tests true.
.IRP sym,<arg1,arg2,...>	Indicates the beginning of an indefinite repeat block in which the symbol specified is replaced with successive elements of the real argument list (which is enclosed in angle brackets).

.IRPC sym,string	Indicates the beginning of an indefinite repeat block in which the symbol specified takes on the value of successive characters in the character string.
.LIMIT	Reserves two words into which the Linker inserts the low and high addresses of the relocated code.
.LIST .LIST arg	Without an argument, .LIST increments the listing level count by 1. With an argument, .LIST does not alter the listing level count but formats the assembly listing according to the argument specified.
.MACRO sym,arg1,arg2,...	Indicates the start of a macro named sym containing the dummy arguments specified.
.MEXIT	Causes an exit from the current macro or indefinite repeat block.
.NARG symbol	Appears only within a macro definition and equates the specified symbol to the number of arguments in the macro call currently being expanded.
.NCHR sym,string	Can appear anywhere in a source program; equates the symbol specified to the number of characters in the string (enclosed in delimiting characters).
.NLIST .NLIST arg	Without an argument, .NLIST decrements the listing level count by 1. With an argument, .NLIST deletes the portion of the listing indicated by the argument.
.NTYPE sym,arg	Appears only in a macro definition and equates the low-order six bits of the symbol specified to the six-bit addressing mode of the argument.
.ODD	Ensures that the assembly location counter contains an odd address by adding 1 if it is even.
.PAGE	Causes the assembly listing to skip to the top of the next page.
.PRINT exp,string	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.

.RADIX n	Alters the current program radix to n, where n can be 2, 4, 8, or 1Ø.
.RAD5Ø string	Generates a block of data containing the Radix-50 equivalent of the character string (enclosed in delimiting characters).
.REPT exp	Begins a repeat block. Causes the section of code up to the next .ENDM or .ENDR to be repeated exp times.
.SBTTL string	Causes the string to be printed as part of the assembly listing page header. The string part of each .SBTTL directive is collected into a table of contents at the beginning of the assembly listing.
.TITLE string	Assigns the first symbolic name in the string to the object module and causes the string to appear on each page of the assembly listing. One .TITLE directive should be issued per program.
.WORD exp1,exp2,...	Generates successive words of data containing the octal equivalent of the expression(s) specified.

Appendix 3: MACRO-11 Assembly-Time Diagnostic Error Codes[3]

If MACRO-11 finds a questionable statement, an error code will be printed near the questionable statement.

An error code is printed as the first character in a source line containing an error. This error code identifies the error condition detected during the processing of the line. Example:

```
Q     26 000236   010102              MOV R1,R2,A
```

The extraneous argument A in the MOV instruction above causes the line to be flagged with a Q (syntax) error.

3. © Copyright 1980 Digital Equipment Corporation.

Error Code	Meaning
A	Assembly error. Because many different conditions produce this error message, the directives which may yield a general assembly error have been categorized below to reflect these error conditions: *Category 1: Illegal Argument Specified.* .RADIX—A value other than 2, 8, or 1Ø is specified as a new radix. .LIST/.NLIST—Other than a legally defined argument is specified with the directive. .ENABL/.DSABL—Other than a legally defined argument is specified with the directive, or the attribute arguments of a previously declared program section. .PSECT—Other than a legally-defined argument is specified with the directive, or the attribute arguments of a previously declared program section change. .IF/.IIF—Other than a legally defined conditional test or an illegal argument expression value is specified with the directive. .MACRO—An illegal or duplicate symbol found in dummy argument list. .TITLE—Program name is not specified in the directive, or first nonblank character following the directive is a non-Radix-50 character. .IRP/.IRPC—No dummy argument is specified in the directive. .NARG/.NCHR/.NTYPE—No symbol is specified in the directive. .IF/.IIF—No conditional argument is specified in the directive. *Category 3: Unmatched Delimiter/Illegal Argument Construction.* .ASCII/.ASCIZ/.RAD50/.IDENT—Character string or argument string delimiters do not match, or an illegal character is used as a delimiter, or an illegal argument construction is used in the directive. .NCHAR—Character string delimiters do not match, or an illegal character is used as a delimiter in the directive.

Error Code	Meaning

Category 4: General Addressing Errors.

This type of error results from one of several possible conditions:

1. Permissible range of a branch instruction (from −128(10) to +127(10) words) has been exceeded.
2. A statement makes invalid use of the current location counter. For example, a ".=expression" statement attempts to force the current location counter to cross program section (.PSECT) boundaries.
3. A statement contains an invalid address expression:

   In cases where an absolute address expression is required, specifying a global symbol, a relocatable value, or a complex relocatable value results in an invalid address expression.

   If an undefined symbol is made a default global reference by the .ENABL GBL directive during pass 1, any attempt to redefine the symbol during pass 2 will result in an invalid address expression.

   In cases where a relocatable address expression is required, either a relocatable or absolute value is permissible, but a global symbol or a complex relocatable value in the statement results in an invalid address expression.

   For example:

   .BLKB/.BLKW/.REPT—Other than an absolute value or an expression which reduces to an absolute value has been specified with the directive.
4. Multiple expressions are not separated by a comma. This condition causes the next symbol to be evaluated as part of the current expression.
5. .SAVE—The stack is full when the .SAVE directive is issued.
6. .RESTORE—The stack is empty when the .RESTORE directive is issued.

Category 5: Illegal Forward Reference.

This type of error results from either of two possible conditions:

1. A global assignment statement (symbol== expression or symbol==:expression*) contains a forward reference to another symbol.
2. An expression defining the value of the current location counter contains a forward reference.

Error Code	Meaning
B	Bounding error. Instructions or word data are being assembled at an odd address. The location counter is incremented by 1.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
E	End directive not found. When the end-of-file is reached during source input and the .END directive has not yet been encountered, MACRO-11 generates this error code, ends assembly pass 1, and proceeds with assembly pass 2. Also caused by assembler-stack overflow. In this case MACRO-11 will place a question mark (?) into the line at the point where the overflow occurred.
I	Illegal character detected. Illegal characters which are also nonprintable are replaced by a question mark (?) on the listing. The character is then ignored.
L	Input line is greater than 132(10) characters in length. Currently, this error condition is caused only during macro expansion when longer real arguments, replacing the dummy arguments, cause a line to exceed 132(10) characters.
M	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a label previously encountered.
N	A number contains a digit that is not in the current program radix. The number is evaluated as a decimal value.
O	Opcode error. Directive out of context. Permissible nesting level depth for conditional assemblies has been exceeded. Attempt to expand a macro which was unidentified after .MCALL search.
P	Phase error. A label's definition of value varies from one assembly pass to another or a multiple definition of a local symbol has occurred within a local symbol block. Also, when in a local symbol block defined by the .ENABL LSB directive, an attempt has occurred to define a local symbol in a program section other than that which was in effect when the block was entered. An error code P also appears if an .ERROR directive is assembled.
Q	Questionable syntax. Arguments are missing, too many arguments are specified, or the instruction scan was not completed.
R	Register-type error. An invalid use of or reference to a register has been made, or an attempt has been made to redefine a standard register symbol without first issuing the .DSABL REG directive.

Error Code	Meaning
T	Truncation error. A number generated more than 16 bits in a word, or an expression generated more than 8 significant bits during the use of the .BYTE directive or trap (EMT or TRAP) instruction.
U	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression; such an undefined symbol is assigned a value of zero. Other possible conditions which result in this error code include unsatisfied macro names in the list of .MCALL arguments and a direct assignment (symbol=expression or symbol=:expression) statement which contains a forward reference to a symbol whose definition also contains a forward reference; also, a local symbol may have been referenced that does not exist in the current local symbol block.
Z	Instruction error. The instruction so flagged is not compatible among all members of the PDP-11 family.

Appendix 4: ASCII Codes[4]

ASCII Character Set

Even Parity Bit	7-Bit Octal Code	Character	Remarks
Ø	ØØØ	NUL	NULL, TAPE FEED, CONTROL/SHIFT/P.
1	ØØ1	SOH	START OF HEADING. ALSO SOM, START OF MESSAGE, CONTROL/A.
1	ØØ2	STX	START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL/B.
Ø	ØØ3	ETX	END OF TEXT; ALSO EOM, END OF MESSAGE, CONTROL/C.
1	ØØ4	EOT	END OF TRANSMISSION (END); SHUTS OFF TWX MACHINES, CONTROL/D.
Ø	ØØ5	ENQ	ENQUIRY (ENQRY); ALSO WRU, CONTROL/E.
Ø	ØØ6	ACK	ACKNOWLEDGE; ALSO RU, CONTROL/F.
1	ØØ7	BEL	RINGS THE BELL. CONTROL/G.
1	Ø1Ø	BS	BACKSPACE; ALSO FEO, FORMAT EFFECTOR. BACKSPACES SOME MACHINES, CONTROL/H.
Ø	Ø11	HT	HORIZONTAL TAB. CONTROL/I.

4. © Copyright 1980 Digital Equipment Corporation.

Even Parity Bit	7-Bit Octal Code	Character	Remarks
Ø	Ø12	LF	LINE FEED OR LINE SPACE (NEW LINE); ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL/J.
1	Ø13	VT	VERTICAL TAB (VTAB). CONTROL/K.
Ø	Ø14	FF	FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL/L.
1	Ø15	CR	CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL/M.
1	Ø16	SO	SHIFT OUT; CHANGES RIBBON COLOR TO RED. CONTROL/N.
Ø	Ø17	SI	SHIFT IN; CHANGES RIBBON COLOR TO BLACK. CONTROL/O.
1	Ø2Ø	DLE	DATA LINK ESCAPE. CONTROL/B (DCØ).
Ø	Ø21	DC1	DEVICE CONTROL 1, TURNS TRANSMITTER (READER) ON, CONTROL/Q (X ON).
Ø	Ø22	DC2	DEVICE CONTROL 2, TURNS PUNCH OR AUXILIARY ON. CONTROL/R (TAPE, AUX ON).
1	Ø23	DC3	DEVICE CONTROL 3, TURNS TRANSMITTER (READER) OFF, CONTROL/S (X OFF).
Ø	Ø24	DC4	DEVICE CONTROL 4, TURNS PUNCH OR AUXILIARY OFF. CONTROL/T (AUX OFF).
1	Ø25	NAK	NEGATIVE ACKNOWLEDGE; ALSO ERR, ERROR. CONTROL/U.
1	Ø26	SYN	SYNCHRONOUS. FILE (SYNC). CONTROL/V.
Ø	Ø27	ETB	END OF TRANSMISSION BLOCK; ALSO LEM, LOGICAL END OF MEDIUM. CONTROL/W.
Ø	Ø3Ø	CAN	CANCEL (CANCL). CONTROL/X.
1	Ø31	EM	END OF MEDIUM. CONTROL/Y.
1	Ø32	SUB	SUBSTITUTE. CONTROL/Z.
Ø	Ø33	ESC	ESCAPE. CONTROL/SHIFT/K.
1	Ø34	FS	FILE SEPARATOR. CONTROL/SHIFT/L.
Ø	Ø35	GS	GROUP SEPARATOR. CONTROL/SHIFT/M.
Ø	Ø36	RS	RECORD SEPARATOR. CONTROL/SHIFT/N.
1	Ø37	US	UNIT SEPARATOR. CONTROL/SHIFT/O.
1	Ø4Ø	SP	SPACE.
Ø	Ø41	!	
Ø	Ø42	"	
1	Ø43	#	
Ø	Ø44	$	
1	Ø45	%	
1	Ø46	&	
Ø	Ø47		ACCENT ACUTE OR APOSTROPHE.

Even Parity Bit	7-Bit Octal Code	Character	Remarks
Ø	Ø5Ø	(	
1	Ø51	)	
1	Ø52	*	
Ø	Ø53	+	
1	Ø54	,	
Ø	Ø55	−	
Ø	Ø56	.	
1	Ø57	/	
Ø	Ø6Ø	Ø	
1	Ø61	1	
1	Ø62	2	
Ø	Ø63	3	
1	Ø64	4	
Ø	Ø65	5	
Ø	Ø66	6	
1	Ø67	7	
1	Ø7Ø	8	
Ø	Ø71	9	
Ø	Ø72	:	
1	Ø73	;	
Ø	Ø74	<	
1	Ø75	=	
1	Ø76	>	
Ø	Ø77	?	
1	1ØØ	@	
Ø	1Ø1	A	
Ø	1Ø2	B	
1	1Ø3	C	
Ø	1Ø4	D	
1	1Ø5	E	
1	1Ø6	F	
Ø	1Ø7	G	
Ø	11Ø	H	
1	111	I	
1	112	J	
Ø	113	K	
1	114	L	
Ø	115	M	
Ø	116	N	
1	117	O	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
ø	12ø	P	
1	121	Q	
1	122	R	
ø	123	S	
1	124	T	
ø	125	U	
ø	126	V	
1	127	W	
1	13ø	X	
ø	131	Y	
ø	132	Z	
1	133	[	SHIFT/K.
ø	134	\	SHIFT/L.
1	135	]	SHIFT/M.
1	136	↑ or ^	UP-ARROW or CIRCUMFLEX.
ø	137	_	UNDERLINE.
ø	14ø	`	ACCENT GRAVE or OPENING SINGLE QUOTE
1	141	a	
1	142	b	
ø	143	c	
1	144	d	
ø	145	e	
ø	146	f	
1	147	g	
1	15ø	h	
ø	151	i	
ø	152	j	
1	153	k	
ø	154	l	
1	155	m	
1	156	n	
ø	157	o	
1	16ø	p	
ø	161	q	
ø	162	r	
1	163	s	
ø	164	t	
1	165	u	
1	166	v	
ø	167	w	

Even Parity Bit	7-Bit Octal Code	Character	REMARKS
Ø	17Ø	x	
1	171	y	
1	172	z	
Ø	173		
1	174	|	
Ø	175		CLOSING BRACE GENERATED BY ALT MODE.
Ø	176	~	TILDE GENERATED BY PREFIX KEY (IF PRESENT)
1	177	DEL	DELETE, RUB OUT.

Appendix 5: Using MACRO-11 with RT-11

Use the system editor EDIT to create the MACRO-11 source module. You can assign any name of your choosing for your source file, subject to the usual naming conventions, provided that you give it the suffix /MAC; in DEC terminology, this is called a file name extension. This file name extension serves two purposes: it identifies the file as one containing only ASCII text, and it further specifies that the text is a MACRO-11 source module. So if you want to call your file TEST, then you should name it TEST/MAC when you are creating it with the editor.

Old and New RT-11 Versions

Some of you may be using very old versions of RT-11; some of you may be using the most recent (version 4.0). It does not make much difference in terms of what you will learn. The basic ideas have not changed, but the appearances have. The next section shows the commands for the older versions of RT-11. Following that we show the commands for RT-11 version 4.0.

Older Versions of RT-11

The RT-11 commands that are necessary to assemble, link, and execute a source module whose file name is "PROG" are shown below:

```
RUN MACRO              Remarks
*PROG,PROG=PROG        * is a prompt
*^Z                      from MACRO
                       ^Z for end-of-file response

RUN LINKER
*PROG=PROG,LIB         * is a prompt
*^Z                    from LINKER

RUN PROG
```

The syntactic conventions used need some explanation. The name PROG has been used over and over again. Each time it is used RT-11 appends a different suffix to it, depending upon the context. RT-11 conventions allow you to use different names if you wish. Suppose our source module was called SM. Then we could have MACRO-11 name the object module it produces OM, and have it assign the name PR to the print file it also produces. Finally, we could give the linker the name OM for its input, to be combined with any other modules to be found in file LIB, and we could call the linker output EX. This is illustrated below.

```
RUN MACRO
*OM,PR=SM
*^Z
RUN LINKER
*EX=OM,LIB
*^Z
RUN EX
```

Afterward you can request that RT-11 send the printout file (PROG.LST in the first example and PR.LST in the second example) to either a CRT or a printer.

If our source module had an .END statement such as

```
.END GO
```

the assembler would encode the address corresponding to GO in the object module it produces. The linker, in turn, would take note of this address, leaving that information available with its output, so that when you come to load and execute your program, the RUN processor will find the address corresponding to GO and use it as your program's entry point.

RT-11 Version 4.0

Instead of merely prompting you with a *, this version prints out the prompt "Files?" If you had a MACRO-11 source module called PROG, the following commands would lead to assembling, linking, and executing it, and also producing an assembly listing:

```
.MACRO              Prompt is .
Files? PROG/LIST    Prompt is ``Files?''
```

The /LIST is optional: it requests producing an assembly listing in a file to be named PROG.LST. You can then display the assembly listing file by using:

```
.PRINT              Prompt is .
Files? PROG.LST     Prompt is ``Files?''
```

The linking is requested by:

```
.LINK
Files? PROG
```

and execution of the linker output is requested by:

```
.RUN PROG
```

These commands (except for RUN) use what is known as the long command form. You can shorten the above by using the short command form, depicted below. In the short command form no prompts are issued for file names.

```
.MACRO PROG/LIST
.PRINT PROG.LST
.LINK PROG
.RUN PROG
```

Finally, you can combine the assembling, linking, and execution into one step by using the EXECUTE command:

```
long form          short form

.EXECUTE           .EXECUTE PROG/LIST
Files? PROG/LIST
```

Once again, the /LIST is optional, to be used if you want an assembly printout file created. All of the above assumes that you have consistently used the file name extension /MAC as part of the file names of the MACRO-11 source modules you are working with.

Appendix 6: Using MACRO-11 with RSX-11

RSX-11 is a real-time multiprogramming operating system. The M version allows several people and/or applications to be working simultaneously; the S version is smaller, for dedicated applications.

There are five ways to use MACRO-11 with RSX-11. Since they are all variations on one theme, we will describe only one way here. You can find the others described in the extensive RSX-11 documentation.

To begin with, you should see the following prompt:

```
MCR>
```

This means that you have the attention of the Monitor Console Routine. If you do not see this, then type ctrl-c (hold ctrl key down while pushing the C key). After you see the MCR> prompt, respond by typing:

```
MAC
```

This is how you invoke the MACRO-11 assembler in RSX-11. You should then see the following prompt:

```
MAC>
```

Respond by typing the same kind of file names you would use with RT-11. That is to say, provide names for the following three kinds of files, in the indicated order, using the "," and "=" as delimiters:

object-file-name, listing-file-name = source-file-name

Finally, with RSX-11, the linker is called a task-builder. To process the assembler's output, the object file, respond to the MCR> prompt by typing:

```
TKB
```

The task-builder TKB will then prompt you by displaying:

```
TKB>
```

Respond by typing two file names, in the format:

save-file-name=object-file-name

You can then execute the save-file by using

```
RUN save-file-name
```

We can summarize the preceding dialog by showing the prompts and responses that would be recorded if we took the MACRO-11 source module named PROG.MAC through all these steps:

```
MCR> MAC
MAC> PROG,PROG=PROG
MCR> TKB
TKB> PROG=PROG
MCR> RUN PROG
```

The RSX-11 task-builder offers many options which are beyond the scope of this extremely brief introduction. You should refer to the extensive RSX-11 documentation for further information.

Appendix 7: Using MACRO-11 with RSTS

RSTS is an acronym for the resource-sharing time-sharing system. When it was originally designed, it was not intended to support the use of MACRO-11. In order to use it you must use either the RT-11/RSTS run-time system or the RSX-11/RSTS run-time system. Determine which of the two is supported by your RSTS installation, then read the appropriate appendix on using MACRO-11, either with RT-11, or with RSX-11. Then come back to the next section.

MACRO-11 Under RSTS/RT-11

You can begin in one of two ways:

(1) type SW RT11
or
(2) type RUN $MACRO.SAV

If you type SW RT11, control is switched to the RT-11 emulator program, which will prompt you with a period. From then on you will use the same commands as apply to standard RT-11, which are described in Appendix 5.

If you typed the second form, RUN $MACRO.SAV, then you will get a * as a prompt. You should respond by typing in the same file names you would use with standard RT-11.

Appendix 8: Using MACRO-11 with UNIX

The UNIX operating system will be supported by the manufacturer of the PDP-11 and LSI-11. UNIX was developed at the Bell Laboratories, and various versions of it run on LSI-11s, PDP-11s, VAX-11s, etc. The use of assembly language is not encouraged by the designers of UNIX, since almost everything you can write with MACRO-11 can be written in the high level language named C. The C compiler comes with the UNIX operating system.

However, it is possible for those who wish to use MACRO-11 with UNIX to do so, and a summary of the necessary commands follows.

Preliminaries

You can prepare a MACRO-11 source module by means of any of the many text editors which run under UNIX. These will not be described here. Suppose you have prepared a MACRO-11 source module and given it the name "f." How do you then proceed to assemble the source module in file "f," link its object module to the run-time support library, and take the linker's output so that it can be loaded and executed?

MACRO-11, LINKER

To invoke the assembler, you provide the name of your source module—say, f—as in

```
macro -ls f
```

Be certain the name f has no periods in it. This invocation will produce two files, whose names will be constructed with the name you used above.

File f.lst is the assembly printout; examine it by using any of the UNIX commands to display text files; you could use "cat f.lst," or "more f.lst," or even use one of the text editors if you prefer. The assembler will take the source file name f and create the name f.obj. File f.obj is the relocatable output file which will become the linker's input file. To link this relocatable file with the system subroutine library, *do not type*

```
linker f.obj
```

Do type

```
linker f
```

The linker will itself add the .obj to the name you give. The linker's output is placed in a file f.out. Do not attempt to display any file whose name ends with .obj or .out. Such files are not ASCII files; they are binary files. So attempting to display them is a waste of time or paper.

The only use for your f.out file is to cause it to be executed. Type

```
f.out
```

to execute it.

The various system macros and supporting subroutines used in the text can easily be made available for use with MACRO-11. If you want to learn more about the UNIX operating system and the services it provides see the section entitled System Software in chapter 19.

bibliography

[AC 80a] Association for Computing Machinery. *Collected Algorithms of the ACM*. Supplement 75 to vol. II, Oct. 1980.

[AC 80b] —. *Computing Reviews* 21 (1980): 135–44

[BA 80] Barnes, J.G.P. An overview of Ada. *Software Practice and Experience* 10 (1980): 851–87.

[BA 81] Bartree, T. C. *Digital Computer Fundamentals*. 5th ed. New York: McGraw-Hill, 1981.

[BE 71] Bell, C. G., and A. Newell. *Computer Structures: Readings and Examples*. New York: McGraw-Hill, 1971.

[BE 78a] Bell, C. G., J. C. Mudge, and J. McNamara. *Computer Engineering, A DEC View of Hardware Systems Design*. Bedford, Mass.: Digital Press, 1978.

[BE 78b] *Bell System Technical Journal* 57, no. 6, part 2 (1978).

[BR 75] Brown, A. C. *Bodyguard of Lies*. New York: Harper, 1975.

[CH 69] Chages, A., and J. B. Weisner. *The ABM System, An Evaluation of the Decision to Deploy an Antiballistic Missile System*. New York: New American Library, 1969.

[CH 80a] Chan, T. F., et al. A numerical library and its support. *ACM Transactions on Mathematical Software* 6 (1980): 135–45

[CH 80b] Chien, Y. P. Multitasking executive simplifies realtime microprogramming systems design. *Computer Design* 19 (1980): 109–17.

[CO a] *Computer Times* (formerly *Minicomputer News*). Boston: Hayden Publishers.

[CO b] *Computerworld*. Framingham, Mass.: CW Communications Inc.

[CO 77] Cooper, J. W. *The Minicomputer in the Laboratory: With Examples Using the PDP-11*. New York: Wiley, 1977.

[CO 80] Coutant, C. A. and C. W. Fraser. A device driver for display terminals. *Software Practice and Experience* 10 (1980): 183–87.

[CR 72] Crary, F. D. *Notes on 1108 Assembly Language Programming*. University of Wisconsin-Madison Mathematics Research Center, MRC TR 1139, September 1972.

[DA a] *Data Communications*. New York: McGraw-Hill.

[DA b] *Datamation*. Barrington, Ill.: Technical Publishing Co.

[DE 70] Denning, Peter. J. Virtual memory. *Computing Surveys* 2 (1970): 153–90.

[DI 71] Digital Equipment Corporation. *PDP11/45 Processor Handbook*. Maynard, Mass., 1971.

[DI 77] ———. *Microcomputer Handbook 1977-78*. Maynard, Mass., 1977.

[DI 78a] ———. *PDP-11 Software Handbook*. Maynard, Mass., 1978.

[DI 78b] ———. *Terminals and Communications Handbook 1978-79.* Maynard, Mass., 1978.
[DI 78c] ———. *Distributed Systems Handbook.* Maynard, Mass., 1978.
[DI 79a] ———. *Microcomputer Interfaces Handbook 1979-80.* Maynard, Mass., 1979.
[DI 79b] ———. *PDP-11 Peripherals Handbook.* Maynard, Mass., 1979.
[DI 79c] ———. *PDP-11 Processor Handbook.* Maynard, Mass., 1979.
[DI 79d] ———. *Microcomputer Processor Handbook 1979-80.* Maynard, Mass., 1979.
[DI 80a] ———. *VAX Architecture Handbook 1980-81.* Maynard, Mass., 1980.
[DI 80b] ———. *VAX Hardware Handbook 1980-81.* Maynard, Mass., 1980.
[DI 80c] ———. *VAX Software Handbook 1980-81.* Maynard, Mass., 1980.
[EA 73] Eames, C., and R. Eames. *A Computer Perspective.* Cambridge, Mass.: Harvard Univ. Press, 1973.
[EC 79] Eckhouse, Jr., R. E,. and L. R. Morris. *Minicomputer Systems. In Organization, Programming and Applications (PDP-11).* 2d ed. Englewood Cliffs, N.J.: Prentice-Hall, 1979.
[EL 80] *Electronics,* Special Commemorative Issue, vol. 53, no. 9, April 17, 1980.
[EL 81] Electronics Magazine Books. *An Age of Innovation.* Hightstown, N.J.: Electronics Magazine Books, 1981.
[EV 80] Evans, R. Communication protocols for dumb terminals. *Computer Communications* 3 (1980): 224–27.
[FI 78] Fitzgerald, J., and T. S. Eason. *Fundamentals of Data Communications.* New York: Wiley, 1978.
[FR 75] Freeman, P. *Software Systems Principles.* Chicago: Science Research Assoc., 1975.
[GE 80] Gear, C. W. *Computer Organization and Programming.* 3d ed. New York: McGraw-Hill, 1980.
[GI 78] Gill, A. *Machine and Assembly Language Programming of the PDP-11.* Englewood Cliffs, N.J.: Prentice-Hall, 1978.
[GO 72] Goldstine, H. H. *The Computer from Pascal to von Neumann.* Princeton, N.J.: Princeton Univ. Press, 1972.
[GO 80] Gorsline, G. W. *Computer Organization: Hardware/Software.* Englewood Cliffs, N.J.: Prentice-Hall, 1980.
[GR 72] Griswold, R.E. *The Macro Implementation of SNOBOL4.* San Francisco: W. H. Freeman, 1972.
[HA 78] Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky. *Computer Organization.* New York: McGraw-Hill, 1978.
[HA 57] Hammer, P.C. ed. *The Computing Laboratory in the University.* Madison: Univ. of Wisconsin Press, 1957.
[HE 72] Heath, F. G. Origins of the binary code. *Scientific American,* August 1972, pp. 77–83.
[HE 73] Hellerman, H. *Digital Computer System Principles.* 2d ed. New York: McGraw-Hill 1973.
[HO 80] Hofstadter, D. R. *Godel, Escher, and Bach: An Eternal Golden Braid.* New York: Random House, 1980.
[IN 78] International Mathematical and Statistical Library. *Numerical Computations Newsletter.* Houston, Texas.
[KA 67] Kahn, D. *The Code-Breakers.* New York: Macmillan, 1967.
[KE 69] Kent, W. Assembler-language macroprogramming. *Computing Surveys* 1 (1969): 183–96
[KE 74] Kernighan, B. W., and P. J. Plauger. *The Elements of Programming Style.* New York: McGraw-Hill, 1974.

[KE 76] ———. *Software Tools*. Reading, Mass.: Addison-Wesley, 1976.
[KE 78] Kernighan, B. W., and D. M. Ritchie. *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall,, 1978.
[KI 75] Kimbleton, S.R., and G.M. Schneider. Computer communication networks: approaches, objectives and performance considerations. *Computing Surveys* 7 (1975): 129–73.
[KI 81] Kidder, Tracy. *The Soul of a New Machine*. Boston: Little, Brown and Co., 1981.
[KN 68] Knuth, Donald E. *Fundamental Algorithms*. Vol. 1 of *The Art of Computer Programming*. Reading, Mass.: Addison-Wesley, 1968.
[KN 70] ———. von Neumann's first computer program. *Computing Surveys* 2 (1970): 247–60.
[KN 73] ———. *Sorting and Searching*. Vol. 3 of *The Art of Computer Programming*. Reading, Mass.: Addison-Wesley, 1973.
[KN 81] ———. *Semi-Numerical Algorithms* 2d ed. Vol. 2 of *The Art of Computer Programming*. Reading, Mass.: Addison-Wesley, 1979.
[KU 80] Kudlick, M. D. *Assembly Language Programming for the IBM Systems 360 and 370*. Dubuque, Ia.: Wm. C. Brown Company, 1980.
[LE 78] Lewin, R. *Ultra Goes to War: the Secret Story*. London: Hutchinson, 1978.
[LE 80] Levy, H. M., and R. H. Eckhouse, Jr. *Computer Programming and Architecture: the VAX-11*. Bedford, Mass.: Digital Press, 1980.
[MC 78] McNamara, J. E. *Technical Aspects of Data Communications*. Bedford, Mass.: Digital Press, 1978.
[MA 74] Madnick, S., and J. Donovan. *Operating Systems*. New York: McGraw-Hill, 1974.
[ME 80] Metropolis, N., J. Howlett, and Gian-Carlo Rota, eds. *A History of Computing in the Twentieth Century*. New York: Academic Press, 1980.
[OE 80] Oettinger, A. G. Information resources: knowledge and power in the 21st century. *Science* 209 (4 July 1980): 191–98.
[OR 78] Organick, E. I., and J. A. Hinds. *Interpreting Machines: Architecture and Programming of the B1700-1800 Series*. New York: North-Holland, 1978.
[PR 72] Presser, L., and J. R. White. Linkers and Loaders. *Computing Surveys* 4 (1972): 149–68
[RA 76] Ralston, A., ed. *Encyclopedia of Computer Science*. New York: Petrocelli/Charter, 1976.
[RA 73] Randell, B., ed. *The Origins of Digital Computers: Selected Papers*. Berlin and New York: Springer-Verlag, 1973.
[RA 80] Ratz, H. C., and J. A. Field. Economic comparisons of data communication services. *Computer Networks* 4 (1980): 143–56.
[RI 66] Richards, R. K. *Electronic Digital Systems*. New York: Wiley, 1966.
[RO 69] Rosen, Saul. Electronic Computers: A Historical Survey. *Computing Surveys* 1 (1969): 7–36
[SA 69] Sammet, Jean E. *Programming Languages: History and Fundamentals*. Englewood Cliffs, N.J.: Prentice-Hall, 1969.
[SA 78] Sammet, Jean E. Roster of programming languages for 1976-77. *SIGPLAN Notices*. 13(11) Nov. 1978, 56–85.
[SC 74] Schneider, B. R. Jr. *Travels in Computerland, or Incompatibilities & Interfaces*. Reading, Mass.: Addison-Wesley, 1974.
[SC 66] *Scientific American* (information issue). Sept. 1966
[SC 72] *Scientific American* (communications issue). Sept. 1972
[SC 77] *Scientific American* (microelectronics issue). Sept. 1977
[SM 80] Smith, Anthony. *The Geopolitics of Information*. New York: Oxford Univ. Press, 1980.

[ST 75a] Stone, H., and D. Siewiorek. *Introduction to Computer Organization and Data Structures PDP-11 Edition.* New York: McGraw-Hill, 1975.

[ST 75b] Struble, G.W. *Assembler Language Programming: The IBM 360.* 2d ed. Reading, Mass.: Addison-Wesley, 1975.

[TA 76] Tanebaum, A. *Structured Computer Organization.* Englewood Cliffs, N.J.: Prentice-Hall, 1976.

[TH 70] Thornton, J. E. *Design of a Computer—The Control Data 6600.* Glenview, Ill.: Scott, Foresman, 1970.

[WA 72] Waite, W. *Implementing Software for Non-numeric Applications.* Englewood Cliffs, N.J.: Prentice-Hall, 1972.

[WE 76] Weizenbaum, J. *Computer Power and Human Reason: from Judgment to Calculation.* San Francisco: W. H. Freeman, 1976.

[WI 69] Wilkes, M. V. The growth in interest in microprogramming; a literature survey. *Computing Surveys* 1 (1969): 139–45

[WI 73] Wirth, N. *Systematic Programming: An Introduction.* Englewood Cliffs, N.J.: Prentice-Hall, 1973.

[WI 74] Winterbotham, F. W. *The Ultra Secret.* London: Weidenfeld and Nicolson, 1974.

solutions for selected problems

Solutions to Selected Problems

The solutions to many of the exercises which follow the chapters are given here. The missing solutions are in the instructor's manual. The missing solutions are flagged with an asterisk to avoid confusion. Note that multipart problems, such as problem 1.9, may have only some of the subproblem solutions in this appendix.

Be aware of the fact that in the world of computing, problems rarely have a unique solution. Most problems can be solved in many different ways. Some solutions may be better than others, even though they are all correct solutions. The solutions presented here are intended to be correct, but they are not necessarily the best possible solutions. You can learn much by trying to improve upon the solutions you will see here.

Selected Solutions for Chapter 1

1.1 Surface area program using fewer memory locations.

```
- S 0    Height in location 0
- S 1    Length in location 1
- S 2    Width in location 2
  R 0    Get H
* R 1    H * L
  S 7    Store in location 7
  R 1    Get L
* R 2    L * W
+ R 7    L * W + H * L
  S 7    Store in location 7
  R 0    Get H
* R 2    H * W
+ R 7    (L * W + H * L) + H * W
```

Here we are using location 7 to accumulate the result, eliminating the need for location 8.

1.2 Program for a + b*x + c*x*x

```
- S 0    Store value for a
- S 1    Store value for b
- S 2    Store value for c
- S 9    Store value for x
* R 2    c * x
+R 1     b + c * x
* R 9    (b + c * x) * x
+R 0     a + ( b * x + c * x * x)
```

For each new value of x, repeat from the S 9 line.

1.3*

1.4 With a 5-bit code you can normally represent only 2^5, or 32, distinct items. However, if you designate one of these as a mode indicator (similar to the role played by a shift key), you could assign two meanings to each of the other 31 codes. In the absence of any use of the mode code, a default interpretation, call it mode 0, can prevail. The next appearance of a mode code forces mode 1 to prevail, reverting to mode 0 upon the next occurrence of the mode code. For instance, let the mode code be represented by $. The character string
THIS IS $A TEST $OF P$A$TIENCE
could be interpreted as
THIS IS a test OF PaTIENCE.
The Baudot code uses two mode codes instead of one. It has a "figures" code and a "letters" code; that way you don't have to remember what mode you were in.

1.5*

1.6 The Hollerith code for the characters +, −, 0, 1, . . ., 9 uses one hole per column for each of these. Each column can accommodate up to 12 holes, one per row. The rows are identified with the names +, −, 0, . . ., 9. The letters of the alphabet (upper case only) use two holes per column. The letter A uses the + and 1 punches, B the + and 2, . . ., I the + and 9. Similarly J–R use −,1 to −,9 respectively. Finally S–Z use 0,2 through 0,9. The other characters use more than two holes per code. For instance, the decimal point (period) uses +,3 and 8 row punches. More often than not, the card reader hardware or software converts the 12-bit Hollerith code seen on the punched card into a more convenient 6- or 8-bit code.

1.7 A 91-bit-wide binary code can accommodate the 88 keys and three pedals of a piano. Since almost any combination of these may occur, you can't reduce the 91 without constraining future composers. We were able to replace the 17-bit-wide calculator code by a 5-bit code only because we knew no more than 17 distinct combinations could occur: you are allowed to push only one calculator key at one time; not so with a piano.

1.8*

1.9 (a) (i) the parity of 1100 is even; (ii)* ; (iii) the parity of 011001 is odd;
(b) (i) 111000 has odd parity; (ii)* ; (iii) 0011001 has odd parity;
(c) (i) 01100 has even parity; (ii)* ; (iii) 1011001 has even parity

Selected Solutions for Chapter 2

2.1 Pay program in contiguous memory:

Address	Content	Comments		
00	0127	LAC	27	Get hours
01	0420	SUB	20	hours-40
02	09 ?	TPL	?	hours>40 ?
03	0121	LAC	21	Get 0
04	0225	STO	25	0 → overtime
05	0127	LAC	27	Get hours
06	0523	MPY	23	* rate
07	0224	STO	24	→ reg pay
08	0226	STO	26	→ tot pay
09	0000	HLT		stop
10	0523	MPY	23	Extra hours * rate
11	0522	MPY	22	* 2
12	0225	STO	25	→ ovtm pay
13	0140	LAC	40	get 40
14	0523	MPY	23	* rate
15	0224	STO	24	→reg pay
16	0325	ADD	25	add ovtm pay
17	0226	STO	26	→ tot pay
18	0000	HLT	stop	
...				
	Constants			
20	0040			
21	0000			
22	0002			
23	——	rate		
24	——	reg pay		
25	——	overtime pay		
26	——	total pay		
27	——	hours		

2.2*

2.3*

2.4 If location 70 held 0170 instead of 0150, the program would use its own first instruction as if it were the number of hours!

2.5 At 500 CPS, assuming 1 character = 1 instruction, we execute 500 IPS, or .5 KIPS (thousands of IPS), or .0005 MIPS.

2.6 The arithmetic instructions require either two fetches/instruction or one fetch and one store per instruction; thus two memory accesses per arithmetic instruction are required. At 1 usec per memory access, we can perform .5 MIPS. The branch (transfer) instructions execute at the rate of 1 MIPS. So a typical program would run in the range of .5 to 1 MIPS, disregarding I/O and any other factors.

2.7*

2.8 A program to zero (clear) ten consecutive locations:

Adr	Content	Comments	
00	0113	LAC	zero
01	0290	STO	90 0→90, 91,...
02	0111	LAC	cnt
03	0412	SUB	one
04	0211	STO	cnt
05	0710	TZE	done
06	0101	LAC	inst
07	0312	ADD	one
08	0201	STO	inst
09	0600	TRA	00
10	0000	HALT	Done
11	0010	cnt	
12	0001	one	
13	0000	zero	

2.9*

2.10*

2.11 It is *not* necessary for data to be placed in low memory, or to place instructions in high memory.

2.12 Fit the sample program into locations 20–26:

Adr	Content
20	0123
21	0634
22	0221
23	0120
24	0321
25	0322
26	0000

2.13*

2.14*

2.15 Typing history while loading a small program:

```
@20/---- 0123
@21/---- 0634
...
@26/---- 0000
@20G
```

2.16*

2.17 A program which creates a table of twenty items, each item being the address of the location in which it is stored. The table is to begin at location 50.

Adr	Content		
00	0108	LAC	adr
01	0250	STO	50
02	0307	ADD	one
03	0208	STO	adr
04	0409	SUB	max
05	0800	TMI	loop
06	0000	HALT	
07	0001		one
08	0050		adr
09	0070		max

2.18 A program which tests for completion before the key addition.

Adr	Content			
83	——	sum		
84	0100	LAC	00	get 1st item
85	0283	STO	83	init sum
86	0199	LAC	99	countdown
87	0498	SUB	98	
88	0299	STO	99	
89	0797	TZE	-	stop if 0
90	0183	LAC	83	get sum
91	0301	ADD	01	
92	0283	STO	83	
93	0191	LAC	91	modify adr
94	0397	ADD	98	
95	0291	STO	91	
96	0686	TRA	86	
97	0000	HALT		
98	0001			
99	0010			

2.19 In the event that a naive user runs the program with a zero or negative count, you would stop the program immediately if the program used a leading decision.

2.20*

2.21 Adding ten numbers by straight-line code takes one LAC, nine ADDs, one HALT, for a total of eleven instructions plus eleven other memory accesses for data transfers.

If we examine the program in figure 2.9, it has thirteen instructions, of which three will be executed once. The other ten instructions will be executed nine times; they involve ten instruction fetches and nine data transfers. This sums to $3+9(10+9)$, or $3+171$, for 174 memory references.

The straight-line program uses 22 memory references, versus 174 for the other program, making the first one 7.9 times faster.

2.22*

2.23*

2.24*

2.25 It is possible for a program to use the 0000 value associated with the HALT instruction as a constant with the value zero. The computer won't mind, but you may come to regret it later, if you make any changes to your program and forget that an instruction was also being used as a numeric constant. It is good practice to use instructions only as instructions, and constants only as numeric data.

Selected Solutions for Chapter 3

3.1*

3.2 (a) Binary 11001 is $16+8+0+1$, or 25 decimal; (b)* ; (c)* ; (d) Binary 11,111,111 is $100,000,000-1$ binary, or 2^8-1, or 255 decimal

3.3 (a) $11001+10101$ binary is 101110 binary; (b)* ; (c)*

3.4*

3.5 (a) 129 decimal contains $128=2^7$, leaving $1=2^0$, so this results in $10,000,000+1 \rightarrow 10,000,001$ binary.

(b) *

(c) 100 decimal contains $64=2^6$, leaving 36, which contains $32=2^5$, leaving 4, which is 2^2. Combining this decomposition of 100 decimal gives $2^6+2^5+2^2 \rightarrow 1,000,000+100,000+100 \rightarrow 1,100,100$ binary.

(d) *

3.6 (a)* ; (b) (i) 123+321 octal is 444 octal; (ii)* ; (iii) 076+753 octal can be worked out as follows:

```
   076
+  753
  ----
    11
   14
   7
  ----
  1051
```

(iv)*

3.7*

3.8*

3.9 The ADD instruction in the sequence

```
LAC     X
ADD     Y
STO     Z
```

can be replaced by the sequence

```
STO     TMPA ; save (AC)
LAC     ZERO ; 0 → AC
SUB     Y    ; -(Y) → AC
STO     TMPB ; → TMPB
LAC     TMPA ; restore AC
SUB     TMPB ; (AC)- (-(Y)) → AC
```

which is equivalent to an ADD Y.

3.10*

3.11*

3.12*

3.13*

3.14 The content of memory locations is generally unknown until a program initializes them. Do not assume anything about the content of a location unless you have initialized it yourself.

3.15*

3.16 A program to add the numbers in locations 1030–1036 follows:

```
1050:  013737,1030,1102     ; MOV 1030,sum
1056:  063737,1032,1102     ; MOV 1032,sum
1064:  063737,1034,1102     ; MOV 1034,sum
1072:  063737,1036,1102     ; MOV 1036,sum
1100:   000000              ; HALT
1102:  0                ; sum
1050            ; entry point
```

3.17*

3.18 One way in which a JMP may have an illegal destination address is for this address to be odd. A JMP may only refer to some other instruction, and instructions always have even addresses. Another way is to have the JMP refer to a nonexisting memory location on computers which do not have a full complement of memory.

3.19*

3.20 A MOV s,d instruction is three words long. Fetching it takes three memory references. Then we get (s), (d), and replace the result at d, for three more memory references, giving us a total of 6 memory references in fetching and executing this instruction.

3.21*

3.22 The pay program, rewritten for the PDP-11, with a rate of 5:

```
1000: 013737,1164,1170 ; MOV zero,ov
1006: 023737,1176,1162 ; CMP hours,K40
1014: 100402           ; SKM
1016: 000137 1076      ; JMP more
1022: 013737,1176,1172 ; reg: MOV hours,pay
1030: 063737,1172,1172 ; ADD pay,pay
1036: 063737,1172,1172 ; ADD pay,pay
1044: 063737,1172,1172 ; ADD pay,pay
1052: 063737,1172,1172 ; ADD pay,pay
1060: 013737,1172,1174 ; MOV pay,total
1066: 063737,1170,1174 ; ADD ov,total
1074: 000000           ; HALT
1076: 013737,1176,1200 ; more: MOV hours,x
1104: 163737,1162,1200 ; SUB K40,x
1112: 063737,1200,1200 ; ADD x,x
1120: 013737,1200,1166 ; MOV x,xx
1126: 063737,1200,1200 ; ADD x,x
1134: 063737,1200,1200 ; ADD x,x
1142: 063737,1166,1200 ; ADD xx,x
1150: 013737,1200,1170 ; MOV x,ov
1156: 000137,1022      ; JMP reg
```

```
1162:   40                  ; K40
1164:   0                   ; zero
1166:   0                   ; xx
1170:   0                   ; ov
1172:   0                   ; pay
1174:   0                   ; total
1176:   0                   ; hours
1200:   0                   ; x
1000                ; entry point
```

The program illustrates two ways of multiplying by 5, when multiply instructions are not available. It follows that larger pay rates would probably require having a loop to perform repetitive additions to effect a multiply, making the process somewhat inefficient for large pay rates.

3.23*

3.24 The first machine-language program of chapter 3, rewritten in hexadecimal, appears below. The loading addresses are to the left of the colons, as usual.

```
0200: 17DF
0202: 020E
0204: 0212
0206: 67DF
0208: 0210
020A: 0202
020C: 0000
020E: 0001
0210: 0008
0212: 0000
```

Selected Solutions for Chapter 4

4.1*

4.2*

4.3*

4.4*

4.5*

4.6 Your source program is assembled (this is also called "translated"), then it is loaded and executed (this is also called "run"). The best answer is thus (d). Later on we will see that loading takes place in two steps; first linking, then the actual loading.

Selected Solutions for Chapter 5

5.1 If-Then-Else

```
            CMP     C,A       ; (C)-(A)
            BMI     MORE      ; branch if (C)<(A)
            BR      ELSE
MORE:       CMP     C,D       ; (C)-(D)
            BMI     SET       ; branch if (C)<(D)
ELSE:       MOV     A,X
            SUB     B,X       ; X=A-B
            BR      NEXT
SET:        MOV     C,X
            ADD     D,X       ; X=C+D
NEXT:       ...
```

5.2 Pay program using registers:

```
; use %0 for reg pay, %1 for ovtm pay, %2 total pay
BEGIN:  MOV     CONZ,%1   ; 0 → OV PAY
        MOV     HOURS,%5
        CMP     %5,CONFR  ; HOURS : 40
        BMI     NOOV      ; HOURS<40
; compute overtime pay
        SUB     CONFR,%5  ; ov time hours
        ADD     %5,%5     ; double time
        MOV     RATE,%4   ; accumulate ov pay in %4
MULA:   SUB     ONE,%4    ; multiply by adding
        BEQ     DOREG
        ADD     %5,%4
        BR      MULA
DOREG:  MOV     CONFR,%5  ; first 40 hours
NOOV:   MOV     %5,%0     ; accum reg pay in %0
        MOV     RATE,%4
MULB:   SUB     ONE,%4
        ADD     %5,%0
        BR      MULB
;
DONE:   MOV     %0,%2     ; reg pay
        ADD     %1,%2     ; +ov → total
        HALT
HOURS:  .WORD   -
RATE:   .WORD   -
CONFR:  .WORD   40
CONZ:   .WORD   0
ONE:    .WORD   1
        .END    BEGIN
```

5.3*

5.4*

5.5 A 1-bit error in bit position 0 (the low or right bit) changes a "B" into a "C", or vice versa, so that such an error cannot be detected without recourse to other information. A 1-bit error in transmitting any ASCII code which is using an odd parity check will necessarily force the code to have even parity; therefore a single-bit error will always be detectable.

5.6*

5.7 The string "help me NOW!" comes out as 062550 070154 066440 020145 047516 020527

5.8 The octal words decode as "This-is---a-TEST.", where a - marks each space or blank. Things seem scrambled as you decode because pairs of bytes in each word read as (second byte, first byte), due to the relationship between byte addresses and word addresses.

5.9*

5.10*

5.11*

5.12 Using a .EVEN after every byte-oriented directive is unnecessary. Some do it believing the beginning address of each byte string must be on a word boundary; this is not required. You need only one .EVEN for each group of byte-oriented directives.

5.13 (a)* ; (b) .ASCIZ /HELLO/ uses 6 bytes (5 for HELLO; 1 for implied 0).

5.14*

Selected Solutions for Chapter 6

6.1*

6.2 Blank skipping program:

```
S1:     .ASCIZ  /TEST ONE TWO/
S2:     .BLKB   40          ; hope its wide enough
        .EVEN               ; important
START:  MOV     #S1,%1
        MOV     #S2,%2
NEXT:   MOVB    (%1)+,%0    ; examine a byte
        BEQ     DONE        ; stop on a zero
        CMPB    %0,SPACE
        BEQ     NEXT        ; skip a blank
        MOVB    %0,(%2)+    ; copy nonblanks
        BR      NEXT
DONE:   HALT
        .END    START
```

6.3*

6.4*

6.5 Character reversal program:

```
GO:     MOV     #A,%1       ; addresses of two
        MOV     #B+40,%2    ;           strings
LOOP:   MOVB    (%1)+,(%2)
        BEQ     DONE        ; stop on zero
        DEC     %2
        BR      LOOP
DONE:   HALT
A:      .ASCII  /ABC/<0>
B:      .BLKB   40          ; must be wide enough
        .EVEN; ← important
        .END    GO
```

A smarter program would first find out how long the A string is, so it could copy the last character from the A string to location B, and so on.

6.6 (a) (3000)=4040; (b)* ; (c)* ; (d) (3000)=0

6.7 (a) F; (b) T; (c) F; (d) F; (e) F

6.8 This program zeroes the array of bytes at AL, AL+1, . . ., AL+14. Then it starts examining the string at LINE, LINE+1, . . . intending to stop when it encounters the first line feed code, code 12. If it finds a byte with a value of 101 or greater (the ASCII code for the letter A), then it decreases the value found by 101, and uses that as an index into the array AL. If it finds the value corresponding to an A, B, C, etc., it increases the corresponding count at AL, AL+1, AL+2, etc. by one. For the string we are given, this leaves the first five values at AL through AL+4 with the values 1, 0, 0, 0, and 4.

6.9 (a)* ; (b) (%1)=2400 (PC)=1004; (c) (%1)=1204 (PC)=1002; (d)* ; (e)*

6.10 Computed jump:

```
LIST:   .WORD   L0,L1,L2,L3,L4
GO:     ...
        ...
ABC:    ADD     %1,%1       ; double it
        MOV     LIST(%1),%1 ; get address
        JMP     (%1)        ; deferred addressing
        .END    GO
```

6.11*

6.12 (a) (iv) or (v); (b)* ; (c) (viii); (d)* ; (e)* ; (f) (x)

6.13 Count "L"s:

```
BEGIN:  CLR     %1        ; 0→count
        MOVB    LTRL,%2   ; "L"→%2
        MOV     #STR,%0   ; string adr→ %0
NEXT:   MOVB    (%0)+,%5  ; get char
        BEQ     DONE      ; FOUND ZERO
        CMPB    %5,%2     ; is it an L?
        BNE     NEXT
        INC     %1        ; yes--count it
        BR      NEXT
        HALT
LTRL:   .ASCII  /L/
STR:    .ASCIZ  / ... /
        .EVEN             ; don't forget this
        .END    BEGIN
```

6.14*

6.15*

6.16*

6.17*

6.18*

Selected Solutions for Chapter 7

7.1 Parity of a 3-bit nibble in bits 2–0.

```
X:      CLR     %2        ; record parity here
        MOVB    DATA,%1   ; get nibble in
        BEQ     DONE      ; if 0, it has even parity
        SUB     BIT2,%1
        BPL     A
        ADD     BIT2,%1   ; restore
        BR      DOBIT1
A:      INC     %2        ; bit 2 was on
DOBIT1: SUB     BIT1,%1
        BPL     B
        ADD     BIT1,%1   ; restore
        BR      DOBIT0
B:      INC     %2        ; bit 1 was on
DOBIT0: DEC     %1
        BPL     C
        BR      DONE
C:      INC     %2        ; bit 0 was on
DONE:   CMP     %2,#3
        B NE     EXIT
        CLR     %2 ; =0 for even, not-zero for odd
```

```
EXIT:   HALT
        .RADIX  2 ; to write binary data
BIT2:   .BYTE   100
BIT1    .BYTE   10
DATA:   .BYTE   ---
        .EVEN
        .END    X
```

7.2 Parity of a 3-bit nibble in bits 15–13.

```
X:      CLR     %2      ; record parity here
        MOV     DATA,%1 ; get nibble in
        BEQ     DONE    ; if 0, it has even parity
        BMI     A
        BR      DOBIT14
A:      INC     %2      ; bit 15 was on
DOBIT14:SUB     BIT14,%1
        BPL     B
        ADD     BIT14,%1 ; restore
        BR      DOBIT13
B:      INC     %2      ; bit 14 was on
DOBIT13:SUB     BIT13,%1
        BPL     C
        BR      DONE
C:      INC     %2      ; bit 13 was on
DONE:   CMP     %2,#3
        BNE     EXIT
        CLR     %2 ; =0 for even, not-zero for odd
EXIT:   HALT
BIT15:  .BYTE   100000
BIT14   .BYTE   040000
BIT13   .BYTE   020000
DATA:   .BLKW   ---
        .END    X
```

7.3*

7.4 (a) −49 decimal is 61 octal, or 177717; (b)* ; (c)* ; (d) 075432

7.5*

7.6 (a) (%1)=1, (%2)=1; (b)*; (c) (%1)=−2, (%2)=−1; (d)*

7.7*

7.8*

7.9 (a)* ; (b) 1,000,000 binary (two's-complement) = $-2^6 = -64$; (c)*

7.10 (a)* ; (b) use a source operand with the value 0; (c)* ; (d) impossible; cannot be zero and negative simultaneously

7.11 Triple-precision add

```
; inputs are (A,A+2,A+4) and (B,B+2,B+4)
        MOV     A+4,C+4 ; preserve the inputs
        MOV     A+2,C+2
        MOV     A,C
        ADD     B+4,C+4
        ADC     C+2
        ADD     B+2,C+2
        ADC     C
        ADD     B,C
```

Timing: three MOVs → 3(3+2)=15 memory references, three adds provide 18 more, and two ADCs for 8, totaling 41.

7.12*

7.13*

7.14 You can use a branch tree:

```
        CLR     CCSAVE  ; save the CC bits here
        BMI     NSET
        BEQ     ZSET
        BVS     VSET
        BCS     CSET
; none set
        HALT
NSET:   ADD     NBIT,CCSAVE ; now see if others are set
        BEQ     ZN
        BVS     VN
        BCS     CN
        HALT ; Z-V-C clear
ZN:     ADD     ZBIT,CCSAVE
        BVS     VZN
        BCS     CZN
        HALT    ; V-C clear
        ... etc. ...
        .RADIX  2
NBIT:   .WORD   1000
ZBIT:   .WORD   0100
VBIT:   .WORD   0010
CBIT:   .WORD   0001
        ...
CCSAVE: .WORD   0
```

Restoring the CC bits can be tricky. The easy way is to use the SE= and CL= instructions we will be seeing, or the special instructions available on some CPUs (e.g. MTPS on the PDP-11/45). Otherwise you have to create the situations which will generate the desired CC bit combinations. For instance, to restore C=V=N=0 and Z=1, a CLR %1 will do it. So by using an ADD or SUB in conjunction with a CLR, you can generate many but not all possible CC values.

7.15 (a) the result is 116552; (b) it is desirable if we are adding unsigned numbers; (c) it is in error if we were adding signed numbers; and (d) N=1, Z=0, C=0, V=1 (V=1 because of the sign change)

7.16 (a) T (but differ in effect on CC bits); (b) F; (c) F; (d) F; (e) T; (f) F because the code to output the message may have been destroyed by your program; (g) T; (h) F

7.17*

7.18 Another jump table:

```
DATA:   .WORD   ZERO,TWO, ...
        ...
CASES:  MOV     DATA(%),%0
        JMP     (%0)
```

7.19*

7.20*

7.21*

7.22 (a) N=1 Z=0; (b)* ; (c) N=1 Z=0; (d)* ; (e) no effect; (f)*

7.23 Another double-precision addition (32 bits):

```
ADD     XL,YL   ; low parts
ADC     YH
ADD     XH,YH
```

7.24*

7.25*

7.26 (a) (%0)=000067 N=0 Z=0; (b)* ; (c) N=0 Z=0; (d)* ; (e) (%5)=1001, (1001)=067; (f)* ; (g) (1000)=177776 N=1 C=V=Z=0

7.27*

7.28*

7.29*

7.30*

7.31*

7.32 A 48-bit precision addition:

```
ADD     B+4,C+4 ; low words
ADC     C+2
ADD     B+2,C+2 ; middle words
ADC     C
ADD     B,C
BVS     ERROR   ; overflow
```

7.33*

7.34*

7.35*

7.36*

7.37*

Selected Solutions for Chapter 8

8.1 2

8.2 (a) fetching MOV ABC,(%0) takes two references, fetching (ABC) one more, and storing using the address in % 0 uses one move reference, for a total of four; (b)*

8.3*

8.4*

8.5*

8.6*

8.7 1000

8.8 (a) T; (b) F; (c) F; (d) F; (e) F; (f) F; (g) T (must be even closer); (h) T

8.9*

8.10 It is wrong to think that the address of a byte or a string of bytes must be copied using a byte instruction. Addresses on the PDP-11 are always 16-bits wide, whether they are addresses of other words or of bytes. If you use a MOVB #A,%1, then only the low 8 bits of the 16-bit address A will be copied, so you may be computing with very strange data!

8.11 (a)* ; (b)* ; (c)* ;(d) (%2)=1052 (%3)=1050; (e) (%2)=1052 (%3)=1050 (1050)=35; (f)* ; (g)* ; (h) (%2)=1052; (i)* ; (j)* ;

8.12 (a) vi; (b) viii; (c) x; (d) ix; (e) ii; (f) vii; (g) xi (h) iii; (i) v; (j) i

Memory references for each addressing mode, if used only to fetch or to store a word (not fetch-alter-replace): (i) 3; (ii) 0; (iii) 1; (iv) 2; (v) 1; (vi) 2; (vii) 2; (viii) 2; (ix) 1; (x) 2; (xi) 1

8.13 Subroutine called with parameters in the stack:

```
ARYADD: MOV     %0,-(SP)          ; save %0-%2
        MOV     %1,-(SP)
        MOV     %2,-(SP)
        MOV     12(SP),%0         ; first adr
        MOV     10(SP,%1          ; second adr
        MOV     6(SP),%2          ; result adr
LOOP:   MOV     (%0)+,(%2)        ; copy to result
        BEQ     DONE     ; check for zero sentinel
        ADD     (%1)+,(%2)+ ; add to result
        BR      LOOP
DONE:   MOV     (SP)+,%2          ; restore %2-%0
        MOV     (SP)+,%1
        MOV     (SP)+,%0
        RTS     PC   ; caller must clear his args
```

8.14*

8.15 Replace all spaces with minus signs:

```
        MOV     #ST,%1     ; get strng adr
LOOP:   MOVB    (%1)+,%0   ; get a char
        BEQ     DONE       ; stop on zero
        CMPB    %0,SPACE
        BNE     COPY
        MOVB    MINUS,%0
COPY:   MOVB    %0,(%1)
        BR      LOOP
DONE:   HALT
SPACE:  .ASCII  / /
MINUS:  .ASCII  /-/
ST:     .BLKB   ?
        ...
```

8.16 Addresses assigned and code generated from the source statements:

```
0: 016700          L:       MOV     A,%0
2: 000002
4: 000775                   BR      L
6: 177777          A:       .WORD   -1
```

8.17 This is a fairly tricky program, but a good one to test your skills.

	DATA1	DATA2	ARG1	ARG2	%5	WHO	FIR	SEC
a)	17	20	DATA1	DATA2	7	?	?	?
b)	20	17	"	"	7	17	DATA1	DATA2
c)	17	20	"	"	ARG2+2	?	"	"
d)	20	20	"	"	"	17	"	"

8.18 (a)* ; (b) Rewriting S1 so it can be invoked using JSR %7,S1:

```
Invocation:     MOV     #A1,%4
                JSR     %7,S1
Subroutine:     as before, except
                RTS %4  becomes RTS PC
```

8.19*

8.20*

8.21*

8.22*

8.23 You can copy the items at and near the top of the stack (using MOV (SP),- and MOV OFFSET(PC),-) into your own program memory area (without disturbing the stack). Then you can easily dump these locations.

8.24 (a)* ; (b) (%1)=2001; (c)* ; (d) (3000)=0

8.25 Program:

```
; needs 2 pointers %0=from  %1=to
SQUASH: MOV     %0,%1     ; set "to" pntr
        MOVB    (%0)+,%2  ; new char
        MOVB    %2,(%1)+  ; copy it
        BEQ     DONE      ; stop if 0
LOOP:   MOVB    (%0)+,%3  ; next char
        CMPB    %3,%2     ; same as previous?
        BEQ     LOOP      ; yes- skip it
        MOVB    %3,%2     ; no- copy it
        MOVB    %3,(%1)+
        BR      LOOP
DONE:   MOVB    %3,(%1)
        RTS     %7
```

8.26*

8.27*

8.28 (a) F; (b) T; (c) F; (d) F

8.29*

8.30 Subroutine to compare strings:

```
CMP:     MOVB    (R0)+,R2          ; assume R0=%0 etc.
         BEQ     R0DONE
         MOVB    (R1)+,R4
         BEQ     R1DONE
         CMPB    R2,R4
         BEQ     CMP       ; look at next pair
         BMI     R0LOW     ; 1 → N bit
         CLR     R1        ; 0 → N bit
R0LOW:   RTS     PC
R0DONE:  CMPB    R1,#0
         BNE     LESS
R1DONE:  CLR
         RTS     PC
LESS:    MOV     #-1,%1
         RTS     PC
```

8.31*

8.32*

8.33 (a) is equivalent to

```
MOV      @#100,@#200
MOV      100,200
```

which is the same as (b) if the two lines are interchanged

8.34 (a)* ; (b)* ; (c) (%1)=1204 (PC)=1002; (d) (1200)=0, (%2)=1200; (e) (%2)=1200, (1200)=1000; (f)* ; (g)*; (h)*

Selected Solutions for Chapter 9

9.1 (a) 1111 1001 0101 1100; (b)* ; (c) Using ? for the original carry bit value, 100? 1001 0101 1100; (d)* ; (e) 1001 0101 001? 1100

9.2 Bit-test-like program:

```
BITTY:   MOV     (R5)+,R0    ; data
         MOV     (R5)+,R1    ; bit position
         SUB     #15,R1      ; make it useful
         ASH     R1,R0       ; shift right by (R1) bits
         CCC                 ; clear CC bits
         BIC     #177776,R0  ; no effect on C
         BEQ     NO          ; not set
         SCC                 ; was set
         RTS     R5
```

9.3 (a) T; (b) F; (c) F; (d) F; (e) F

9.4*

9.5*

9.6*

9.7 It can trigger a carry, but this will be detected when the next pair of digits is added. The last "carry propagation" step should be followed by a check, in case it did trigger a carry.

9.8*

9.9*

9.10 Parity of an 8-bit byte:

```
Invocation:     MOVB    data,R1
                JSR     PC,PAR
Subroutine:
PAR:    ; byte in R1, leave parity in N bit
        MOV     #8,R2     ; loop count
        MOV     #1,R0     ; working reg
NEXT:   BIT     #1,R1     ; test low bit
        BEQ     ZERO
        NEG     R0        ; negate Ro for each 1
ZERO:   ROR     R1        ; get next bits
        SOB     R2,NEXT
        RTS     PC        ; N=1 for odd parity
```

9.11 Outline of an algorithm to convert binary to decimal: the basic idea is subtracting the largest power of 10 that fits, recording that it did fit, then repeating the procedure on the remainder, and so on. Consider the binary number 1011. We can try subtracting 100 decimal, but that won't fit (1,100,100 is bigger than 1,011). Try 10 decimal, or 1010 binary: it fits, leaving 1. Record one 10. Try next power of 10, which is 1, on the remainder, which is also 1. It fits: record one 1, remainder 0. Stop when the remainder is 0. Output recorded values, from first to last.

9.12 Octal output of value in R2:

```
; get sign digit, then right digit, then next-to-right, etc.
00:     CLR     R3
        TST     R2          ; test left bit
        BPL     PLUS
        INC     R3
PLUS:   ADD     #60,R3      ; convert to ASCII
        MOVB    R3,S        ; 1 digit done
        MOV     #5,R1       ; loop count
NEXT:   MOV     R2,R3       ; copy
        BIC     #177770,R3  ; isolate low 3 bits
        ADD     #60,R3      ; convrt to ASCII
        MOVB    R3,S(R1)    ; save it
        ASH     #-3,R2      ; shift right 3 bits
        SOB     R1,NEXT
        RTS     PC
S:      .BLKB   6           ; octal ASCII digits
```

Selected Solutions for Chapter 10

10.1*

10.2*

10.3*

10.4 Morse-to-ASCII program, for letters only, using bit-count:

```
; code: use 0 for . and 1 for -
        .RADIX  2
MA:     .BYTE   10,01    ; bit count,code for A
        .BYTE   11,100   ; B
        ...
        .BYTE   100,1100         ; Z
MEND:
        .RADIX  8
; invocation of Morse-to-ASCII subroutine
        MOV     X,R0     ; bit count,Morse code
        JSR     PC,MTA
        MOVB    R1,XX    ; returns ASCII in R1
; subroutine MTA
MTA:    MOV     #MEND-MA,R1      ; table length
        MOV     R1,R2
        ADD     R2,R2    ; double it
MTALP:  CMP     MA-2(R2),R0
        BEQ     HIT
        SOB     R1,MTALP
```

```
        CLR     R0      ; bad code
        RTS     PC
HIT:    ASR     R2      ; use offset
        ADD     #100,R2 ; create ASCII
        MOVB    R2,R1   ; leave it in R1
        RTS     PC
```

10.5 ASCII-to-Morse for letters; no bit count.

```
; use 10 for . and 11 for -
; invocation    MOVB    X,R1 ; ASCII
        JSR     PC,ATM
        MOV     R2,Y    ; double-bit Morse
; subroutine
ATM:    SUB     #100,R1 ; create offset from ASCII A-Z
        ASL     R1      ; double it
        MOV     ASC-2(R1),R2
        RTS     PC
        .RADIX  2
ASC:    .WORD   1011    ; A .-
        .WORD   111010  ; B -..
        ...
        .WORD   11111010        ; Z --..
        .RADIX  8
```

10.6*

10.7*

10.8*

10.9*

10.10 (a) 64 seconds; (b) 16 seconds; (c) 4 seconds

10.11*

10.12 (a) even; (b) even; (c) odd; (d) 101 decimal = 144 octal, odd

Selected Solutions for Chapter 11

11.1 (a) Define missing addresses as follows:

```
OUTSTA= 177564    ; device CSR adr
OUTBUF= OUTSTA+2  ; device's output buffer adr
OUTADR= 64        ; interrupt vector adr

CLKSTA= 177546    ; clock CSR adr
CLKADR= 100       ; interrupt vector adr
```

(b) MAIN places the addresses of the two interrupt handlers at the interrupt vector locations for each of these, then it enables the clock for interrupts and goes into an infinite loop at X. Whenever the clock interrupts (it will, every 1/60 second), this causes an activation of its interrupt handler CLK, which increments %1 and returns, unless the count is up to 59. Then it will increment location MBOX and enable the output device for interrupts. This should immediately trigger an interrupt, since the device is not busy. So we enter its interrupt handler at OUTPUT. As long as MBOX is not 72, we output (send to the device's output buffer register) the character in MBOX. If it was a 72, it is reset to 57, output, and we return. (c) Every minute, the output device will receive and display a character from the list /, 0, 1, . . .,9, :, and then repeat the sequence.

11.2*

11.3 (a) Interrupts are used in conjunction with I/O to provide I/O devices with a means of signaling the CPU when they need attention, or need to report a change in their activity status. This makes it possible to more fully overlap utilization of the CPU and the I/O devices.

(b) An interrupt handler can call subroutines. As long as every use made of the system stack respects the first-in, last-out sequencing, all will be well.

11.4*

Selected Solutions for Chapter 12

12.1*

12.2 A subtract macro:

```
.MACRO   SUBSI    A,B
MOV      R1,-(SP)          ; save R1
MOV      A,R1
NEG      R1                ; negate (A)
ADD      R1,B              ; (B)-(A) → B
MOV      (SP)+,R1          ; restore R1
.ENDM
```

Using SUBSI is almost equivalent to using a SUB; unfortunately, the final MOV will change the CC bit settings, which might be crucial.

12.3*

12.4*

12.5 Use one of the following statements:

```
MULSW = 0 ; 0 for PDP-11/20
MULSW = 1 ; 1 for PDP-11/34
```

then whenever you want to multiply, always load one operand into R1.
If the other operand is in B, use:

```
      MOV      A,R1
.IF MULSW EQ 0          ; for PDP-11/20
      JSR      R5,MULSUB
      .WORD    B
.ENDC
.IF MULSW EQ 1          ; for PDP-11/34
      MUL      B,R1
.ENDC
```

12.6 If we are allowed to use macros, we can define one as shown:

```
      .MACRO   MULIT A,B
      MOV      A,R1
.IF MULSW EQ 0          ; for PDP-11/20
      JSR      R5,MULSUB
      .WORD    B
.ENDC
.IF MULSW EQ 1          ; for PDP-11/34
      MUL      B,R1
.ENDC
      .ENDM
```

Then we can use MULIT A,B when we need to multiply (A)*(B).

12.7 Copy macro:

```
.MACRO   COPY    S,D,L ; source & dest adr, length
MOV      R1,-(SP)
MOV      R2,-(SP)
MOV      R3,-(SP)
MOV      #L,R1
MOV      #S,R2
MOV      #D,R3
MOVB     (R2)+,(R3)+
SOB      R1,.-2   ; can't use a label here
MOV      (SP)+,R3
MOV      (SP)+,R2
MOV      (SP)+,R1
.ENDM
```

12.8*

12.9 Rename the registers using a macro:

```
.MACRO  REGNM ; no args needed
Reg0 = %0
Reg1 = %1
...
.ENDM
```

Then we must invoke the macro we have just defined, in order for the "="s to be effective.

12.10 Reserve words macro:

```
.MACRO  BLOCK   A ; A is a word count
. = . + A + A ; need to double the count
.ENDM
```

12.11*

Selected Solutions for Chapter 13

13.1 The PDP-11 representations and the components are:

	Octal	Sign	Biased-exp	Full fraction
(a)	140000	1	200	10000000000000
(b)	041423	0	206	10010011000000
(c)	141776	1	207	11111110000000
(d)	037314	0	175	11001100146315

13.2*

13.3 For a 3-by-4 array of bytes, assuming row-major ordering:

```
   MOV   I,R1
   DEC   R1
A: MUL   #4,R1  ; (I-1)*no of cols
   ADD   J,R1
B: DEC   R1 ; +(J-1)
   MOVB  ARR(R1),R0 ; fetch byte from array
```

(a) for a 3-by-4 array of words, insert `ASL R1` after line B;
(b) for a 5-by-8 array of bytes, change the 4 in line A to 8.

13.4*

13.5*

13.6 As in 13.3, change the 4 in line A to 16.

13.7 (a)* ; (b) 9; (c)* ; (d) 11; (e)* ; (f)* ; (g) 13 ; (h)* ; (i)* ; (j)* ; (k) syntax error; (l)* ; (m)* ; (n) 8.

13.8 (a) 1; (b) .1; (c) approximately 2^{114}, or $4*10^{34}$.

13.9*

13.10*

13.11*

13.12 Convert a 3-by-4 row major array A into a 3-by-4 column major array B. Approach: step through each element of A (assumed to be bytes), copying each to its new position in B.

Positions:
A: 11 12 13 14 21 22 23 24 31 32 33 34

B: 11 21 31 12 22 32 13 23 33 14 24 34

```
        MOV     #12.,R0     ; copy 3*4 bytes
        MOV     #A,R1
        MOV     #B,R2
        MOV     #B,R3
NEXT:   MOVB    (R1)+,(R2)  ; step R1
        ADD     #3,R2       ; 3=no of rows
        CMP     R2,A+11.
        BLOS    NEXT
        INC     R3          ; step R3
        CMP     R3,#B+3
        BEQ     DONE
        MOV     R3,R2
        BR      NEXT
DONE:   ...
```

13.13 A fetch array element macro:

```
        .MACRO  GET ARR,COLS,I,J
        MOV     I,R1
        DEC     R1
A:      MUL     #COLS,R1    ; (I-1)*no of cols
        ADD     J,R1
B:      DEC     R1          ; +(J-1)
        MOVB    ARR(R1),R0  ; fetch byte from array
        .ENDM
```

13.14 Subroutine to compare two floating-point numbers, without using optional floating-point compare instructions:

```
Invocation:     JSR     R5,CMPFP
                .WORD   FPNA,FPNB ; 2 addresses
Subroutine:
CMPFP:  MOV     (R5)+,R1        ; call it adr A
        MOV     (R5)+,R2        ; call it adr B
        CMP     (R1),(R2)
        BNE     DIFF
; case 1 First words match
LOWWRD: CMP     2(R1),2(R2)     ; look at low words
        BNE     DIFFA
        RTS     R5      ; identical numbers
DIFFA:  BHI     A2H     ; treat low wrds as unsigned #s
        SEN         ; A+2 low
        RTS     R5
A2H:    CLN         ; A+2 high
        RTS     R5
; case 2  First words differ
DIFF:   TST     (R1)            ; check signs
        BMI     AM              ; A -
        TST     (R2)
        BMI     APBM            ; A +, B -
; A PLUS, B PLUS
        CMP     (R1),(R2)
        RTS     R5
; A -
AM:     TST     (R2)
        BMI     AMBM            ; A - B -
; A MINUS, B PLUS
        SEN                     ; A LOW
        RTS     R5
; A PLUS, B MINUS
APBM:   CLN                     ; A HI
        RTS     R5
; both minus
AMBM:   MOV     (R1),R3
        ADD     #100000,R3      ; clear sign bit
        MOV     (R2),R4
        ADD     #100000,R4      ; clear sign bit
        CMP     R3,R4
        BHI     AMBMA           ; mag R3 < mag R4, so
        CLN                     ; -|R3| > -|R4|
        RTS     R5
AMBMA:  SEN
        RTS     R5
```

13.15 Comparing double-precision floating-point numbers. Given two numbers, at (A,A+2,A+4,A+6) and (B,B+2,B+4,B+6), we can use the code from exercise 13.13. It needs additional code for the case (A)=(B) and (A+2)=(B+2). Then we use unsigned conditional branches following the comparisons (using CMP) of (A+4) and (B+4). If they are identical, we repeat the same test on (A+6) compared to (B+6).

13.16*

13.17*

Selected Solutions for Chapter 14

14.1*

14.2*

14.3

(a)

```
        Invocation
            MOV #22,R0
            IOT
            ...
IOT Interrupt Service Routine:
IOTSR:  ASL R0
        RTI
```

(b)

```
        Invocation
            IOT
            .WORD  X
            ...
IOT Interrupt Service Routine:
IOTSR:  MOV @(SP),R2 ; copy arg
        ADD R2,R2
        ADD @(SP),R2
        RTI
```

The solutions a and b are mutually exclusive, since they use different calling sequences. Otherwise we would need to provide the IOT service routine with an argument to specify which service is being requested so it would know which kind of calling sequence is being used.

14.4 (a) F; (b) T; (c) F; (d) F; (e) F; (f) T; (g) T; (h) F; (i) F; (j) F

14.5 The control unit stores the PC and the PS using two memory references, as part of entering an interrupt handler. The RTI will restore them, using two memory references, but the RTI itself uses one memory reference in being fetched.

14.6*

14.7*

14.8 Instead of copying all of the disk from track 0, sector 0 on up to the maximum track and sector numbers, a "smart" backup program reads the disk directory (prepared by the operating system). It then proceeds to read and copy each file listed in the directory, verifying that the directory information and the information read from file headers and trailers is consistent. It also keeps track of how much space is used up by these files, and whether that agrees with the space-availability map usually recorded on each disk. Since the files are likely to be randomly distributed, more seek time and rotational delays will be incurred. This takes much longer, but it provides a valuable consistency check.

14.9*

Selected Solutions for Chapter 15

15.1*

15.2 (a) Macro definition:

```
        .MACRO  EXTREM,LIST,MAX,MIN
        MOV     #100000,MAX
        MOV     #77777,MIN
        .IRP    X,<LIST>
        M1      X
        .ENDM
        .ENDM   EXTREM

        .MACRO  M1 X,?A,?B
        CMP     X,MAX
        BLE     A
        MOV     X,MAX
A:      CMP     X,MIN
        BGE     B
        MOV     X,MIN
B:
        .ENDM   M1
```

(b) Macro expansion:

```
        MOV     #100000,%0
        MOV     #77777,%1
        CMP     A,%0
        BLE     64$
        MOV     A,%0
64$:    CMP     A,%1
        BGE     65$
        MOV     A,%1
65$:    CMP     B,%0
        BLE     66$
        MOV     B,%0
66$:    CMP     B,%1
        BGE     67$
        MOV     B,%1
67$:
        CMP     (%6),%0
        BLE     68$
        MOV     (%6),%0
68$:    CMP     (%6),%1
        BGE     69$
        MOV     (%6),%1
69$:
        CMP     16(%2),%0
        BLE     70$
        MOV     16(%2),%0
70$:    CMP     16(%2),%1
        BGE     71$
        MOV     16(%2),%1
71$:
```

15.3 Code produced by the assembler:

```
000000   010046
000002   012700   000016'
000006   004767   000000g
000012   000167   000026
000016      040      124      110
000021      111      123      040
000024      111      123      040
000027      101      116      040
000032      101      130      101
000035      115      120      114
000040      105      040      012
000044   012600
000046   000137   173160
```

15.4*

15.5 (a)

```
LL:     .WORD   20,1$
1$:     .WORD   25,2$
2$:     .WORD   30,3$
3$:     .WORD   10,0
```

(b)

```
TWLL:   .WORD   0,20,1$
1$:     .WORD   TWLL,25,2$
2$:     .WORD   1$,30,3$
3$:     .WORD   2$,10,0
```

15.6*

15.7 Using .IRPC:

```
.MACRO   RGDF
.IRPC    A,<012345>
MOV      %'A,-(SP) ; notice concatenation
.ENDM
.ENDM
```

15.8*

15.9*

15.10 For SAM ABC, the expansion is:

```
        MOV     ABC,%0
        JSR     %7,JOAN
```

For SAM #PQ,ED, the expansion is

```
        MOV     #PQ,%0
        JSR     %7,JOAN
        BCC     64$
        JMP     ED
64$:
```

15.11 (a) F; (b) F; (c) T; (d) T; (e) F

15.12 (a) 3; (b) 1101001, 0001111, 1110000; (c)* ; (d) error in third bit (from the left) for a code 7 (0111): instead of seeing 0001111, we get 0011111. The parity checks produce:

```
        c1: 1
        c2: 1
        c3: 0 Pointer c3 c2 c1 → 011 →
bit 3.
```

Correct bit 3 by complementing it.

15.13 Macro definition which generates optimal code:

```
        .MACRO  S1,S2,LEN,?C ; LEN in bytes
        X = LEN/2            ; no of words
        XX = LEN - <2*X>     ; get remainder
        MOV     #X,R0        ; length in words
        MOV     #S1,R1
        MOV     #S2,R2
C:      MOV     (R1)+,(R2)+  ; copy words
        SOB     R0,C
    .IIF EQ XX-1,MOVB (R1),(R2)
        .ENDM
```

The expansion only includes the final MOVB if necessary. This is not a very useful macro unless string lengths are known at assembly time.

15.14 (a) through (j) are all false.

15.15 (a) This assembles as:

```
CLR     0
FISH = 3
```

(b) It takes two memory references to fetch the CLR 0, and one more to store a zero in relative location 0.

15.16 Recursion can be implemented on any computer. Having stack operations is not essential; it only makes things much easier. Here we would have to simulate a stack. It would be necessary to explicitly save the return address in a memory array used to simulate a stack.

15.17 If we use DB X,X+2, a DUMP request will be generated. If we use DB with no arguments, a .SNAP request will be generated. If we used DB Y, then an invalid DUMP request will be generated.

15.18*

15.19*

15.20*

15.21 (a) F; (b) T; (c) T; (d) F; (e) T; (f) F; (g) F; (h) T; (i) F; (j) F

15.22*

15.23*

15.24 Expansion of the RET macro:

```
        MOV     %5,64$
        MOV     (%6)+,%5
        MOV     64$,-(%6)
        RTS     %7
64$:    .WORD   0
```

RET rewritten to be more efficient:

```
        .MACRO  RET
        MOV     (%6),-(%6) ; duplicate TOS
        MOV     %5,2(%6)   ; copy RA
        MOV     (%6)+,%5   ; restore %5
        RTS     %7
        .ENDM
```

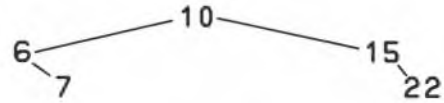
15.25*

15.26*

15.27 (a) Stored at assembly time, using:

```
ROOT:   .WORD   LA,10,RA
RA:     .WORD   0,15,RB
RB:     .WORD   0,22,0
LA:     .WORD   0,6,RRB
RRB:    .WORD   0,7,0
```

(b) Sorted tree, given the sequence 10,15,6,22,7 is:



(c) 6, 7, 10, 15, 22

15.28*

15.29 A SHIFT ABC generates two instructions, each using two words, if ABC is a memory address. A SHIFT %0 generates two instructions, each using one word.

15.30*

Selected Solutions for Chapter 16

16.1*

16.2 Maximum program size on a PDP-11 is 64KB; on an 11/70 the program part and the data part can each be 64KB. Most personal computers use 16-bit byte addresses, limiting them to 64KB programs. Most have smaller limits because part of the operating system must also be resident. On the 11/70 the operating system can be somewhere else in the 2MB memory. File space on an 11/70 can be in the many-MB range (even GB). On a personal computer, on-line file space is usually limited to what one or two diskette drives can hold: as little as 200KB, as much as 1MB. For performance, since an 11/70 has cache memory, DMA disk controllers, and perhaps floating-point hardware, it could run much faster than a personal computer. On the other hand, if you have to share it with fifty people at one time, your share may be much slower. There are many other factors to be considered in making these kinds of comparisons (e.g., the cost of money, ease of expansion, risk of obsolescence, etc.).

16.3*

Selected Solutions for Chapter 17

17.1 You can implement a NOT A by having the relay normally closed so that a voltage (corresponding to logic level 1) from a power source is transmitted over the relay contacts whenever A is 0. Whenever A is 1, A is used to force the relay open, leaving the relay output at logic 0.

17.2*

17.3*

17.4 Dial-up access makes it impractical to restrict access to a computer merely by restricting physical access to a building. The operating system must require that each user identify himself and use a unique, secret password. Dial-up access gives rise to the possibility that a user may accidentally be disconnected. Then the operating system must be able to sense this situation and take care to force a proper log-out, so no intruder can accidentally sneak in on that line. Then the operating system must make it possible for the user to recover whatever work was in progress.

17.5 The transmission speed is not directly related to the transmission efficiency. In asynchronous communication, 8 information bits can be sent for every 10 bits transmitted; this is an efficiency of 80 percent. In synchronous communication, if we sent 1KB blocks, each with 20 bytes of header and trailer, this provides 980 information bytes per block, for an efficiency of 98 percent.

17.6*

17.7*

Selected Solutions for Chapter 18

18.1*

18.2*

18.3 In writing a disassembler, you have to make many assumptions. Does the first instruction of the binary code occur at its beginning? What run-time support routines were available? Does the program modify itself? Does the program use the same word as both an instruction and a numeric constant? Unless a great deal is known about how the program was originally generated (e.g., by a compiler whose structure you know well), it is practically impossible to write a useful disassembler that won't require some human intervention some of the time.

18.4*

PDP 11 Instructions

(Also see appendix 1)

MACRO 11 Directives

(Also see appendixes 2 and 3)

Special Characters

a

b

C

d

e

f

g

q

r

S

t

#439