

DANIEL D. McCRACKEN

a guide to

IBM 1401

programming

JOHN WILEY & SONS, INC., NEW YORK • LONDON

Reprinted by permission, with minor revisions,
from *Principles of Programming*, Sections 1-12,
© 1961 by International Business Machines Corporation

All rights reserved. This book or any part
thereof must not be reproduced in any form
without the written permission of the publisher.

Library of Congress Catalog Card Number: 62-15331

Printed in the United States of America

A GUIDE TO **IBM 1401** PROGRAMMING

DANIEL D. McCRACKEN
McCRACKEN ASSOCIATES, INC.

a guide to
IBM 1401
programming

JOHN WILEY & SONS, INC., NEW YORK • LONDON

PREFACE

This book has been written for the person who wants to get a rapid grasp of the use of the IBM 1400 Series equipment in business data processing. The computers in this line have been widely accepted for a large variety of applications, creating the need for a textbook that introduces all the basic concepts quickly and simply.

The reader is not assumed to have any background in computing or punched-card methods. The early chapters therefore present the basic ideas of data processing methods and equipment before proceeding to a detailed discussion of programming concepts and techniques for this line of computers. The person who does know punched cards or who knows programming for some other computer will not find the presentation too elementary but will be able to move more rapidly.

It is anticipated that the book will prove useful in a variety of situations.

1. It may be used for self-study before attending a programming course or by those who are not able to attend a formal course.

2. It may be used for formal or informal training programs on the 1400 Series equip-

ment, either as a text or as a supplemental reference.

3. It may be used as the text for a college course in programming or as a supplemental reference in a course on the general principles of electronic data processing.

This book is not intended to be a programming manual for any computer but rather to introduce the *principles* of programming for the 1400 Series, with examples in terms of the IBM 1401. A number of case studies illustrate programming principles and at the same time give an indication of typical applications of the equipment. There are numerous exercises, with answers to approximately half of them.

It is a pleasure to acknowledge the cooperation of the International Business Machines Corporation, which made the publication of this book possible. Special mention must be made of the many contributions of Bill Lee, Norm Patton, George Gerken, Bill Kelly, Abe Kaufman, Lou Robinson, and Helen Taft. Mrs. Bea R. Boxer did most of the typing.

DANIEL D. McCRACKEN

*Ossining, New York
March 1962*

CONTENTS

- 1 The Nature of Data Processing 1**
 - 1.1 Introduction, 1
 - 1.2 Basic Data Processing Ideas, 2
 - 1.3 An Example of Sequential File Processing, 4
 - 1.4 An Example of Random Access File Processing, 10
Exercises, 12
- 2 Introduction to Computing Equipment 13**
 - 2.1 The IBM Punched Card, 13
 - 2.2 IBM 1401 Data Processing System Components, 16
 - 2.3 The Card Sorter and Collator, 23
 - 2.4 System Components Used in Sequential File Processing, 26
 - 2.5 Representation of Information in a Computer, 29
Exercises, 30
- 3 Coding Fundamentals 31**
 - 3.1 Computer Storage and Its Addressing, 31
 - 3.2 Instructions, 33
 - 3.3 Storage of Instructions, 38
 - 3.4 Arithmetic and Control Registers, 40
 - 3.5 Addition and Subtraction, 43
Exercises, 45
- 4 Symbolic Programming 47**
 - 4.1 Fundamentals of Symbolic Programming, 47
 - 4.2 Further Information on the SPS Language and Processor, 52
 - 4.3 Case Study: Payroll, 54
Exercises, 59
- 5 Branching 62**
 - 5.1 Fundamentals of Branching, 62
 - 5.2 Further Branching Operations, 66
 - 5.3 Case Study: Parts Explosion and Summary, 70
Exercises, 73
- 6 Address Modification and Loops 75**
 - 6.1 Computations on Addresses, 75
 - 6.2 Program Switches, 77
 - 6.3 Program Loops, 78
 - 6.4 Address Modification Loops, 79
 - 6.5 Indexing, 84
Exercises, 88
- 7 Miscellaneous Operations 89**
 - 7.1 Editing and Format Design, 89
 - 7.2 Printer Carriage Control, 92
 - 7.3 Input and Output Timing, 94
 - 7.4 Buffering, 97
 - 7.5 Program Timing, 98
 - 7.6 Subroutines and Utility Programs, 100
Exercises, 101
- 8 Magnetic Tape Operations 103**
 - 8.1 Physical Characteristics of Magnetic Tapes, 103
 - 8.2 Magnetic Tape Instructions, 106
 - 8.3 Tape Programming with Auto-coder and IOCS, 111
 - 8.4 Inventory Control Case Study, 116
Exercises, 120

9 Random Access File Storage 122

- 9.1 Basic Concepts, 122
- 9.2 The IBM 1405 and 1301 Disk Storage Units, 122
- 9.3 Disk Storage Programming for the IBM 1405, 125
- 9.4 Disk Organization and Addressing, 132
- 9.5 Disk Storage Utility Routines, 134
- 9.6 Case Study: Wholesale Grocery, 135
Exercises, 138

10 Planning and Installing a Computer Application 139

- 10.1 Problem Statement, 139
- 10.2 Problem Analysis, 140
- 10.3 Block Diagram and Program for Inventory Control Processing, 142
- 10.4 Program Checkout, 146
- 10.5 Going Into Operation, 148
- 10.6 Documentation, 149
- 10.7 Summary, 149
Exercises, 150

11 Additional Programming Methods 151

- 11.1 Introduction, 151

- 11.2 Decision Tables, 151
- 11.3 The FORTRAN Coding System, 152
- 11.4 The Report Program Generator, 154
- 11.5 The COBOL Programming System, 156
- 11.6 Fundamentals of COBOL Programming, 157
- 11.7 COBOL Program for Inventory Control Case Study, 161

Appendix 1 IBM 1401 Instructions with Symbolic Programming System Mnemonics 163

Appendix 2 Autocoder Operation Codes 167

Appendix 3 Card and Computer Character Codes 170

Appendix 4 Instruction Timing Data 171

Appendix 5 IBM 1401: Configuration Assumed in Text 172

Glossary 173

Bibliography 177

Answers to Selected Exercises 179

Index 197

1. THE NATURE OF DATA PROCESSING

1.1 Introduction

This book is intended to provide a basic understanding of what is required to make an electronic computer do useful work. In order to reach such an understanding, it is necessary to discuss four topics:

1. What kinds of things can computers do in business data processing?
2. What is required to specify the processing to be done?
3. What are the components and functions of a computer and how do they work?
4. What is "programming" and what are the important programming techniques and principles?

The emphasis in this book is largely on the last of these four areas. The other questions are treated briefly in the first two chapters and indirectly throughout the book. The first two questions relate more directly to the subject of *system analysis* or *procedure design*. They are crucial to the successful utilization of an electronic data processing system but cannot be studied properly without a background in programming. The third question relates to computer engineering; the programmer needs to know a few general characteristics of the subject but almost none of the detail.

It is assumed that the reader of this book has had no experience with computers or with punched cards. No knowledge of mathematics or accounting is required.

The reader who completes a careful study of this book may expect to have learned a good deal about programming. He will know all the basic principles of the subject, a little of how to get started on a project, and what the major steps are. He will find it much simpler to learn programming for another computer or to proceed with a detailed study of the machine (the IBM 1401), which is sketched here. He should not expect, however, that he will have acquired enough skill to undertake by himself the programming of a major application. Learning programming takes a certain amount of practice and, ideally, an opportunity to work on one or two applications with an experienced person.

The programming ideas presented here are illustrated in terms of the IBM 1401 Data Processing System. This machine was chosen primarily because of its wide distribution. It should be realized, however, that most of the basic principles of programming are applicable to any computer. The reader who studies the material in this book thoroughly will have relatively little difficulty learning any other system. There are a few features of the IBM 1401 that are not strictly typical of all computers, but they constitute a small part of the subject when compared with the broad general principles. No one should be concerned that he is studying material here that will not be useful to him. It should also be recognized that *every* computer has specialized features. Finally,

it should be noted in this connection that no attempt has been made to cover all of the features of the IBM 1401 system; this book should be regarded as an introduction to programming, not as a manual of programming for the IBM 1401.

One last point of information about the book itself. The review questions and exercises are important. The review questions, which appear at the end of most sections, allow the reader to be assured that he understands the material thoroughly before proceeding. If the material has been understood, these questions will be relatively easy; if they seem difficult, a rereading would probably be a good idea. The exercises at the end of each chapter provide an opportunity to apply the principles that have been studied. In some cases they continue the development of a topic that could not be treated fully in the text for lack of space. Answers to some exercises are given at the end of the book.

1.2 Basic Data Processing Ideas

Electronic computers are used in business for a variety of reasons. When properly applied, they can save time or money (or both) in producing reports for management and government, in preparing checks and earnings statements for employees, in issuing statements to customers, and in keeping records of accounts payable to suppliers. In many situations they make it possible to obtain information that would otherwise not be economically justifiable. In some cases they provide the basis for improved management control of a business that would not be feasible for time or money reasons without a computer. They are also widely used for engineering and scientific computations.

In carrying out these functions, a number of basic computer operations are performed. Information appearing on punched cards is *listed* (printed). Various *calculations* are performed on data. Detailed information is *summarized* (totaled) often according to several classifications. Information is *edited*, which means two rather different things. In one meaning, *source data* (input information) is checked for validity and accuracy before it is used in further processing. In the other meaning, editing refers to the rearrangement of results for easy reading by inserting dollar signs, decimal points, and commas, deleting zeros in front

of numbers, and providing adequate space between numbers.

These *operations* are performed on *data*. It is necessary also to consider how the data is organized, since the arrangement of the information has a most significant effect on the way the processing is done. This brings us to a fundamental concept in data processing, that of a *file*.

A file is a collection of *records* containing information about a group of related accounts, people, stock items, etc. For instance, an accounts receivable file contains a record for each customer, showing at least the customer's name, address, account number, and amount owed. It may also contain his credit limit, the length of time the amount owed has been due (the "age" of the account), and other information, depending on the needs of the particular business. In a payroll file the record for each employee contains such information as name, payroll number, department, sex, social security number, number of dependents, pay rate, year-to-date gross earnings, year-to-date taxes withheld, year-to-date social security tax, and often many other things.

These examples relate to *master files*, which contain semipermanent information, some of which is *updated* (modified) periodically. A *transaction file*, on the other hand, contains information used to update a master file. Examples: a file containing a record for each customer purchase or a file of labor vouchers used to calculate gross pay. In addition to master and transaction files, there are *report files* that contain information extracted from a master file. An example is provided by the quarterly social security reports required by the Federal Government.

It is obviously necessary to have some way to identify each record in a file. This is usually accomplished by establishing one item in the record as the *key* or *control field* of the record. The key distinguishes each record from all others and is used in almost all file operations. Examples of keys: the customer's account number in an accounts receivable application, the employee's pay number in a payroll, the part number in an inventory control application, the salesman's number in a sales commission job.

Almost all data processing involves operations on files. It is frequently necessary to *sort* the records in a file, that is, to put the records into ascending sequence (or descending, sometimes), according to the keys of the records. For example,

it may be necessary to sort employee labor vouchers into sequence on payroll number before this transaction file can be processed against the payroll master file. As we shall see later in this chapter, data processing methods fall into two broad and rather different classes, according to whether the files do or do not require sorting before the primary processing can be done.

Another common file operation is the combining of two or more files to form one file. If the combined file contains all records from the separate files, this operation is called *merging*; if some of the original records are omitted from the combined file, it is properly called *collating*. (The distinction between the two terms is not always observed in practice.)

Careful planning is required to combine the basic operations so that the files are properly processed and the desired results produced. It is necessary to establish goals for the application, the time schedules that must be met, the exact nature of the operations to be carried out, etc. All of this takes more time than might first be expected for two reasons that are fundamental to a proper understanding of electronic data processing.

1. All processing, with a very few exceptions, must be defined in advance. For instance, it often happens that a customer sends in a check for an amount different from the amount shown on his bill. The person planning the accounts receivable job cannot proceed on the assumption that all payments will be for exact billed amounts and say, "I'll worry about that problem when it happens." The processing operations for such a situation must be planned *in advance*. Again, it is necessary to decide what to do about possible processing errors *before* an application is placed on the machine.

2. A machine cannot exercise judgment unless it has been given explicit directions for making a decision. A machine can be set up to make relatively complex decisions if they are expressible in quantitative terms, but it must be *told* how to make the decisions and what to do in each alternative. We can say to a computer, in suitable language, "If a man's deductions exceed his gross pay, omit as many deductions as necessary; the order in which to omit them is specified in the following table, in which the first deduction is the least crucial." We *cannot* say, "If anything unusual comes up, do what you think is best."

When the task has been properly defined in terms of *what* is to be done, the next step is to decide *how* to do it with the computer. In this step the processing is expressed in terms of operations that can be carried out with the available computing equipment. One of the primary tools of this step is the *flow chart*, which shows the sequence of operations in graphic form. Several flow charts appear in Sections 1.3 and 1.4.

The next step is *programming*. This includes two activities, one of which is *block diagramming*. A block diagram is a detailed flow chart, showing in greater depth exactly what is to be done at each stage of the computer processing. The other activity in programming is *coding*, which is the primary subject of this book.

The fundamental problem is this: The "language" in which the computer can accept "instructions" is very different from the language in which we ordinarily describe data processing. One way or another, the procedure to be followed must be translated into the computer's language. For instance, we say, "Summarize sales by salesman and district." The computer understands instructions like "Add these two numbers," "Go to the print steps if these two numbers are not the same," or "Read a card and place the information in the card input area."

Coding is the process of stating a procedure in a language acceptable to the computer. (The word comes from the fact that the computer's basic language consists of instructions that are written in a "coded" system of numbers and letters.) In few cases are we required to do the entire job of translation, all the way to the final form of the instructions as they will be obeyed ("executed") by the computer. Usually, we write instructions in a symbolic form that is rather similar to the machine's language but considerably more convenient for us. The last step of the translation is then performed with the machine's assistance. In other situations we are able to write the machine procedure in a language quite similar to ordinary English, with the bulk of the translation being done with the aid of a special computer *program* (set of instructions).

Programming and coding involve so much detailed work that most programs do not operate correctly when first tried. Thus it is necessary to *debug* the program (locate and correct the errors) and to *test* it with *test cases* to be sure that it properly processes the data. All of this goes under the name of *program checkout*.

One more activity remains before the program is ready to be used: the master file must be prepared. This usually requires converting the file from the form in which it was used with the previous manual methods. File conversion can be a sizable task in itself and one that often must be started well before the program is completed.

REVIEW QUESTIONS

1. What are some of the basic data processing operations?
2. Give an example of source data.
3. What is a file? Record? Master file? Transaction file? Give examples.
4. Define sorting and merging.
5. A computer can make decisions, under certain circumstances. Give two qualifications to this general statement.
6. What is the difference between programming and coding and between flow charting and block diagramming?

1.3 An Example of Sequential File Processing

Some of the ideas introduced in Section 1.2 may be clarified by considering a typical example of data processing.

A certain company has a system of sales districts, each district employing several salesmen. For each sale a transaction record shows the following:

1. Product number.
2. Quantity.
3. Salesman number.
4. District number.

These transaction records are prepared in the form of punched cards at the data processing center from reports sent in from the districts. Records such as these, with certain other information included, would ordinarily start a whole chain of data processing: instructions would be prepared for the shipping department; the customer's account would be charged with the amount of the purchase, less discounts; the inventory file of finished goods would be updated. For our purposes here, however, we shall consider only one aspect of the total data processing activity based on these records: the preparation of sales statistics.

For various purposes it is desirable to obtain

sales figures summarized monthly in several classifications:

1. Total sales of each merchandise item for the month.
2. Total sales of each salesman for the month.
3. Total sales of each district for the month.
4. Total sales of the company for the month.

Before outlining the sequence of operations required to produce these reports, we must consider how the characteristics of the master file affect the planning. The master file for this simplified problem is needed only to get the unit price for each product; in a full-scale application it would do much more. Our master file contains a record for each product, each record showing a product number and the price of one unit of that product. The most important consideration for our purposes is that the file is in product number order. The record for the product with the smallest product number is first in the file, the record for the product with the next larger product number is next, etc.

The first computation is to get from the master file the unit price of each product sold. This means that for each transaction record we must look up the unit price, which could be done by searching the master file once for each transaction record. This can be done if the master file is stored in a form that makes it convenient to get at any record with a minimum of delay, the subject of the next section. However, when the master file is stored on cards or magnetic tape, this approach has serious drawbacks. The difficulty is that to find any one record in a file of cards (or on tape) it is necessary to inspect every record until the desired one is found. With this kind of equipment it is not possible to thumb through a deck of cards until the approximate area is located and then search in detail for the correct card; it is necessary to read every card in sequence.

This is the basis of the term *sequential access* file: each record is available only in the order in which it appears as the file is read *in sequence*. To be explicit, this means that the first record in the file is available with little delay, but to get the last record requires reading the entire file. This is contrasted with a *random access* file (see Section 1.4), in which any record is available just about as quickly as any other.

The fact that our master file is the sequential access type determines to a considerable extent how the processing must be done. We clearly do

not want to have to read the master file once for each transaction card. Instead, we will put the transaction file into the same sequence as the master file and then get all the information we need from the master file in one *pass*, that is, in one reading of it.

The resequencing of the transaction file is called *sorting*. Since our transaction file is a *deck* of cards, it can be sorted with a card sorter, as described a little more fully in Section 2.2. When this operation is completed, the deck of sales cards will be in the same sequence as the master deck, that is, the sales card with the smallest product number will be at the front of the deck, etc. There may be more than one card with the same product number, of course: several customers no doubt will have bought the same item.

Recall that what we are trying to do at this point is to get the unit price of each product from the master file so that the total price of each sale can be computed (the sales cards give only the number of units sold). There are several ways of doing this; the choice depends on the characteristics of the computer to be used. We shall assume that this job is to be done on a computer that can read only one deck of cards and has no magnetic tapes but can separate the cards into two stacks after reading them. This corresponds to the equipment available on an IBM 1401 Card System.

With the master and detail (transaction) decks now in the same sequence, we use the card *collator* to merge them. By this operation, each detail card is placed behind the master having the same product number. Several things can happen when this is done.

1. There may be *unmatched masters*, that is, masters for which there is no corresponding detail. This means simply that the particular product was not sold in the month. With the collator we have the choice of including such unmatched masters in the merged deck or of selecting them to fall into a separate pocket. We shall leave them in to avoid having to put them back later. This will require a little extra work in the computer program but nothing very difficult.

2. There may be unmatched details, that is, sales cards with product numbers not in the master file. This means that a product number was written or punched incorrectly. We cannot process these cards, and the collator must be set up to select them into a separate pocket. The error cards must be

Master File			
Product Number	Unit Price	Product Number	Unit Price
1120	6.90	3495	2.70
1190	4.32	4192	8.09
1200	10.60	4377	21.90
1213	25.50	4992	10.20
1655	.80	5009	8.00
1656	18.00	5062	1.47
2441	2.57	5100	7.75
2702	74.00	5211	43.50

Figure 1.1a. Sample master file for sequential file processing example.

corrected and either reinserted in the deck or saved until the report for the next month is run. We shall do the first. If there are only a few error cards, they can be inserted by hand; if there are many, another collator run can be used to insert them.

3. It could happen, in a mixup in card handling, that one or both of the decks are out of sequence. The collator can be set up to check for this possibility and stop if an error is detected. Such errors are not an everyday occurrence, but since they *can* happen and since they are fairly easy to check we may as well do so.

Before describing the rest of the procedure, we may review what has been covered so far in terms of some sample data. In a typical application of this type, the master file would contain perhaps 10,000 records and there might be 20,000 sales cards in a month. The sample data is based on a master file of 16 records and also 16 sales cards, as shown in Figures 1.1a and 1.1b.

The first step, sorting the detail deck, puts the details into the order shown in Figure 1.2. The merged deck is shown in Figure 1.3, in which asterisks are written after the master file product numbers for clarity. Note that the master card appears in front of its associated detail cards.

When the two decks are merged, the sales card for product number 4190 will fall out as an unmatched detail. Suppose that investigation shows that product number to have been incorrectly punched; it should have been 1190. After the card has been repunched, it can be placed anywhere in the group of sales cards for product 1190; it is shown in Figure 1.3 at the front of the group.

Now we are ready for the first calculation step. We must get the unit price of each product, *extend* the price for each sale (multiply unit price by the number of units sold), summarize the total sales dollars for each product (add up the price of each sale), and summarize all sales for the month. This

Detail File

Product Number	Quantity	Salesman	District
4992	8	31	3
4192	12	20	1
1190	55	32	3
1213	2	20	1
1655	80	31	3
4190	100	41	2
4992	11	10	1
5062	20	6	2
1655	20	6	2
1213	1	41	2
5062	75	32	3
1190	30	10	1
1190	16	61	3
1655	150	61	3
4192	7	32	3
1656	4	6	2

Figure 1.1b. Sample data (unsorted) for sequential file processing example.

Product Number	Quantity	Salesman	District
1190	55	32	3
1190	30	10	1
1190	16	61	3
1213	2	20	1
1213	1	41	2
1655	80	31	3
1655	20	6	2
1655	150	61	3
1656	4	6	2
4190	100	41	2
4192	12	20	1
4192	7	32	3
4992	8	31	3
4992	11	10	1
5062	20	6	2
5062	75	32	3

Figure 1.2. Sorted detail file for sequential file processing example.

Product Number	Unit price or Units Sold	Salesman	District
1120 *	6.90		
1190 *	4.32		
1190	100	41	2
1190	55	32	3
1190	30	10	1
1190	16	61	3
1200 *	10.60		
1213 *	25.50		
1213	2	20	1
1213	1	41	2
1655 *	.80		
1655	80	31	3
1655	20	6	2
1655	150	61	3
1656 *	18.00		
1656	4	6	2
2441 *	2.57		
2702 *	74.00		
3495 *	2.70		
4192 *	8.09		
4192	12	20	1
4192	7	32	3
4377 *	21.90		
4992 *	10.20		
4992	8	31	3
4992	11	10	1
5009 *	8.00		
5062 *	1.47		
5062	20	6	2
5062	75	32	3
5100 *	7.75		
5211 *	43.50		

Figure 1.3. Merged deck for sequential file processing example.

is now easily done. As the combined deck is read by the computer, the unit price for each product can be obtained from the master card and stored for computing the price of each sale. As each sales card is read, the number of units can be multiplied by the unit price and this sale price added to the total of sales for the product. For later operations a new detail card which contains all of the old information plus the extended price of each sale must be punched. When all of the sales cards for one product have been read and extended, the total sales for that product must be printed. When all cards have been read, the total sales for the month is printed. Finally, as the cards are read, the original details (which may now be discarded) should

be stacked separately from the masters (which must be saved for use next month).

When this first computer operation is completed, the master deck will be unchanged. The new detail deck will be the same as the original except that the price of each sale will be punched on each sales card. The product summary will be as shown in Figure 1.4.

Before proceeding with a description of the remainder of the processing (the summary of sales by salesman and district), we may investigate a way of presenting graphically the steps so far covered.

This may be done with a *work-flow chart*, or simply *flow chart*. A flow chart is a graphic representation of the complete system in which the input data is converted to final documents. In other words, a flow chart shows *what* the major processing steps are, without detailing *how* they are done; detailing is the concern of a *block diagram*, to be discussed later.

A flow chart uses lines and arrows to connect symbols that stand for documents and operations. Some of the standard symbols are shown in Figure 1.5. They are most easily drawn with the IBM Charting and Diagramming Template. A *source document* is any representation of information that becomes input to a data processing operation. In our example the only source documents are the sales reports. The symbol shown for a file applies only to a card file, of course; the file concept is broader than its card implementation, as we shall see.

The *card punch* is naturally a device for punching holes in cards. In our example it happens that only numbers are punched; most card punches can

Product Number	Total Sales
1190	868.32
1213	76.50
1655	200.00
1656	72.00
4192	153.71
4992	193.80
5062	139.65
	1703.98

Figure 1.4. Summary of sales by product in sequential file processing example.

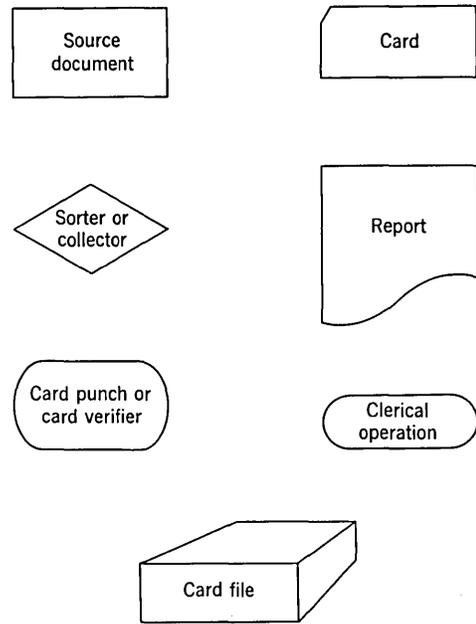


Figure 1.5. Some of the standard flow charting symbols and their meanings.

also punch letters and certain other symbols. The card verifier looks about like the card punch but has no mechanism for punching holes. Punched cards are run through the verifier, with the verifier operator pressing the keys in the same way (hopefully) as the punch operator did. When a key on the verifier is pressed, the equipment checks that the key corresponds to the hole punched in that column. If the entire card proves to be correct, it is notched at the end to show that it has been verified. If the card is wrong or the verifier operator makes a mistake, a red light is turned on. The verifier operator can now try again; if the card is actually wrong, it is notched at the top to show that it must be repunched. It is, of course, possible that the two operators will make the same mistake, but this is sufficiently unlikely to make the technique acceptable in most situations.

With a flow chart we can represent the operations in our example that have been described so far, as shown in Figure 1.6. It may be seen that the pictorial representation is much easier to follow than the verbal description, once the basic concepts are understood.

With the simplification made possible by the tool of the flow chart, the rest of the procedure is much easier to describe. We are required to produce a summary of sales by district and salesman. To do so, the new details must be resorted, so that

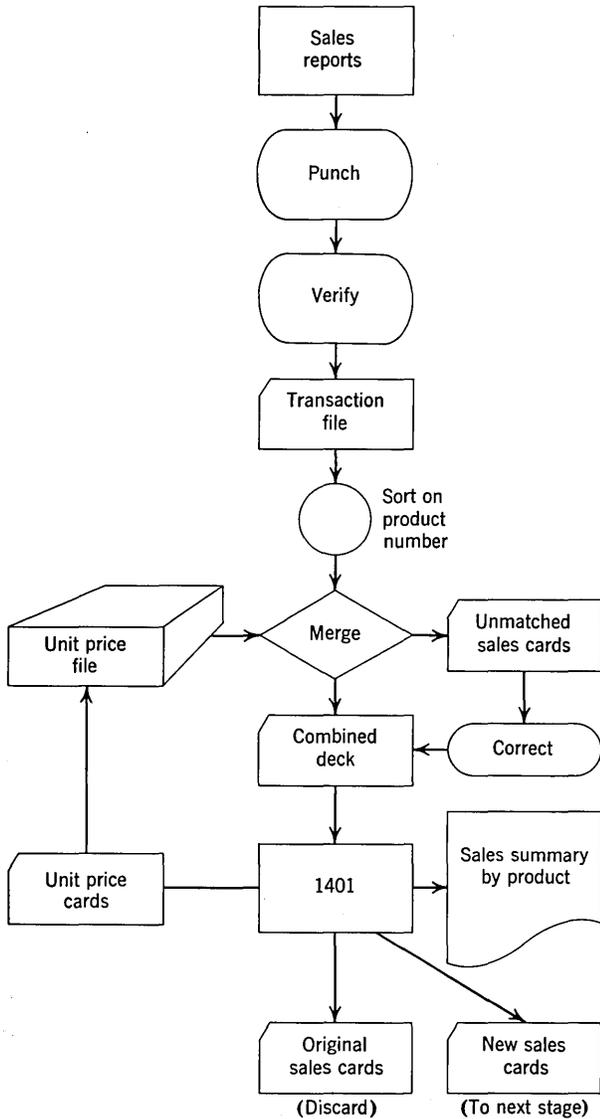


Figure 1.6. Flow chart of sequential file processing example, through the sales summary by product.

all the cards for one district are together; within each district, all the cards for each salesman must be together. After sorting, another computer run produces the summary required. As the cards are read by the computer, the sales total for the month is computed again. This figure obviously should be the same as the total at the end of the previous summary to give a check on the correctness of the processing. A number such as this is often called a *control total*; the term is also used to denote a total that has no other purpose than that of checking accuracy.

The flow chart of this part of the processing is shown in Figure 1.7. The new details, after they have been sorted by salesman and district, appear in Figure 1.8, and Figure 1.9 is the sales summary by salesman and district.

This discussion has said nothing about how the various computer operations are to be carried out. We are still not prepared to go into the details of this matter, but we can at least consider the overall picture of what the major computer operations are and their sequence. For this purpose it is convenient to employ a *block diagram*, which is considerably more detailed than a flow chart. A flow chart outlines the major steps in the processing as the work "flows" from machine to machine. A block diagram, on the other hand, shows *how* the task of each machine is accomplished. In this book our primary concern is the computer, and block diagrams are used to give the sequence of data movements, computation, and decisions on which the computer is operating.

The symbols used in flow charting are also used in block diagramming but with different meanings. Figure 1.10 explains the meanings attached to the symbols needed in this example.

Figure 1.11 is a block diagram of the computer operation in the second part of our example, the

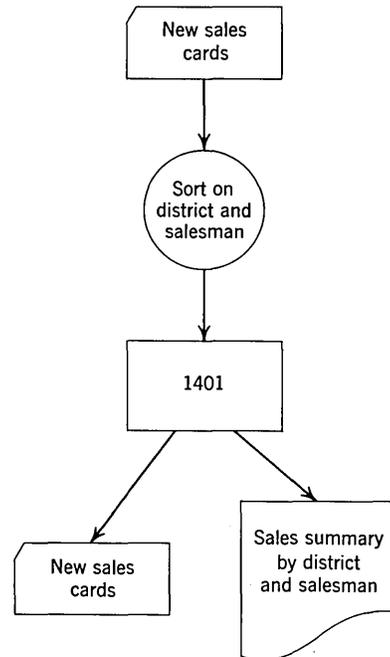


Figure 1.7. Flow chart of sequential file processing example, from new sales cards to sales summary by district and salesman.

summarization by month, district, and salesman. The first step is to read a card. This means that the information on the card is copied into the storage of the computer, where it is available for later operations. Some of this data is then copied in other places in storage so that it will be available after the data from another card has been copied into the storage areas once occupied by the data from the first card. These data transfers are shown in three different boxes in the block diagram because some of them are used at different times later. An arrow, in this connection, means "goes to."

When the first card has been read and the data

Product Number	Quantity	Salesman	District	Sales Price
1190	30	10	1	129.60
4992	11	10	1	112.20
1213	2	20	1	51.00
4192	12	20	1	97.08
1655	20	6	2	16.00
1656	4	6	2	72.00
5062	20	6	2	29.40
1213	1	41	2	25.50
1190	100	41	2	432.00
1655	80	31	3	64.00
4992	8	31	3	81.60
1190	55	32	3	237.60
4192	7	32	3	56.63
5062	75	32	3	110.25
1190	16	61	3	69.12
1655	150	61	3	120.00

Figure 1.8. Sorted new details for sequential file processing example.

Salesman	District	Total
10		241.80
20		148.08
	1	389.88
6		117.40
41		457.50
	2	574.90
31		145.60
32		404.48
61		189.12
	3	739.20
		1703.98

Figure 1.9. Summary by district and salesman.

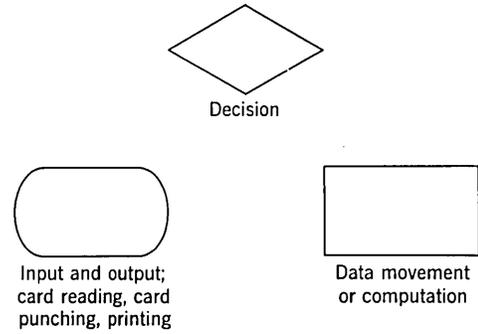


Figure 1.10. Some of the standard block diagramming symbols and their meanings.

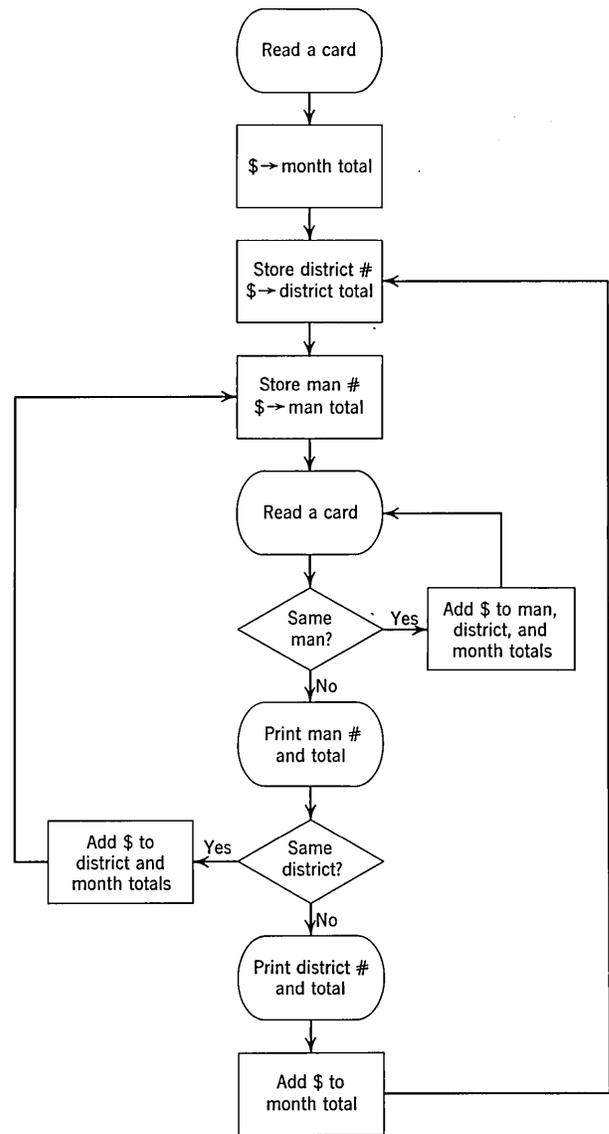


Figure 1.11. Block diagram of the computer run to produce a sales summary by district and salesman.

moved, another card is read. Now a decision is necessary. Does this card belong to the same salesman? If it does, then the price of the sale should be added to each of the three *accumulators* that will develop the three totals. If it refers to a different salesman, then the number and total of the previous man should be printed. Next, a check must be made to determine whether this card refers to the same district. If it does, then the sales price should be added to the district and month totals, the salesman number stored as the one against which to compare the next card, and the sales price stored in the accumulation for that man. This will destroy the previous salesman total, which has now been printed. Next another card is read and the process is repeated. If the second test indicates a new district, then the district total must be printed and the other operations carried out as shown.

The student is strongly urged to follow through this block diagram in detail, using the data presented previously. The test required to detect the last card of the deck is not shown here; Exercise 3 considers this problem.

This example brings out a number of important concepts, which may be summarized as follows:

1. *The unit record concept.* A punched card has the important advantage that it moves as a unit; this is not necessarily true of other storage media. When the key is used to control sorting or collating, all the other information on the card moves with the key. For this reason, the conventional IBM card equipment is often referred to as *unit record* equipment. (Of course, this example can be done entirely with unit record equipment; in fact, it is a typical application.)

2. *Control levels.* The salesman number controls one level of totals and the district number, a higher level that includes the salesman totals. The total by salesman is called a minor total, and the total by district, a major total. There could also be one or more intermediate levels of control; for example, if each district had branches out of which the salesmen worked.

3. *Sequential file processing.* When the files to be processed can be done only sequentially, it is necessary to arrange all of the files into the same sequence before proceeding. This becomes the basic consideration in organizing the processing.

4. *Batch processing.* Since it is necessary to read the entire master file, including those records not affected, in order to process even a few trans-

action records, it is necessary to save the transactions until a *batch* has been accumulated. In many applications, as in the example above, this is a natural mode of operation; in others, it is a decided disadvantage, and we turn to the *random access* file storage methods.

REVIEW QUESTIONS

1. Describe how you would look up a telephone number if the directory had to be "processed" sequentially. What is the "key"?
2. Why would it not be feasible to have several sales reports punched on each card of the transaction deck?
3. In the example there can be several details with the same product number. Could there ever be several masters with the same product number?
4. What would happen in the merging operation if the last card of the sorted transaction deck were inadvertently placed at the beginning of the deck?
5. In the sample data for this example there was a misspelled sales card which was detected because there was no master card corresponding to the incorrect product number. Suppose the erroneous product number had been the same as some master card product number. Would the error have been detected?
6. Suppose that in the various card-handling operations between the two summarization runs one sales card got lost. What would signal the error?
7. If a salesman could report sales in more than one district, could the salesman and district summaries be produced in one summarization run?

1.4 An Example of Random Access File Processing

The outstanding feature of sequential file processing is that the *entire* file must be read each time *any* transactions are processed against it. This, in turn, forces us to sort the transactions so that the master file need be read only once. Furthermore, it is necessary to accumulate the transactions into batches of fair size before doing any processing in order to reduce the number of times the entire file must be read.

In many cases the nature of the job is such that these factors are not actually restrictions. For instance, in the example of the last section, it is completely natural to accumulate a month's sales cards before running the monthly summary. The effort of sorting is more than compensated by the economy of sequential file storage media.

In other situations, however, the application de-

mands that records be kept on a current basis or that the information be more readily available than it usually is with a sequential file. In such cases the random access file becomes necessary.

For an illustration, let us consider the problem in the last section but with two additional requirements. Besides preparing monthly sales statistics, we are required to keep records on the inventory of each product and to be able to answer on short notice several kinds of inquiries on inventory status and sales position.

As before, there is a master file consisting of one record for each product, but now the records contain more information. Besides the product number and the unit price, each record contains the number of units available for sale and the accumulated sales amount for the month. There is also a separate record for each salesman and each district showing sales for the current month.

As the sales information is received at the data processing center, cards are punched and processed against the master file immediately, perhaps as often as several times a day. The processing of a sales card now consists of the following steps:

1. Locate the master record for the product number.
2. Determine whether there is enough of the item in stock to be able to fill the order. If so, proceed with the processing; if not, write an out-of-stock notice.
3. Subtract the number of units sold from the number of units in stock.
4. Extend the price, that is, multiply the number of units by the unit price.
5. Add the total sales price to the accumulated sales of the product for the month, which is now also in the master record.
6. Write the modified master record back in the file.
7. Locate the record for the salesman who sold the order, add the total sale price to the total of his sales for the month, and write the updated record back in the file.
8. Do the same for the district in which the salesman works.

This procedure is shown in the block diagram of Figure 1.12. The only new technique here is the location of the proper master record. How this is accomplished depends on the physical device used

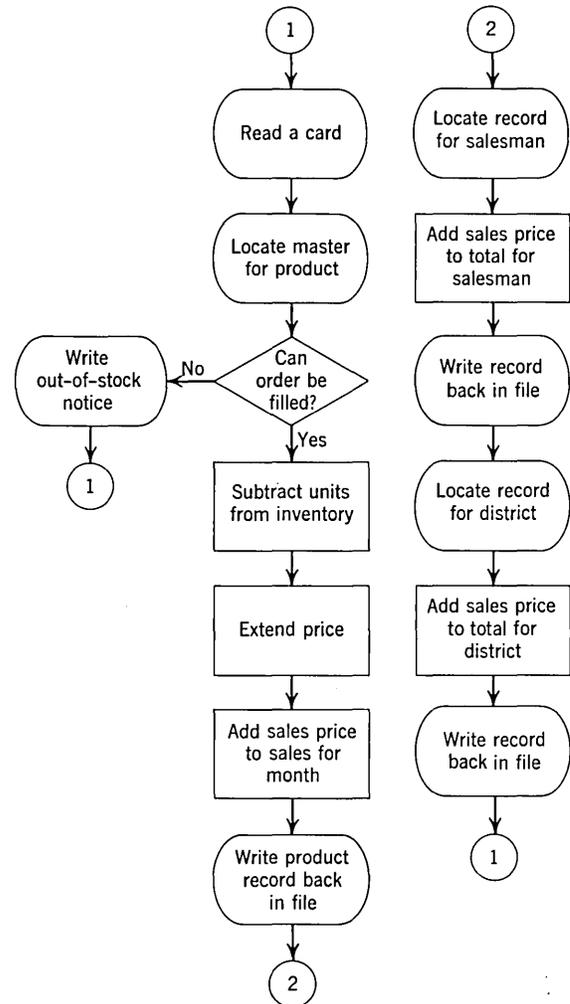


Figure 1.12. Block diagram of the computer operations in updating a sales statistics file, using a random access storage device.

to store the master file, a subject to which we shall return in Chapter 9.

So far we have considered the handling of the sales cards. The complete system must obviously be able to do considerably more. If it is to keep inventory records, there must clearly be some way to enter information on the addition to stock of items manufactured by the company or received from suppliers. This can easily be done by setting up a card rather similar to a sales card—that is, containing a product number and a number of units—but which also contains a code that will be recognized by the computer program as indicating an addition to stock rather than a sale.

There must, as before, be a procedure for producing sales summaries. Besides the monthly sum-

mary, however, it is now possible to provide a summary on demand at any time during the month. This would not ordinarily be a complete report but only the statistics on selected items. Such spot summaries could be requested by other specially coded cards similar to the sales card, showing the product, salesman, or district for which a summary is desired.

It goes almost without saying that this is a considerably oversimplified example. The inventory portion, in particular, does not take into account many factors that would be required in any actual application.

Comparing this procedure with the sequential file processing example, the following characteristics stand out:

1. Sales reports can be processed as quickly as they are received rather than as batches are accumulated, since only the correct master records need be read. The master file information is thus always up to date.
2. Information from the master file is available on demand, with little delay.
3. No sorting of the transactions is required.

Why, then, are random access files not universally used? The simple answer is that the same capacity costs more in a random access storage medium than in a sequential access medium. Furthermore, many applications have little need for hourly or daily availability of the latest information; for them, a random access file represents a pointless expense.

In short, when immediate access to master file information is not needed, a sequential file is perfectly adequate and is less expensive. When immediate access is essential, a random access file provides conveniences that more than compensate for the additional cost.

REVIEW QUESTIONS

1. What is the most important characteristic of a random access file?
2. Would it introduce any complications in this example if a salesman could work out of more than one district?

3. In this example we did not mention error checking. This is partly because there are fewer card-handling steps where errors could be made, but at least one of the error checks in the sequential file version can still be made. What is it?

4. At the end of each month a sales summary for the month would be produced. After doing so, what should be done to the file information to prepare for accumulating the next month's statistics?

EXERCISES †

*1. In the block diagram of Figure 1.11 insert the additional operations necessary to produce a count of the number of sales made by each salesman.

2. In the block diagram of Figure 1.11 insert the operation necessary to sequence-check the detail cards on district number—that is, determine that no district number is smaller than the preceding one.

*3. Suppose that after reading the last card of the deck an indicator is automatically turned on. Insert in the block diagram of Figure 1.11 the operations necessary to test such an indicator and to use the result of the test to wrap up the processing when the last card has been read. This will require printing all three totals, including now the total for the month.

4. Draw a block diagram of the computer operations in the summarization by product in the sequential file version. Assume that there is some simple way to distinguish between a master and a detail, for example, a decision box which asks "master or detail?" following the reading of a card.

Suggestions. Draw the block diagram first without any consideration of how to start or end the process. This will be the heart of the run: obtaining the unit price of each product, extending each detail, and getting the summary for each product. To do this much, it will be necessary to use the detection of a new master card to cause the printing of the summary for the preceding product and to store the new unit price for extending the next details. Be sure that nothing is printed when two masters in succession are read and that after printing each summary the total storage area is cleared.

After this part has been worked out, it should not be too difficult to add the operations necessary to start properly and to make use of a last card test for stopping.

*5. Suppose that the sales card in the sequential file in this example had only the salesman number and no district number. There is an auxiliary master file that gives the district in which each salesman works; this file is in salesman-number sequence. Extend the flow chart of Figure 1.6 to include the steps necessary to punch new sales cards with district numbers.

† Answers to starred exercises are given at the end of the book.

2. INTRODUCTION TO COMPUTING EQUIPMENT

In Chapter 1 we surveyed some of the methods and concepts used in electronic data processing. In this section we introduce the "hardware" that carries out these operations. After these preliminaries, the next chapter begins the discussion of specifying the nature of the desired processing to the data processing system.

2.1 The IBM Punched Card

The starting point for entering data into most electronic data processing systems is the punched card. We must therefore become quite familiar with the way punched cards are used to contain and transmit information.

An IBM card is a piece of light flexible cardboard $7\frac{3}{8}$ wide and $3\frac{1}{4}$ in. high. It is composed of 80 vertical columns, numbered from left to right. Each column may contain one of the digits 0 to 9, the letters A to Z, or any special characters such as the dollar sign or per cent sign. There are 12 vertical punching positions in each column, of which the punching positions for 0 through 9 are identified by printing on the card. Numerical information is recorded on the card by punching a single hole in a given column in the position representing that digit. For example, a single hole punched in the 2 position always means the digit 2 to IBM machines.

Alphabetic information is represented by a combination of two punches, a *numerical*

punch and a *zone* punch. Positions 1 to 9 are referred to as the numerical positions. There are three zone punching positions:

12 zone—at the top edge of the card.

11 zone—just below the 12 zone position.

Zero zone—just below the 11 zone position.

A punch in the 12 zone position is sometimes called a Y punch and a punch in the 11 zone position is sometimes called an X punch. This terminology has nothing to do with the representation of the letters X and Y, and because of the possible confusion it is not used here. The zero zone position is the same as the numerical zero and is labeled on the card. The 12 and 11 zone positions are not labeled, since this part of the card is usually set aside for the printing of headings.

The codes (combinations of punches) for the letters of the alphabet are shown in Figure 2.1. To understand the basic idea of this coding, it may be helpful to note that the letters A through I are made up of a 12 zone punch and one of the digits 1 through 9; the letters J through R are made up of an 11 zone punch and one of the digits 1 through 9; the letters S through Z are composed of a zero zone punch and one of the digits 2 through 9. The first letter represented with a zero zone is composed of a zero zone and a numerical punch 2, not 1.

The combination of a zero zone and a numerical 1 is used to represent the symbol / (slash). The various other special symbols are made up either of a 12 zone or an 11

A 12 and 1	J 11 and 1	S 0 and 2
B 12 and 2	K 11 and 2	T 0 and 3
C 12 and 3	L 11 and 3	U 0 and 4
D 12 and 4	M 11 and 4	V 0 and 5
E 12 and 5	N 11 and 5	W 0 and 6
F 12 and 6	O 11 and 6	X 0 and 7
G 12 and 7	P 11 and 7	Y 0 and 8
H 12 and 8	Q 11 and 8	Z 0 and 9
I 12 and 9	R 11 and 9	

Figure 2.1. Punched-card coding of the letters of the alphabet.

zone alone, or the combination 8-3 or 8-4 alone or with the various zone punches. Thus an ampersand (&) is represented by a 12 zone only; a minus sign (-), by an 11 zone only. The per cent sign is represented by the combination of an 8, 4, and zero zone. The combinations used for all of the standard allowable characters on an IBM card are shown in Figure 2.2. The word *character* describes any digit, letter, or special symbol that can appear in one column of an IBM card.

There is no need to memorize these codes because they are automatically punched by the depression of the keys on the card punch and are read automatically by the various IBM machines that can accept information from cards. Furthermore, a card punch can be equipped with a printing device that prints at the top of the card the character

represented by each column. For cards produced by some means other than a card punch, the characters represented by selected columns can be printed at the top of the card by a machine called an *interpreter*.

Some cards have a distinctive colored stripe or have one of their corners cut. These features are provided for ease of handling and recognition by machine operators and have no meaning to the computer. When it is necessary for any of the various machines to distinguish between different types of cards, the characteristic information must be punched in the card. This may be done in many ways. One of the most common is the use of a 12 or 11 zone punch as an identification. For instance, in the example in Section 1.3 the master cards might have been identified by an 11 punch in some column set aside for this purpose.

In normal usage columns on an IBM card are grouped into *fields*. A field is composed of one or more columns which together express one piece of information. For example, Figure 2.3 shows a card layout for the sequential file processing example of Section 1.3. On this card columns 1 to 4 make up the product number field. The card punch operator will always punch the digits representing the product number in these columns, and other card machines and the computer will be set up to recognize that field as always representing the product number.

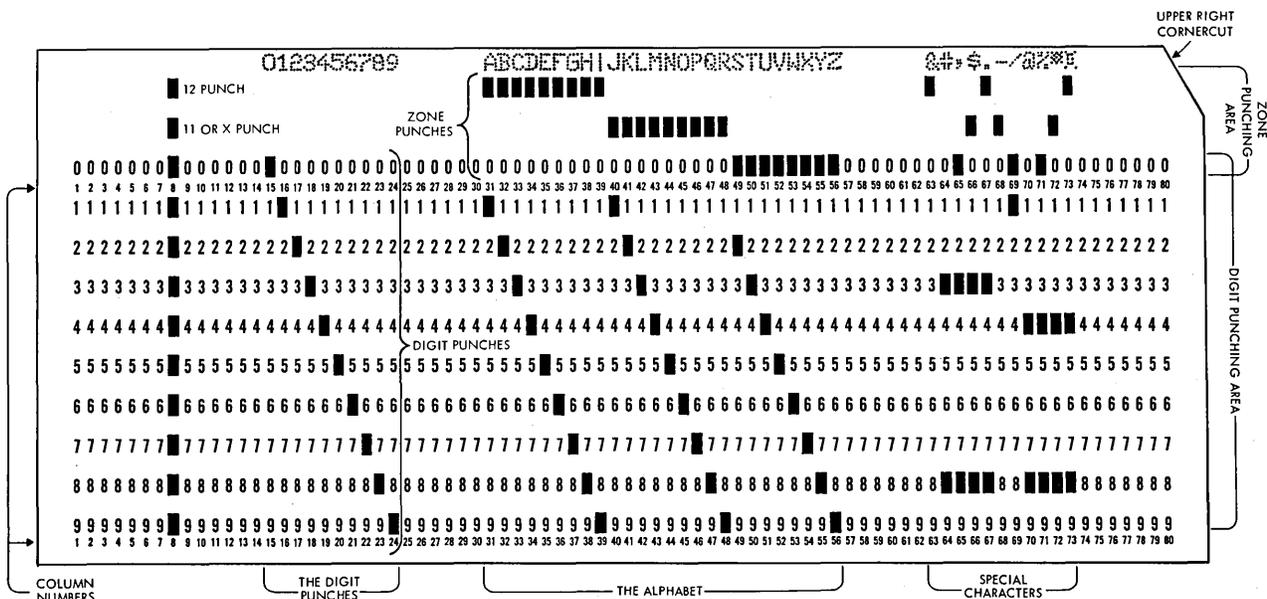


Figure 2.2. Arrangement of information on an IBM card.

in combination with a numerical punch to represent one of the letters J through R. Again, it might be used in the units position of a field in combination with one of the digits 0 to 9 to signal that the entire field is a credit amount, or, mathematically, a negative number.

REVIEW QUESTIONS

1. Using only numerical punches, it would be possible to represent letters by combinations of a single punch in each of two columns. (This is actually done to represent letters within a number of computers.) What would be the disadvantage of this scheme compared with the zone and numerical punch method?
2. How is a zero numerical punch distinguished from a zero zone punch?
3. Suppose that an 11 zone punch has been punched over the least significant digit of a field to indicate that the field is negative. If that column is interpreted (without any special control panel wiring to separate the punches), what will be printed at the top of the card for that column?
4. Why must field assignments on cards be made early in the design of a data processing procedure?

5. Give four possible meanings (in different situations) of a punch in the 11 zone position in a column.

2.2 IBM 1401 Data Processing System Components

Most computing systems are composed of a number of individual pieces of equipment that perform some part of the complete data processing operation. We may begin to get a more complete picture of what is involved in electronic data processing by considering briefly the major components of a 1401 system.

Processing unit. Every computer system has at its heart an assembly that does the actual arithmetic processing of data. The central processing unit of the IBM 1401 is illustrated in Figure 2.4. The central processing unit also contains the electronic circuitry for carrying out arithmetic and other operations on data, for interpreting the coded instructions that must be placed in storage to direct



Figure 2.4. The processing unit of the IBM 1401.

the operation of the system, and for controlling all of the other parts of the system as a result of the interpretation of these instructions. It also contains the operator's console, which provides a certain amount of access to the information in the computer and allows for manual control of the system when it is being started.

IBM 1402 Card Read-Punch. This is the component that reads the information from punched cards; that is, it interprets the punched holes and translates the information into the form in which it is stored internally. The 1402 Card Read-Punch is able to punch cards with the information that results from internal processing operations. Both reading and punching are controlled by the central processing unit as the result of suitable instructions placed in storage by the programmer. This unit is shown in Figure 2.5.

The read section, which is on the right in Figure

2.5, is able to read cards at a maximum rate of 800 per minute. The actual speed is ordinarily less than the maximum because complex processing operations require more time than is available while reading cards at top speed.

Figure 2.6 is a schematic diagram of the card transport mechanism in the card read punch. Looking at the read side, we see that a card is actually read twice. No information is transferred into storage by the first reading; the only action here is to make a count of the number of holes in each column of the card. When the information is actually read into storage, at the second reading station, a second hole count is made and compared with the first. If the two are not the same, reading stops and a red light comes on to signal the error. Some checking is performed on all input and output operations in the 1401, although the hole count method obviously can apply only to card reading and punching.

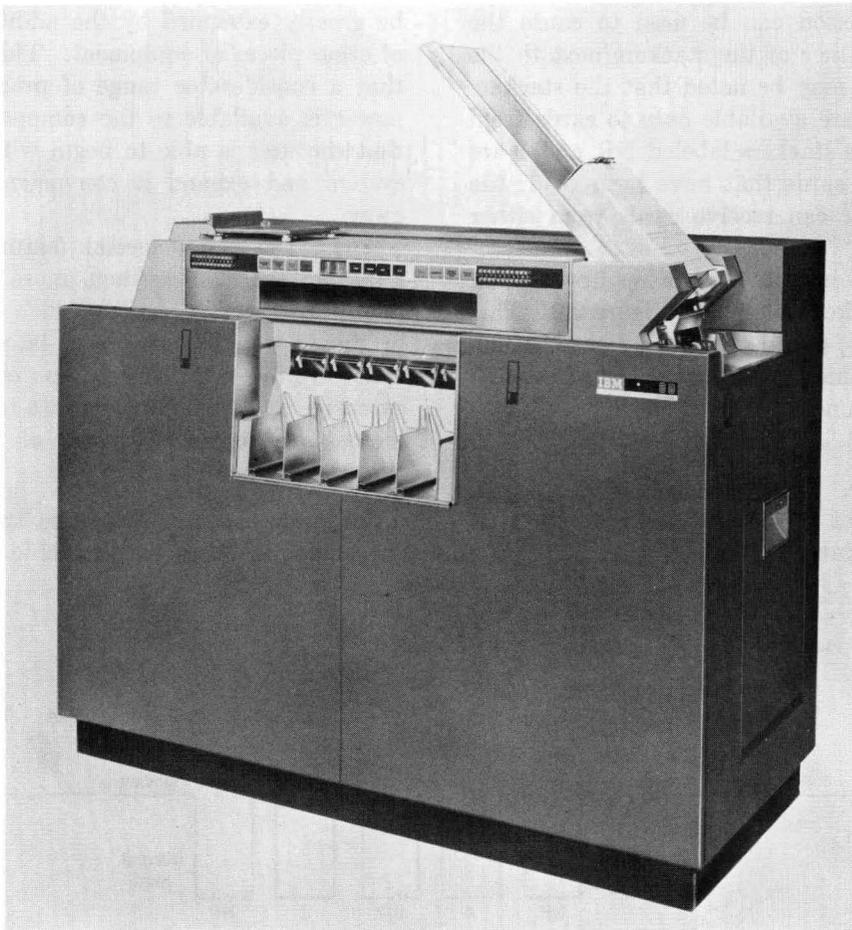


Figure 2.5. The IBM 1402 Card Read-Punch.

After the information on a card has been read into storage, the card may then be directed to one of three stackers. When no special action is taken, the card will fall into the normal read (NR) pocket. If it is desired to have the card stacked in one of the other two pockets to which a card can be directed from the read side, then a simple instruction can cause this stacker selection.

Figure 2.6 also shows the sequence of operations for punching, which can be done at a maximum speed of 250 cards per minute. A blank card from the hopper moves past a blank station and is punched, as directed by information from the central processing unit storage, at the punch station. As the holes are punched, a hole count is made on each column and this hole count is verified at the punch check station. As with reading, if the hole count is not the same for every column, the machine stops and a light comes on to signal the error.

After the card has moved past the punch check station, it is stacked. If no special action is taken, the card will stack in the normal punch stacker (NP). An instruction can be used to cause the card to stack in either of the stackers next to the normal punch. It may be noted that the stackers labeled NP and 4 are available *only* to cards from the punch side; the stackers labeled NR and 1 are available only for cards that have been read; the stacker labeled 8/2 can receive cards from either side.

It is possible to install another reading station at the position marked "blank" in Figure 2.6. This is a special feature, called *punch feed read*, which allows the programmer to read a card on the punch side and then to punch new information back into the same card.

IBM 1403 Printer. In addition to punching cards, it is possible to get information out of the 1401 system with the printer shown in Figure 2.7. This

component is able to print a complete line of 100 characters at one time (it can be increased optionally to 132 positions). The maximum speed is 600 lines per minute. Each of the 100 (or 132) positions in a line can print any one of 48 different characters; these are the 26 letters, the 10 digits, and 12 special characters.

The printing is accomplished with a chain assembly illustrated schematically in Figure 2.8. The alphabetic, numerical, and special characters are assembled on this chain. As the chain travels in a horizontal direction, each character is printed as it reaches a position opposite a magnet-driven hammer that presses the form against the chain. Before a character is printed, it is checked against the corresponding position in the print area of the central processing unit storage to insure accuracy of printer output.

The equipment described so far can be used as a complete computing system, called the IBM 1401 Card System. However, in common with most computers, the computing capacity of the 1401 can be greatly expanded by the addition of a number of other pieces of equipment. This means not only that a considerable range of price and computing power is available to the computer user, but also that the user is able to begin with an inexpensive system and expand it conveniently as his needs grow.

There are many special features that may be added to the basic system, more or less as minor modifications. We shall consider a number of them in the appropriate places in later chapters. The following three optional system elements are more extensive and, when appropriate to the application, increase the computer power of the system by a considerable amount.

Magnetic tapes. There are two principal uses for magnetic tapes in electronic data processing.

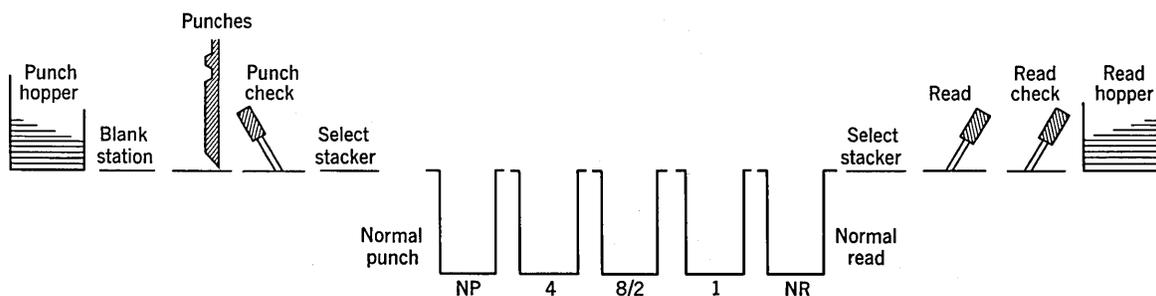


Figure 2.6. Schematic representation of the card transport mechanism in the 1402 Card Read-Punch.



Figure 2.7. The IBM 1403 Printer.

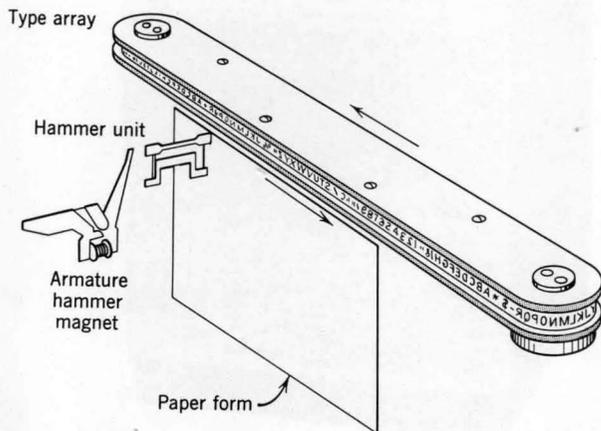


Figure 2.8. Schematic representation of the chain printing mechanism in the 1403 Printer.

The first is to increase the storage capacity of the computing system. The amount of information that can be stored in the central processing unit is limited for economic reasons. When it is desired to store large quantities of data for use during processing, it is necessary to employ some form of external storage. With the 1401, master files are frequently stored on magnetic tapes. It is still not possible to store the entire file within the central processing unit, and the principles of sequential file processing are much the same as with card files. The big advantage is that information from magnetic tape can be read and written a great deal more rapidly than cards can be read and punched. Furthermore, the same information on magnetic tape can be read repeatedly. For instance, it is possible to sort a file by using magnetic tapes without the card handling that is required with a card sorter.

The second principal use of magnetic tape is in connection with input and output. On large computing systems, such as the IBM 7080 and 7090, the speed of reading cards and of printing results is very much slower than the internal processing of information. This means that it can become uneconomical to use such a large system for input and output operations that utilize only a small fraction of the computing power of the entire system. For this reason it is preferable to transfer information from punched cards to magnetic tape, using either a special converter or a 1401 system, and then read the input data from tape. This is justifiable, as we have noted, in view of the cost of the large system and in view of the fact that tape reading can easily be 50 to 100 times as fast as card reading. The same considerations apply to output. A large system can write problem results on magnetic tape at high speed and then go on to other work while a converter or the 1401 is used to print the results from the magnetic tape.

As indicated, there are special devices that have

no other function than to perform these card-to-tape and tape-to-printer conversions. The 1401, on the other hand, can be used to check the validity of the input data as it is being read, develop control totals, and perform other operations that are described in later sections. Similarly, the large computer system can write its output information on magnetic tape in a condensed form at high speed. The 1401 can then transform the condensed output into a readable format and print it.

Some 1401 systems are intended primarily for this sort of input-output conversion and editing. The machine system intended primarily for output can be obtained without a card reader and punch, and one intended primarily for input can be obtained without a printer.

All IBM computers use the same type of magnetic tape. It comes on a 10½-in.-diameter reel with either 1200 or 2400 ft of tape. The tape itself is a plastic ribbon ½ in. wide and coated with a magnetic oxide material. A 2400-ft tape can be used to record as many as 14 million characters. The



Figure 2.9. The IBM 729 (left) and 7330 (right) Magnetic Tape Units.

format in which information is recorded on tape is considered in detail in Chapter 8.

Magnetic tape is read and written in a magnetic tape unit, of which there are three types. The IBM 729 and 7330 Magnetic Tape Units are pictured in Figure 2.9. There are two models of the 729 tape unit, designated 729-II and 729-IV. The differences between these three models are entirely in the speed with which they can read and write information. A tape may be written on one and read on any of the others; thus we say that the tapes produced by the three are *compatible*. (Tapes of different manufacturers are generally not compatible, although converters are available to translate from one type of tape to another.)

From one to six magnetic tape units may be attached to a 1401 system. Figure 2.10 shows a typical 1401 Tape System with card read punch, printer, and three 729 tape units. The spectrum of available computer systems runs from a few small computers that do not permit tapes to large systems that can handle more than 100.

IBM magnetic tape units are provided with a number of automatic checking features to insure the accuracy of transmission of data. Certain additional information is automatically written on the tape to provide part of this checking. This is the subject of parity checking discussed in connection with tapes in Chapter 8. (A similar technique is used within the central processing unit also.) A second checking feature is provided by the presence of a device to read the information recorded on tape

immediately after it is written. This is the *two-gap head* principle that is used on IBM tape units. The third checking feature is concerned with the electronics of the reading process and is called *dual level sensing*.

The IBM 1405 Disk Storage Unit. This storage unit provides the random access bulk storage described in Section 1.4. It is composed of either 25 (Model 1) or 50 (Model 2) metal disks which are coated with a magnetic oxide material. The total capacity of a disk storage unit is either 10 or 20 million characters, divided into records of 200 characters each. This is in the approximate storage capacity range of a single reel of magnetic tape. However, there is a fundamental difference between tape and disk storage, as we saw in the preceding section.

Tape must be accessed sequentially; that is to say, if the tape is positioned at its beginning, there is no way to read a record in the middle of the tape without passing over all of the intervening records. At worst, this can cost several minutes. In computations that are properly organized for the use of tapes, this is not a disadvantage. However, if it is necessary to have access to records on a random basis in which access time in minutes would be unacceptable, then the additional cost-per-character-stored of a disk system becomes justified. Any 200-character record anywhere in disk storage can be obtained in at most 0.8 sec.

Magnetic disk storage is considered external

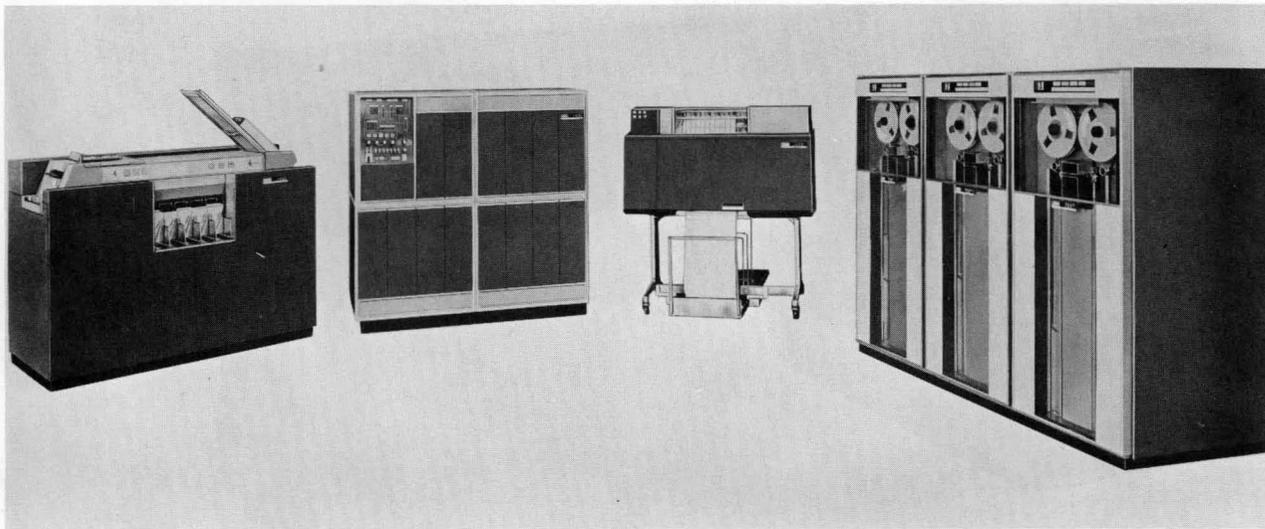


Figure 2.10. A typical 1401 Tape System with Card Read-Punch, Printer, and three 729 Tape Units.

storage, as is magnetic tape. Transfer of information between disk storage and the central processing unit must be initiated by the execution of appropriate instructions. This subject is treated in detail in Chapter 9.

IBM 1407 Console Inquiry Station. With the batch processing made necessary by the use of card or tape files, all processing requirements are accumulated until the master file is to be processed. With the random access bulk storage, however, it is feasible to set up the system to accept inquiries about the status of stored information on a random basis whenever an operator requests it. This facility is provided by the console inquiry station, which is illustrated in Figure 2.11.

When the console operator desires information from the system, he presses a request button. With appropriate instructions, the computer can detect the presence of this request and call for it to be typed in from the inquiry station typewriter. This request must be in a prescribed coded format established when the system was programmed. Instructions in the computer can then determine what in-

formation is desired, obtain it from disk storage, and type it out.

It should be realized that such requests would normally not be the major function of the computer. The computer would be set up to carry out some other primary function; the console inquiry requests would be interruptions of the primary program. Careful planning is obviously required to ensure that the main program and the console program do not interfere with each other in any undesirable way.

This facility might be used in an inventory control application in which orders are processed on a random basis as described in Section 1.4, when it is necessary to determine whether some urgent order can be filled. Facilities similar to the console inquiry station are available for many computer systems, but not all. In some cases the facilities are considerably more elaborate.

It is useful to picture the various components in terms of their relation to one another, as shown schematically in Figure 2.12. We see that the internal storage of the central processing unit is

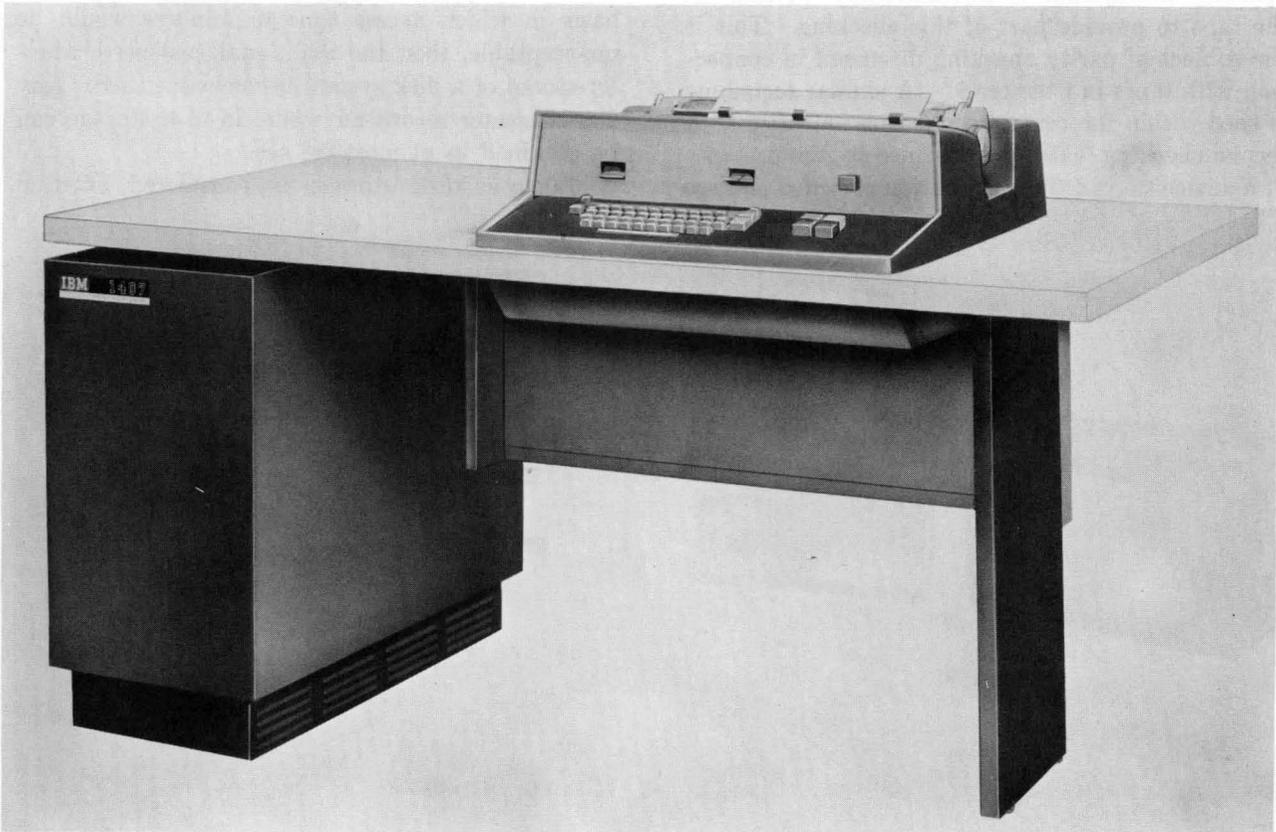


Figure 2.11. The IBM 1407 Console Inquiry Station.

The reader should satisfy himself by experimenting with an example that it is necessary to start with the least significant digit, not the most significant, remembering that after each sort pass the entire deck is picked up from the pockets and re-assembled. That is, it is not normal procedure to keep the cards from each pocket separate after a sort pass.

Sorting cards on an alphabetic control field is somewhat more complicated, requiring either two complete passes on each column or special circuitry in the sorter. It is not frequently necessary to sort

cards on an alphabetic control field, although it is not uncommon to do an alphabetic sort on magnetic tape records, using the computer.

The card collator. The IBM 85 Collator is shown in Figure 2.15. The collator has two card hoppers, called *primary* and *secondary*, the primary hopper being the one on the bottom. The collator has four pockets which are used in a way somewhat analogous to the stackers on the 1402 Card Read Punch. If we number the pockets from 1 to 4 from the right, cards from the primary feed can be moved

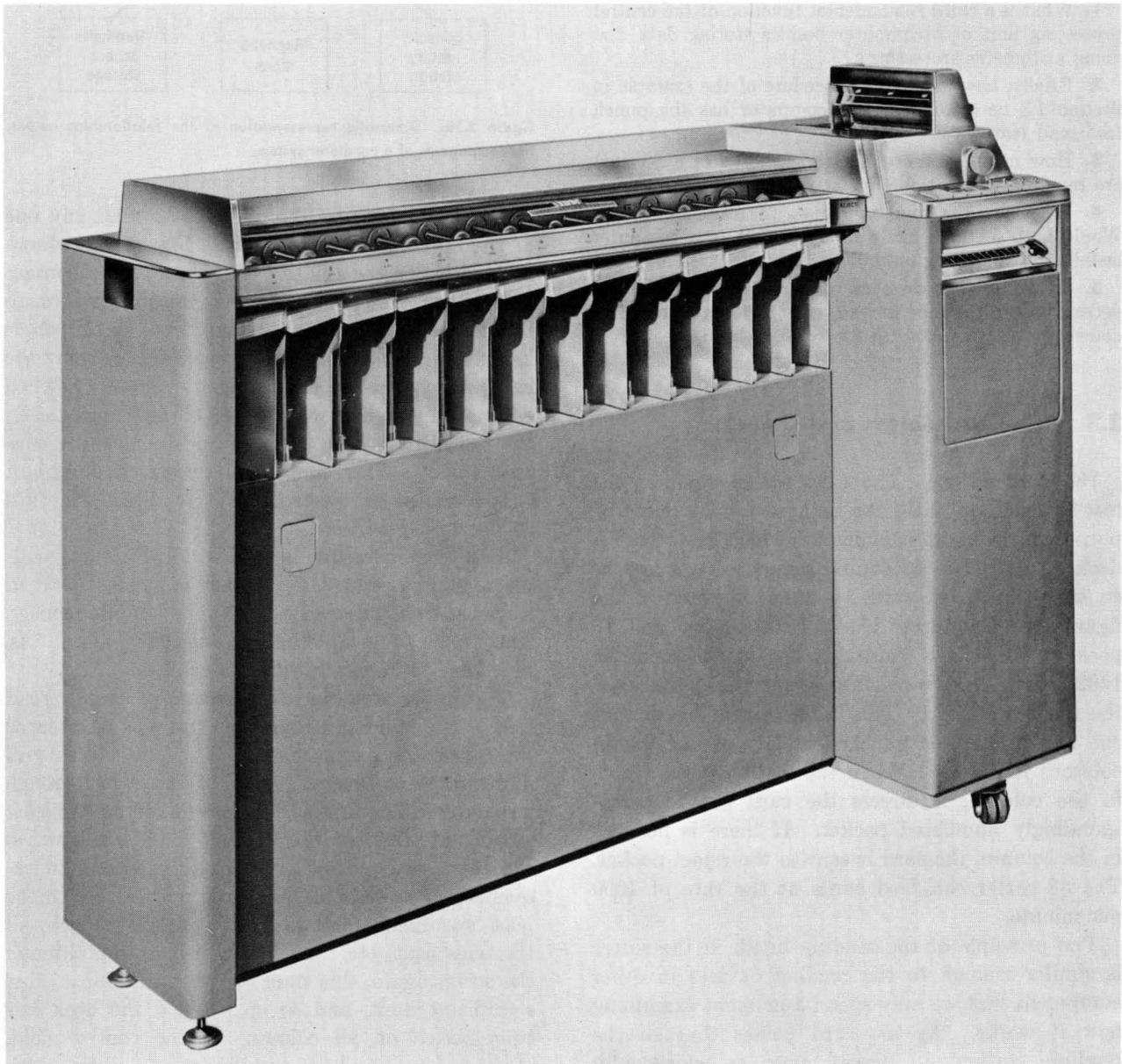


Figure 2.13. The IBM 83 Card Sorter.

the nerve center of the entire system. All input and output devices communicate with it. It contains the instructions that are used by the control section of the central processing unit to determine the actions of every part of the system. Any data to be processed by the arithmetic section of the central processing unit must be located in internal storage, although the data may have been brought to the internal storage from an external storage device such as a magnetic tape unit.

REVIEW QUESTIONS

1. What is a third fundamental function of the central processing unit of a computer besides storing data and doing arithmetic processing?
2. Briefly, how might the procedure of the example in Section 1.3 be modified if the computer has the punch feed read feature?
3. How many milliseconds (thousandths of a second) are required to print one line at full speed?
4. What are the two basic uses of magnetic tapes? Would both of them likely be applicable to a computer installation consisting *only* of an IBM 1401 system?
5. Assuming that an error in writing a tape will be detected when the tape is read, what is the advantage in detecting it *immediately* as the two-gap head does?

2.3 The Card Sorter and Collator

The card sorter. The card sorter, such as the IBM 83 in Figure 2.13, can be used for a variety of purposes, the most common of which is to sort a deck of cards into ascending sequence on a key or control field in the card. It may be noted in the figure that the sorter has a card hopper and 13 pockets, which are similar to the stackers on the 1402. As a card leaves the hopper, it moves past the reading station. This consists of a *brush* that can detect the hole punched in any one of the 80 columns in a card. When the brush senses a hole in the column, it directs the card to the correspondingly numbered pocket. If there is no hole in the column, the card is sent to the reject pocket. The 83 sorter can feed cards at the rate of 1000 per minute.

The principle of the reading brush in the sorter is similar enough to the reading devices in other equipment that we may spend a moment examining how it works. As the card passes through the reading station, it passes over an electrically charged contact roller. While the card is passing *over* the contact roller, it passes *under* a brush. The

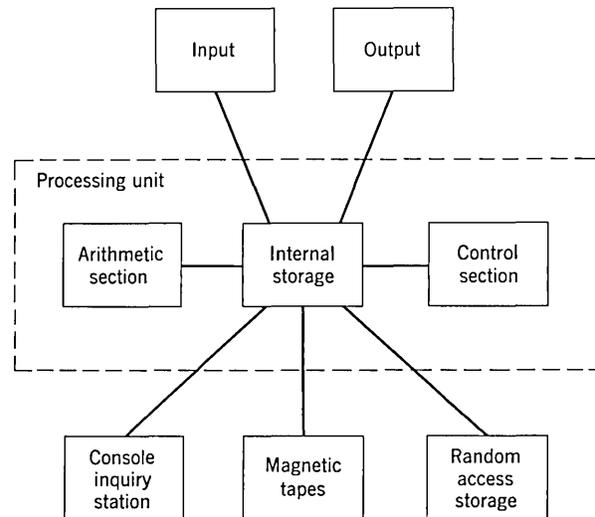


Figure 2.12. Schematic representation of the relationships among the components of a computer system.

brush may be set by the operator to read any one of the 80 columns in the card. Figure 2.14 shows schematically the relative positions of the contact roller, card, and brush. As the card passes through the machine, bottom or 9-edge first, the brush is kept from touching the copper contact roller by the card, which acts as an insulator. However, when a punched hole is reached (a 4 hole in Figure 2.14), the brush drops into the hole and touches the contact roller. This completes an electrical circuit that actuates an electromagnet to direct the card to the proper pocket.

This same principle is used in all IBM card-reading machines, except that other machines have 80 such brushes and read all columns simultaneously. The result of completing the electrical circuit is, of course, different in other machines.

The single brush in a sorter can, of course, read only one column at a time. To sort a deck of cards into ascending sequence on a control field of several digits requires several passes of the deck through the sorter. The first sort pass is made on the *least* significant digit of the control field, after which the cards are picked up from the pockets in sequence. This puts all the cards with a zero in the least significant digit at the front of the deck, all the ones next, etc. Then the deck is run through the sorter again, this time sorting on the next most significant digit, and so on. When the deck has been sorted on all columns of the control field, starting from the least significant and ending with the most significant, the deck will be in sequence on the entire control field.

The reader should satisfy himself by experimenting with an example that it is necessary to start with the least significant digit, not the most significant, remembering that after each sort pass the entire deck is picked up from the pockets and re-assembled. That is, it is not normal procedure to keep the cards from each pocket separate after a sort pass.

Sorting cards on an alphabetic control field is somewhat more complicated, requiring either two complete passes on each column or special circuitry in the sorter. It is not frequently necessary to sort

cards on an alphabetic control field, although it is not uncommon to do an alphabetic sort on magnetic tape records, using the computer.

The card collator. The IBM 85 Collator is shown in Figure 2.15. The collator has two card hopper, called *primary* and *secondary*, the primary hopper being the one on the bottom. The collator has four pockets which are used in a way somewhat analogous to the stackers on the 1402 Card Read Punch. If we number the pockets from 1 to 4 from the right, cards from the primary feed can be moved

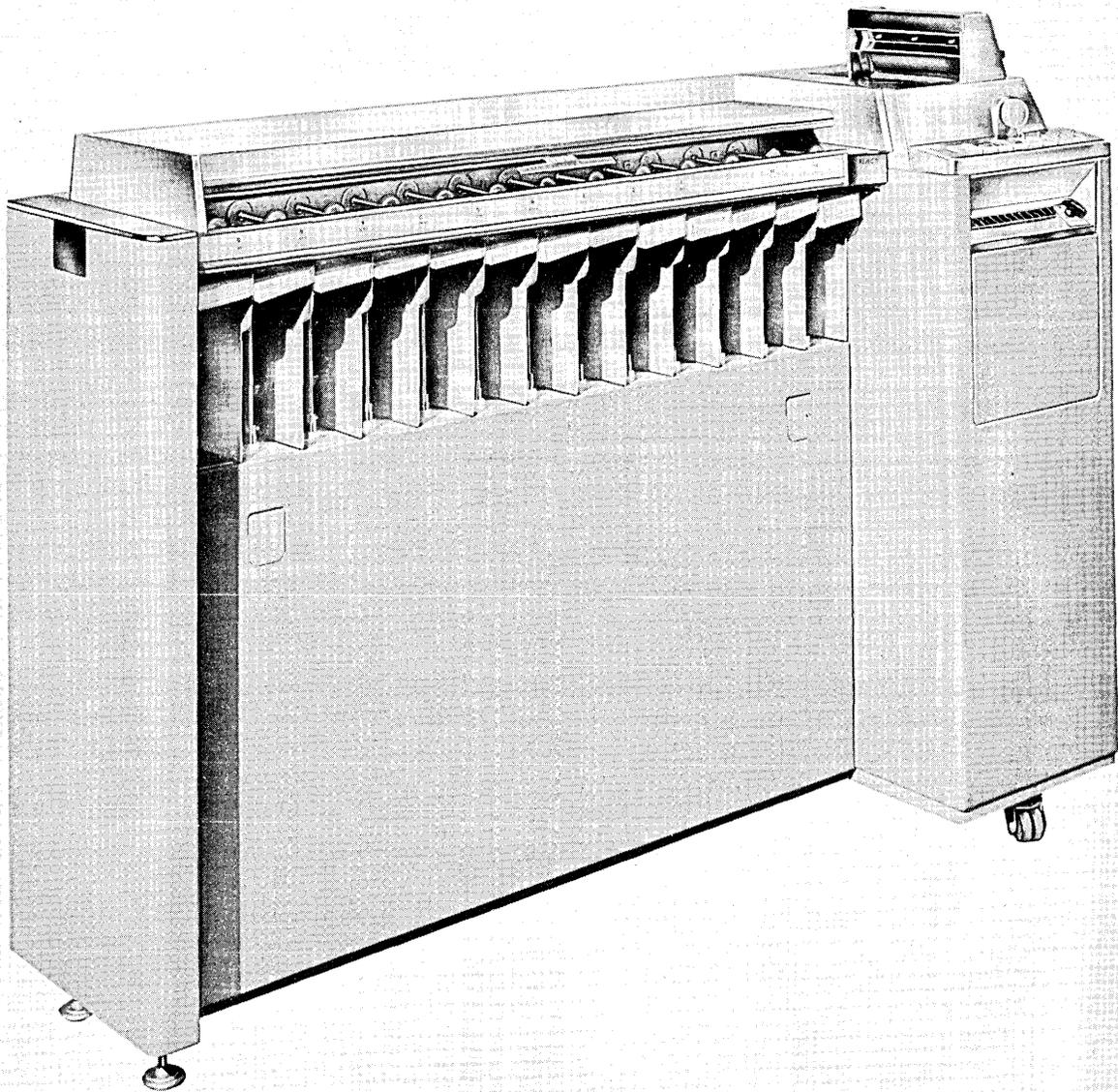


Figure 2.13. The IBM 83 Card Sorter.

to pockets 1 or 2, and cards from the secondary feed can be moved to pockets 2, 3, or 4. Thus the cards from the two feeds can be merged, which is the most common application of the collator.

The basic principle of the operation of the collator may be better understood with the help of Figure 2.16. It may be seen from this figure that there are two sets of brushes in the path followed by cards from the primary feed hopper. They are identified as *primary sequence read* and *primary read*. The secondary cards can be read at only one station. There are 80 brushes at each of these read stations so that all 80 columns can be read.

The heart of the collator's operation is based on the *selector unit* and the *primary sequence unit*. Looking first at the selector unit, we see that information can go to it from both the secondary read brushes and the primary read brushes. This selector unit is able to compare the information from

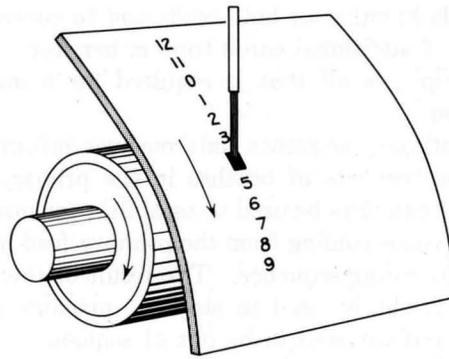


Figure 2.14. Schematic representation of the relative positions of the contact roller, card, and brush in a card-reading mechanism.

the two sources and determine whether the primary field is less than, equal to, or greater than the information in the secondary field. The selector unit output can then be used to control the stacking of

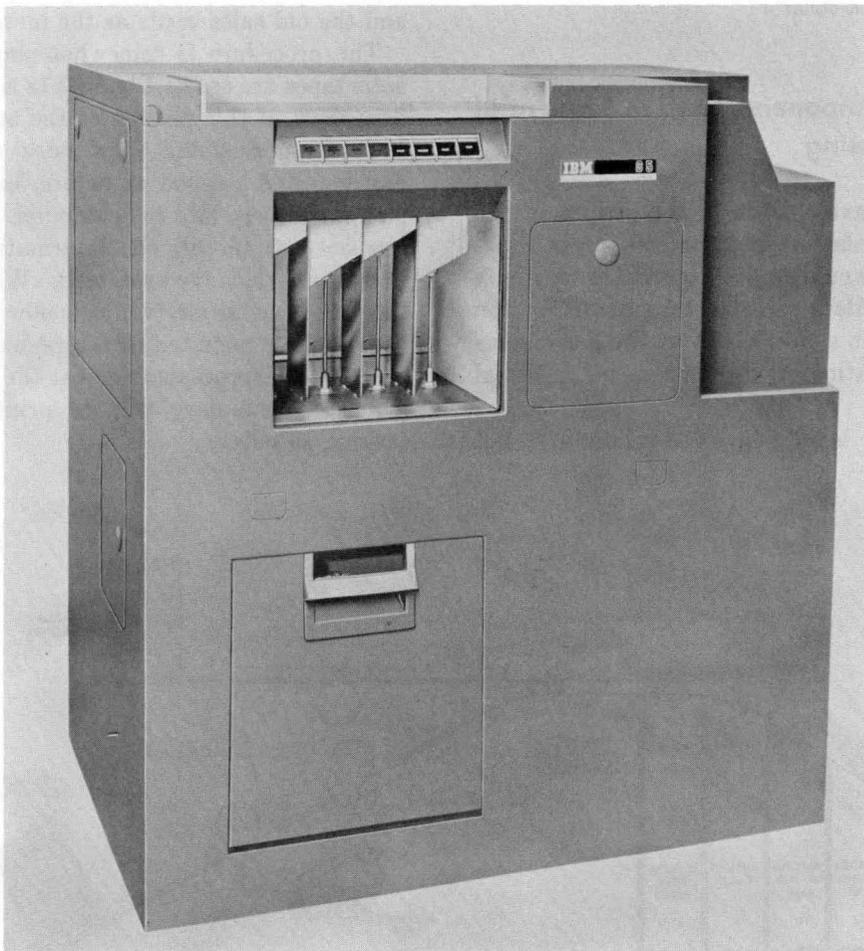


Figure 2.15. The IBM 85 Collator.

the cards in either or both feeds and to control the feeding of additional cards from either feed. This, in principle, is all that is required for a merging operation.

The primary sequence unit receives information from the two sets of brushes in the primary feed path. It can thus be used to establish, for instance, whether cards coming from the primary feed hopper are in ascending sequence. The result of this comparison might be used to stop the machine if the cards are discovered to be out of sequence.

With these basic functions, and with a control panel by which the operator can select the columns to be used to control them, a wide variety of operations can be performed.

REVIEW QUESTIONS

1. To sort a deck of cards into ascending sequence on a control field, which column should be sorted on first?
2. Can the primary and secondary cards both be sequence-checked in a collator?

2.4 System Components Used in Sequential File Processing

Now that we have a little better picture of the equipment that makes up the IBM 1401 Data Processing System, it would be well to take a new look at the sequential file processing example in Section 1.3. Figure 2.17 is a flow chart of the sales summarization application, drawn in a more informal style.

This flow chart is largely self-explanatory, but

we may note one or two of its features. After the sales reports have been punched and verified and the resulting sales cards sorted, they are merged with the master file. We note, incidentally, that the detail deck goes into the secondary feed of the collator and the master deck into the primary. If there are no unmatched details, the entire merged file will appear in pocket 2; any unmatched details would go in pocket 3. After the incorrect sales cards have been corrected, they must be placed at the proper point in the merged deck. Ordinarily, there will not be many and they can be inserted by hand. If, however, there are a great many of them, they can be merged by another collator operation. The merged deck is read by the 1402 Card Read Punch and the sales summary written by the 1403 Printer. As the procedure has been described, new sales cards are punched with the extended price of each product ordered. These appear in the normal punch pocket of the reader punch. The stacker selection can be used to separate the master file and the old sales cards as the merged deck is read.

The procedure is somewhat simpler when magnetic tapes are used. Figure 2.18 shows an informal flow chart of this portion of the application with a 1401 Tape System. The sales reports must be punched and verified as before, but they can then go immediately into the computer. Magnetic tapes can be used to sort the information on the cards after the cards have been read. When this has been done, the master file (which is now also on magnetic tape) can be mounted on a tape unit and the sorted sales records processed against the tape master file. The sales summary will be printed in the same fashion as before.

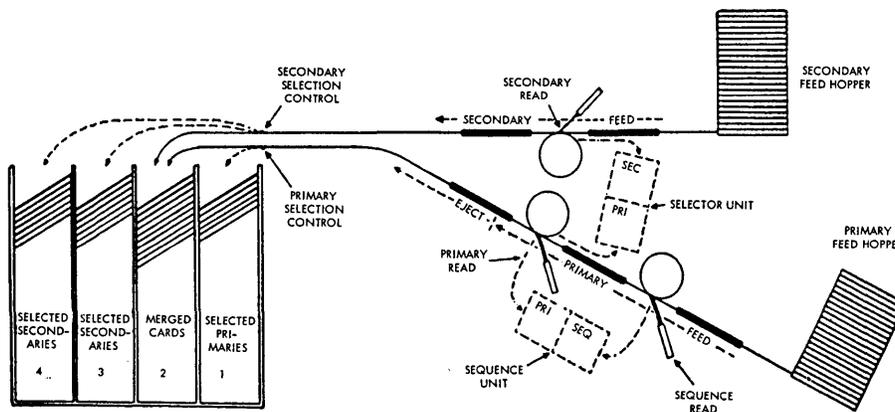


Figure 2.16. Schematic representation of the card transport mechanism of an IBM Collator.

There is one difference here: we have not shown the procedure for handling unmatched details. The proper way to handle this problem would depend somewhat on the total size of the job, the expected number of unmatched details, and the use to which the report is to be put. If in a normal month there

are only a few incorrect sales cards, then the value of the sales summary may not be diminished significantly by simply ignoring them or by making manual corrections on the report if one or two large orders were omitted. If, on the other hand, it should happen by some sort of consistent error

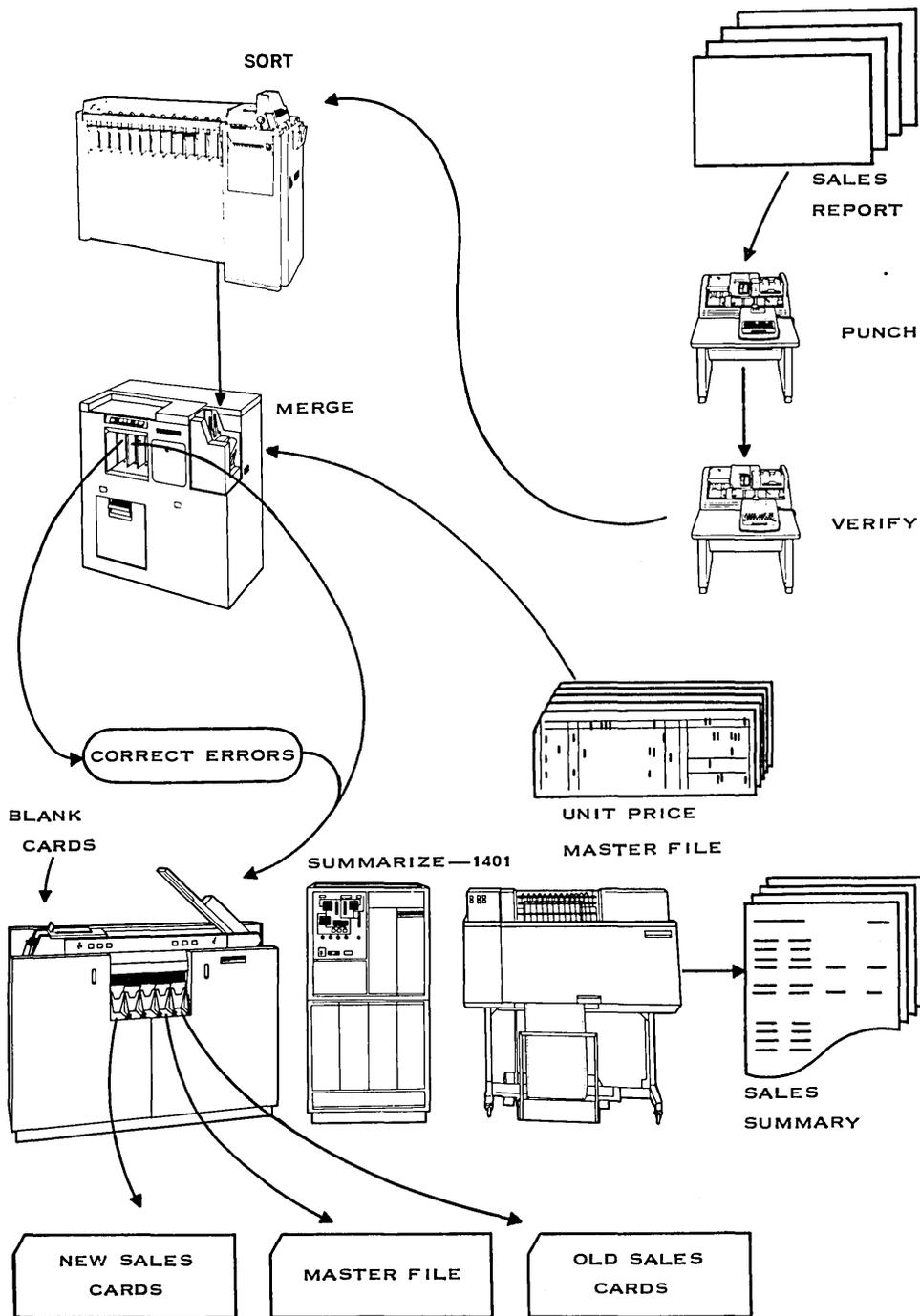


Figure 2.17. Flow chart of the sales summarization example of Section 1.3, drawn in an informal style.

that a large number of sales cards are incorrectly punched, then part of the procedure could be repeated to correct the sales summary. One way to handle the problem would be to punch a card each time a sales record was found to be unmatched.

The handling of errors is highly relevant to any

discussion of the work of programming. It often happens that these considerations require a significant fraction of the total time to plan the job. To a large extent such questions are properly the concern of the system designer who establishes error procedures before the work of the program-

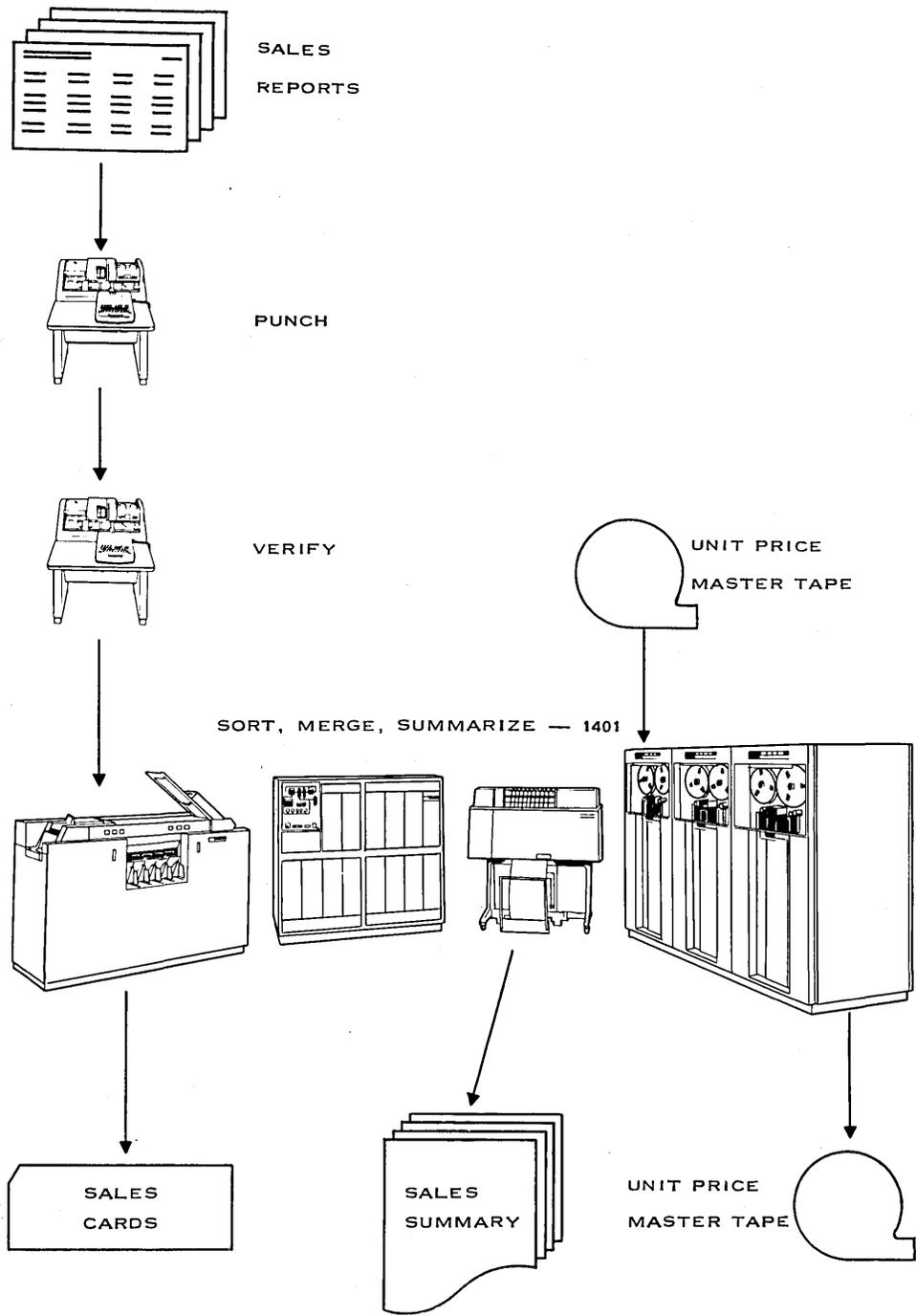


Figure 2.18. Flow chart of the sales summarization example of Section 1.3 using a tape system, drawn in an informal style.

mer begins. However, errors must always be of some concern to the programmer. Furthermore, the programmer who wishes to progress to systems work must be highly conscious of their importance.

2.5 Representation of Information in a Computer

Information in a computer is represented in a form that requires each storage or processing element to be able to take on only two distinct states. For instance, one of the most common methods of storing information within a central processing unit is by the use of a *magnetic core*. A core is a doughnut-shaped piece of a special ceramic material about the size of a matchhead. It has certain rather special magnetic properties that make it very useful in the design and construction of a storage unit. The storage unit operates so that each core is always fully magnetized in one direction or the other; a core is not allowed to operate so that its magnetization is anywhere between these two extreme conditions. Thus each core can be used to represent exactly two symbols.

Since there are 48 different symbols (and some other things) to be represented, a character must be represented within the computer by a *combination* of individual cores. Six cores would be sufficient for all of the characters because there are 64 different combinations of the directions of magnetization. As we shall see a little later, each character in the 1401 is in fact represented by eight cores, the extra two being used for other purposes.

It would be inconvenient to talk for very long in terms of "combinations of directions of magnetization of magnetic cores." We therefore look for some simpler way to describe the two directions of magnetization. Actually any two convenient symbols or terms would do: north and south, on and off, yes and no, etc. The conventional way to represent the two states of a core is to call them zero and one, but this is simply a convenience of terminology. A device or condition which has exactly two possible states is described as being *binary*. It is then said to represent a *binary digit*, which is commonly abbreviated *bit*. We speak of the method of representing the 48 characters with six bits as *binary coding*.

Figure 2.19 shows the binary coding of the 26 letters, 10 digits, and the 12 special characters. (A few other combinations used within the computer

Character	Binary Code	Card Code	Character	Binary Code	Card Code
0	1 00 1010	0	O	0 10 0110	11-6
1	0 00 0001	1	P	1 10 0111	11-7
2	0 00 0010	2	Q	1 10 1000	11-8
3	1 00 0011	3	R	0 10 1001	11-9
4	0 00 0100	4	S	1 01 0010	0-2
5	1 00 0101	5	T	0 01 0011	0-3
6	1 00 0110	6	U	1 01 0100	0-4
7	0 00 0111	7	V	0 01 0101	0-5
8	0 00 1000	8	W	0 01 0110	0-6
9	1 00 1001	9	X	1 01 0111	0-7
A	0 11 0001	12-1	Y	1 01 1000	0-8
B	0 11 0010	12-2	Z	0 01 1001	0-9
C	1 11 0011	12-3	&	1 11 0000	12
D	0 11 0100	12-4	.	0 11 1011	12-3-8
E	1 11 0101	12-5	∩	1 11 1100	12-4-8
F	1 11 0110	12-6	-	0 10 0000	11
G	0 11 0111	12-7	\$	1 10 1011	11-3-8
H	0 11 1000	12-8	*	0 10 1100	11-4-8
I	1 11 1001	12-9	/	1 01 0001	0-1
J	1 10 0001	11-1	,	1 01 1011	0-3-8
K	1 10 0010	11-2	%	0 01 1100	0-4-8
L	0 10 0011	11-3	#	0 00 1011	3-8
M	1 10 0100	11-4	@	1 00 1100	4-8
N	0 10 0101	11-5	blank	1 00 0000	

Figure 2.19. Binary coding of the characters in a standard 1401 system.

are considered later.) We see in the figure that only seven of the eight bits are shown; the eighth is called the *word mark* bit and is discussed below. The seven bits that are shown are seen to have conventional designations: CBA8421. The four rightmost bits (8421) are called the numerical bits, since they correspond in an approximate way to the numerical punches in card coding. The BA bits are called the zone bits for the same reason. The C bit is the parity bit; we shall consider its function after looking into the coding scheme displayed in Table 2.2.

We note that the decimal digits all have representations in which the zone bits are both zero; this corresponds to the fact that their card representations have no zone punches. The letters A to I have zone bits which are both 1, corresponding in a way to a zone punch of 12. Similarly, the coding of the letters J to R have zone bits of 10, corresponding to an 11 zone punch. The letters S to Z have zone bits of 01, corresponding to a zone punch of zero.

The word *parity* is used here in the sense in which it refers to oddness or evenness. A careful study of Table 2.2 will show that the representation of a character in the 1401 always involves an *odd* number of ones. This is done to provide checking of the accuracy of machine operation at certain crucial points within the machine. The number of ones in each character passing by these points is checked to determine that it is odd. If it is not, an error is signaled on the console and the machine is stopped. Parity checking, therefore, provides a very high degree of assurance that the machine is operating correctly.

The eighth bit associated with each character in core storage is called the *word mark* bit. We shall have to give this matter very careful consideration in later chapters. We may suggest for the time being that the word mark bit is used to signal to the computer the beginning and ending of fields of information in the storage. We saw previously that the assignment of card columns to fields is a matter of interpretation that must be handled by the user of the equipment; in the case of card machines this requires proper wiring of control panels. The 1401 system, however, contains no control panels, and some other technique must be used to signify within the computer where fields begin and end. This is the function of the word mark bits, which we shall be considering in much greater detail in later sections.

Algebraic signs of fields within the computer are indicated by the zone bits of the least significant digit of the field. If the zone bits are 10 (one-zero), the entire field is taken to be negative. If the zone bits are any other combination (00, 01, or 11), the entire field is taken to be positive. Unless there are special reasons to handle the matter differently, a positive field is ordinarily denoted by zone bits of 11.

REVIEW QUESTIONS

1. Can you find a relationship between the card codes for the special characters (\$, %, etc.) and their binary representations?

2. How many individual cores are required to store 1400 characters in core storage?

EXERCISES

*1. These exercises concern a tape version of the first summarization in Section 1.3. Suppose that the master file is as before, except that it is on magnetic tape. Each tape record gives the product number and unit price of one product; the file is still in product-number sequence. Assume that the sales records are on another tape and that they are already in sequence on product number. Assume, for this exercise, that there are no unmatched sales records and that there is only *one sale per product*. There will be unmatched masters, however: there were no sales of some products. Draw a block diagram of the computer operations required to

- a. read a sales record;
- b. read master records until the one having the same product number as the sales record is found.
- c. When it is found, multiply the unit price (from the master record) by the number sold; print the product number and total price of the sale.

2. Extend the block diagram of Exercise 1 to process the entire sales tape. This will require a relatively simple modification of the flow chart to return to the reading of another sales record repeatedly. To stop the process when the sales tape has been completely read, use an end-of-file test. After the last sales record, there is a special mark on the tape that indicates that the end of the file has been reached. The end-of-file indicator may be checked each time the sales tape is read. The indicator will *not* be turned on by reading the last record but by trying to read the "next" record, which will instead be the end-of-file mark. Thus when this mark is detected the processing is finished (in this version of the problem). All that need be done is to rewind the two tapes and stop. (We are still assuming only one sale per product and no unmatched details.)

3. Extend the block diagram of Exercise 2 to handle the normal condition of many sales per product. Probably the simplest way to do this is to read successive sales records, summarizing the units sold as long as a comparison shows that sales records for the same product are being read. When a new product number is detected, save that sales record until after finding the master record for the previous set of sales records and completing the processing of that set. Then pick up again with the next sales record (which has already been read, remember). *Hints.* Be sure that the comparison of successive sales records is started correctly; note that when the end-of-file mark on the sales tape is detected the processing of the last set of sales records has not been completed.

It may still be assumed that there are no unmatched sales records. Do not try to test for errors in sequencing of the two tapes and do not try to handle the possibility of tape reading errors.

3. CODING FUNDAMENTALS

In order to process data with a computer, it is necessary to provide the machine with a *program* of *instructions*. A computer instruction is a command to the machine, expressed in a coded combination of numbers and letters, to carry out some simple operation. Once the basic data processing task has been completely defined, the job of *coding* is to put together a suitable set of these elementary instructions to do the task. When the set of instructions, which is called a *program* or *routine*, is loaded into the internal storage of the computer, the instructions can be executed by the machine and the desired data processing carried out.

In this chapter we shall learn what a few of the simpler instructions are and how they operate. We must begin, however, by investigating the characteristics of the internal storage of the computer.

3.1 Computer Storage and Its Addressing

The *storage* of a computer (also sometimes called the *memory*) is the part of the machine where instructions must be placed before they can be executed and also where the data being processed by the instructions must be placed. By this definition we refer to *internal* storage; such things as magnetic tapes and magnetic disks are *external* storage. Instructions can be executed only from internal storage, and the data currently being processed must be put into internal storage before any processing can be done on it. When data in external storage is to be proc-

essed, it must first be read into internal storage by the execution of instructions.

We saw that in working with cards it was necessary to deal with groups of columns called *fields*. We saw that the interpretation of a group of columns constituting a field was completely under the control of the user of the equipment. Similarly, in working with the internal storage of a computer we must work with groups of characters, which are called *words*. A computer word may be defined as any collection of characters that is treated as a unit. For instance, when the sales cards of Sections 1.3 and 1.4 are read into computer storage, such things as the product number and the unit price are words. An instruction is also considered to be a word.

In many computers the number of characters in a word is fixed by the design and construction of the machine. A typical size is 10 characters. Such machines are said to have *fixed word length*. Other computers, including the IBM 1401, permit words of any length, from one character up to (in principle) the size of the storage. Such machines are said to have *variable word length*.

In any computer, whether of fixed or variable word length, it is necessary to be able to identify every location in storage where a word can be stored. For this purpose an *address* is assigned to every word location in a fixed word-length machine and to every character location in the variable word-length case. The addresses start at zero and run up to one less than the number of storage locations.

Note carefully that an address identifies a word *location*, not a word. For example, the

address 593 in the 1401 refers to a *place* in which a character may be stored; it does not by itself tell us what is stored there. A location that contains the character A at one time may be used a moment later to store the digit 7. We must always make a most careful distinction between the address of a location and the word or character currently stored at the location identified by that address.

The internal storage of the 1401 is made up of magnetic cores, as pointed out previously. The smallest model of the 1401 is able to store 1400 characters of instructions or data; larger versions, which store 2000, 4000, 8000, 12,000 or 16,000 characters, are available. In this book we assume a machine that can store 4000 characters. Each of the 4000 positions is able to store any one of the 48 digits, letters, or special symbols; it is also possible to store 16 other symbols that have various meanings within the computer. Thus each of the 4000 character positions is able to hold any one of 64 different characters.

Each character position is composed of eight magnetic cores, each core holding one bit. Six cores are required for six bits of this character itself, as discussed in Section 2.5. One core holds the parity bit, and the eighth is used for the word mark bit. This last has the function of defining the length of words within storage. Any character position in which the word mark bit is *one* is thereby identified as being the *high-order* (left-most) position of a word. As we shall see, word mark bits can be *set* (made one) or *cleared* (made zero) by the execution of appropriate instructions.

When a data word is referenced by the computer, it is always by the address of its low-order (right-most) character position. The machine is built so that addresses increase as the character positions are taken from left to right, which means that the low-order character of a word has the largest address. To summarize: when a character position is addressed for data, the computer takes the character in that position and all higher order (but lower address) characters as comprising a word, until it reaches a character with the word mark bit on. If the character position that is addressed has its word mark bit on, the word will consist of just that one character.

We shall see in Section 3.2, in connection with instructions, that all storage addresses are written as three 1401 characters. The first thousand addresses are written simply as numbers between zero and 999. Addresses of 1000 and over are handled

in a special manner to fit into three characters by the following scheme. The numerical parts of the three characters are always the hundreds, tens, and units digits of the address. The zone bits of the high-order (hundreds) character are regarded as the thousands digit, according to the following pattern:

If the zone bits are		then the thousands digit is
B	A	
0	0	0
0	1	1
1	0	2
1	1	3

Thus the binary coded form of the address 1234 would be

01	0010	00	0011	00	0100
1	2		3		4

(Word mark and parity bits not shown.) The address 3789 would be coded

11	0111	00	1000	00	1001
3	7		8		9

Naturally, we do not want to have to show the binary coded form of such addresses; instead, we write them as though the high-order character were the character represented by the combination of zone and numerical bits. Looking at the table on page 33, we see that the address 1234 would be written S34 and 3789 would be written G89. The complete pattern of three-character addresses is shown in Figure 3.1, for addresses up to 3999. Larger addresses are handled by using the zone bits of the units digit in a similar system.

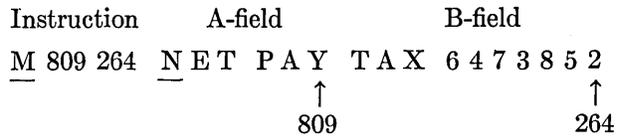
REVIEW QUESTIONS

1. What is the difference between internal and external storage?
2. Explain the following statement: instructions can be *stored* in external storage but they cannot be *executed* while in external storage.
3. What is the three-character form of the address 1643? 2700?
4. The eight character positions 678 to 685 contain the characters 93865274; underlining a character means that the character position has a word mark. If we address character position 684 for data, what word will result?

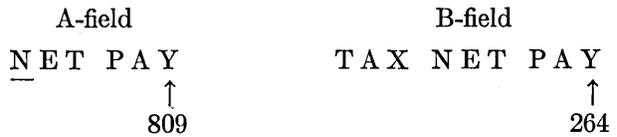


There are several important things to remark about this example. First, the storage positions *from which* the word was moved were not affected. To be technical about it, the word is not really "moved" but "copied and moved." Second, the previous contents of the storage positions *to which* the word is moved are destroyed. As a completely general principle, any time anything is placed in storage locations the previous contents of the locations are erased. It is the programmer's responsibility to be sure that the previous contents are no longer needed. Third, the Move instruction does not change word marks. In fact, word marks are not affected by most instructions; when word marks are to be set or cleared, special instructions are used. Thus, when word marks are set to define fields (words) in storage, the definitions stay in effect until deliberately altered.

For another example, suppose that the instruction and storage contents are as follows:



Here the first word mark is encountered in the A field. The result of this instruction is



The space between NET and PAY is no accident. In showing the example this way, it is assumed that a blank space between NET and PAY is desired. In order to obtain it, a character position must be set aside for the purpose. The character "blank" is thus a character with the same status as any other.

The essential information above the instruction Move Characters to A or B Word Mark is summarized in the box. In order to make this summary box a source of all the reference information about the instruction, it is necessary to list some things that are not explained until later.

Move Characters to A or B Word Mark

FORMAT

Mnemonic	Op Code	A-address	B-address
MCW	<u>M</u>	xxx	xxx

FUNCTION The word in the A-field is moved to the B-field. The data in the A-field is not changed; the previous data in the B-field is lost.

WORD MARKS The first word mark encountered in either field stops the operation. If the first word mark is in the A-field, the character at that position is moved; if the first word mark is in the B-field, that position receives a character from the A-field. Word marks are not disturbed in either field. If the fields are the same length, only one of them need have a word mark.

TIMING $T = 0.0115 (L_I + 1 + 2L_w)$ ms.

We see that to move words within storage (and in fact to do almost any data manipulation) it is necessary to have word marks set. This naturally means that some way must be provided for setting and clearing word marks within a program of instructions. This facility is provided by two instructions called Set Word Mark and Clear Word Mark. These instructions may have one or two addresses, allowing us to deal with one or two word

marks at a time. The operation code (,) is recognized by the computer as meaning Set Word Mark, so that the instruction

, 200258

would mean to set the word mark bits of character positions 200 and 258. ("Setting" the word mark bit means making it a one, and clearing means making it a zero. It is convenient to use phrases

printer is called for by the Write a Line instruction, which has the operation code 2. The line printed consists of the 100 characters in the print area, positions 201 to 300.

The IBM 1403 Printer can be equipped optionally with 132 printing positions, in which case the print area consists of positions 201 to 332.

It is often necessary to clear an area of storage. For instance, suppose that certain data and results are to be moved into the print area and printed. The words moved into the area will ordinarily not occupy every position, and we naturally want to

erase the contents of the unused positions before printing to eliminate the unwanted characters. Furthermore, it is often necessary to clear word marks in an entire area of storage; once again the print storage area is a good example. The Clear Storage instruction makes it possible to clear as many as 100 positions with one instruction, putting the character *blank* in all, and clearing all word marks. The operation code is /, which is technically called a *virgule* but is more commonly referred to as a *slash* or *slant*.

Since one of the functions of this instruction is

Read a Card

FORMAT

Mnemonic	Op Code
R	<u>1</u>

FUNCTION A card feeds and the 80 columns of information are read into storage locations 001 to 080.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)ms + I/O$

Punch a Card

FORMAT

Mnemonic	Op Code
P	<u>4</u>

FUNCTION The data in storage locations 101 through 180 is punched into an IBM card.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)ms + I/O$

Write a Line

FORMAT

Mnemonic	Op Code
W	<u>2</u>

FUNCTION The data in storage locations 201 to 300 (or 201 to 332) is transferred to the printer. The printer takes one automatic space after printing a line.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)ms + I/O$

to clear word marks, it obviously cannot depend on the detection of a word mark to stop its action. Instead, the computer is built to clear all positions from the one addressed down to and including the nearest hundreds position. If the instruction $_ / 799$

is executed, positions 799, 798, 797, ..., 701, 700 are set to blank and word marks cleared. If the instruction $_ / 801$ is executed, positions 801 and 800 would be cleared. The instruction $_ / 400$ would clear position 400 only.

Clear Storage

FORMAT

Mnemonic	Op Code	A-address
CS	$_ /$	xxx

FUNCTION Clearing starts at the A-address and continues leftward through the nearest hundreds position. The cleared area is set to blanks, and word marks are cleared.

WORD MARKS Word marks are not required to stop the operation.

TIMING $T = 0.0115 (L_I + 1 + L_x)$ ms.

To illustrate the use of the instructions described so far, suppose that we are required to read a card and print some of the information on it in a readable format. The card format for the sales card of Section 1.3 was

Columns	1-4	Product number
	5-8	Units sold
	9-11	Salesman
	12-13	District

Suppose that we are required to print this same information in the following positions:

Printing positions	1-4	Product number
	10-13	Units sold
	19-21	Salesman
	27-28	District

This spaces the numbers out so that they can more easily be read.

As we start this operation, we do not know what is in the read and print areas—and even if we did know it probably would be unwanted information and the word marks would likely be in the wrong places. In the course of carrying out a complete program, there are ordinarily several different types of cards to be read and lines to be printed so that word marks must be set properly for each type before trying to use the information from cards or trying to move information to the print area.

For these reasons we must begin the program

by clearing the read and print areas, which can be done with three Clear Storage instructions (we assume that the printer has the additional print positions):

$_ / 080$
 $_ / 299$
 $_ / 332$

As soon as a card has been read, it will be necessary to move the four words from the read area to the print area, which will require word marks to define the length of the fields. As far as we are concerned in this particular example, it would not matter whether the word marks were set before or after reading the card. However, the normal situation would be to read and print numerous cards, all having the same format, in which case we would repeat part of the program each time a card is read and the line printed. When this is done, it is pointless to set the word marks for every card; reading a card does not erase them. Therefore, it is desirable to set the word marks before reading the card.

We recall that the Move Characters to A or B Word Mark instruction is stopped by a word mark in either the A- or the B-field so that it is not necessary to set word marks in both the read and print areas. In this example it really does not matter much which area has them; we shall set the word marks in the read area. Remembering that the word mark of a data word must be in the

high-order position, we need to set word marks in positions 1, 5, 9, and 12. This can be done with the instructions

2 001 005
2 009 012

Now the card can be read, which requires only the operation code 1. With the data from the card in the read area, we can move the words to the print area, which requires the following four instructions:

M 004 204
M 008 213
M 011 221
M 013 228

Recall that a Move instruction addresses the low-order position (but largest address) and moves characters until it encounters a word mark in the high-order position of either field, in this case the A-field.

The data in the print area can now be printed, which requires only the operation code 2. The program is shown in Figure 3.2.

REVIEW QUESTIONS

1. What does the operation code of an instruction do?
2. In the example on page 33 suppose there had been a word mark in position 870. Would the word moved have been the same or different?

3. Suppose there had been a word mark in position 234. What would have been moved?

4. Can word marks be set or cleared with a Move instruction?

5. Suppose that storage contains the following characters:

A-field	B-field
2 5 8 <u>D</u> P 7 F G 5	H K L M 8 9 5 3 V
↑	↑
604	709

What will the contents of the storage positions be if we execute the three instructions

2 709
M 604 709
□ 709

Starting with the original contents again, what would result from

M 602 704

6. Can you suggest why the computer was designed so that the Clear Storage instruction clears to blanks rather than zeros?

3.3 Storage of Instructions

We have so far spoken of instructions in terms of what they cause the machine to do and have not said anything about how the machine deals with the instructions themselves.

IBM 1401 PROGRAM CHART							FORM X24-6437-0 PRINTED IN U.S.A.		
Program: _____									
Programmer: _____							Date: _____		
Step No.	Inst. Address	Instruction				Remarks	Effective No. of Characters		
		P	A/I	B	d		Inst.	Data	Total
		/	080						
		/	299						
		/	332						
		,	001	005					
		,	009	002					
		/							
		M	004	204					
		M	008	213					
		M	011	221					
		M	013	228					
		2							

Figure 3.2. Program segment to clear storage, set word marks, read a card, and print some of the information from the card.

The first and most important thing to realize is that the program of instructions must be prepared *before* the processing is done and that the program must be *in storage* before it is executed. We write the program, punch it on cards in a suitable format, load the instructions into storage, and the instructions then control the machine without any further action on our part.

This means, among other things, that when we write the program we must anticipate everything that the machine will have to do. We must know, for instance, the maximum sizes of the fields that the computer will process, but we cannot know the actual numbers that will be dealt with. The instructions must be set up to handle *any* data of the general type that it is designed to handle. If something comes up that we did not anticipate, the program will still do what the instructions say to do, even though the results may be meaningless. The fundamental consideration is that by the time the instructions are executed by the machine, *we* are no longer in the picture.

Another consequence of the storage of instructions is that they must be capable of being stored in the same storage that is used for data and they must be set up so that the machine can determine where one ends and another begins. Since it is frequently necessary to repeat the execution of groups of instructions or to skip around in the program, we must have some way to identify an instruction by its location in storage.

This brings us to a discussion of how instructions are stored within the computer, which is one of the most important topics in the entire study of programming. The crucial concept is that instructions are brought to the control unit for execution from the internal storage of the computer, where they are stored in the same way data is stored. We may therefore talk about where instructions are stored in much the same way as we talk about where data is stored.

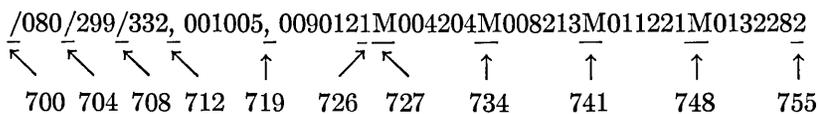
In the IBM 1401 instructions are executed from consecutively higher numbered storage locations, unless special action is taken to break the consecutive sequence. The operation code of every in-

struction *must* have a word mark. Every character of each instruction is stored in a character position; an instruction is identified by the address of the operation code. Note that the operation code is the *high-order* character of the instruction so that the addressing of instruction words is opposite that of data words. Furthermore, instruction words are picked up from storage from left to right, whereas data words are picked up from right to left.

For an application of these ideas, consider the program that was developed in the preceding section. This could be stored in any location that does not conflict with the storage of data; it should be obvious that since the program is stored just as data is, the program storage must not overlap the data storage. A storage location can store either one character of an instruction or one character of data, but not both at the same time, obviously. In this example the only locations used for data are the read and print areas; the program could in theory be placed anywhere else. We avoid the punch area, however, on general principles: in most cases it will be needed for storing information to be punched on cards, and we prefer not to get in the habit of putting instructions where they could get in the way in some problems.

Let us make the arbitrary choice of storage location 700 for the first character of the first instruction of this illustrative program. The complete program in storage is viewed the same way we view data. The underscoring represents word marks, as with data. The characters with word marks are, of course, the operation codes. Simply by counting character positions from the first location of the program, we can determine the address of each character of the program. The most important location for each instruction is the one that contains the operation code, since it is by the address of the operation code that we refer to an instruction.

It would obviously be inconvenient to show instructions strung out along a line this way, which was done to emphasize the similarity of storage of data and instructions. The normal way to write instructions is on a 1401 Program Chart, as shown in Figure 3.2. On this form the step number is



The characters of instructions in the program shown in Figure 3.2, as they would appear in storage.

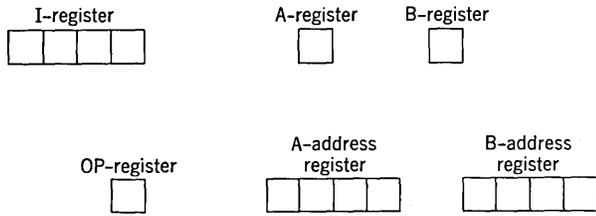


Figure 3.3. 1401 Processing Unit registers.

used at the discretion of the programmer for his convenience. It may be used to identify the instruction when it is punched on a card; it does not enter the computer or have anything to do with the computer's operation. The instruction address is the address of the storage location in which the operation code is stored. OP stands for the operation code. A/I is the address of the A data field, or, as we shall see a little later, the address of the next instruction. B is the address of the B data field—if there is one, of course; d is the d-modifier, which we shall also consider later. The Remarks space may be used to explain what the instruction does for ease of understanding by other programmers or as a reminder to the original programmer. (It is surprising how unfamiliar one's own work can seem after six months.) The "effective number of characters" column is used to determine how much computer time will be used by the instruction; we shall not be greatly concerned with this problem.

It is important to be clear on how much of this gets into the computer: *only* the instruction itself. The instruction address is not part of the instruction; it merely tells *us* where the instruction is located in storage (or, rather, *will* be located after the program is put into storage). The other parts—the step number, the remarks, and the effective number of characters—are strictly for the convenience of the programmer.

As we have seen, an instruction for the 1401 can vary in length, whereas in most computers the length is fixed. It is necessary to fill in only as many boxes on the form as are used on each instruction.

REVIEW QUESTIONS

1. Does the fact that instructions are stored in a manner very similar to the way data is stored suggest that it might be possible to do arithmetic on instructions?

2. For both data and instructions, the word mark is placed in the high-order character. Are data and instructions both addressed in the same way also?

3. Can you tell, without knowing anything about the program organization, whether a given character belongs with data or instructions?

4. What is the use of the instruction address column on the coding sheet?

3.4 Arithmetic and Control Registers

The computer carries out its work of interpreting instructions and processing data by use of several *registers*, a register being an electronic device that can hold one or more characters. Some registers are involved in the transmission of information between internal storage and other parts of the machine. Some are used to hold the parts of an instruction while it is being executed. Others are used to hold the data or results of arithmetic operations.

There are six registers in the part of the 1401 that interprets instructions and operates on information in the internal storage of the machine, as shown in Figure 3.3. (A number of other registers are involved in transferring information between internal storage and input or output devices, but we shall not be concerned with them.)

The most heavily used register is the *B-register*, which holds one character. Every character leaving core storage enters the B-register and is then directed elsewhere, depending on what is being done at the moment. If the character is the first in an instruction, which is the operation code, it is sent to the *OP-register*, where the machine inspects the character and determines what is to be done by this instruction.

If the character from storage is part of the A/I-address, it is sent to the proper position of the *A-address register*, which is a three-character register that will later determine (in most cases) the address of the next data character to be obtained from storage. If the character entering the B-register is a part of the B-address, it is sent to the proper position of the *B-address register*, also three characters, where it will later determine (in most cases) the next location to which to send a character in storage.

The A-address and B-address registers are actually three-character registers, corresponding to the three characters in a 1401 storage address, but for convenience they are displayed on the console

of the machine in four-character form. For this reason they are shown as four characters in the diagram of Figure 3.3.

The d-modifier is not stored in a separate register.

We see that the A- and B-address registers are used primarily to keep track of the addresses of data characters. The *I-address register* performs the same function for instructions. This clearly is necessary; since instructions are stored as data is, the machine must have some way of keeping track of where the next instruction character is to come from.

The operation of the registers may be explained more fully in terms of an example. Suppose that the instruction to be executed is

$$\begin{array}{c} \underline{M} \ 4 \ 1 \ 0 \ 7 \ 8 \ 9 \\ \uparrow \\ 350 \end{array}$$

In order to execute the instruction, the I-address register must contain 350. The 350 would normally be there as the result of the execution of the preceding instruction; that is, the last character of the preceding instruction was located in position 349, and we said that the register always contains the address of the *next* instruction character to be obtained from storage.

The machine operates in two *phases*: an *instruction phase* and an *execute phase*, or *I-phase* and *E-phase*. The I-phase is used to obtain and interpret the instruction, and the E-phase is used to carry out the instruction. When the I-phase begins, the machine uses the contents of the I-register to determine where in storage to obtain the first character of the instruction, which is always the operation code. When this character is obtained from core storage, it moves through the B-register and into the OP-register. The machine "looks at" the operation code in the OP-register, with suitable electronic circuitry, and determines what the function of this instruction will be. This also tells the machine something about the function of the remaining characters of the instruction as they are obtained.

As soon as the first character of the instruction has been obtained, the contents of the I-register are increased by one, giving the address of the next character of the instruction. This is then obtained; it goes through the B-register to the A-address register. The I-register is again increased by one, the next character is obtained, placed in the A-address register, and so on. In our example of a

Move instruction with two addresses, this process would continue until both addresses had been obtained and placed in the A- and B-address registers. At this point the I-register would contain 357, the address of the next instruction character from storage. This character would also be obtained from storage and placed in the B-register, but the machine would detect a word mark, since this character would be the operation code of the next instruction, whatever it is. The word mark would signal the machine that this instruction is complete and would thus end the I-phase.

No data has been moved yet! This much simply gets the instruction from storage and prepares for the *execution* of the instruction, which may now begin. The starting addresses of the two fields are in place in the A-address register and the B-address register, and circuits in the control section of the machine have been set up to carry out the Move function as a result of interpreting the M in the OP-register as meaning "move."

The first step of the E-phase obtains the first character of the A-field, the address of which is given by the contents of the A-address register. This character is brought from storage, placed in the B-register, checked to see whether it has a word mark, moved to the A-register, and placed back in storage at the location specified by the contents of the B-address register. As the character is stored, the machine is able to determine whether the position at which it is being stored has a word mark. This completes the movement of one character. The contents of the A-address register and the B-address register are both decreased by one, to prepare for dealing with the next character. If a word mark was detected in either storage position, the instruction execution is completed and we go back into the I-cycle to obtain and interpret the next instruction; if not, the next character is moved. This process of getting one character, moving it to another location, and checking both places for word marks to determine when the movement is finished is repeated until a word mark is finally detected.

For the purposes of things we sometimes want to do next, it is important to realize the status of the three address registers when the instruction is finished. The I-address register contains the address of the first character (the operation code) of the next instruction in storage. We have seen that this character must have a word mark so that the control circuits may recognize the end of the current instruction. In most cases the next in-

struction will be the next one in sequence in storage, but we shall see several important exceptions. The A-address register contains the address of the next character after the last one transferred. Since the data characters are picked up from storage from right to left, this address will be one less than the address of the last character obtained. Stated another way, it is the address of the next higher order character after the last one moved. This will often be the low-order character of another data word—a fact that can sometimes be useful. The B-address register, similarly, contains the address of the next character position after the last one into which a character was moved. This will also often be the low-order character of another word.

It is frequently possible to take advantage of the contents of the registers after the completion of an instruction, using a technique called *chaining*. Whenever the B-address register already contains the desired address, it is permissible to omit the B address of the next instruction; if *both* address registers already contain the desired addresses, both addresses may be omitted. This saves storage space, obviously, and also saves the time that would have been spent in obtaining the longer instruction from storage. This is a unique feature of the IBM 1401 system.

For an example of how chaining can be used, suppose storage contains the following:

2 3 4 5	9 8 7 6 5	A B C D E F G H I J K
↑	↑	↑ ↑
880	619	739 743

Suppose that it is desired to move the field in 877–880 to 740–743 and to move the field in 615–619 to 735–739. Note that the two fields are to be placed in consecutive locations. If we first execute the instruction

M 880 743

we will move one field as specified, leaving in storage

2 3 4 5	9 8 7 6 5	A B C D E F G 2 3 4 5
↑	↑	↑ ↑
880	619	739 743

Now what are the contents of the A- and B-address registers? The A-address register contains 876, the address of the character to the left of the last one picked up from storage. This fact is of no value to us, since that is not where the next field is to come from. The contents of the B-ad-

dress register *are* useful, however: 739 is just the address we would have to write on the next Move instruction. We may therefore omit it and write

M 619

The control section will recognize the word mark on the next instruction (whatever it is) as terminating this instruction without having picked up a new B-address. Therefore, the contents of the B-address register will not be disturbed and the preceding contents will be used. The effect is the same as if the instruction had been

M 619 739

Either way, the storage contents will now be

2 3 4 5	9 8 7 6 5	A B 9 8 7 6 5 2 3 4 5
↑	↑	↑ ↑
880	619	739 743

The technique used here, in which an A-address is written but no B-address, is called *partial chaining*. It applies only to the Move and Load instructions, since other instructions are handled differently in the internal registers of the machine.

For another example, suppose that storage contents were:

L M N O P Q R S T U V W X Y Z
↑ ↑
395 400
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8
↑ ↑
495 500

The problem is to move 390–395 to 490–495, and 396–400 to 496–500. Since the fields are consecutive in both places, we should be willing to do it all in one instruction—but this cannot be done because the first word mark will stop the movement. However, if we make the first instruction

M 400 500

then the A- and B-address registers will be properly set up to move the second field, since they will contain 395 and 495, respectively. The second instruction can consist of just the operation code M.

Chaining thus saves six instruction characters and a certain amount of computer time. Properly used, it can be quite valuable. (But never try to do things with chaining that cannot be done! It is not possible to use chaining unless the following

field is immediately to the left of the preceding one. Furthermore, it is not possible to omit the A-address and write a B-address; the machine will always put the first address it finds into the A-address register and has no way of "knowing" that you meant it to go into the B-address register. Therefore, if the A-address register is properly set up but the B-address register is not, chaining is not applicable.)

REVIEW QUESTIONS

1. What is a register?
2. Which of the registers is involved in every transfer of information out of storage?
3. What is the difference between an I-phase and an E-phase?
4. What are the contents of the A-address register and the B-address register after any movement of data?
5. How does the control section "know" that it has reached the end of an instruction, bearing in mind that an instruction can be one to eight characters in length?
6. What is chaining?

3.5 Addition and Subtraction

The basic idea of addition is that the number in the A-field is added to the number in the B-field and the sum replaces the B-field. The B-field must

have a word mark because it is this word mark that stops the instruction execution. The A-field is required to have a word mark only if it is shorter than the B-field; in this case the A-field is added only until its word mark is reached but all carries in the B-field are completed. We may illustrate the addition operation with the following storage contents:

```

2 8 4 7 3 2 5           5 7 3 9 9 4 9
                ↑                 ↑
                608                473
    
```

The instruction

A 608 473

would give the resulting B-field:

5 7 4 0 2 7 4

The word mark in 606 signals the end of the A-field but all carries in the B-field are completed.

If the instruction had been

A 608 470

with the original storage contents, the result would have been

5 7 6 4 9 4 9

The word mark in 469 stops the operation, without a word mark having been detected in the A-field.

Add

FORMAT

Mnemonic	Op Code	A-address	B-address
A	<u>A</u>	xxx	xxx

FUNCTION The data in the A-field is added algebraically to the data in the B-field. The result is stored in the B-field.

WORD MARKS The B-field must have a defining word mark because it is this word mark that stops the operation.

The A-field must have a word mark only if it is shorter than the B-field. In this case the transmission of data from A to B stops when the A-field word mark is sensed. Carries within the B-field are completed.

If the A-field is longer than the B-field, the high-order positions of the A-field that exceed the limits imposed by the B-field word mark are not processed.

If the A- and the B-fields have like signs, the result has the sign of the B-field. If the signs are different, the result has the sign of the larger.

If the fields to be added contain zone bits in other than the high-order position of the B-field and the sign positions of both fields, only the digits are used in a true-add operation. B-field zone bits are removed except for the units and high-order positions in a true-add operation. If the A- and B-fields have unlike signs, a complement add takes place, and zone bits are removed from all but the units position of the B-field.

If an overflow occurs during a true-add operation, a special overflow indicator is set, and the overflow indications are stored over the high-order digit of the B-field:

Condition	Result
First overflow	A-bit
Second overflow	B-bit
Third overflow	A- and B-bits
Fourth overflow	No A- or B-bits

For subsequent overflows repeat conditions 1 to 4.

The Branch If Indicator On (B xxx Z) instruction tests and turns off the overflow indicator and branches to a special instruction or group of instructions if this condition occurs. There is only one overflow indicator in the system. It is turned off by a Branch If Indicator On instruction.

TIMING 1. If the operation does not require a recomplement cycle,

$$T = 0.0115 (L_I + 3 + L_A + L_B)\text{ms.}$$

2. If a recomplement cycle is taken,

$$T = 0.0115 (L_I + 3 + L_A + 4L_B)\text{ms.}$$

Subtraction, as might be expected, is very similar to addition. The A-word is subtracted from the B-word; the difference replaces the B-word in storage. The word-mark requirements are the same as in addition: if the fields are the same length, the A may have a word mark but need not; if the A field is shorter, both must have word marks. (In

any case the A-field cannot be longer because the B-field word mark stops the operation.)

Subtraction is algebraic, as is addition. The sign of the result depends on the signs of the two fields and on which of them is larger, as shown in the summary box.

Subtract

FORMAT

Mnemonic	Op Code	A-address	B-address
S	<u>S</u>	xxx	xxx

FUNCTION A-field is subtracted algebraically from the B-field. The result is stored in the B-field.

A-field sign	+	+	-	-
B-field sign	+	-	+	-
Sign of result	+ if B-field value greater - if A-field value greater	-	+	+ if A-field value greater - if B-field value greater

WORD MARKS A word mark is required to define the B-field. An A-field requires a word mark only if it is shorter than the B-field. In this case the A-field word mark stops transmission of data from A to B.

TIMING 1. Subtract—no recomplement:

$$T = 0.0115 (L_I + 3 + L_A + L_B)\text{ms.}$$

2. Subtract—recomplement cycle necessary:

$$T = 0.0115 (L_I + 3 + L_A + 4L_B)\text{ms.}$$

REVIEW QUESTIONS

1. On addition and subtraction, when must the A-field have a word mark?
2. If the A-field is shorter than the B-field, why cannot the execution of an Add instruction stop when the end of the A-field is reached?
3. What is the sign of the result when a large negative number is subtracted from a small negative number?

EXERCISES

*1. Given the following storage contents,

1 2 3 4 <u>5</u> 6 7	9 8 7 6 5 4 3
↑	↑
800	200

show the result of executing:

M 799 200
M 796 196

2. Given the following storage contents,

6 4 3 7 8	1 2 <u>6</u> 4 5
↑	↑
339	881

show the result of executing A 339 881.

3. Given the storage contents,

6 4 3 7 8	1 2 <u>6</u> 4 5
↑	↑
339	881

show the result of executing A 339 881.

*4. Given the storage contents,

5 0 <u>2</u> 8	6 2 <u>3</u> 4 8
↑	↑
497	508

show the result of executing S 497 508.

5. Given the storage contents,

6 2 1 8 9 6	6 2 8 <u>3</u> 2 4
↑	↑
500	600

show the result of executing S 500 600.

*6. Write a program segment to read a card and then punch another card with the same information.

7. Write a program segment to read a card and print part of the information in it as follows:

Card Columns	Print in Print Positions
2-7	2-7
8-11	15-18
43	22
44-70	30-56

*8. Write a program segment to read a card and print a line as follows:

Print Positions	Print
1-8	Contents of card columns 1-8
20-22	Contents of card columns 10-12
30-35	Sum of contents of card columns 15-18, 19-22, and 23-26

9. A report form and a card form are shown on the next page. Write a program segment to read the card and print the required information as shown on the report.

Note that leading zeros have been deleted in printing: where the card might have 09285, the report shows 9285. This *zero suppression* is obtained by using the *Move Characters and Suppress Zeros* instruction, which has the operation code Z. The A-field must have a word mark; word marks in the B-field have no effect and in fact are erased.

4. SYMBOLIC PROGRAMMING

The simple examples of computer instructions that we have seen so far have used actual machine addresses, which is the way the machine must have them. However, very few programs are actually written this way. Writing programs with actual (also called *absolute*) addresses leads to problems in assigning data to storage locations, makes it difficult to write cross references within a program, leads to difficulties when several people must work on the same job, and produces programs that are very difficult to correct and to modify.

For these reasons almost all programming is done with a *symbolic programming system*. For the 1401 system there are three rather similar symbolic programming systems available, called SPS-1, SPS-2, and Autocoder. In this section we discuss the features of the use of SPS-1; all of this material is also applicable to SPS-2 and to Autocoder, since these systems are extensions of SPS-1.

After establishing the fundamental ideas of symbolic programming in this section, almost all later examples are written in the SPS language. This allows the reader ample time to become thoroughly familiar with symbolic programming, bearing in mind that almost no absolute programming is done in applications.

4.1 Fundamentals of Symbolic Programming

The basic idea of symbolic programming is that *symbols* are written in place of actual

machine addresses. After the entire program has been written in the symbolic language, the symbols are translated into absolute addresses by a *processor*. The processor is itself a large program, which can be run on the same machine as the eventual absolute language program. The program, as initially written in symbolic language, is called a *source program*; the processor program translates the source program into an *object program*. The object program may then be run to produce problem results.

It is worth emphasizing before proceeding further that (1) the processor is itself a program, not a machine, and (2) the processor *only* translates the source program into an object program—it does not cause the object program to be executed.

We may begin to get a clearer idea of how symbolic programming is used by considering an example. Figure 4.1 is a program written on a Symbolic Programming System coding sheet. The purpose of this very simple illustrative program is to read four cards, each of which contains an amount in dollars and cents in columns 1 to 10. The program is to form the sum of these four amounts, round the sum to the nearest dollar, and print the total in dollars on the printer in print positions 1 to 9. This, of course, is vastly simpler than anything we would normally do with a computer, but it will serve to illustrate the symbolic programming principles that are our concern at the moment.

A glance at Figure 4.1 shows that all addresses are written as symbols, with the exception of a few at the beginning. The symbols used in the program happen to have

either five or six characters. In general, a symbol may have one to six letters and digits; the first character must be a letter. The invention of symbols is completely under the control of the programmer. It is often convenient to choose symbols that are descriptive of the information referenced by them, such as TOTAL to stand for the address of the field in storage where a total is stored. On the other hand, symbols are not *required* to have any meaning, and none is attached to them by the processor.

We see that it is possible to use absolute addresses where convenient. The processor is easily able to distinguish between symbolic and absolute addresses by the fact that the first character of a symbol is always a letter, whereas the first character of an absolute address is always a digit. We note that absolute addresses are written in *four*-digit form. This is also true of addresses over 999. With SPS we are not required to figure out the three character form of addresses. If we want to write the address 1234, we write it just that way

rather than as S34. The processor will convert the four-digit addresses to the three-character form required inside the machine. (If an address like S34 were used it would be misinterpreted as a symbol.)

In looking at the program in Figure 4.1, it will be noted that the operation codes are written in a new way. These are *mnemonic operation codes*. "Mnemonic" means "aiding the memory"; these substitute operation codes are used because they are easier to remember than the actual machine operation codes. "CS" is the mnemonic operation code for Clear Storage, which is indeed easier to remember than / and SW is easier to remember than a comma. The mnemonic operation codes for the instructions that have been discussed so far are shown in Figure 4.2.

It is still permissible to use the actual operation codes. If this is done, the code should be written in column 16, whereas mnemonic operation codes are always started in column 14.

IBM		1401		Symbolic Programming System		Coding Sheet		Page No. <u>1</u> of <u>1</u>								
Program _____				Programmed by _____				Date _____								
								Identification <u>76</u> <u>80</u>								
LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				COMMENTS				
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	START	CS	0,0,0,0											CLEAR, READ,
0	2	0		CS	0,2,9,9											AND, PRINT
0	3	0		CS	0,3,3,2											STORAGE AREAS
0	4	0		SW	0,0,0,1											SET WM
0	5	0	REPEAT	R												READ
0	6	0		MCW	READI					TOTAL						CARDS
0	7	0		R												AND
0	8	0		A	READI					TOTAL						FORM
0	9	0		R												TOTAL
1	0	0		A	READI					TOTAL						
1	1	0		R												
1	2	0		A	READI					TOTAL						
1	3	0		A	ROUND					TOTAL						ROUND TO \$
1	4	0		MCW	TOTAL	-0,0,2				PRINTI						MOVE \$ ONLY
1	5	0		W	REPEAT											PRINT & REPEAT
1	6	0	02	ROUND	DCW*			5,0								
1	7	0	1,1	TOTAL	DCW*											
1	8	0		READI	DS	0,0,1,0										
1	9	0	09	PRINTI	DCW	0,2,0,9										
2	0	0		END	START											

Figure 4.1. Example of a Symbolic Programming System (SPS) program. Four cards are read, after which the rounded sum of one field from each card is printed.

We may now investigate the program shown in Figure 4.1 in detail. We see that the first instruction has a label of START. This label becomes the symbolic address of the instruction; that is to say, when this source program is translated by the processor, the symbol START will always be associated with the location in storage to which the processor assigns the operation code of the Clear Storage instruction. Any instructions elsewhere in the program that must refer to this instruction may be written with the symbolic address START instead of an absolute address.

The addresses of the three Clear Storage instructions are absolute. This is done because these addresses could never change; no program modification or correction could ever involve changing the read and print storage areas. The address of the Set Word Mark Instruction is also absolute, on the theory that the field to be read from the card will always start in column 1. We shall discuss later the consequences of this assumption.

The Read a Card instruction presents no new concepts. The Move Characters to A Word Mark moves the data field from its position in the read storage area to the locations in which the total will be accumulated. The following six instructions read the other three cards and add their data fields to the locations in which the total is accumulated. The next instruction adds a 50 to the total. Remembering that the data fields were assumed to represent dollars-and-cents amounts, 50 added to the least significant part of the total is, in effect, \$0.50. This means that if the cents amount is 49 or less, adding 50 will not change the dollar amount. However, if the cents amount is 50 or over, adding 50 to it will increase the dollar amount by 1. This is exactly what we want in order to round the total to the nearest dollar.

The next instruction moves the dollars portion of the total to a section of the print storage area. *Character adjustment* is used on this instruction. When it is processed, 2 will be subtracted from the address which is established as the equivalent of the symbol TOTAL. This approach is necessary because we do not know what the equivalent address will be—since the processor has not yet defined it. If we *did* know the actual address corresponding to TOTAL, we could write an address 2 less than that to get only the dollars portion. The effect of the character adjustment is just what we need: the eventual address will be 2 less than

Instruction	Actual Mnemonic	
Move Characters to A or B Word Mark	M	MCW
Set Word Mark	,	SW
Clear Word Mark	□	CW
Read a Card	1	R
Punch a Card	4	P
Write a Line	2	W
Clear Storage	/	CS
Add	A	A
Subtract	S	S

Figure 4.2. Mnemonic operation codes.

whatever address becomes the equivalent of TOTAL.

The last instruction writes the contents of the print storage area on the printer. We see here a variation of the Write a Line instruction: an address is given in the A-operand field. We recall that the Write instruction always refers to the print area so that no address is required for the data. This is our first example of an instruction address that refers to another instruction, this being the Write and Branch instruction. When the line has been written, the control section of the machine will automatically take the next instruction from the location specified by the address in this Write instruction. This is the reason the first address of an instruction is referred to as the A/I-address: it can refer either to a data address or to an instruction address. The idea here is that after the first group of four cards has been read and totaled we would like to return to the beginning of the program to read another group. This process would be repeated indefinitely as long as cards remained in the hopper. (We shall consider in the next section how a test might be set up to detect the last card of the deck.)

The next four instructions are used to define symbols in the program and in one case to define a constant that is referenced by a symbol. They are not instructions to the computer but to the processor; they will not result in the creation of any instructions to be executed in the object program.

DCW stands for Define Constant with a Word Mark. Taking the first of these, we have an instruction to the processor to set up a constant two characters long, as specified by the number in the *count* field, columns 6 and 7. The constant is

shown, starting in column 24, to be 50. The asterisk in column 17 tells the processor that the constant may be assigned to any convenient locations in storage. As we shall see later, the constant in fact would be assigned to the two locations immediately following the last instruction of the program. The symbol ROUND is associated with the low-order character position of this two-character field. The DCW instruction that defines the symbol TOTAL is slightly different. It is specified as eleven characters, which is the number needed to hold the sum of four 10-digit numbers. However, nothing is written starting in column 24. This, in effect, defines the constant as consisting of 11 blanks. The situation here is that we need to specify the length of this field and to have the symbol TOTAL established as being equivalent to the low-order character of the field, but we do not actually need to enter a constant. Here we are only setting up a storage area with a word mark and defining the meaning of the symbol associated with it.

The next instruction is a DS, for Define Symbol. It establishes 0010 as the absolute equivalent of the symbol READ1 but *without* causing anything to be loaded into storage with the object program. This is necessary here because we are dealing with the read area, which is used during object program loading; it is not permissible to use a DCW to set a word mark in this area. The DS, combined with the Set Word Mark instruction in the object program, accomplishes the same result but does not set the word mark until *after* the object program is loaded. The DCW defining the symbol PRINT1

is acceptable, since the print area is not used during loading.

The last "instruction" is again strictly an instruction only to the processor. The END specifies that the end of the program has been reached and that the processor may complete the production of the object program. We write in the A-operand address portion of the END instruction the address of the first instruction that should be executed when the object program is later executed.

The way this program has been written, the processor would put the first character of the program in storage location 333. All succeeding characters would be stored in sequential locations in this example.

The translation of the source program into an object program, which is also called *assembly*, may be outlined as follows. The source program cards are punched from the coding exactly as shown in Figure 4.1, with one card per line. The processor program must be in storage and will have complete control of the computer during the assembly; the source program is not executed during assembly. The processor reads the source program cards and translates the program into absolute form. The procedure varies somewhat, depending on whether the machine on which the assembly is done has tapes. On a card machine there is an additional card-handling step during the assembly. In either case the result of the assembly by the processor is a deck of cards containing the object program. It is also possible to get a *post listing* or *assembly listing*, which shows both the original source pro-

PG	LIN	CT	LABEL	OP	A OPERAND	B OPERAND	D	LOC	INSTRUCTION	COMMENTS
1	010	4	START	CS	0080			0333	/ 080	CLEAR READ
1	020	4		CS	0299			0337	/ 299	AND PRINT
1	030	4		CS	0332			0341	/ 332	STORAGE AREAS
1	040	4		SW	0001			0345	, 001	SET WM
1	050	1	REPEAT	R				0349	1	READ
1	060	7		M CW	READ1	TOTAL		0350	M 010 411	CARDS
1	070	1		R				0357	1	AND
1	080	7		A	READ1	TOTAL		0358	A 010 411	FORM
1	090	1		R				0365	1	TOTAL
1	100	7		A	READ1	TOTAL		0366	A 010 411	
1	110	1		R				0373	1	
1	120	7		A	READ1	TOTAL		0374	A 010 411	
1	130	7		A	ROUND	TOTAL		0381	A 400 411	ROUND TO \$
1	140	7		M CW	TOTAL -002	PRINT1		0388	M 409 209	MCVE \$ ONLY
1	150	4		W	REPEAT			0395	2 349	PRINT & REPEAT
1	160	2	ROUND	DCW	*		50	0400		
1	170	11	TOTAL	DCW	*			0411		
1	180		READ1	DS	0010			0010		
1	190	9	PRINT1	DCW	0209			0209		
1	200			END	START				/ 333 080	

Figure 4.3. Assembly listing of the program of Figure 4.1.

gram and the final absolute object program produced from it.

The assembly listing for the program in Figure 4.1 is shown in Figure 4.3. Note that the listing gives the instructions and data as originally written in the source program and also the assembled object program input. The count field is seen to contain a value for all lines, including instructions; the instructions are provided by the processor for the programmer's convenience. Notice that the addresses shown for the assembled input are correct for both instructions and constants: high-order for instructions and low-order for constants.

The program has not been executed yet! All that has been accomplished so far is the translation of the symbolic source program into an absolute object program and the production of a deck of cards containing the object program. Now the object program may be loaded into the machine and run. It is only at this point that the cards containing problem data are placed in the hopper and read. In short, it is only now that the program that we have written is in control of the computer system.

Let us now consider what would be required to make a change in this program. Suppose that after the program has been written and assembled, the problem specifications change so that it is necessary to form the sum of the dollars-and-cents amounts on *five* cards and that the fields are in columns 14 to 23 instead of 1 to 10.

To incorporate these changes in the program requires adding a Read a Card and an Add instruction and changing all of the addresses that refer to the read area. If the program had been written in absolute, it would mean inserting the two instructions at some appropriate place, such as just before the rounding, and changing a number of absolute addresses. The insertion of the two instructions would require moving all instructions following the insertion, and changing the addresses would require rewriting all those instructions and repunching the instruction cards. Even in this elementary program, we can see that a small change can result in program changes requiring nearly as much work as the initial programming.

To change the symbolic program we start with the source program deck. Since almost nothing in the source program commits us to specific locations in storage, changes in the source program are much easier. The two instructions can be punched on cards and inserted at the proper place in the source

program. At this point we may note a feature of line numbers that are preprinted on the form: they all end in zero. This means that as many as nine instructions can be inserted between any two original instructions without destroying the sequence of line numbers. For instance, if the Read a Card and the Add instructions were to be inserted between lines 120 and 130, they could be given the line numbers 121 and 122 without in any way disturbing the line number sequence. This is valuable, for by using a page number (at the upper right of the form) and a line number the sequence of the source program cards can be defined as a protection against mistakes in handling the source program deck. While we are on the general subject, we may note also that the program identification can be punched in columns 76 to 80 to provide an identification of the program deck, further reducing the possibility of mixups.

With the program written in symbolic form, the change in the location of the card field is almost completely solved by changing the address of the DS instruction that defines the field. On line 180 it is necessary only to change 0010 to 0023, which will change the absolute equivalent of the symbol when the program is reassembled. However, remember that a word mark was set in the high-order position of this field and that an absolute address was used. If this address is not changed, the field will be incorrectly defined. This could be handled by changing the address of the Set Word Mark instruction to 0014, but a better procedure would have been to write the address in symbolic, with character adjustment, so that no change in field position could create this particular problem.

Now when the program is reassembled, which is a simple matter, all of the addresses in the program that are written as READ1 will be changed. As a matter of fact, we note further that the insertion of the two new instructions changes the storage assignments of the ROUND and TOTAL fields so that the reassembly changes virtually every address in the program. This is of no concern to us, since the processor takes care of the whole matter in a few minutes. The new assembly listing is shown in Figure 4.4.

It may seem a little strange to put so much emphasis on designing programs so that modifications are easy to make. It might be thought that once the program is written it could be forgotten. The actual fact is, that virtually all programs change constantly in use, either because improvements in

the program are possible or because the problem specifications themselves change. It is not unusual for one programmer to be assigned the exclusive responsibility of making program changes. The wise procedures designer and programmer give considerable thought to ease of modification *before* the programming is done. Symbolic programming, properly used, is of great value in providing this simplicity of modification.

REVIEW QUESTIONS

1. Which of the following are allowable SPS symbols? CAT, K, F67YN, 674N, ABCDEDF, H&89, GROSSPAY, NET PAY, NETPAY.
2. Explain the relation between the source program and the object program. When is the object program executed in relation to the assembly?
3. Could SPS be used to write a program with no mnemonic operation codes and no symbolic addresses?
4. When character adjustment is used, do the symbolic address and the character adjustment ever get into the object program *separately*?
5. Absolute addresses are written on the SPS coding form as four digits. Does this mean that they appear as four digits in the object program?

4.2 Further Information on the SPS Language and Processor

DCW and END are only two of the "instructions" to the processor. After considering some of the other *symbolic instructions* or *pseudo instructions*,

we shall consider in a little more detail how the processor translates from a source program to an object program.

DCW automatically puts a word mark in a high-order character position of the constant that is defined with it. The DC pseudo-instruction, which stands for Define Constant (no word mark), performs exactly the same functions as DCW but does not enter a word mark. It is ordinarily used to define the value of a symbol and the length of a field, in a situation in which the word mark is specified on the other field in a two-address instruction.

Both DCW and DC create constants which are punched on cards in the object program deck and are actually loaded into storage when the object program is loaded. This is true even if the constant is all blanks. The DS pseudo-instruction, on the other hand, performs the functions of defining the length of a field, reserving space in storage for this field, and, if desired, associating a symbolic address with the field—but it does not enter a constant into storage with the object program. It can be used when it is necessary to set up a storage location for intermediate or final results when a word mark is not needed in the field. The DS pseudo-instruction can also be used for the *sole* purpose of defining the absolute equivalent of a symbol by not putting anything in the count field. In the example of Section 4.1, for instance, a DS instruction was used to specify to the processor that READ1 was to stand for 0010.

PG	LIN	CT	LABEL	OP	A OPERAND	B OPERAND	D	LOC	INSTRUCTION	COMMENTS
1	010	4	START	CS	0080			0333	/ 080	CLEAR READ
1	020	4		CS	0299			0337	/ 299	AND PRINT
1	030	4		CS	0332			0341	/ 332	STORAGE AREAS
1	040	4		SW	READ1	-009		0345	, 014	SET WM
1	050	1	REPEAT	R				0349	1	READ
1	060	7		MCW	READ1	TOTAL		0350	M 023 419	CARDS
1	070	1		R				0357	1	AND
1	080	7		A	READ1	TOTAL		0358	A 023 419	FORM
1	090	1		R				0365	1	TOTAL
1	100	7		A	READ1	TOTAL		0366	A 023 419	
1	110	1		R				0373	1	
1	120	7		A	READ1	TOTAL		0374	A 023 419	
1	121	1		R				0381	1	
1	122	7		A	READ1	TOTAL		0382	A 023 419	
1	130	7		A	ROUND	TOTAL		0389	A 408 419	ROUND TO \$
1	140	7		MCW	TOTAL	-002	PRINT1	0396	M 417 209	MOVE \$ ONLY
1	150	4		W	REPEAT			0403	2 349	PRINT & REPEAT
1	160	2	ROUND	DCW	*		50	0408		
1	170	11	TOTAL	DCW	*			0419		
1	180	11	READ1	DS	0023			0023		
1	190	9	PRINT1	DCW	0209			0209		
1	200			END	START				/ 333 080	

Figure 4.4. Assembly listing of a slightly modified version of the program of Figure 4.1.

Situations will often arise in which this is useful.

Origin is a pseudo-instruction which has the symbolic operation code ORG. The only other field on such an instruction should be an absolute address in the A-operand portion. The processor will interpret this as an order to place the next character of the program in the location specified by the absolute address. This is most commonly used to indicate where the first instruction of the program should be located. In the absence of such an ORG at the beginning of the program, the first instruction is automatically placed in 333, which is the first location beyond the print area. It is also permissible, however, to have an origin instruction elsewhere than the beginning, or even to have several of them. Several might be useful, for instance, if it were desired to place the constants and the working storage in a group separated from the program.

A clearer understanding of the mechanics of the assembly process will be useful in writing correct SPS programs and avoiding certain types of errors. The operation of this processor program can best be explained in terms of an example. Let us see what the processor program would do in assembling the program of Figure 4.1.

With the program punched on cards having the column assignments shown on the coding sheet, the assembly can begin. The operation of the processor in doing this assembly consists of two rather distinct phases or *passes*. In the first pass the processor does little more than establish the meanings of the symbols and translate the mnemonic operation codes to actual. The processor does this by determining the storage location to be associated with each symbol, as it reads the entire program.

At the beginning of the program the processor assumes that the first character of the program will later be loaded into location 333 unless it finds an origin card that specifies some other starting location. In the program in Figure 4.1, therefore, the label START would be entered into a label table along with its absolute equivalent of 333. In order to keep track of the amount of storage required by the instructions and data in a program, the processor must inspect each instruction or data word to determine its length. The first instruction would require four characters (the operation code and one address). If a label appeared on the next instruction, therefore, it would be given the absolute equivalent of 337. Proceeding in this manner, we see that symbol REPEAT would be entered into the label table with the absolute equivalent 349. The

technique by which the processor keeps track of the location to which each symbol is equivalent involves what is called the *location counter*. This is a field within the processor program which is started at 333, or whatever location is specified by the origin instruction, and is increased—as each card is read—by the number of characters needed to store the object program information created by that card. Any time another origin card is detected the location counter is given the value specified on the origin card without regard for previous contents of the location counter.

In our example the location counter would start at 333 and be increased by the length of each instruction, as all of the instructions are read. Finally, it would reach the first constant at the end of the instructions; now the location counter must be increased by the length of the constant as given in the count field. Furthermore, constants and data are addressed by their low-order characters rather than the high-order by which instructions are addressed. Therefore, the label ROUND is associated with the address 400, not 399. Proceeding similarly, TOTAL would be entered in the label table as equivalent to 411. READ1 is made equivalent to 0010 and PRINT1 is made equivalent to 0209, since the actual addresses are specified in the source program. When the processor detects the END card, it stops reading cards and prepares for the second pass.

Notice that the processor has really not done much with the instructions so far. No symbolic addresses have been changed to absolute; this would clearly be impossible. For instance, the processor could not translate the symbolic address READ1 into an absolute address because at the time it finds this address it has not yet established the absolute equivalent of the symbol. This is the basic idea behind the two-pass operation.

On the second pass the processor uses the information in the label table to assemble absolute instructions as the source program cards are read again. The information in the label field (columns 8 to 13) is not used on the second pass; this information was needed only to define the absolute equivalents of the symbols. This time, as the first card is read, the four-digit address is converted to the three-character form. The assembled instruction is then punched into a card along with information to tell a subsequent loading program where in storage to put the instruction and its word mark. This process is carried out for each instruc-

tion as the cards are read. Whenever a symbolic address is found, the processor consults the label table to find the absolute equivalent in order to assemble the instruction. When an instruction is found that has character adjustment, the amount of the adjustment is added to or subtracted from the absolute equivalent found in the label table.

The constants are recognized by their operation codes as constants rather than as instructions and are assembled properly. In our program the 50, which is referred to by the symbol ROUND and the absolute address 400, would be punched on a card for loading with the object program. The other DCW constants are blanks; these would also be put on cards for loading. For the purpose of our program it would be necessary only that sufficient information be punched on the card for setting a word mark; however, there is no provision in the SPS system for doing anything but literally loading the blanks. On the second pass the END card in the source program would cause the creation of a *transition card* in the object program. This card would be the last of the object deck and therefore would be read after the entire program had been loaded. It later causes the object program to take control of the computer system, starting with the instruction specified by the address in the END instruction.

It should be emphasized once again that the result of the assembly is only the creation of an object deck. The object program is *not* executed and it is not even left in storage ready to be executed. With the assembly complete (and ordinarily with some checking for correctness), the object program can *then* be loaded into storage and executed.

REVIEW QUESTIONS

1. What is the difference between DCW and DC? Does either of them allow a symbol to be defined without loading anything into storage with the object program deck? How can this be done?
2. Why must SPS use two passes in assembling a program?
3. Suppose the same origin were given before the instructions and before the constants. What will happen when the object program is loaded?
4. What would happen if a symbol were used in the address part of the instructions in the program but never appeared in the label column? Would the processor have any way of establishing the absolute equivalent of the symbol?
5. What would happen if a symbol were written in *two* places in the label column? Would the processor

have any way of knowing which one establishes the definition of the symbol?

4.3 Case Study: Payroll

The following case study gives us another opportunity to see how symbolic programming is used and at the same time introduces three new instructions.

The problem is a greatly simplified element of a payroll calculation. We are given an input deck which consists alternately of payroll master cards and labor vouchers. The first card of the deck is a master, the second is a detail for the same man, the third is a master for the next man, the fourth is that man's detail, etc. Master cards have the pay number in columns 1 to 5, the name in columns 10 to 29, and the hourly pay rate in columns 53 to 56. The pay rate is given in dollars per hour to three decimals. A detail card has the pay number in columns 1 to 5 and the hours worked, to hundredths of an hour, in columns 12 to 15. Both cards in practice would contain other information. Our job is to read the cards, compute the gross pay, assuming no overtime, and print the pay number, name, and gross pay (to the nearest penny) on the printer. This is to be done for each man in the deck, without considering how to detect the last card (this problem is considered in the next section). The gross pay is to be printed with a dollar sign and a decimal point and with any leading zeros suppressed.

The source program is shown in Figure 4.5, in which separate pages have been used for instructions and constants. This incidentally is a common way to write a symbolic program; often the constants are entered on a separate page as they are first used in the program.

The program begins in this case with an origin instruction, which is used here primarily to illustrate the technique. It might be used in practice to avoid some other standard routine in the first part of available storage. After that, we clear the read and print storage areas and set word marks, as before. Then we read the first card of the data deck, which is a master card. We move the information on it from the read area to the print area and to a working storage area. This is done with a new instruction called Load Characters to A Word Mark. This instruction is somewhat analogous to the Move instruction but with a significant differ-

ence in the treatment of word marks. It requires that only the A-field have a word mark, and it is this word mark that stops the transmission of characters. Any word marks in the B-field are cleared, and the word mark from the A-field is transferred to the corresponding position in the B-field. This instruction can obviously be used only if the field to which the data is being moved is the same length as the source field; however, this is often the case and, when it is, the instruction removes the necessity of setting a word mark in the B field.

The third LCA instruction, written with character adjustment, moves the hourly pay rate to the multiplier field. The next instruction reads the detail card, obtains the hours worked in the HOURS field, and we are ready to multiply to get the gross pay.

Multiplication in the 1401 is a special feature that permits use of built-in machine hardware. (In the absence of this special feature multiplication can be programmed.) On a Multiply instruction the A-address specifies the units position of the multiplicand; this field must have a word mark. The B-address of a Multiply instruction addresses the units position of a rather special field which initially contains the multiplier and in the end contains the product. The multiplier must be in the *high-order* positions of this special field before the instruction is executed. The field must be one character position longer than the sum of the number of digits in the multiplier and the multiplicand. For instance, in our case we have four digits each; therefore, the field has been established as nine character positions long. (This requirement is based on the way the machine multiplies.)

One of the numbers that are multiplied in this

operation has three places to the right of the decimal point, and the other has two to the right. These decimal points are, of course, not punched on the card; they are *understood*. To interpret the result, we must decide where we understand the decimal point of the product to be. This can be obtained by applying the usual rule: the number of places to the right of the point in the product is equal to the number of places to the right of the point in the multiplier plus the number of places to the right in the multiplicand. This means that the eight-digit product will have five decimal places. We want to round this product to the nearest cent, which requires adding a 5 one position to the right of the pennies amount. This turns out to be two characters to the left of the units position of the field, and we add the 5 to the product field with a character adjustment of -2 .

The gross pay is now available in storage, rounded to the nearest penny. Before printing it, however, we would like to insert a decimal point between the dollars and cents, arrange to print a dollar sign, and delete any zeros in front of the first significant digit. All of this can be done with the Move Characters and Edit instruction. This instruction requires the use of an *edit word* that contains the characters to be inserted in the edited amount, along with (in our case) a character to signal the use of zero suppression. The edit word is first loaded to the print storage area. This edit word is \$bbO.bb, where the b's stand for blanks, as shown in the constants in Figure 4.5. When the Move Characters and Edit instruction is executed, the data from the A-field is inserted into the character position in the B-field occupied by blanks or zeros, and high-order zeros are replaced with blanks.

Load Characters to A Word Mark

FORMAT

Mnemonic	Op Code	A-address	B-address
LCA	<u>L</u>	xxx	xxx

FUNCTION This instruction is commonly used to load data into the printer or punch areas of storage and also to transfer data or instructions from the read-in area to another storage area. The data and word mark from the A-field are transferred to the B-field, and all other word marks in the B-field are cleared.

WORD MARKS The A-field must have a defining word mark because the A-field word mark stops the operation.

TIMING $T = 0.0115 (L_I + 1 + 2L_A)$ ms.

Multiply**FORMAT**

Mnemonic	Op Code	A-address	B-address
M	@	xxx	xxx

FUNCTION The multiplicand (data located in the A-field) is repetitively added to the data in the B-field. The B-field contains the multiplier in the high-order positions and enough additional positions to allow for the development of the product. At the end of the multiply operation the units position of the product is located at the B-address. The multiplier is destroyed in the B-field as the product is developed. Therefore, if the multiplier is needed for subsequent operations, it must be retained in another storage area.

Rule 1. The product is developed in the B-field. The length of the B-field is determined by adding "1" to the sum of the number of digits in the multiplicand and multiplier fields.

Example.

1246	4-digit multiplicand
× 543	3-digit multiplier
———	+ 1
	———

8 positions must be allowed in the B-field.

Rule 2. A word mark must be associated with the high-order positions of both the multiplier and multiplicand fields.

Rule 3. A- and B-bits need not be present in the units positions of the multiplier and multiplicand fields. The absence of zone bits in these positions indicates a positive sign. At the completion of the multiply operation the B-field will have zone bits in the units position of the product only. The multiply operation uses algebraic sign control:

Multiplier sign	+	+	-	-
Multiplicand sign	+	-	+	-
Sign of product	+	-	-	+

Rule 4. Zone bits that appear in the multiplicand field are undisturbed by the multiply operation. Zone bits in the units position of the multiplicand are interpreted for sign control.

WORD MARKS A word mark must be associated with the high-order positions of the multiplier and multiplicand fields.

TIMING The average time required for a multiply operation is

$$T = 0.0115$$

$$(L_I + 3 + 2L_C + 5L_C L_M + 7L_M) \text{ms.}$$

L_C = length of multiplicand field.

L_M = length of multiplier field.

Note. The first addition within the multiply operation inserts zeros in the product field from the storage location specified by the B-address up to the units position of the multiplier.

A few examples will show what can be done with this powerful instruction.

A-field	B-field before	B-field after
<u>0</u> 8828	\$bb0.bb	\$b88.28
<u>0</u> 8828	\$bbb.bb	\$088.28

The zero in the edit word calls for zero suppression and also defines the rightmost character position to which it is to be applied, as this example shows:

<u>0</u> 0067	\$b0b.bb	\$bb0.67
---------------	----------	----------

Zero suppression applies to commas to the left of the first significant digit:

000294368 b, bbb, bb0.bb bbbb2, 943.68

This instruction performs certain other editing operations also, as described in the summary box.

The A-field on a Move Characters and Edit instruction is required not to have more characters than the number of zeros and blanks in the edit word. Since the multiplication process always puts a zero in the high-order character of the product, it is necessary to set a word mark one position to

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS	
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.			
0 1 0			ORG	0600										
0 2 0		BEGIN	CS	0080										CLEAR READ
0 3 0			CS	0299										AND PRINT
0 4 0			CS	0332										STORAGE AREAS
0 5 0			SW	PAYNO	-004			NAME	-019					SET
0 6 0			SW	PAYRTE	-003			HOURS	-003					WM
0 7 0		PRG	R											MASTER
0 8 0			LCA	PAYNO				PRINT1						PAY NUMBER
0 9 0			LCA	NAME				PRINT2						NAME
1 0 0			LCA	PAYRTE				MULT	-005					PAY RATE
1 1 0			R											DETAIL
1 2 0			M	HOURS				MULT						GET GROSS PAY
1 3 0			A	ROUND				MULT	-002					ROUND TO CENTS
1 4 0			LCA	EDIT				PRINT3						
1 5 0			SW	MULT	-007									
1 6 0			MC	MULT	-003			PRINT3						EDIT GROSS
1 7 0			CW	MULT	-007									
1 8 0			W	PRG										PRINT AND REPEAT
1 9 0														
2 0 0														

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS	
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.			
0 1 0			ORG	0800										
0 2 0		PAYNO	DS	0005										
0 3 0		NAME	DS	0029										
0 4 0		PAYRTE	DS	0056										
0 5 0		HOURS	DS	0013										
0 6 0		PRINT1	DS	0205										
0 7 0		PRINT2	DS	0230										
0 8 0		PRINT3	DS	0243										
0 9 0		01 ROUND	DCW*			5								
1 0 0		07 EDIT	DCW*			#	0							
1 1 0		09 MULT	DCW*											
1 2 0			END	BEGIN										
1 3 0														

Figure 4.5. SPS program to compute and print gross pay from hours worked and pay rate read from cards.

the right of the high-order character in order to satisfy this rule.* After the editing has been performed the word mark should be removed so that

*An alternative solution, and perhaps a better one, would be to make the edit word one character longer by adding a blank at the left.

it will not disturb later operations with this field when the next card is read.

With the edited gross pay in the print area, it is now possible to write the line on the printer and branch back to PROG to read another card and start over.

Move Characters and Edit

FORMAT

Mnemonic	Op Code	A-address	B-address
MCE	<u>E</u>	xxx	xxx

FUNCTION The Move Characters and Edit instruction modifies the data in the A-field by the contents of the edit-control word in the B-field and stores the result in the B-field.

Define the *body* of the edit-control word as the part beginning with the rightmost blank or zero and continuing to the left until the A-field word mark is sensed. The remaining portion is called the *status* portion.

The following rules control the editing operation.

Rule 1. All numerical, alphabetic, and special characters can be used in the control word. However, some of these have special meanings:

Control Character	Function
b (blank)	This is replaced with the character from the corresponding position of the A-field.
0 (zero)	This is used for zero suppression and is replaced with a corresponding character from the A-field. Also the right-most 0 in the control word indicates the right-most limit of zero suppression.
. (period)	This is undisturbed in the punctuated data field, in the position where written.
, (comma)	This is undisturbed in the punctuated data field, in the position where written, unless zero suppression takes place, and no significant numerical characters are found to the left of the comma.
CR (credit)	This is undisturbed in the status portion if the data sign is negative. It is deleted if the data sign is positive. Can be used in body of control word without being subject to sign control.
- (minus)	Handled in the same way as CR.
& (ampersand)	This causes a space in the edited field. It can be used in multiples.
* (asterisk)	This can be used in singular or in multiple, usually to indicate class of total.
# (dollar sign)	This is undisturbed in the position where it is written.

Rule 2. A word mark with the high-order position of the B-field controls operation.

Rule 3. When the A-field word mark is sensed, the remaining commas in the control field are set to blanks.

Rule 4. The data field can contain fewer, but must not contain more, positions than the number of blanks and zeros in the body of the control word.

TIMING $T = 0.0115 (L_I + 1 + L_A + L_B + L_Y)$ ms.

The constants are shown preceded by an origin instruction, which once again is used mostly for illustrative purposes.

The definitions of the read and print area fields are all made with DS instructions, since nothing can be accomplished with any of them by loading word marks into storage. Word marks are not needed in the print area, and anyway it is not permissible to load constants into the read area.

The DCW with the label ROUND is used to enter a 5 for rounding; the EDIT DCW puts into storage the edit constant; and the MULT DCW sets up the working storage location for the multiplier and the product. These last three DCW instructions are shown with an asterisk in the address field, to indicate that the processor may assign these constants in sequence as the program is assembled. Notice on the assembly listing in Figure 4.6 that the rounding constant is to be loaded into character position 800; the seven pseudo-instructions between the origin and this DCW had no effect on the location counter since they specified absolute locations for the symbols. The END instruction, as usual, specifies that no more source program cards follow, and the address will cause the object program to begin executing instructions at the address shown.

Note that the comments that were written on the coding sheets have been transferred to the assembly listing. They have no effect on the assembly and

are provided for the convenience of the programmer and for others who may have to read the program. The use of comments is strongly recommended.

REVIEW QUESTIONS

1. What is the difference between the instructions Move Characters to A or B Word Mark and Load Characters to A Word Mark?
2. Describe the operation of the Multiply instruction.
3. What characters in the control word (edit word) are always replaced by characters from the A-field?
4. Discuss the reasons for using a combination of DS and DCW pseudo-instructions in this program. Could DCW be used throughout? Could DS be used throughout?
5. How would the object program be changed if both ORG instructions were omitted? Would the execution of the object program give the same results?

EXERCISES

- *1. Give the absolute equivalent of each symbol in the program of Figure 4.1 (before the corrections).
2. Give the absolute equivalent of each symbol in the program of Figure 4.5.
- *3. "Assemble" the program shown in Figure 4.7 "by hand"; that is, carry out the same analysis of the symbolic program that the SPS processor would do, ending with an absolute program.
4. Assemble the program shown in Figure 4.8 by hand.

PG	LIN	CT	LABEL	OP	A OPERAND	B OPERAND	D	LOC	INSTRUCTION	COMMENTS
1	010			ORG	0600					
1	020	4	BEGIN	CS	0080			0600 / 080		CLEAR READ
1	030	4		CS	0299			0604 / 299		AND PRINT
1	040	4		CS	0332			0608 / 332		STORAGE AREAS
1	050	7		SW	PAYNO -004	NAME -019		0612 , 001 010		SET
1	060	7		SW	PAYRTE-003	HOURS -003		0619 , 053 012		WM
1	070	1	PROG	R				0626 1		MASTER
1	080	7		LCA	PAYNO	PRINT1		0627 L 005 205		PAY NUMBER
1	090	7		LCA	NAME	PRINT2		0634 L 029 230		NAME
1	100	7		LCA	PAYRTE	MULT -005		0641 L 056 811		PAY RATE
1	110	1		R				0643 1		DETAIL
1	120	7		M	HOURS	MULT		0649 @ 015 816		GET GROSS PAY
1	130	7		A	ROUND	MULT -002		0656 A 800 814		ROUND TO CENTS
1	140	7		LCA	EDIT	PRINT3		0663 L 807 243		
1	150	4		SW	MULT -007			0670 , 809		
1	160	7		MCE	MULT -003	PRINT3		0674 E 813 243		EDIT GROSS
1	170	4		CW	MULT -007			0681 # 809		
1	180	4		W	PROC			0685 2 626		PRINT AND REPEAT
2	010			ORG	0800					
2	020		PAYNO	DS	0005			0005		
2	030		NAME	DS	0029			0029		
2	040		PAYRTE	DS	0056			0056		
2	050		HOURS	DS	0015			0015		
2	060		PRINT1	DS	0205			0205		
2	070		PRINT2	DS	0230			0230		
2	080		PRINT3	DS	0243			0243		
2	090	1	ROUND	DCW	*		5	0800		
2	100	7	EDIT	DCW	*		\$ 0.	0807		
2	110	9	MULT	DCW	*			0816		
2	120			END	BEGIN					/ 600 080

Figure 4.6. Assembly listing of the program of Figure 4.5.

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS			
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0,1,0			ORG	0,5,0,0												
0,2,0		A,B,C	C.S	0,0,8,0												
0,3,0			C.S	0,2,9,9												
0,4,0			C.S	0,3,3,2												
0,5,0			S.W	A,1					-0,0,4	A,2						
0,6,0			S.W	A,3					-0,0,4	B,1						
0,7,0		B,C,D	R													
0,8,0			A	A,1						T,Ø,T						
0,9,0			A	A,2						T,Ø,T						
1,0,0			S	A,3						T,Ø,T						
1,1,0			A	H,A,L,F,D						T,Ø,T						
1,2,0			M.C.S	T,Ø,T					-0,0,2	B,1						
1,3,0			W	B,C,D												
1,4,0		A,1	D.S	0,0,0,5												
1,5,0		A,2	D.S	0,0,1,0												
1,6,0		A,3	D.S	0,0,1,5												
1,7,0		B,1	D.S	0,2,1,0												
1,8,0		0,6 T,Ø,T	D.C.W*							0,0,0,0	0,0					
1,9,0		0,1 H,A,L,F,D	D.C.W*						5							
2,0,0			END	A,B,C												

Figure 4.7. SPS program for Exercise 3.

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS			
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0,1,0			ORG	0,7,0,0												
0,2,0		H,E,R,E	C.S	0,0,8,0												
0,3,0			C.S	0,1,8,0												
0,4,0			S.W	D,A,T,A,1					-0,0,4	D,A,T,A,2						
0,5,0		R,E,A,D	R													
0,6,0			L,C,A	D,A,T,A,1						P,U,N,C,H,1						
0,7,0			L,C,A	E,D,I,T						P,U,N,C,H,2						
0,8,0			M,C,E	D,A,T,A,2						P,U,N,C,H,2						
0,9,0			P	R,E,A,D												
1,0,0		D,A,T,A,1	D.S	0,0,0,5												
1,1,0		D,A,T,A,2	D.S	0,0,2,3												
1,2,0		P,U,N,C,H,1	D.S	0,1,0,5												
1,3,0		P,U,N,C,H,2	D.S	0,1,1,5												
1,4,0		1,0 E,D,I,T	D.S.W*						\$		0.					
1,5,0			END	H,E,R,E												
1,6,0																

Figure 4.8. SPS program for Exercise 4.

5. What is wrong with the following reasoning? It is desired to set up a program to handle data cards on which the fields are of variable position. To handle this, the absolute addresses of the field-defining DS instructions are given by additional numbers on the data cards.

6. Extend the program of the Case Study as follows. For each man, there are *three* cards: the first and second are as before, and the third gives the man's deductions. The format of the deductions card is

Cols	1-5	Pay number
	6-8	Social security
	9-12	Withholding tax
	13-16	Savings bonds
	17-20	Union dues

The processing now consists of computing the gross pay and the net pay. For each man, a line should be printed as follows:

Positions	1-5	Pay number
	11-30	Name
	36-41	Gross pay
	47-50	Social security
	56-60	Withholding tax
	66-70	Savings bonds
	76-80	Union dues
	86-91	Net pay

The six dollar amounts should be printed with decimal points but without dollar signs.

5. BRANCHING

5.1 Fundamentals of Branching

In Chapter 4 we saw an example of a branching operation in connection with the last instructions of the two sample programs. These instructions were set up so that after the line had been printed the next instruction executed was *not* the next one in storage but the one specified by the address part of the Write instruction. This is the simplest example of branching, which is the process of breaking out of the one-after-the-other sequence of storage locations from which instructions are normally executed in the 1401 (and in most machines).

The Write and Branch instruction is an example of an *unconditional branch*: the next instruction is to be taken from the location specified by the address of the branch instruction, regardless of any condition in the machine. This can also be done as a separate operation, not combined with input or output, by using the Branch instruction. The actual and mnemonic operation codes are the same, B. The unconditional Branch

instruction has one address, which specifies the location of the next instruction to be executed. This is called the I-address, or instruction address, to emphasize that it refers to an instruction, but it is written in the same position as the A-address.

The more powerful application of the branching idea is in the use of *conditional* branch instructions. With these, the next instruction is taken from the specified address *only if* some condition in the machine is present; otherwise, the next sequential instruction is executed. There are several conditional branch instructions in the 1401. The simplest of them is the *Branch If Indicator On* instruction. Here the d-character is used to specify what condition in the machine is to be tested, as shown in the summary box.

On a Branch If Indicator On instruction the instruction operates as an unconditional Branch if the d-character is a blank. This means, in effect, that if the last instruction of a program is a Branch, with blank storage following, there is no need to put a word mark in the character position immediately following the last instruction. (It is necessary to do so otherwise.)

For an example of the use of a conditional Branch, consider the following simple example. We are required to read a deck of less than a hundred cards, print certain items of the information on them, and print the total of one of the fields when the last card has been read. Suppose that the field assignments are as listed in Table 5.1.

The first two printing fields are to be zero-suppressed, that is, any leading zeros are to be omitted in the printing.

TABLE 5.1

Card field	Printing field
7-13	1-7
4-5	11-12
18-30	16-28
37-40 (field to be summed)	30-35 sum on line below body of report

This is not so different from the examples we have seen before, for there are only two new features. The detection of the last card of the deck can be done with a Branch If Indicator On instruction in which the d-character is A, which designates the last card indicator. If sense switch A is on and the last card in the hopper has been read, the branch is taken. If sense switch A is on and cards remain in the hopper, the next sequential instruction is taken. If sense switch A is off and the last card has been read, the machine halts.

The suppression of leading zeros is a matter of ease of use of reports. In most business reports the meaning of a number such as 0008904 is not changed by printing it as 8904, and the report has a neater appearance with the zeros omitted. This applies only to leading zeros; the number should not be printed as 89 4. This suppression of leading zeros is easily accomplished with the Move Characters and Suppress Zeros instruction, which has the actual operation code Z and the mnemonic MCS. The instruction moves characters from the A-field to the B-field, stopping upon detection of a word mark in the A-field. Word marks in the B-field are not inspected and are in fact erased. Any high-order zeros are then replaced by blanks.

The symbolic program to do this job is shown in Figure 5.1. As usual, we begin by clearing the read and print storage areas and setting word marks in the read area. The Read a Card instruction is given a label so that it is possible to refer to it with a later instruction. The numbers for the first two printing fields are moved with a Move Characters and Suppress Zeros instruction, and the third (which was not to be zero-suppressed) is moved with a Load Characters to a Word Mark instruction. Then the card field which is being summed is added into a counter called TOTAL.

This much of the program sets up the printing line and forms the sum. When the line has been printed, the next instruction asks whether the card just read was the last; note the A in the d-character column of the coding sheet. If this was the last card, we branch to the symbolic location FINAL, where there are instructions to print the final total. If this was not the last card, the next sequential instruction is executed, which is also a Branch, but this time an unconditional one which takes us back to read the next card.

When the last card has been processed, the conditional Branch goes to a Clear Storage instruction to erase the recently printed detail line. The total is moved to the designated print position and the total printed.

The last instruction, called Halt and Branch, is a new one. Nothing was said in the problem specification about what should be done once the total is printed. We therefore assume that the machine should be stopped to wait for another problem to be loaded. The Halt and Branch instruction stops the execution of instructions until the start button on the console is pressed, at which time the next instruction is taken from the location specified by the I-address. In this case the I-address was made the address of the first instruction of the program. This was done because of the possibility that when one deck of cards had been read, printed, and totaled it might be desirable to do the same thing with another deck. If this had not been thought necessary, the Halt instruction could have been written without an address, in which case pressing the start button would have caused the next sequential instruction to be executed. In our case the next "instruction" is not an instruction at all, as it happens, but the total that has just been printed. What might happen when the control circuits try to carry

Branch

FORMAT

Mnemonic	Op Code	I-address
B	<u>B</u>	xxx

FUNCTION The next instruction is unconditionally taken from the storage location specified by the I-address.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)$ ms.

Branch If Indicator On

FORMAT

Mnemonic	Op Code	I-address	d-character
B	<u>B</u>	xxx	x

FUNCTION The d-character specifies the indicator tested. If the indicator is on, the next instruction is taken from the location specified by the I-address. If the indicator is off, the next sequential instruction is taken. The valid d-characters and the indicators they test are as follows:

d-character	Branch On
blank	Unconditional
9	Carriage channel # 9
@	Carriage channel # 12
A	"Last card" switch (sense switch A)
B	Sense switch B *
C	Sense switch C *
D	Sense switch D *
E	Sense switch E *
F	Sense switch F *
G	Sense switch G *
K	End of reel *†
L	Tape transmission error *†
+	Reader error if I/O check stop switch is off †
-	Punch error if I/O check stop switch is off †
P	Printer busy (print storage feature) *
+	Printer error if I/O check stop switch is off †
/	Unequal compare (B ≠ A)
R	Printer carriage busy (print storage feature) *
S	Equal compare (B = A) *
T	Low compare (B < A) *
U	High compare (B > A) *
Z	Overflow †
%	Processing check with process check switch off †

* Special feature.

† Conditions tested are reset by a Branch If Indicator On instruction.

The indicators tested are not turned off by this instruction except as noted by †. When carriage tape-channel 9 or 12 is sensed, the corresponding indicator is turned on. These carriage channel-indicators are turned off when any other carriage tape-channel is sensed. The next Compare instruction turns off the compare indicators.

WORD MARKS Not affected.

TIMING T = 0.0115 (L_I + 1)ms.

Move Characters and Suppress Zeros

FORMAT

Mnemonic	Op Code	A-address	B-address
MCS	<u>Z</u>	xxx	xxx

FUNCTION The data in the A-field is moved to the B-field. After the move high-order zeros are replaced by blanks in the B-field. The sign is removed from the units position of the data field.

WORD MARKS The A-field word mark stops transmission of data. B-field word marks encountered during the Move operation are erased.

TIMING $T = 0.0115 (L_I + 1 + 3L_A)$ ms.

Page No. 1/2 of 2

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				COMMENTS				
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	START	C.S	0080											CLEAR, READ
0	2	0		C.S	0299											AND, PRINT
0	3	0		C.S	0332											AREAS
0	4	0		S.W	R1		-006	R2		-001						SET
0	5	0		S.W	R3		-012	R4		-003						WMS
0	6	0	READ	R												
0	7	0		M.C.S	R1			P1								MOVE DATA
0	8	0		M.C.S	R2			P2								TOTAL, PRINT
0	9	0		L.C.A	R3			P3								AREA
1	0	0		A	R4			TOTAL								ACCUMULATE
1	1	0		W												
1	2	0		B	FINAL											A LAST CARD Q
1	3	0		B	READ											
1	4	0	FINAL	C.S	0299											CLEAR, STORAGE &
1	5	0		M.C.S	TOTAL			P4								WRITE
1	6	0		W												FINAL TOTAL
1	7	0		H	START											
1	8	0														

Page No. 1/2 of 2

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				COMMENTS				
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	06 TOTAL	D.C.W*				000	000							
0	2	0	R1	D.S	0013											
0	3	0	R2	D.S	0005											
0	4	0	R3	D.S	0030											
0	5	0	R4	D.S	0040											
0	6	0	P1	D.S	0207											
0	7	0	P2	D.S	0212											
0	8	0	P3	D.S	0228											
0	9	0	P4	D.S	0235											
1	0	0		END	START											
1	1	0														

Figure 5.1. SPS program illustrating the Branch instruction.

out this number as an instruction depends, of course, on what the number is. At any rate, it is not a very desirable situation. It is probably good practice to put an address on all final halts to avoid the possibility of this kind of confusion. If there is really nothing more to be done at this point, the I-address of the Halt can be the location of the Halt instruction itself, so that if the start button is pressed the machine will simply halt again.

REVIEW QUESTIONS

1. What is a conditional branch instruction?
2. Does the Move Character and Suppress Zeros instruction remove all zeros from the field? Does its action depend on a word mark in the B field?
3. Is the last card switch changed by testing it with a Branch If Indicator On instruction?
4. On a Halt and Branch instruction, when does the branch occur?

5.2 Further Branching Operations

There are a number of other types of branching operations besides those mentioned so far. After mentioning one of these briefly, we shall consider the most important application of the concept: its use in comparison of data fields.

The next instruction to be considered is a rather special one that tests a single character, called Branch If Word Mark and/or Zone. The B-address specifies a character position to be tested. The I-address tells where to find the next instruction if the position satisfies the conditions on the word

mark and/or zone bits specified by the d-character. The tests are described in the summary box.

This instruction, it may be seen, can test for all combinations of word mark and zone bits. This feature, which is frequently applicable, saves a great deal of trouble. We shall find several applications for it in later chapters.

The most useful application of branching is in combination with the Compare instruction, which lets us compare two fields in storage. The contents of the A- and B-fields are compared; if they are not the same, the *unequal indicator* is turned on. A Branch If Indicator On instruction can then be used to test this indicator.

The status of the unequal indicator is not affected by testing it with a Branch If Indicator On instruction. Therefore, it may be tested several times after being set once, if desired.

As an optional special feature, the 1401 can be equipped with the High-Low-Equal compare device, which considerably expands the power of the Compare instruction. With this device installed, the comparison turns on a separate *equal indicator* if the two fields are equal and either the *high* or *low* indicator as well as the unequal indicator if they are not the same. "High" and "low" here refer to a scale in which the characters of the machine are ranked from smallest to largest. In this scale the "smallest" character is a blank, the letters of the alphabet run from A as smallest to Z as largest, and the digits follow the alphabet. The various special characters fit in at the positions shown in Appendix 3.

Halt, Halt and Branch

FORMAT

Mnemonic	Op Code	I-address
H	·	
H	·	xxx

FUNCTION The execution of instructions is stopped and the stop-key light on the console is turned on. Pressing the start key causes the program to start at the next sequential instruction if no I-address is written and to start with the instruction specified by the I-address if one is written.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)$ ms.

Branch If Word Mark and/or Zone

FORMAT

Mnemonic	Op Code	I-address	B-address	d-character
BWZ	<u>V</u>	xxx	xxx	x

FUNCTION The single character at the B-address is examined for a particular bit configuration, as specified by the d-character. If the bit configuration is present as specified, the program branches to the I-address for the next instruction:

d-character	Condition
1	Word mark
2	No zone (No-A, No-B-bit)
B	12-zone (AB-bits)
K	11-zone (B, No-A-bit)
S	Zero-zone (A, No-B-bit)
3	Either a word mark or no zone
C	Either a word mark or 12-zone
L	Either a word mark or 11-zone
T	Either a word mark or zero-zone

WORD MARKS As explained.

TIMING $T = 0.0115 (L_I + 2)$ ms.

Compare

FORMAT

Mnemonic	Op Code	A-address	B-address
C	<u>C</u>	xxx	xxx

FUNCTION The data in the B-field is compared with an equal number of characters in the A-field. The bit configuration of each character in the two fields is compared. The comparison turns on an indicator that can be tested by a subsequent Branch If Indicator On instruction. The indicator is reset by the next compare instruction.

WORD MARKS The first word mark encountered stops the operation. If the A-field is longer than the B-field, extra A-field positions at the left of the B-field word mark are not compared. If the B-field is longer than the A-field, an unequal-compare results.

TIMING $T = 0.0115 (L_I + 1 + L_A + L_B)$ ms.

Note. Both fields must have exactly the same bit configurations to be equal. For example, $\underline{000}^+$ compared with $\underline{000}$ results in an unequal comparison.

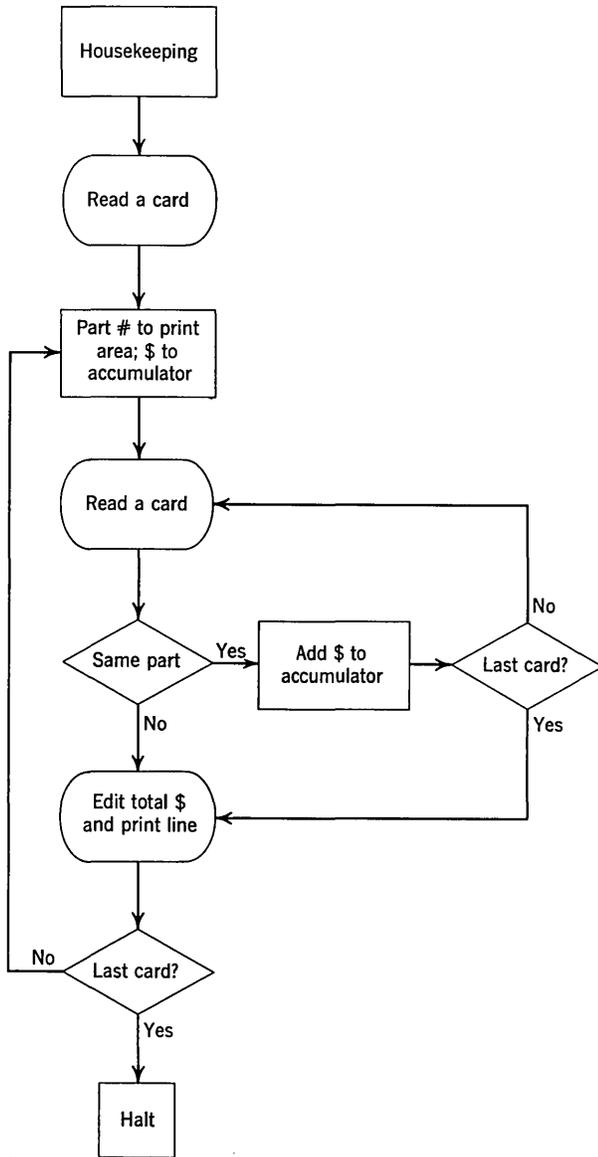


Figure 5.2. Block diagram of the computer operations in producing the sales summarization by product in the example in Section 1.3.

When it is necessary to determine which of two signed numerical fields is algebraically larger, it is best to subtract one from the other and use a Branch If Word Mark and/or Zone instruction to determine the sign of the difference. The Compare instruction cannot be used if the fields could have different signs because it will treat the sign bits as the zone bits of a letter. This is what we want in comparing two alphabetic fields, but not what we want for algebraic comparison.

For a practical illustration of the use of the Compare instruction, we may write the program to perform the first summarization in the sequential file

processing example of Section 1.3, with one simplification. It may be recalled that in the example we read the merged master and transaction deck, obtained the unit price from the master and used it to extend the price of each sale and summarized the total sales for each product. This complete job is considered in Exercise 5. Here we shall simplify the task by assuming that the input deck consists only of the extended sales cards, that is, we are required to summarize the new sales deck. This deck contains a card for each sale, showing the product number, district, salesman, number of units sold, and the total price of the sale. The deck is in product number order. We are required to produce a summary showing the total sales of each product for the month.

A block diagram of the computer processing for this job is shown in Figure 5.2. We begin with what are called "housekeeping" operations, which are the preparatory instructions at the start of the program, to clear storage and set word marks. After reading the first card the part number is moved to the print area and the sales price is moved to a storage field where the total sales for the product are accumulated. Such a field is often called an *accumulator*. Now another card is read and a comparison is used to determine whether it has the same part number. If so, its sales amount is added to the accumulator, and a check is made to determine whether this was the last card; if it was not, another card is read and the process repeated.

When it is found that a card has a different part number from the preceding one, the situation is this: the information from the new card is in the read storage area, the part number of the preceding group is in the print area, and the sum of the sales amounts for the preceding group is in the SUM. It only remains to edit the total and print the line.

The two last card tests are necessary for the following reasons. It is convenient to use the same editing and printing steps for the last group of cards as applied to all others. This dictates a Branch to the same steps—after which the computer, of course, has no way of "knowing" that the steps were reached by a different path and that something different should be done on completing them than is normally done. This is the reason for the second test, after the output box. It is important to know that testing the last card indicator does not turn it off; this is not true of some of the other indicators.

This problem presents an excellent example of a principle that the programmer must never forget: you have to plan for everything. What would hap-

pen if the last card of the deck were the only card for a product number? The comparison would show that the preceding card was the last of a group, the line for that group would be printed—and the last card test would stop the program without ever processing the last card! (This does not happen if the last card is part of a group of cards having the same product number.) The simplest solution is to put a blank card at the end of the deck, which will take care of the special situation without causing any trouble in the normal case.

(Although this is a simple method, it is not particularly desirable for the computer operator. Exercise 8 considers a better solution.)

As a general rule, it is an excellent idea to check

every block diagram to be sure that such special cases as the first card, the last card, and single-card groups are properly handled. And it is also an excellent idea to be sure that test cases are designed to examine these diagrams. It is most disconcerting to discover after four months of operation that a program does not properly handle some special condition.

With the clear picture of the logic of the program that is provided by a careful study of the block diagram, the program shown in Figure 5.3 presents no difficulties. The only instruction not previously illustrated is the Compare, which is used here to determine whether the part number in the read area is the same as the part number in the print area.

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.		
010		START	CS	0080									H.O.U.S.E.K.E.E.P.I.N.G.
020			CS	0299									
030			CS	0332									
040			SW	RPN	-003			R.D.Ø.L.L	-005				
050			R										
060		MØVE	LCA	RPN				P.P.N					P.R.Ø.D.U.C.T.N.U.M.B.E.R.
070			MCW	R.D.Ø.L.L				SUM					S.A.L.E.S.A.M.Ø.U.N.T.
080		READ	R										
090			C	RPN				P.P.N					S.A.M.E.P.R.N.Ø.Q
100			B	P.R.I.N.T									/ P.R.I.N.T.I.F.N.Ø.T
110			A	R.D.Ø.L.L				SUM					A.C.C.U.M.I.F.S.A.M.E.
120			B	P.R.I.N.T									A.L.A.S.T.C.A.R.D.Q
130			B	READ									B.A.C.K.R.E.A.D.I.F.N.Ø.T
140		P.R.I.N.T	LCA	EDIT				P.D.Ø.L.L					
150			MC	SUM				P.D.Ø.L.L					
160			W										
170			B	H.A.L.T									A.L.A.S.T.C.A.R.D.Q
180			B	MØVE									
190		H.A.L.T	H	H.A.L.T									
200			H										

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.		
010		RPN	DS	0004									
020		R.D.Ø.L.L	DS	0019									
030		P.P.N	DS	0204									
040		P.D.Ø.L.L	DS	0217									
050	07	SUM	DCW*										
060	09	EDIT	DCW*					Ø					
070			END	START									
080													

Figure 5.3. SPS program to do the processing defined in the block diagram of Figure 5.2.

After the comparison, a Branch If Indicator On instruction with a d-character of *slash* is used to test the Unequal Compare indicator. If it is on, the program branches to the edit and print instructions. The Halt instruction is written with an I-address that is the same as its label, so that if the start button is pressed after the program is completed the Halt will simply be repeated. This prevents an accidental attempt to restart the program when there is nothing more to do. After the Halt and Branch, there is another Halt. This is provided merely to make sure that there is a word mark in the position following the last instruction to be executed, since every instruction except an unconditional Branch must be succeeded by a word-marked character.

REVIEW QUESTIONS

1. If a second Compare instruction were executed immediately after another Compare, what net effect would the first Compare have on the Unequal Compare indicator?
2. Suppose the instruction BWZ 0600 0800 1 is located in 800. What would it do?
3. In the program of this section, what happens to the information from the first card of a new group while the line for the previous group is being printed?

5.3 Case Study: Parts Explosion and Summary

In a manufacturing operation parts explosion and summary is the process of getting the total parts requirements from a prescribed production schedule of finished goods. In the somewhat simplified example to be considered in this case study we are given a production schedule deck containing one card for each model to be manufactured, each card showing the quantity of this product required. Each product has its own parts requirements, which are given in a parts requirements deck. This deck contains, for each product the company makes, as many parts cards as there are different parts in the model. Each parts card shows the product number, the part number and description, and the quantity of this part required for the model. The basic task is to find the total number of each part required by the entire production schedule.

The following listing shows in semischematic style the information for two hypothetical models from the catalogue of a furniture manufacturer:

Schedule card:	model 5392 table; 40 required
Part card:	5392 table requires 1 top, part 278
Part card:	5392 table requires 4 legs, part 339
Part card:	5392 table requires 2 braces, part 447
Part card:	5392 table requires 12 screws, part 2285
Schedule card:	model 5673 table; 36 required
Part card:	5673 table requires 1 top, part 276
Part card:	5673 table requires 4 legs, part 339
Part card:	5673 table requires 2 braces, part 447
Part card:	5673 table requires 1 front plate, part 663
Part card:	5673 table requires 18 screws, part 2285

We see that the first model creates a need for

40 tops, part 278
160 legs, part 339
80 braces, part 447
480 screws, part 2285

The second model creates the need for

36 tops, part 276
144 legs, part 339
72 braces, part 447
36 front plates, part 663
648 screws, part 2285

The explosion portion of the application produces this kind of information, in our example in the form of one card for each type of part required by each model. The summary portion gets the total requirements for each part. In this sample the summary would show

36 tops, part 276
40 tops, part 278
304 legs, part 339
152 braces, part 447
36 front plates, part 663
1128 screws, part 2285

This is the general idea of the job. Now we may consider in a little more detail the implementation of the application in terms of the card and report forms to be used here.

A flow chart of the processing is shown in Figure

5.4. The first step in the job is to obtain the parts requirements of each model and multiply by the number of models to be built. In order to do this, the production schedule is punched into cards having the following format:

Cols. 1 to 5 Model number
 Cols. 6 to 9 Number of this model to be built

The cards are next sorted into model number order, for collating with the master parts requirements file. This file, which is in model number order, consists of cards having the following format:

Cols. 1 to 5 Model number
 Cols. 6 to 10 Part number
 Cols. 11 to 30 Part description
 Cols. 31 to 33 Number of this part required for one of this model

The deck will contain, for each model, as many cards as there are parts in the model.

Not every model in the catalogue will be built in any one production period, ordinarily, so that when the parts requirements master file is collated with the schedule cards there will be unmatched masters. These could be left in the deck, but it will simplify our block diagramming and coding work here if we assume that they are selected out of the deck and returned to the file. In fact, it might work out in practice that the unmatched masters would make a much larger deck than the matched and that it would be entirely reasonable to remove the unneeded ones to avoid wasting the computer time required to read them.

The deck that now goes to the computer consists of sets of what may be called "packets," each containing a schedule card giving the number of a certain model to be built, followed by parts requirements cards showing the parts required to build one of the model and how many of each. The task of the computer run is to "explode" the parts needed for each model, that is, to multiply the number of each model to be built, by the quantity of each of the various parts used to built it. This will produce another deck of cards, each card giving the quantity of some part needed to build the specified number of units of one model.

After these cards are sorted on part number, a second computer run can easily summarize the number of each part needed by the various models in which it is used.

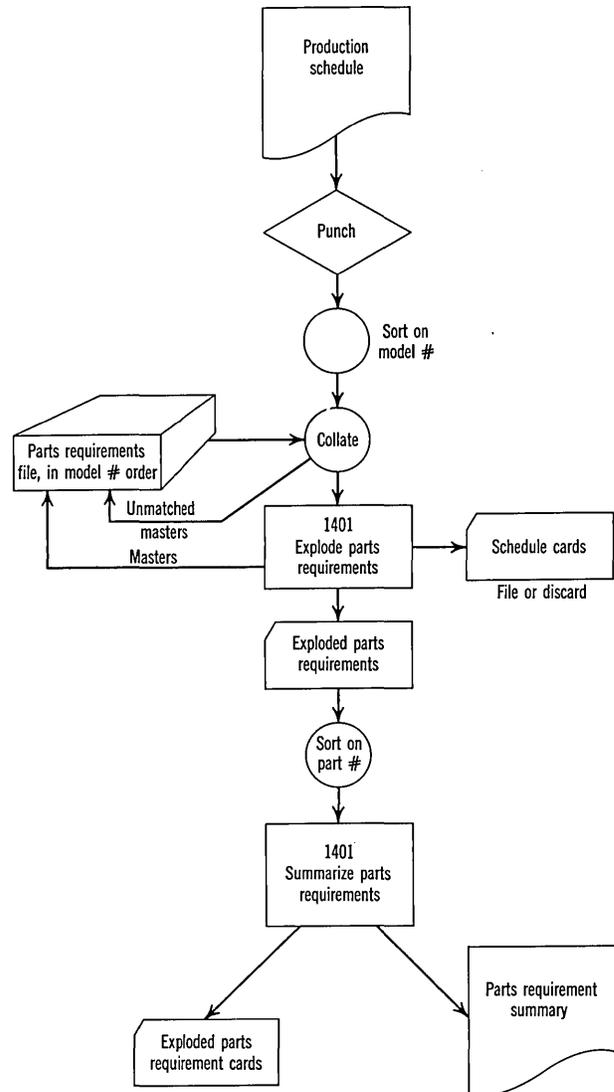


Figure 5.4. Flow chart of a procedure for parts explosion and summary.

These cards have the following format:

Cols. 1 to 5 Model number
 Cols. 6 to 10 Part number
 Cols. 11 to 30 Part description
 Cols. 31 to 35 Quantity of this part required to build the specified number of this model

The format of the parts requirement summary is the following:

Positions 1 to 5 Part number
 Positions 10 to 16 Quantity required

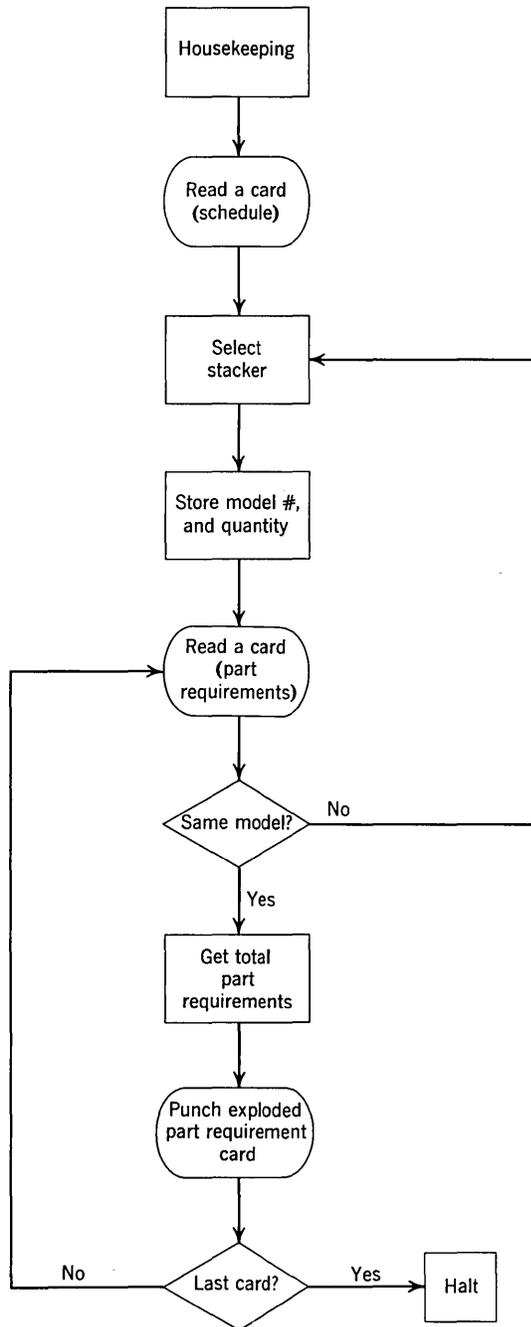


Figure 5.5. Block diagram of the explosion part of the procedure charted in Figure 5.4.

A block diagram of the computer processing to explode the parts requirements of each model in the production schedule is shown in Figure 5.5, and the symbolic program in Figure 5.6. The housekeeping is much the same as before, except that now we clear the punch area instead of the print area.

The setting of word marks is done in absolute as a concession to the necessity of taking advantage of the similarity of card formats in order to avoid setting word marks separately for each of the two different types of cards that must be read.

The first card in the deck should be a schedule card. After reading it, we use stacker selection to put this card in the 1 pocket instead of in the normal read pocket in which the parts requirement cards are stacked. Stacker selection requires the Select Stacker instruction. The actual operation code is K and the mnemonic SS. The instruction needs only a d-character besides the operation code to determine which of the stackers is to be selected.

Next, the model number is placed in the punch area from which it will be punched, and in which it may be used to compare with the model number from succeeding cards to determine when a new model is to be processed. The model quantity is next moved to a working storage location named QTY, for later use in multiplying by the number of each part required.

Now another card is read, which should be a parts requirement card this time. (There obviously must be at least one part to each model.) As later cards are read, however, we shall eventually come to the model requirement card for the next model. Therefore, the first thing to do now is to determine whether the model number on this card is the same as that on the preceding card. If it is the same, we set up the information for the total parts requirement card, multiply the number of models by the quantity of parts required for one model, and punch the card. If the comparison showed a different model number, then the preceding model must have been completely processed, this must be a new schedule card, and we go back to select the stacker and proceed with the processing of the new model.

After punching the card, a last-card test is used to determine whether the end of the deck has been reached. If it has, we halt; if not, we return to read another card.

The summarization run is not hard to write and is left as an exercise.

REVIEW QUESTIONS

1. It is rather essential to the procedure of the block diagram that there not be two schedule cards for the same model. What would happen if there were? Would it make any difference whether this happened at the front of the deck instead of the middle or the end?

2. What would happen if there were a schedule card for which there were no corresponding parts requirements cards? Would it make any difference where in the deck this happened?

3. This procedure contains no error checking of any kind. Can you think of a way to use a total count of all parts in all models to provide some measure of checking of card handling?

EXERCISES

*1. Using the High-Low-Equal compare feature, move whichever of the two fields DATA1 and DATA2 is larger to location BIG. (Do not write a complete program, that is, assume word marks are set and that the symbols

are defined elsewhere.) Draw a block diagram and write a program segment.

2. Move whichever of the three fields DATA1, DATA2, and DATA3 is largest, to location BIG. (Assume that all the words are different.) Draw a block diagram and write a program segment with the same assumptions as in Exercise 1. *Hint.* Place the larger of DATA1 and DATA2 in BIG, then compare this number with DATA3, and replace it with DATA3 if DATA3 is larger.

*3. Read a card. If column 23 contains an 11 zone punch—regardless of what else the column contains—punch another card containing in columns 1 to 40 the information in columns 41 to 80 of this card. If column 23 does not have an 11 zone punch, print in positions

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS			
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	EX.PS.U.M	C.S	0	0	8	0								H.O.U.S.E.K.E.E.P.I.N.G.
0	2	0		C.S	0	1	8	0								
0	3	0		S.W	0	0	0	1			0	0	0	6		
0	4	0		S.W	0	0	1	1			0	0	3	1		
0	5	0		R												R.E.A.D. S.C.H.E.D. C.A.R.D.
0	6	0	S.C.H.C.D	S.S												S.E.L.E.C.T. S.T.A.C.K.E.R.
0	7	0		L.C.A.R.1							P	R	Ø	D	N	Ø
0	8	0		M.C.W.R.2							Q	T	Y			P.R.Ø.D.U.C.T. N.U.M.B.E.R.
0	9	0	P.A.R.T.C.D	R												Q.U.A.N.T.I.T.Y. S.C.H.E.D..
1	0	0		C	R	1					P	R	Ø	D	N	Ø
1	1	0		B	S	C	H	C	D							S.A.M.E. P.R.Ø.D. N.Ø. Q.
1	2	0		L.C.A.R.3							P	A	R	T	N	Ø
1	3	0		L.C.A.R.4							D	E	S	C		N.Ø-N.E.W. S.C.H. C.A.R.D.
1	4	0		M.C.W	Q	T	Y				M	U	L	T		Y.E.S.-P.A.R.T. C.A.R.D.
1	5	0		M	R	5					M	U	L	T	-0	0
1	6	0		M.C.W	M	U	L	T			M	U	L	T		
1	7	0		P												S.E.T.U.P. P.U.N.C.H. A.R.E.A
1	8	0		B	H	A	L	T								G.E.T. T.Ø.T.A.L.
1	9	0		B	P	A	R	T	C	D						Q.U.A.N.T.I.T.Y.
2	0	0	H.A.L.T	H	H	A	L	T								P.U.N.C.H. E.X.P. P.T. R.E.Q
																A.L.A.S.T. C.A.R.D. Q.
																N.Ø-R.E.A.D. N.E.X.T. C.D.

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS			
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	R.1	D.S	0	0	0	5								
0	2	0	R.2	D.S	0	0	0	9								
0	3	0	R.3	D.S	0	0	1	0								
0	4	0	R.4	D.S	0	0	3	0								
0	5	0	R.5	D.S	0	0	3	3								
0	6	0	R.R.Ø.D.N.Ø	D.S	0	1	0	5								
0	7	0	P.A.R.T.N.Ø	D.S	0	1	1	0								
0	8	0	D.E.S.C	D.S	0	1	3	0								
0	9	0	Ø.5 T.Ø.T.Q.T.Y	D.C.W	Ø	1	3	5								
1	0	0	Ø.4 Q.T.Y	D.C.W	*											
1	1	0	Ø.8 M.U.L.T	D.C.W	*											
1	2	0		E.N.D	E.X.P.S.U.M											
1	3	0														

Figure 5.6. Program of the computer operations diagrammed in Figure 5.5.

1 to 40 the information in columns 41 to 80. Write a program segment, in absolute if desired.

4. In the block diagram and program of Section 5.2 add the steps necessary to produce a sales total for the month for all products.

5. Draw a block diagram and write a complete program to do the extension and first summarization of the example in Section 1.3.

The format of the master cards is

Cols. 1 to 4	Product number
Cols. 5 to 9	Unit price
Col. 10	11 zone

The format of the sales cards is

Cols. 1 to 4	Product number
Cols. 5 to 8	Units sold
Cols. 9 to 11	Salesman
Cols. 12 to 13	District

See Exercise 4 of Chapter 2 for a description of the processing and some hints on how to proceed.

*6. Suppose that cost information is included in the parts cards in the case study of Section 5.3 as follows. Columns 34 to 38 contain the total cost of the number of parts required to build one of this model. Modify the block diagram and program as necessary to provide a parts cost total for each model scheduled for production. This should be printed during the explosion run in the following format:

Position 1 to 5	Product number
Position 10 to 13	Number to be built
Position 18 to 25	Total cost of all parts required to build this many of this model. Print with decimal point.

7. After the exploded parts requirements cards have been sorted into part number order they must be summarized by part number. Draw a block diagram and write a complete program.

8. Extend the block diagram of Figure 5.2 to handle the special case of a one-card "group" at the end of the deck without the requirement of a blank card at the end.

6. ADDRESS MODIFICATION AND LOOPS

6.1 Computations on Addresses

We have seen in preceding sections that instructions are stored within a computer in much the same way data is stored. An instruction is made up of the same characters that are available for storing data, the instruction characters are placed in the same storage as data, and in the 1401 instructions are required to have word marks in their high-order positions just as most data words have. As long as an instruction is simply being stored, it is literally indistinguishable from data. It is only when an instruction is to be *executed* that any differences arise. The fact that in the 1401 instructions are brought from storage in a left-to-right fashion, whereas data is accessed from a right-to-left, is really only a matter of design convenience and is not fundamental.

The one thing that actually distinguishes an instruction from data is the *time* at which it is brought from storage, that is, during the instruction phase or during the execution phase. If a word is read out of storage during the instruction phase, it goes to the control registers and is treated as an instruction. If a word is read out of storage during the execution phase, it goes wherever the operation code dictates that it should go to execute the processing prescribed by the instruction. This distinction between instruction and data is the same for all stored-program computers. It does not depend on the fact that the 1401 has a variable word

length, that word marks are involved, or that most instructions have two addresses or on any of the other features of the 1401 that are not typical of all computers.

What significance has all of this to us as programmers? In a nutshell, the answer is that we are able to operate on instructions in storage just as though they were data. If one instruction says to add a constant to the address of another instruction, there is no confusion in the machine in doing so. The first instruction, which calls for the addition, is accessed during the instruction phase and goes to the control registers. The address part of the second instruction, on which arithmetic is being performed, is accessed during the execution phase of the first instruction. Similarly, if a Move instruction is used to transfer an instruction from one place in storage to another, this is perfectly legitimate. It is also permissible to have one instruction change the operation code of another instruction.

The facility for carrying out processing operations on instructions is one of the most important aspects of a stored-program computer. In short, it makes it possible to set up a program to *modify itself*, according to the results of its own data processing operations. This ability, combined with the ability to repeat a section of a program that is provided by the various branching instructions, is by all odds the most important single feature of the stored program concept.

For a first example of the application of

TABLE 6.1

Merchandise Class	Printing Position
1	1-10
2	11-20
3	21-30
4	31-40
5	41-50
6	51-60

this concept, consider the following sales summarization problem. A previous computer run has produced a deck of cards containing (among other things) a sales amount and a code number that gives the class of merchandise represented by the sale. There are six classes of merchandise, represented by the codes 1 through 6. The merchandise code is punched in column 43 and the amount of the sale in dollars and cents is punched in columns 17 to 22. In this highly simplified example we are required only to print in a single line the total sales for each merchandise class. The required printing positions are shown in Table 6.1.

This example presents only one problem: how to determine to which of the six total accumulators the sales amount on each card should be added. The reading of the cards, the last-card test, and the printing of the total line will cause us no difficulty. The choice of the proper accumulator *could* be handled by a series of comparisons and branches, or somewhat more conveniently by the use of an instruction that we have not considered, the Branch If Character Equal instruction. However, by either of these methods, the testing and branching would run to 15 or 20 instructions, which makes us wonder whether there might not be some simpler way to accomplish the same result.

Indeed there is a simpler way. Consider what we would get if we were to multiply the merchandise code by 10 and add the product to 200 (see Table 6.2).

Thus it appears that, if we carry out this simple computation on the merchandise code and use the result as the address of an instruction, then in each case we will have the address of the proper field in the print area. Since the entire computation produces only one line of output, there is no reason not to use the print area itself for the six accumulators. Therefore, once the address of the proper

position in the print storage area has been computed, it can be placed in the address part of an Add instruction and the addition performed in *whichever accumulator is specified by the computed address*.

Since we are concerned in this example with other things, the program in Figure 6.1 is shown without the initial housekeeping operations of clearing storage and setting word marks. After reading a card, we proceed immediately to compute the address of the accumulator to which the sales amount from this card should be added. The address of this accumulator will be developed in a three-position field which has been given the symbolic address WKSTOR, for working storage. We begin by moving the constant 200 into this working storage. Then the merchandise code is added to this field with character adjustment of minus one, which has the effect of adding the code into the tens position of the field. Therefore, the sum in WKSTOR will be 200 plus ten times the merchandise code, which as we saw is the address of the proper accumulator. This address is next moved into the B-operand address part of the Add instruction, which follows immediately. This is done with the MCW instruction in which the B-operand address is COMPAD with character adjustment of plus six. Looking at the labels of this program, we see that COMPAD is the symbolic location of the Add instruction. Remembering that the location of an instruction refers to the location of the operation code, we see that to obtain the address of the rightmost character of the B-operand does require character adjustment of plus six.

The Add instruction which will form the sum is shown with a B-address of 0000. This is to remind us that this address is computed by the program itself. The situation is this. When the object program is loaded into storage, the B-address

TABLE 6.2

Merchandise Code	10 × Code +200
1	210
2	220
3	230
4	240
5	250
6	260

of this instruction is 0000. However, *by the time the instruction is executed* the program itself will have placed the address of one of the six accumulators in this part of the instruction.

With the sales amount added to the proper accumulator, we make a last-card test. If this was the last card, we branch to print the total and halt; if it was not the last card, we branch back to read another card and repeat the entire process.

It is important to realize just what this example shows: that computations on addresses are possible and useful. This particular example, however, is unrealistic for a reason that is important in itself. What would happen if a merchandise code were mispunched and entered the computer as 7 or K? The answer is simply that the program as written would carry out the address computation on the bad code and then add the sales amount to whatever location it computed. The result would be at least wrong—and perhaps disastrous: the program might well be destroyed by the addition.

The point of all this is that one should think twice before putting so much faith in data. In this example we could make a check before the address computation to determine that the code really is a digit between one and six. Different ways to guarantee the correctness of a computed address may be found in other situations.

In any case, the example illustrates well the principle of address computation and perhaps gives a hint of the usefulness of the technique.

REVIEW QUESTIONS

1. How does a computer distinguish between instructions and data?
2. Would the concept of storing instructions like data and the consequent ability to perform arithmetic on instructions be significantly different in a machine in which instructions have three addresses?
3. Why was it necessary to develop the address of the correct accumulator in a working storage area rather than directly in the address part of the Add instruction? *Hint.* Consider what would happen when the second and subsequent cards were read, and what the word mark problems might be.
4. This particular program is completely dependent on the fact that each of the printing fields is exactly 10 columns long. Still using the same basic address computation technique, how could you revise the program if each of the printing fields were 15 columns long?

6.2 Program Switches

For another example of the concept of a program modifying itself, consider the “storage” of decisions, by the use of program switches. It not infrequently happens that a decision made at one point in the program has a bearing at one or more later points. Sometimes it is possible simply to repeat the Branch instruction that made the decision in the first place. In other cases, however, the information on which the decision was originally made is no longer available—or it may happen that

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS			
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	READ	R												
0	2	0		M.C.W	C.200											A.D.D.R.E.S.S. -
0	3	0		A	C.Ø.D.E											C.Ø.M.P.U.T.A.T.I.Ø.N
0	4	0		M.C.W	W.K.S.T.Ø.R											S.T.Ø.R.E. A.D.D.R.E.S.S
0	5	0	C.Ø.M.P.A.D	A	S.A.L.E											C.Ø.M.P.U.T.E.D. B.-A.D.D.R.S
0	6	0		B	P.R.I.N.T											A L A S T C A R D T E S T
0	7	0		B	R.E.A.D											N.Ø
0	8	0	P.R.I.N.T	W												P.R.I.N.T. T.O.T.A.L.S
0	9	0		H	*											
1	0	0		H												
1	1	0	03	C.200	D.C.W	*										
1	2	0	03	W.K.S.T.Ø.R	D.C.W	*										
1	3	0		C.Ø.D.E	D.S	0.043										
1	4	0		S.A.L.E	D.S	0.022										
1	5	0		E.N.D	R.E.A.D											
1	6	0														

Figure 6.1. Program illustrating address computation.

the decision involves a number of instructions, making it wasteful to repeat them when the result of the decision is needed later.

When such situations arise, it is desirable to be able to store the result of the decision. This can be done in many ways. One possibility is to store either a zero or a one in some location, depending on the outcome of the test. Then, when it is later necessary to know what the result of the test was, this storage location can be checked to see whether it contains a zero or a one. The most common technique, however, at least for storing the results of two-way decisions, is to change the operation code of instructions.

This instruction modification is most frequently done by using the unconditional Branch and the No Operation instructions. No Operation is an instruction that causes no action to take place anywhere in the computer. Stated otherwise, there is no execution phase on this instruction. It is provided partly for such situations as we are describing and partly to make it possible to eliminate the effect of unwanted instructions when it is not feasible to reassemble. It has many other valuable uses. The operation code is N and the instruction may have any of the other parts of the instruction; any other parts besides the operation code will, of course, have no effect. The mnemonic operation code is NOP.

To describe the operation of program switches a little more concretely, consider the following situation. A comparison is to be made early in a program. If the comparison shows equal, then at three subsequent points in the program it is necessary to branch to special routines to handle this case. If the comparison shows unequal, then at each of those three points the program should continue in sequence. The technique is to write the three Branch instructions at the points at which the program should transfer out to the special routines, as though the branch would always occur. When the test is made, one of two short routines is executed. If the test shows unequal, then the operation codes of the three Branch instructions are changed to N. If the test shows equal, the three operation codes are set to B. When the three instructions are subsequently executed, they will either cause the Branches or allow the program to continue in sequence, depending on the result of the test.

It actually is necessary to go to the trouble of

setting the operation code of the three switches for *each* outcome of the test. It might be thought that if the instructions were originally written as Branches, then they could simply be left alone if the initial test showed that the branch should be executed and changed to N's if the program should continue in sequence. This would indeed work correctly the first time through the program and possibly for a few later executions. However, as soon as the operation codes are once changed to N, then they need to be reset to B's if the Branch should be executed.

Naturally there are many other programming techniques that can be used to store the result of a decision. One that comes to mind immediately is the possibility of changing the address part of a Branch instruction. The choice of the method to be used in setting up a program switch depends on such factors as the number of different possible outcomes the decision has, how many places the switch must operate, and how much trouble it is to repeat all or part of the original decision. In other computers the choice will also depend on the programming characteristics of the machine.

In all cases, however, the general principle is simply that a decision made at one point in the program is being used to control the subsequent action of the same program on one or more later occasions. This further example of the modification of a program by itself finds fairly frequent application in many programs. We shall see a few examples of the technique in later sections.

REVIEW QUESTIONS

1. Describe how a program switch could be set up to use modification of the address of a Branch instruction.
2. Does the concept of a program switch depend on using the result of a decision at *more than one* subsequent point in the program?

6.3 Program Loops

It must be readily apparent that a program involving no repeated executions of instructions would not be practical. If a program were able to proceed only sequentially through its instructions and on completion had to be replaced by another program, then it is clear that the stored program computers would be of little value. Most

of the time would be spent in loading instructions.

Fortunately, however, there is no such restriction on the organization of programs, and a whole body of technique has been built up around the methods for repeated execution of program segments. This technique is known as *looping*. We have, in fact, already seen a number of elementary examples of loops. The illustrative program in Section 5.1 is a loop in the following sense: after some preliminary housekeeping operations, we read a card and perform certain computations on the data read from it. Then we test the last card indicator to determine whether all of the cards have been read. If not, we return to the instruction for reading a card and repeat the entire program except for the initial housekeeping operations.

We have here almost all of the normal parts of a loop. There is an *initializing* section, which gets the loop started properly and is only executed once. There is a *computation* section which does the actual work of the program—in this case, reading a card and performing the calculations. There is a *testing* section which determines whether the work of the loop is completed. Most loops also contain a *modification* section which changes some of the instructions in the computation section of the loop or changes the data on which the computation section operates. In a certain sense, even the simple program in Section 5.1 has a modification section if we regard the reading of a new data card as a modification of the data being operated on.

This example is a loop that consists of an entire program, which is a rather broad application of concept. We more commonly find loops that are only small segments of a total program. Frequently, one loop has within it one or more additional loops. The sales summarization program of Section 1.3 can be viewed in this manner. The innermost loop is the one that obtains the sales total for each salesman. This loop is “inside” the loop that computes the totals for each district, which, in turn, is “inside” the total program loop that reads the entire deck of sales cards. In each case there is an initialization section that consists of the housekeeping operations for the total program loop and of the special handling of the sales amount on the first card of each group for the other two loops. In each case there is a computation section; this must be applied broadly to the total program loop, since it consists of all operations

contained in the other two loops. In the two summarization loops the computation consists of the summarization and of the processing that is done when it is found that the last card of a group has been read. In each case there is testing, in one to detect the last card (although this test was not written in the program earlier) and in the other two to determine when the first card of a new group has been read.

The loop concept provides the best example of the unique power of a stored program digital computer. It is probably the most important single topic in the study of programming. We shall see immediately below that one of the most powerful types of loops involves the repetitive modification of the instructions within the loop itself, most commonly the addresses.

REVIEW QUESTIONS

1. Name the four parts of a loop and give examples of each.
2. Must the four parts of a loop always be executed in the order in which they are named in the text?
3. If a loop is used only once in a program, that is, never started again with new data, is it logically necessary to initialize?

6.4 Address Modification Loops

In this type of loop we have, as before, the four parts of initialization, computation, testing, and modification, although not necessarily always in that order. The modification now consists of changing one or more addresses within the computation section of the loop. The testing most commonly involves determining whether the computation section has yet been carried out a specified number of times.

For an example of this type of loop consider the following inventory usage application. A deck of cards contains one card for each part in the inventory of a certain manufacturing company. Each card shows the part number and the usage for each of the 12 months of the calendar year. The task is to produce a report with one line for each part, showing the average monthly usage of the part and the number of the month in which maximum usage occurred.

The card format is as follows:

Columns	Field
1-9	Part number
9-13	January usage
14-18	February
19-23	March
24-28	April
29-33	May
34-38	June
39-43	July
44-48	August
49-53	September
54-58	October
59-63	November
64-68	December

The format of the report is as follows:

Printing Positions	Field
1-8	Part number
12-16	Average monthly usage
20-21	Number of month of maximum usage

This program may be thought of as consisting of two parts: dividing the sum of the monthly usages by 12 to get the average and determining which of the months has the heaviest usage. In the final version of the program these two parts are combined in one loop. However, to get a clear picture of the workings of an address modification loop, we first write a program to get the average only and then add the instructions for finding the heaviest usage.

After reading a card the object of the summing loop is to add to an accumulator the usage for each of the 12 months. This, of course, *could* be done with a Move and 11 Adds. However, we shall see that it can be done with fewer than 12 instructions. We begin by setting the accumulator to zeros in order to remove the sum developed there from the previous card. We also set to zero a two-position counter that is used to determine when the last monthly usage has been added to the accumulator. The 12 monthly usages are picked up from the read area by a single Add instruction, the address of which is modified each time through the loop. Since, after reading one card, this address will be incorrect for starting the accumulation of the usages from the next card, we initialize this address by setting it to 13, the address of the first data field.

Each time around the loop another data field is added to the accumulator, the A-address of the Add instruction is increased by 5, and 1 is added to the counter. A comparison is made each time to see whether this counter has reached 12. If it has, then all 12 monthly usages have been added into the total and we are finished; if it has not, then the loop is repeated. When the total usage for the year has been developed, we divide by 12, print the line for this inventory item, make a last-card test and, if cards remain, return to the Read instruction.

A block diagram of this procedure appears in Figure 6.2 and a symbolic program in Figure 6.3. Once again, the program is shown without the preliminary housekeeping operations of clearing storage and setting word marks. In the program a new instruction is used to place zeros in the accu-

Zero and Add

FORMAT

Mnemonic	Op Code	A-address	B-address
ZA	$\begin{array}{c} + \\ 0 \\ - \end{array}$	xxx	xxx

FUNCTION The entire B-field is set to zeros; then the data from the A-field is moved to the B-field with zone bits stripped from all but the units position. If A is shorter than B, zeros are placed in the high-order positions of B.

WORD MARKS The B-field must have a word mark; the A-field must have a word mark only if it is shorter than the B-field.

TIMING $T = 0.0115 (L_I + 1 + L_A + L_B)$ ms.

mulator and count fields. The Zero and Add instruction is just like an Add except that the B-field is cleared to zeros before the addition takes place. This instruction is therefore analogous to a Move or Load instruction, with the significant difference that in a Zero Add the zone bits of all but the low-order character are removed during the transmission. This feature itself is often of value. In our case, the advantage of a Zero Add over a Move is that we can set up a field consisting of only a single zero and clear the *entire* B-field. What the instruction actually does, in our case, is to clear the entire B-field to zero and then add the one-character constant of zero that we specify with the A-address.

The desired initial address of the Add instruction that picks up the monthly usages is transferred with an MCW instruction.

The variable-address Add instruction, which has the symbolic label of ADDINS, is shown with an A-address of 0000; this address is computed by the program and will have some value other than 0000 by the time it is first executed. After the first monthly usage is added into the accumulator, using character adjustment to add into the high-order part, we add a 1 to the counter and a 5 to the variable address of the Add instruction. This last is done with character adjustment to add the 5 into the units position of the A-address. It might appear that there is a word-mark problem here, but it happens that the attempt to propagate carries when the 5 is added to the address will not affect the operation code of the Add instruction. If it were desired to be double safe on this, a word mark could be set in the high-order position of the A-address before the addition of the 5 and then cleared afterwards.

Next the count is compared with 12 and a Branch If Indicator On instruction tests for equality. If the indicator shows that the two are unequal, we branch back to the Add instruction and pick up another monthly usage and continue the loop. If the unequal indicator is off, then the branch does not occur and we proceed to find the average. This could be done with the Divide instruction, an optional feature on the 1401, or by a programmed division routine. Here, however, we have chosen to multiply by $\frac{1}{12}$ rather than divide by 12. This is done simply to save the time that would be required to describe division in the 1401, since we shall have no further occasion to use it.

The constant $\frac{1}{12}$, which is taken as equal to

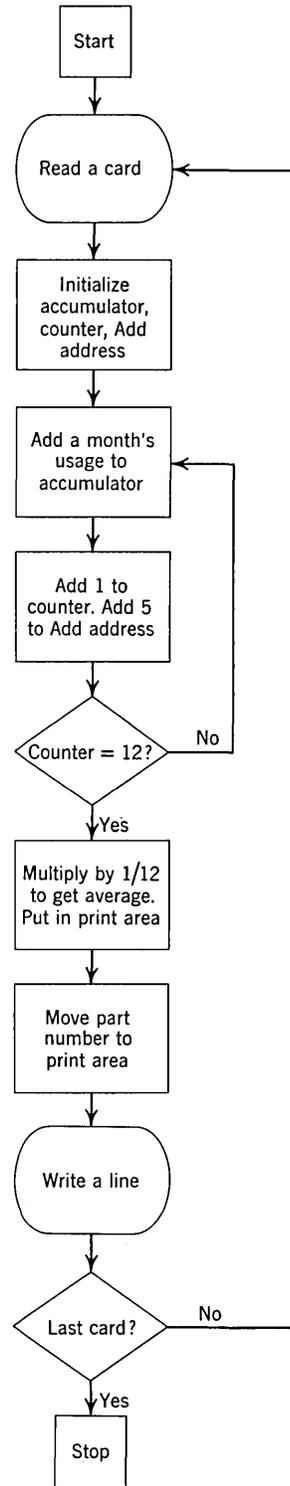


Figure 6.2. Block diagram of a procedure to compute the average of 12 numbers on a card.

.083333, has six places to the right of the decimal point. Therefore, after the multiplication the average can be rounded to the nearest unit by adding a 5 to the fifth digit to the left of the units position. The rounded monthly usage is then moved with zero suppression to the printing position, the part number is also moved with zero suppression, the line is printed, and a last-card test is made.

It is worth emphasizing what this program illustrates. We have here a fairly representative example of an address modification loop. There is an initializing section, where we put zeros in locations that could have left-over data from the preceding execution of the complete loop and where we start an address at its correct initial value. There is a computation section, consisting in this loop of just the one variable-address Add instruc-

tion. The modification section consists of the addition of 1 to the counter and of 5 to the address of the Add instruction. The testing involves determining whether the counter has reached 12 and returning to another execution of the loop if it has not. The instructions that follow the test are not part of this loop.

Note that the complete loop takes eight instructions, including the initialization. Without a loop, the same summation would take 12 instructions: one MCW and 11 Adds. However, the loop version requires the execution of 63 instructions: three for initialization and 12 times around the five instructions in the repeated portion of the loop. Thus we see that a loop saves space at the expense of time. This is a completely general statement.

This example is quite important for what it shows

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				COMMENTS		
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.			
0.1.0		START	R											
0.2.0			ZA	ZERØ				ACCUM						INITIALIZE
0.3.0			ZA	ZERØ				CØUNT						TØTAL & CØUNTER
0.4.0			MCW	INT.ADD				ADD.INS+0.03						INIT. ADDRESS
0.5.0		ADD.INS	A	0.0.0				ACCUM -0.07						VARIABLE ADDRESS
0.6.0			A	ØNE				CØUNT						MØDIFY CØUNT
0.7.0			A	FIVE				ADD.INS+0.03						& VAR. ADDR.
0.8.0			C	CØUNT				TWLVE						FINISHED Ø
0.9.0			B	ADD.INS										NØ
1.0.0			M	CI				ACCUM						YES. MULT. 1/12
1.1.0			A	FIVE				ACCUM -0.05						RØUND
1.2.0			MC'S	ACCUM -0.06				0.21Ø						SET
1.3.0			MC'S	0.0.Ø				0.2.Ø8						UP
1.4.0			W											AND PRINT
1.5.0			B	LAST										A LAST CARD Ø
1.6.0			B	START										NØ
1.7.0		LAST	H	* -0.03										YES
1.8.0			H											
1.9.0														

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				COMMENTS		
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.			
0.1.0		1.4	ACCUM	D.C.W*										1.4 CHAR. FOR MULT
0.2.0		0.2	CØUNT	D.C.W*										
0.3.0		0.1	ZERØ	D.C.W*				0						
0.4.0		0.3	INT.ADD	D.C.W*				0.1.3						
0.5.0		0.1	ØNE	D.C.W*				1						
0.6.0		0.1	FIVE	D.C.W*				5						
0.7.0		0.6	CI	D.C.W*				0.8.3	3.3.3					
0.8.0		0.2	T.WLVE	D.C.W*				+1.2						
0.9.0			END	START										
1.0.0														

Figure 6.3. Program to compute an average, as diagrammed in Figure 6.2.

about the way computers are programmed. The student is urged to understand this example thoroughly before proceeding.

With this basic loop clearly understood, we can without too much difficulty extend it to include finding the month with the largest usage. This can be done by starting with the initial assumption that January has the largest usage. A 1 is stored in a location that will contain the number of the month having the largest usage. January's usage is then compared with February's; if January's is larger, then January is still the largest of those considered so far and the 1 is maintained as the number of the month having the largest usage. If, on the other hand, February has larger usage, then February's usage is moved to the location containing the largest usage so far and a 2 is placed in the location containing the number of the largest month. This "largest usage to date" is continually compared with each succeeding month and either left where it is if it is larger or replaced by another month.

In order to simplify the loop, we will actually begin by comparing January's usage with itself. Thus we avoid having to set up a somewhat longer initializing section to get the loop properly started, bearing in mind that in this testing we are still accumulating the total usage in order to develop the average. This may seem like a waste of time, which it is, but it is worth it: we are saved the complication and the space that would be required to make the loop operate differently the first time. This also is a rather general situation. Simplicity is usually a virtue in programming, since it reduces the likelihood of making mistakes. Furthermore, the time that might be saved by repeating the loop one less time would probably be completely offset by the extra instructions that would be required to get it started properly.

A block diagram is shown in Figure 6.4 and a symbolic program in Figure 6.5. In this complete program there are a number of additional things to initialize. Besides the accumulator and the counter, there are now three variable addresses to start properly. January's usage must be moved to LARGE and a 1 must be put in the month number. This last is transferred with a Zero Add to get the one-digit constant of 1 into the two-digit field.

The variable address Add instruction is as before. We next compare the current month's usage with what is so far the largest usage. The first time

through January is compared with January, but no damage is done and a few instructions are saved. If the one that has been largest so far is larger than the current month's usage, we branch directly to

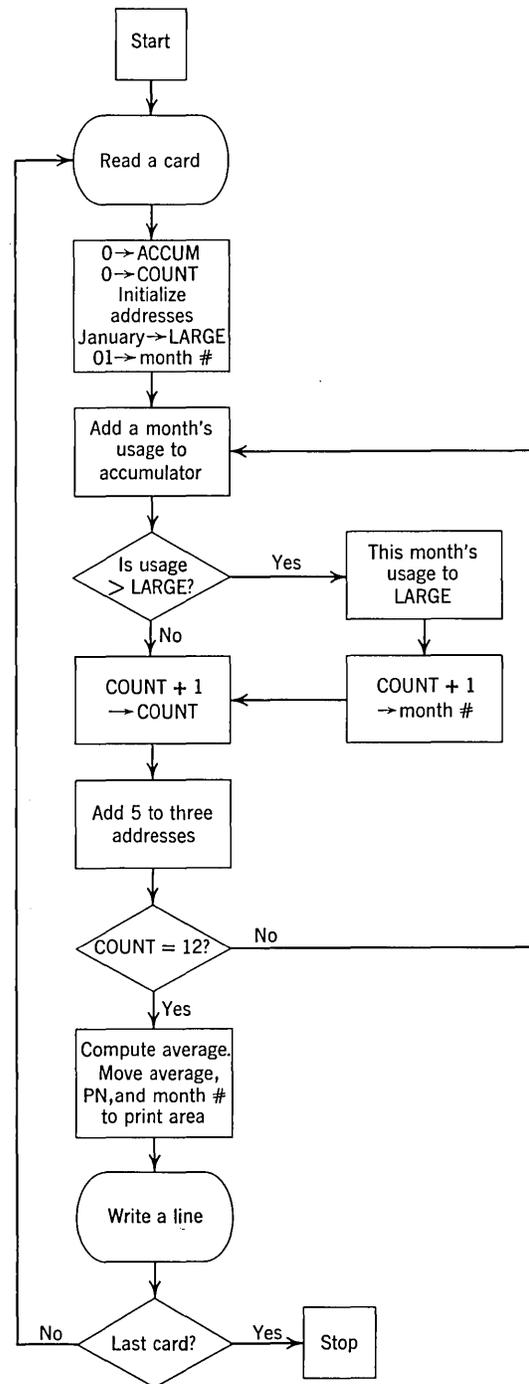


Figure 6.4. Block diagram of a procedure to compute the average of 12 numbers on a card and find which of the 12 is largest.

the modification section. If it is not, the current month's usage is moved to LARGE, the count is moved to the month number, and one is added to this count. This is necessary because the count as set up here is always one less than the number of the month with which we are currently dealing.

The modification section is just about as before, except that there are three addresses to modify. The final instructions of the program are the same, except that the new field must also be moved to the print area.

REVIEW QUESTIONS

1. Suppose the COUNT had been initialized to 1 instead of zero. What constant would have to be changed?
2. Suppose the COUNT had been tested *before* adding 1 to it. What should the COUNT be tested against in this case?
3. What would happen if a Zero and Add instruction were used to transfer alphabetic data?

6.5 Indexing

We see in the program just completed that a fair number of instructions were used in doing nothing but modifying addresses. In many programs a rather high fraction of the total instructions are involved in operations that are required to get the program to operate correctly but which do not themselves directly process any data. A valuable machine feature in reducing this kind of red tape is the indexing of addresses.

The basic idea of indexing is to leave the addresses of the variable address instructions unchanged as they appear in storage and to modify them with the contents of an *index register* each time they are executed by the object program. Between executions of the repeated portions of the loop we can change the contents of the index register. This will have the effect of changing the effective address but will not actually change the instruction as it appears in storage, because the addition of the index register contents to the address

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS		
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.				
3	5	6	7	8	13	14	16	17	23	27	28	34	39	40	55
010		START	R												
020			ZA	ZERO							ACCUM			INIT. TOTAL	
030			ZA	ZERO							COUNT			& COUNTER	
040			MCW	INTADD							ADDINS+0.03			INIT.	
050			MCW	INTADD							COMINS+0.03			ADDRESSES	
060			MCW	INTADD							MOVINS+0.03			X	
070			MCW	0.013							LARGE			JAN. USAGE	
080			ZA	ONE							MONTHN			MONTH. NO. 1	
090		ADDINS	A	0.000							ACCUM -0.07			SUM - VAR. ADDRS	
100		COMINS	C	0.000							LARGE			COMP. USAGE	
110			B	MODIFY										UBR. IF LARGER	
120		MOVINS	MCW	0.000							LARGE			THIS MO. LARGER	
130			MCW	COUNT							MONTHN			MOVE NO. TO NO.	
140			A	ONE							MONTHN			IF LARGEST MONTH	
150		MODIFY	A	ONE							COUNT			INC. COUNT	
160			A	FIVE							ADDINS+0.03			MODIFY	
170			A	FIVE							COMINS+0.03			ADDRESSES	
180			A	FIVE							MOVINS+0.03			X	
190			C	COUNT							TWLV			FINISHED Q	
200			B	ADDINS										NO	

Figure 6.5. Program to carry out the procedure diagrammed in Figure 6.4.

as written is carried out in the address registers and not in storage. To summarize: instead of actually changing the addresses of instructions that vary, we specify that before execution the address as written should be incremented by the contents of an index register. This process does not change the instruction as it appears in storage; we can get the effect of a variable address simply by changing the index register contents.

In a loop in which only one address has to be modified, this procedure does not offer any strong advantages unless there are specialized instructions for doing combination operations on the index registers. Even in the absence of such features, however, the indexing principle becomes very valuable if there are several instructions that have to be changed, since the same index register can be used to modify any number of instructions. The initialization now consists of just the one instruction required to put the proper initial contents into the index register, and the modification consists only of adding the required constant to the index regis-

ter. Furthermore, the index register now also serves as a counter that can be used to determine when the loop operation is completed.

In the 1401 there are three index registers which are named 1, 2, and 3. Index one consists of storage locations 087 to 089; index two, 092 to 094; index three, 097 to 099.

To add the contents of an index location to the address of an instruction, we tag the address that should be modified. This is done, in actual machine language, by using the zone bits of the tens position of the address in the following pattern:

TABLE 6.3

Index Location	Tens Position Zone Bits	Zone Punch
1	01	Zero
2	10	Eleven
3	11	Twelve

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS																								
				ADDRESS	±	CHAR. ADJ.	INC.	ADDRESS	±	CHAR. ADJ.	INC.																										
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55																					
0	1	0	M	C	1						A	C	C	U	M	YES. MULT. 1/12.																					
0	2	0	A	F	I	V	E				A	C	C	U	M	R	ROUND.																				
0	3	0	M	C	S	A	C	C	U	M	-	0	0	6	0	2	1	6	S	E	T																
0	4	0	M	C	S	0	0	0	8		0	2	0	8								U	P														
0	5	0	M	C	S	M	Ø	N	T	H	N		0	2	2	1							A	N	D												
0	6	0	W																						P	R	I	N	T								
0	7	0	B	L	A	S	T																		A	L	A	S	T	C	A	R	D	Ø			
0	8	0	B	S	T	A	R	T																													
0	9	0	L	A	S	T	H	*		-	0	0	3																								
1	0	0	H																																		
1	1	0	1	4	A	C	C	U	M	D	C	W	*																								
1	2	0	0	2	C	Ø	U	N	T	D	C	W	*																								
1	3	0	0	1	Z	E	R	Ø		D	C	W	*																								
1	4	0	0	3	I	N	T	A	D	D	D	C	W	*																							
1	5	0	0	1	Ø	N	E			D	C	W	*																								
1	6	0	0	1	F	I	V	E		D	C	W	*																								
1	7	0	0	6	C	I				D	C	W	*																								
1	8	0	0	2	T	W	L	V	E		D	C	W	*																							
1	9	0	0	5	L	A	R	G	E		D	C	W	*																							
2	0	0	0	2	M	Ø	N	T	H	N	D	C	W	*																							
										E	N	D	S	T	A	R	T																				

Figure 6.5 (Continued).

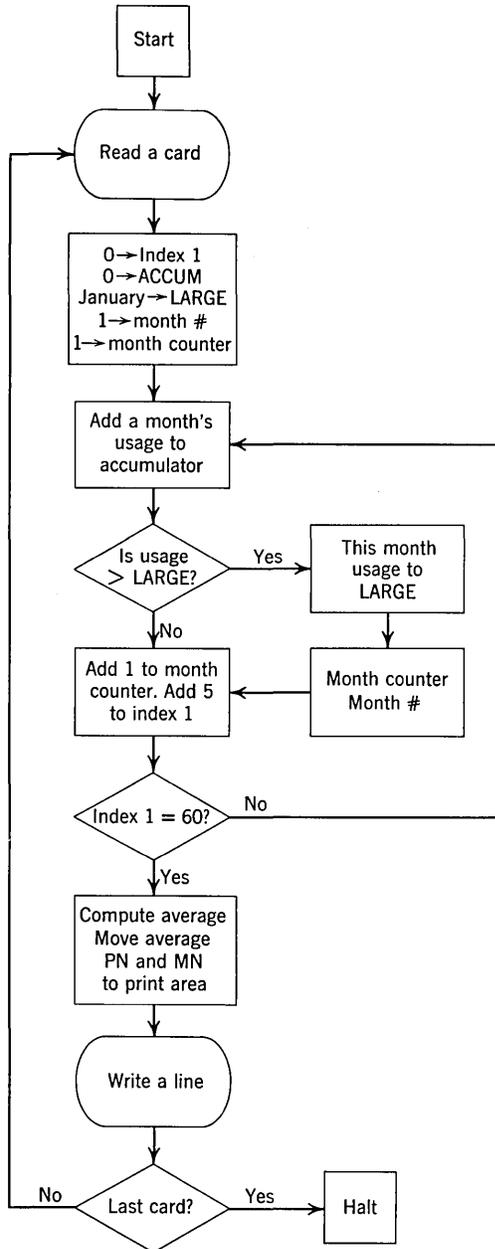


Figure 6.6. Figure 6.4, modified to show the use of indexing.

On the symbolic programming sheet it is necessary only to write the number of the desired index location in the appropriate IND column—that is, column 27 or 38.

When an indexed instruction is executed, the sequence of operations within the machine is as follows. The instruction is first brought to the control section registers just as it always is. During this process the zone bits of the tens position are de-

tected as specifying indexing. The contents of the specified index location are obtained from storage and added to the contents of the address register. The instruction is then executed. Note that the instruction as it appears in storage is not changed by indexing. (The index locations are, of course, not changed either.) The address as modified by the contents of an index location is called the *effective address*.

In order to change the effective address, it is necessary only to change the contents of the index location, which may be done with ordinary 1401 instructions. To do this, the index locations will ordinarily have to have word marks, since the locations are not treated any differently than any other locations in storage, except as they are called on by the execution of an indexed instruction. When the index locations are not being used for indexing, they may be used for other purposes.

We may see how indexing can be used by re-writing the program of the last subsection. The basic logic is not appreciably different. There are fewer instructions, for by initializing the one index location we initialize the *effective* address of the three instructions that must have variable addresses, and one instruction that adds 5 to the index changes the effective address of all three. Furthermore, the index can also be used as the counter. We write the instructions that are to have variable *effective* addresses with *actual* addresses of 0013. The index location, which is chosen to be 1 in the program shown below, is initialized to zero. Each time through the loop 5 is added to this location; loop testing consists of asking whether index 1 contains 60.

The block diagram for this program is shown in Figure 6.6 and the program, in Figure 6.7.

In this particular program it is still necessary to have a counter that counts by ones to know the month number as we make the comparisons to find the month having the largest usage. In this particular case it would not matter much whether the loop testing were done by using the index register or by comparing this month counter against 13. In many problems, of course, there would not be this choice. We see that even though it is necessary to have what amounts to two loop counters, the program is still somewhat shorter than the unindexed version. We note that the three instructions that have variable effective addresses were written with actual addresses of 0013 and that index 1 is specified in column 27 in each case. It

happens not to be necessary here, but it is also permissible to index B-addresses.

This example nicely illustrates the power of the indexing technique in reducing the auxiliary operations of an address modification loop. Since we are concerned primarily with the concept and not with the details of operation, we are omitting a complete

description of what the machine does in certain situations involving addresses over 999 and a number of other matters that are important when using the 1401 but are not crucial to the indexing concept.

Index registers, which are also sometimes called *B-boxes*, or *indexing accumulators*, are available on most computers. In some machines there are more

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				COMMENTS				
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	START	R												
0	2	0		ZA	ZERØ					0089					ZERØ TØ INDEX 1	
0	3	0		ZA	ZERØ					ACCUM					ZERØ TØ TAL	
0	4	0		MCW	0013					LARGE					JAN. USAGE	
0	5	0		ZA	ØNE					MØNTHN					MØNTH. NØ. 1	
0	6	0		ZA	ØNE					MTHCTR					1 TØ MØNTH. CØUNT	
0	7	0	ADDINS	A	0013				1	ACCUM	-007				INDEXED	
0	8	0	CØMINS	C	0013				1	LARGE					X	
0	9	0		B	MØDIIFY										UBR. IF LARGER	
1	0	0	MØVINS	MCW	0013				1	LARGE					INDEXED	
1	1	0		MCW	MTHCTR					MØNTHN					NØ. LARGEST MØN.	
1	2	0	MØDIIFY	A	ØNE					MTHCTR					ADD 1	
1	3	0		A	FIVE					0089					INDEX 1 + 5	
1	4	0		C	0089					SIXTY					FINISHED Q	
1	5	0		B	ADDINS										NØ	
1	6	0		M	CI					ACCUM					YES. MULT. 1/12	
1	7	0		A	FIVE					ACCUM	-005				ROUND	
1	8	0		MC'S	ACCUM	-006				0216					SET	
1	9	0		MC'S	0008					0208					UP	
2	0	0		MC'S	MØNTHN					0221					AND	

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				COMMENTS				
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0		W											PRINT	
0	2	0		B	LAST										LAST CARD Q	
0	3	0		B	START										NØ	
0	4	0	LAST	H	*	-003									YES	
0	5	0		H												
0	6	0	1.4	ACCUM	DCW*											
0	7	0	0.1	ZERØ	DCW*	0										
0	8	0	0.1	ØNE	DCW*	1										
0	9	0	0.1	FIVE	DCW*	5										
1	0	0	0.6	CI	DCW*	0.833	3.3									
1	1	0	0.5	LARGE	DCW*											
1	2	0	0.2	MØNTHN	DCW*											
1	3	0	0.2	MTHCTR	DCW*											
1	4	0	0.2	SIXTY	DCW*	+60										
1	5	0	0.3		DCW	0089										
1	6	0		END	START											
1	7	0														
1	8	0														

Figure 6.7. The program of Figure 6.5, modified to use indexing.

than three, 10 being a typical number. In some machines the contents of the index register are subtracted from the actual address instead of added to it. In a number of computers there are specialized instructions that make indexing even more powerful. At least one computer has an instruction that is a combination of a conditional branch and a subtract, making it possible to write useful loops that have only two repeated instructions.

REVIEW QUESTIONS

1. Does indexing change the indexed instruction as it appears in storage?
2. Is indexing done in the processor or in the object program?
3. Is it possible to have both character adjustment and indexing of a single address? Explain the effect of each and when each is done.
4. What are the advantages of indexing?

EXERCISES

- *1. In the program of Figure 6.3 the variable address of the Add instruction could be used as a counter to determine when the loop has been executed 12 times. Rewrite the program accordingly. *Hint.* Determine carefully what the loop testing constant should be.
2. Modify the block diagram of Figure 6.4 and the program of Figure 6.5 to produce on the report the number of the month having the *smallest* usage as well as the largest. Write with or without indexing.
3. Modify the block diagram of Figure 6.4 and the program of Figure 6.5 to print an X behind the month number if two or more months had a usage larger than all others. Write with or without indexing.

*4. Draw a block diagram and write a program to do the following. Read a card and move columns 1 to 20 to 0401 to 0420; read another card and move columns 1 to 20 to 0421 to 0440; read another card and move columns 1 to 20 to 0441 to 0460, etc. When columns 1 to 20 of 20 cards have been moved to the new locations (the information from the last card goes to 0781 to 0800), leave a blank line in your program for writing the 400 characters in 0401 to 0800 onto magnetic tape. Then assemble another block of 400 characters in 0401 to 0800 and indicate writing on tape. Continue writing on tape until all cards have been read.

The number of cards in the deck is not necessarily a multiple of 20, so a last-card test must be made after moving each group of 20 characters. When the last card is detected, indicate the writing on tape of the last block, even though it is most likely not full. Write with or without indexing.

5. Columns 21 to 22 of an invoice card contain a two-digit state number between 01 and 50. Write a program segment to find the four-character alphabetic abbreviation corresponding to the number and place the abbreviation in positions 0237 to 0240 in the print area. There is a table in storage as follows:

State Number	Address of Abbreviation
01	0785
02	0789
03	0793
04	0797
.	.
.	.
.	.
50	0981

A loop is not necessary.

7. MISCELLANEOUS OPERATIONS

7.1 Editing and Format Design

Many business applications of computers require that reports be printed. These include such things as checks and earnings statements, sales summaries, bills to customers, deduction registers, and inventory summaries. In the printing of most such reports it is necessary to spend a fair amount of effort in planning for ease of readability. This area includes a number of activities such as planning the proper spacing of the information on the report, numbering of pages, printing of headings, proper placement of total lines, insertion of dollar signs, commas, and decimal points, and suppression of unwanted zeros in the high-order positions of numbers. This planning of the format and appearance of reports can take a considerable amount of time, and it can also easily happen that a sizable fraction of an entire program will be taken up with editing results for printing.

In this section we consider the horizontal placement of information within a line. In Section 7.2, on carriage control, we discuss the vertical positioning of the lines on a page.

The fundamental consideration in planning the spacing of information on a line is that sufficient space must be allotted to contain the largest quantity that can ever be printed, with at least a few additional spaces to make the reading easier. Some printing fields are of constant length; a social security number, for instance, always has nine digits and is almost always printed with two hyphens. Many other fields are of variable length, such as a man's name or almost any

dollar amount. The first step in planning, therefore, is to determine the maximum size of each field to be printed.

Next we decide the sequence of information across the line. Sometimes this is specified in advance; at others, it is left to the programmer to decide. Usually a fairly logical scheme will suggest itself. For instance, if a sales summary is to be printed, it would be uncommon to begin the line with anything but the product number. Sometimes the arrangement of information on a line—and perhaps even the spacing—is predetermined by the use to which the report will be put. For instance, W-2 forms for withholding summaries are usually available in preprinted form; the program designer must put the information in the spaces allowed. This touches on the whole broad area of forms design, which is somewhat outside the scope of this book.

Next, we consider any editing that is to be done on each field as it is printed. For instance, most dollar amounts are printed with a decimal point, commas, and a dollar sign. Naturally, these punctuation marks must be included in planning the amount of space required for each field. Very commonly, leading zeros at the beginning of the number are deleted in printing.

The computer techniques by which the fields are printed with punctuation marks and by which zeros are suppressed naturally depend on the instructions available in the particular computer. As we have seen, there are several specialized instructions available in the 1401 that greatly simplify editing. The Move Characters and Suppress Zeros

TABLE 7.1

Field	Symbol	Maximum Number of Characters
Customer number	CUSTNO	5
Customer name	CUSTNA	25
Invoice number	INVNO	5
State	STATE	2
District	DIST	2
Invoice month	MONTH	2
Invoice day	DAY	2
Invoice amount	AMOUNT	6

instruction makes a simple matter of deleting the leading zeros, if that is all that is required. The Move Characters and Edit instruction, as we have seen, greatly simplifies the insertion of punctuation symbols and, in fact, is able to do a good deal more. It may be worthwhile to review the basic functions of this instruction before pointing out one or two additional features of it.

The Move Characters and Edit instruction requires that an edit control word be placed in the output storage area before the data is edited. This edit word will contain any punctuation marks that are to be inserted in the field. When the instruction is executed, characters from the A-field are moved to the B-field, working from right to left, except that characters in the B-field other than zero and blank are not disturbed. This means ordinarily that dollar signs, commas, and decimal points are left unchanged in the B-field. However, almost any other character may be put into the edit word and will also be left unchanged. An exception is the ampersand, which, if present in the edit word, will be replaced by a blank and the blank will not be disturbed by the movement of characters from the A-field. This makes it possible to insert blank spaces in the edited field, such as between a dollar amount and the letters CR for credit.

If suppression of leading zeros is desired, a zero should be inserted in the control word. After the A-field has been transferred to the B-field, then leading zeros will be deleted down to and including the original position of the zero. The point of this qualification on the position of the zero is that there is usually a maximum amount of zero suppression desired. For instance, if we have set up the program to print amounts in dollars and cents, we

ordinarily want amounts under one dollar to be printed in the form .xx. The limit is indicated to the machine by the position of the rightmost zero in the control word. The zero suppression part of the editing operation also replaces with blanks any commas to the left of the first significant digit in the field.

The editing operation can be used to do something else. In planning output formats, it is always necessary to decide what is to be done with negative amounts. These are most commonly printed with a minus sign; they can also be indicated by the letters CR. The question arises: if the field to be printed can be either positive or negative, how do we handle the decision to print the minus sign (or the credit symbol)? The Move Characters and Edit instruction makes provision for this problem. To describe its action, we must define the *body* of the control word and the *status* portion of the control word. The body is the part beginning with the rightmost blank or zero and continuing to the left until the character at which the A-field word mark is sensed. The remaining portion of the control field is referred to as the status portion. The handling of negative amounts is as follows: we insert the minus sign or the characters CR, whichever is desired, in the status portion of the control word. The Edit instruction automatically determines whether the number in the A-field is positive or negative. If it is negative, the minus sign (or the CR) is left in the field; if the data field is positive, the symbols are blanked out.

For an example of the use of these features, consider the printing of a line of an accounts receivable register. Shown in Table 7.1 is the information that must be printed, the symbol assigned to each field in the program of Figure 7.1, and the maximum number of characters in each field.

The customer name is alphabetic and is to be printed exactly as it appears in storage. All the other fields except the dollar amount are to be printed with simple zero suppression. The date is to be printed in two columns to separate the month and the day. The invoice amount is to be printed with a dollar sign, comma, and decimal point. It must also be printed with a CR if the amount is negative. (This would, of course, indicate an "invoice" sent out to show a credit from an overpayment.) The CR is to be printed one position to the right of the amount—that is, with one blank between the pennies and the C.

IBM **1401** Symbolic Programming System Coding Sheet

Program _____ Page No. 1 of 2

Programmed by _____ Date _____ Identification 76 80

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				COMMENTS				
				ADDRESS	±	CHAR. ADJ.	IND	ADDRESS	±	CHAR. ADJ.	IND					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	M,C,W	C,U,S,T,N,A						0,2,2,5						
0	2	0	M,C,S	C,U,S,T,N,Ø						0,2,3,3						
0	3	0	M,C,S	S,T,A,T,E						0,2,3,8						
0	4	0	M,C,S	D,I,S,T						0,2,4,3						
0	5	0	M,C,S	I,N,V,N,Ø						0,2,5,1						
0	6	0	M,C,S	M,Ø,N,T,H						0,2,5,6						
0	7	0	M,C,S	D,A,Y						0,2,5,9						
0	8	0	M,C,W	E,D,I,T,W,D						0,2,7,4						
0	9	0	M,C,E	A,M,Ø,U,N,T						0,2,7,4						
1	0	0	H	*												
1	1	0	/ 2 E,D,I,T,W,D	D,C,W	*					Ø						
1	2	0														

Figure 7.1. Program illustrating editing operations.

In the absence of a predetermined format or an existing form that *must* be used, we are free to make our own decision as to the order in which these data fields should be printed on the line, as well as their spacing. It seems reasonable to begin the line with either the customer number or the customer name and to group the customer name, number, and location at the left of the line. The invoice number, date, and amount could reasonably be printed in that order toward the right side of the line. Let us agree rather arbitrarily to print the customer name first. Now, considering the editing symbols that will be inserted in the invoice amount, allowing, say, three spaces between fields, and assuming that we begin printing with print

position 1, we arrive at the assignments for the fields on the report shown in Table 7.2.

The program segment required to set up the printing line once the data is in the symbolic location shown is presented in Figure 7.1. For simplicity in studying the principles of editing, the addresses are shown in absolute. We understand that in ordinary practice these should be symbolic.

Note that the edit control word is shown with an ampersand, which will cause the edited field to contain a blank space at that position. The credit symbol is shown as the characters CR; these are deleted by the execution of the instruction if the amount is positive (as, of course, it is in most cases). The zero in the edit control word indicates the rightmost limit of zero suppression. The comma is deleted if the amount is less than a thousand dollars. If zero suppression does occur, there will be blanks between the dollar sign and the first digit of the amount. (An optional feature on the 1401, called expanded print edit, would make it possible to move this dollar sign so that it would be immediately to the left of the first significant digit.)

TABLE 7.2

Field	Position
Customer name	1-25
Customer number	29-33
State	37-38
District	42-43
Invoice number	47-51
Month	55-56
Day	58-59
Amount	63-74

REVIEW QUESTIONS

1. What does the zero in an edit control word do?
2. Name some of the considerations in deciding on the placement of information in a printed line.

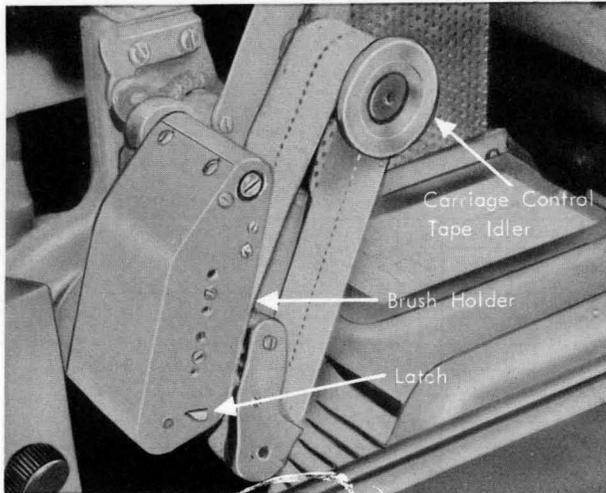


Figure 7.2. Tape-reading mechanism of the tape-controlled carriage in the 1403 Printer.

7.2 Printer Carriage Control

Control of the vertical spacings of lines on a report is necessary for a variety of reasons. Sometimes a heading line must be printed at the top of a page and separated from the body lines by one or two lines of space. Often a preprinted form requires that the printing appear in specified positions on the form, making it necessary to space the paper to these positions before printing. Sometimes there is a variable amount of information to be printed on each page, followed perhaps by a total line, after which the form must be spaced to the top of the next page. This might happen, for instance, if all the purchases by one customer have to be listed, starting on a new page, followed by a total line. After this, of course, the information for the next customer should start at the top of a new page.

Control of the spacing of the output document is accomplished by a combination of programmed signals to the printer carriage and a control tape in the carriage itself. The control tape is prepared for each application, or each group of similar applications, with holes in proper positions to indicate where carriage spacing is to stop once it is started. The tape-reading mechanism is shown in Figures 7.2 and 7.3. The mechanism is seen to be in some ways analogous to a card-reading system. That is, brushes are kept from making contact with an electrically charged roller, except where holes appear in the loop of paper tape.

A control tape has 12 columns of positions indicated by vertical lines. These positions are called channels. Holes can be punched in each channel throughout the length of the tape, which is ordinarily the same length as one complete page. A maximum of 132 lines can be used to control a form, although for convenience the tape blanks are slightly longer. Horizontal lines are spaced six to the inch, the entire length of the tape, which corresponds to the height of one line of printing. If the form has fewer than 132 lines (most do), then the tape can be cut off at the desired length. Round holes in the center of the tape are provided for the pin feed drive that advances the tape in synchronism with the movement of a printed form through the carriage. The effect is exactly the same as if the control holes were punched along the edge of each form.

At any point in a program where skipping of the form to a new printing position is desired, a signal can be given by using the Control Carriage instruction. The B-character of this instruction specifies which channel of the tape is to stop skipping, as shown in the summary box. For example, if a skip to channel 6 is called for, the paper will start

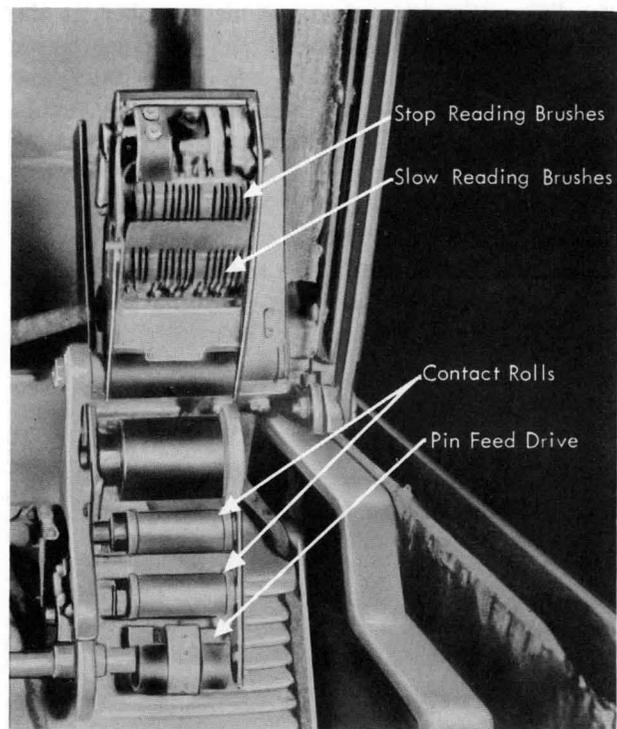


Figure 7.3. Carriage tape brushes.

moving and will stop only when a hole in channel 6 is detected. Depending on the purpose of the form control operation, there may be only one hole in a channel or there may be several holes, and, of course, there may be unused channels. The Control Carriage instruction may also be used to cause the spacing of one, two, or three lines not under control of the control tape. This function is also shown in the summary box.

It is essential in using the two types of spacing (not skipping) to realize that there is normally one

space after printing. If an *immediate* space is used, then there will be as many lines spaced over as are called for by the d-character. If a J is written, then one line will be spaced, etc. After spacing, the line is printed and then the paper is spaced one line as normally. When spacing *after* printing is called for by writing a slash, S or T, the number of spaces prescribed will be the *total* number of spaces after printing, *including* the one that normally occurs. Thus a Control Carriage instruction with a d-character of slash has no net effect. A

Control Carriage

FORMAT

Mnemonic	Op Code	d-character
CC	<u>F</u>	x

FUNCTION This instruction causes the carriage to move, as specified by the d-character. A digit causes an immediate skip to a specified channel in the carriage tape. An alphabetic character with a 12 zone causes a skip to a specified channel after the next line is printed. An alphabetic character with an 11 zone causes an immediate space. A zero zone character causes a space after the next line is printed. The table shows the function of the d-character. If the carriage is in motion when a Control Carriage instruction is given, the program will stop until the carriage comes to rest. At this point the new carriage action is initiated, and the program advances to the next instruction in storage.

d	Immediate skip to	d	Skip after print to
1	Channel 1	A	Channel 1
2	Channel 2	B	Channel 2
3	Channel 3	C	Channel 3
4	Channel 4	D	Channel 4
5	Channel 5	E	Channel 5
6	Channel 6	F	Channel 6
7	Channel 7	G	Channel 7
8	Channel 8	H	Channel 8
9	Channel 9	I	Channel 9
0	Channel 10	+	Channel 10
#	Channel 11	.	Channel 11
@	Channel 12	□	Channel 12

d	Immediate space	d	After print space
J	1 space	/	1 space
K	2 spaces	S	2 spaces
L	3 spaces	T	3 spaces

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)$ ms plus remaining form-movement time if carriage is moving when this instruction is given. The form-movement time is determined by the number of spaces the form moves. Allow 20 ms for the first space, plus 5 ms for each additional space.

d-character of S will call for one *additional* space and a d-character of T will call for two additional spaces. Finally, it must be realized that, although the immediate skip and the immediate space cause the requested action to take place as a result of *this* instruction, the skip after printing and the space after printing become effective only *after* the next line is printed. An after-print skip or space would have no effect if another line were never printed.

If the carriage is already in motion when a Control Carriage instruction is executed, the program waits until the carriage comes to rest. At this time the new carriage action is initiated and then the program advances to the next sequential instruction.

The Control Carriage instruction, as we have described it, is a two-character instruction. It is also possible, however, to write an I-address, in which case the instruction is called Control Carriage and Branch. After carrying out the prescribed carriage action, the next instruction is taken from the location specified by the I-address.

One of the most common and at the same time simplest forms of operation is skipping to a new page when one page has been printed, perhaps after first printing a total line. There are two rather different ways to sense the end of a page. One way is to set up a program counter to count the number of lines already printed. When this counter reaches the number of lines in a complete page of the particular report, a Control Carriage instruction can be executed. (Although there is no logical necessity for doing so, the skipping to the top of a new page is most commonly controlled by a punch in channel 1.)

The other way is to put a punch in channel 9 or 12 in a position corresponding to the last printing position on the page. The detection of a hole in either of these channels turns on a corresponding indicator, which stays on until a punch in another channel is sensed. This makes it possible to print lines without counting them and to detect the end of the page by detecting the proper punch in the carriage control tape. This has the advantage of not requiring a program counter, which can sometimes be inconvenient.

Whichever method of end-of-page detection is used, we often set up the signal so that it indicates only the end of printing in the *body* of the page. A typical page format consists of a heading line, a certain maximum number of body lines, and a total or summary line. The signal that the end

of the page is about to be reached is needed when the last body line has been printed. We then commonly skip a line before printing the total and go on to the next page. The presence of two additional lines after the last body line must, of course, be taken into account in setting up the constant against which the line counter is tested or in punching the hole in channel 9 or 12 of the carriage control tape.

REVIEW QUESTIONS

1. State precisely what action is caused by the instruction

CC 0800 S

2. What would you do if a form were so short that the corresponding length of carriage control tape was not long enough to go around the tape reading mechanism?

7.3 Input and Output Timing

The discussion so far has said little about the timing of reading or punching a card or printing a line. The maximum speeds have been given, but these are hardly the whole story—in the 1401 or in any other computer. We must be concerned also with a number of other questions:

1. If the maximum speed cannot be obtained, does the speed drop to some lower figure in a large jump?
2. For what portion of the total reading or writing cycle is the computer waiting on the input-output operation and unable to do processing? Conversely, for what portions of the total cycle is it possible for the computer to be carrying out processing?
3. At what point during the cycle for one operation is it necessary to give the impulse to start another one if the device is to operate at maximum speed?

Questions of this general sort must be considered in planning the programming of input and output operations in any computer. However, the features of individual machines vary so greatly that it is hard to make generalizations about all machines. Therefore, we turn to a detailed consideration of the 1401 as generally indicative, although not everything we say applies exactly in the same form to other machines.

Card reading in the IBM 1401 is carried out at a

maximum speed of 800 cards per minute. This works out to 75 ms for the reading of one card. The 75 ms are divided into three portions as shown in Figure 7.4. The *read start time* of 21 ms is the interval between the starting of the cycle and the time when information actually begins to move into core storage. It is spent in moving the card from the hopper to the point where the nine row is under the brushes and information starts to transfer into storage. The card reading time of 44 ms is taken up with the reading of the 12 rows on the card and the transfer of the information into storage. The remaining 10 ms of processing time may be used for processing the information on this card and still maintain the reading of cards at maximum speed. If the instruction to read the next card is executed before the 10 ms processing time is completed, cards will read at full 800-per-minute speed; if the processing time exceeds 10 ms, then the card-reading speed drops in one single jump to 400 cards per minute.

This jump is caused by the fact that there is only one point in the cycle of the card reader at which an impulse to start the card-reading operation can be obeyed.¹ If the impulse comes before the end of processing time, it will be obeyed—that is, another card will be read, without any delay. If the impulse comes after the end of processing time, the mechanism will wait for another complete cycle to elapse before obeying the impulse. This means that there is no steady card-reading speed between 800 and 400 per minute. It can happen, however, that in some cases the succeeding card will be read with no delay and in other cases that there will be a delay of one or more cycles between the reading

¹ An optional feature called Early Card Read provides three starting points, thereby speeding up card reading considerably in some applications.

of successive cards. In such a case the *average* card-reading speed may be some intermediate figure.

It is important to realize that the computer is completely idle during read start time and card-reading time. Stated otherwise: when a Read a Card instruction is executed, the next instruction is not executed until card-reading time for that instruction has been completed—that is, a minimum of 65 ms later. We say that the computer is *interlocked* during the read start time and the card-reading time. (A special feature called *read release* is available for the 1401 to make the read start time available for processing.)

Card punching is carried out at a maximum rate of 250 cards per minute, which works out to 240 ms per card. This cycle is divided into three parts also as shown in Figure 7.5. The punch start time of 37 ms is the interval between the starting of the card motion and the beginning of actual punching. The punching time of 181 ms begins with the 12 row. The 22 ms remaining is available for processing.

The computer is interlocked during punch start time and punching. (If a special feature called *punch release* is installed, punch start time is available for processing.) To maintain full card-punching speed of 250 per minute, the instruction to punch the following card must be executed before the end of processing time. However, in the case of punching, there are four points during the cycle, occurring at 60-ms intervals, at which an impulse to start punching can be obeyed. Therefore, if the instruction to punch another card is given shortly after the end of the part of the cycle shown as processing time, the punching speed will not be slowed down to half the maximum. Instead, the following card will take 300 ms—that is, the nor-

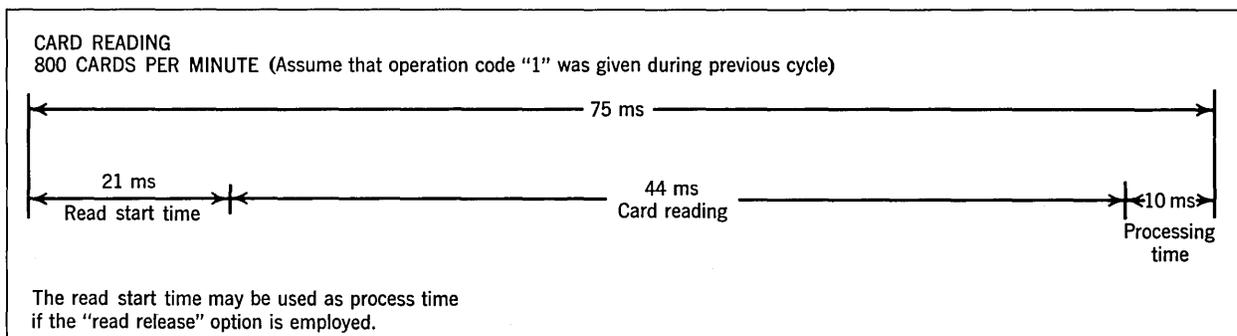


Figure 7.4. Timing diagram for card reading with the 1402 Card Read-Punch.

mal 240 ms plus the 60 ms during which the computer will wait until another punch impulse can be accepted. Thus the "penalty" for not getting the instruction to punch another card executed during processing time is not so heavy with punching as it is with reading.

Printing is carried out at a maximum of 600 lines per minute, which is 100 ms per line. The cycle is divided into two basic parts, with another part of the total operation overlapping one of them, as shown in Figure 7.6. The printing time is 84 ms; during this period the computer is interlocked. The remaining 16 ms are available for processing; if the next print instruction can be given during this processing time, printing will be carried out at full speed. As we have seen before, printing is always followed by a single line space unless a Control Carriage instruction has been executed to specify otherwise. The normal single spacing takes 20 ms, which completely overlaps processing time. It is important to note that any skipping that may have been specified, either immediate or after-print, does *not* overlap any of the processing time. On the other hand, the computer is not interlocked during

the skips unless the skip instruction happens to be executed when the carriage is already in motion, in which case the computer is interlocked only until the preceding movement is completed.

The printer is able to accept an impulse to print a line at any time. If the instruction to print the next line can be given before the end of processing, printing will proceed at full speed. If the instruction to print the next line cannot be given during processing time, the only time penalty is the excess over processing time; we do not have to wait until some specified point in the following cycle. In short, the printing cycle begins whenever the Write a Line instruction is executed.

The total time for input and output operations in the 1401 can be somewhat reduced if it is feasible to use combined operations. The Write and Read instruction, for instance, combines the functions of Read a Card and Write a Line. Its mnemonic operation code is WR and its actual operation code is 3. When this instruction is executed, the printer takes priority and the print cycle is completed before the actual card-reading operation takes place. However, the execution of the instruction is set up

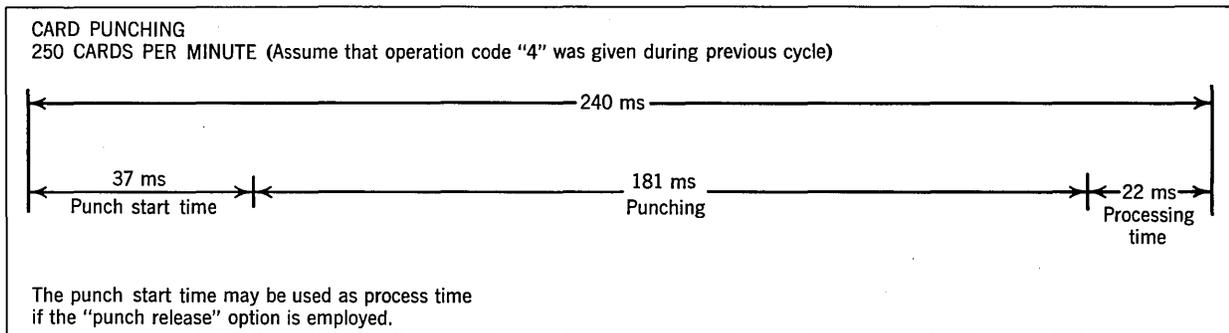


Figure 7.5. Timing diagram for card punching with the 1402 Card Read-Punch.

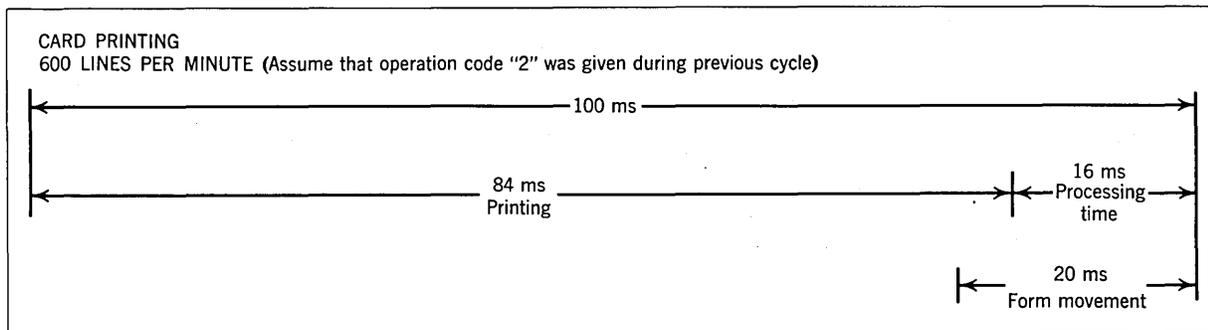


Figure 7.6. Timing diagram for printing with the 1403 Printer.

so that the signal to start the reader is accepted before the end of the print cycle. Thus read start time overlaps the print cycle, with a reduction in the total time required for the two operations. This total time is 150 ms, of which the last 21 ms are available for processing.

The Read and Punch instruction (mnemonic RP and actual 5) combines these two operations with an even more favorable overlap. Here, the two operations occur simultaneously, with the total time, 240 ms, being that for punching a card. The time available for processing is 22 ms.

Write and Punch (mnemonic WP and actual 6) combines these two functions. The situation on overlaps is about the same as with Write and Read: the printer takes priority, but the start punch signal is automatically given by the machine before the end of the print operation. Therefore, the punching begins very shortly after the printing is completed. The total time for the two output operations is 300 ms, of which the last 28 ms are available for processing.

The Write, Read, and Punch instruction combines all three operations. The mnemonic code is WRP and the actual code is 7. Here the printing takes place first, immediately after which reading and punching occur simultaneously. The total time is 300 ms, of which the last 28 ms are available for processing.

The effective use of these combined instructions naturally depends on rather careful planning of the program to insure that the information in the read, punch, and print areas is set up in such a way that the combined operations produce correct results. To take a simple example, suppose that we are required merely to read a deck of cards and print the information on each of them. We immediately think of using a Read and Write combination. However, this can obviously not be done on the *first* card because as we begin there is nothing in the print area to be printed. Therefore, we start with a Read a Card instruction. After the information has been moved from the read area to the print area, possibly with rearrangement of the data fields and editing, then the Read and Write combination can be used effectively to print the information in the print area and then immediately read another card without having to wait for the read start time. In every case, after setting up the information in the print area, we make a last-card test and, when it is satisfied, execute only the Write instruction.

REVIEW QUESTIONS

1. Outline the timing differences between Read, Punch, and Print.
2. Under what conditions can a complete job be done in just the time required for input and output?
3. How much time does the WRP instruction save over doing the three operations separately?

7.4 Buffering

In most business data processing applications there is a relatively large amount of input and output. The total time required to do the job is often largely taken up with reading and punching cards, printing reports, and, as discussed in Chapter 8, reading and writing tapes. We have seen in the preceding section that a relatively small amount of the input and output time is available for processing in the 1401. In the absence of the buffering facilities to be discussed now, this is true in most computers. In fact, in some machines the *entire* cycle is unavailable for processing. This can very well mean that the total time to do the job is the sum of the times required for each of the individual input and output operations plus the total processing time. If the processing time is very much less than the input and output time, or if the processing takes a great deal longer than the input and output, then there is really not much to be saved by trying to overlap the input and output with processing.

However, it frequently happens that the input and output time is about the same as the processing time, in which case it becomes very desirable to set up the machine so that processing can continue during most of the time required for the input and output cycles. To take a specific instance, consider printing. We would like to be able to move the information to be printed from core storage to a small auxiliary storage (this can be done at electronic speeds) and then continue with normal processing while the information is moved from the small storage to the printer at the mechanical speeds of the printing device.

The essence of *buffering* is this: on output, information goes from core storage to the small auxiliary storage, which is called the buffer. Since no mechanical operations are required for this transfer between two electronic devices, it can be done at very high speeds. Then processing may continue while the information is sent out to the output de-

vice at the speeds required of the mechanical device. On input, the process is simply reversed: information is accumulated in the buffer storage as the input medium is read, and, when all of the information has been assembled, it is transferred to core storage at high speed.

Some computers have no buffering; others buffer virtually every input and output operation. The 1401 can be equipped with an optional special feature called *print storage*, which puts it in an intermediate class. Print storage gives us buffering of printing only. However, since in many applications printing time is a fairly sizable fraction of the total job time, this can mean a very significant reduction in the time required to do each job and, therefore, an increase in the data processing capability of the equipment. This is all the more true because print storage permits virtually all of the print cycle to be used for other processing, including other input and output operations. For instance, it is possible to keep both printer and reader running at 400 per minute and still have more than half the total job time available for processing.

The 1401 print storage feature operates in just the manner described for output buffering in general. When the Write a Line instruction is executed, the information in the print area, positions 201 through 300 (or 201 through 332) is moved to a special nonaddressable buffer storage. This transfer requires only 2 ms, and it is only during these 2 ms that the computer is interlocked from processing. As soon as the information has been moved to the print storage buffer, processing can continue for the remaining 98 ms of the print cycle. As we have noted, it is possible to initiate other input-output operations during this time. If another Write a Line instruction is given before the completion of the total 100-ms print cycle, the computer will interlock until the completion of the preceding cycle, at which time the next cycle will begin.

The effective use of buffering requires a certain amount of preplanning of the program organization. For instance, if two Write instructions are given in sequence, then the execution of the second one will be interlocked until the first one is completed. Buffering will have saved nothing on the first instruction. Therefore, whenever possible, we try to space out the printing operation so that a computer will be interlocked as little as possible.

In the 1401, with its capability of buffering only one operation, the planning requirements are not

really severe. Even if the programmer gives no special thought to buffering and simply puts his Write instructions wherever he would put them if the machine did not have print storage, the feature will save a certain amount of time, although it may not be used to full advantage. In some of the larger computers, however, where all input and output operations are buffered, the effective use of the complete computing system requires very extensive programming systems to attempt to keep all of its components in operation as much of the time as possible. These input and output packages are prepared by a special programming group and are then utilized by all other programmers.

REVIEW QUESTIONS

1. Buffering can be described as a way of matching the speed of electronic storage with the much slower speeds of input and output devices. How does buffering save time?
2. Why does buffering not save much of the total percentage of job time when processing already takes much longer than input and output?

7.5 Program Timing

It is frequently necessary to estimate the amount of time that a program will require. Obtaining an accurate estimate of this sort requires a number of pieces of information and careful consideration of a variety of factors that affect the total time to execute the program.

The basic idea is simply to take the total time required for input and output, add to it the total time required for the execution of internal processing instructions, and subtract the amount of any overlapping of input and output with processing. Doing this requires, first of all, estimates of the total amount of input and output, together with timing information on these operations. This much is fairly simple, provided that the volume estimates are reasonably accurate.

The timing of the internal processing operations is a little more difficult. The time required to execute each instruction is fairly readily obtained from the programming manuals. This information has been shown for the 1401 in the summary boxes throughout this manual and is described shortly. This, however, is not the end of the estimating job. Complexity is added by the fact that most programs have alternative paths that may be followed for

different conditions existing in the input. Often these paths are not the same length so that the total time must be derived from a weighted average based on the expected fraction of the time that each of the paths is followed. And this must be a *weighted* average. If the normal path for processing information on a card takes 40 ms, whereas in special cases arising only 2 per cent of the time the processing takes only 10 ms, then one gets a very misleading picture of the total time if the average time for processing one card is taken to be 25 ms. This is one source of complexity.

A second and more serious complication is the fact that processing is often *partly* overlapped with input and output. Related to this problem are other considerations; for example the card reading cycle in the 1401 can begin only at specified times. It can also very easily happen that the processing for some types of data will be *completely* overlapped with input-output, whereas the processing of other data that takes longer will be only *partly* overlapped. This can lead to erroneous estimates if the "variable overlapping" is not taken into account.

For instance, an average processing time may be short enough to allow complete overlapping, but this sort of "average" is very misleading. The time "lost" on the longer-than-average processing cases is not offset by the shorter-than-average cases because once the processing is completely overlapped there is no more time to save.

This is not the place to enter into a complete and detailed explanation of how to handle all these considerations, since the subject depends too strongly on the features of the particular computer being used. We shall have to be content with the observation that if high accuracy of time estimates is required, then extreme care must be exercised in making the time estimate. Carelessly made time estimates are notoriously inaccurate.

We may close this very brief consideration of time estimating by mentioning the 1401 timing formulas given in the summary boxes for the various instructions.

The timing of the IBM 1401 is described in terms of one complete core storage cycle, which is 11.5 μ s (microseconds, or millionths of a second) or 0.0115 ms (milliseconds, or thousandths of a second). The time required for any internal processing instruction is always a multiple of this interval of time. The timing formulas are given in terms of certain characteristics of the instruction under consideration and of the data fields being operated on. The

symbols used for these variables are shown in Figure 7.7.

For an example of the application of these formulas, consider the equation for the Move Characters to A or B Word Mark, which is

$$T = 0.0115 (L_I + 1 + 2 L_W)ms$$

Looking at Figure 7.7, we see that L_I stands for the length of the instruction and L_W stands for the length of whichever data field is shorter. A Move instruction without chaining has seven characters. Suppose that we are moving a field of 11 characters. The total time is therefore

$$0.0115 (7 + 1 + 2 \cdot 11)ms = 0.345 ms$$

We may note in passing how these storage cycles are used. It clearly takes one cycle to get each instruction character from storage to the control registers and one extra to get the operation code of the next instruction and recognize its word mark. This is the basis of the $L_I + 1$ in the formula.

The movement of each character of the data field takes two storage cycles: one to get it from the A-field and one to place it in the B-field. Thus the number of cycles spent in data movement is twice the number of characters moved, which in

SYSTEM TIMINGS

Key to abbreviations used in formulas

L_A	= Length of the A-field
L_B	= Length of the B-field
L_C	= Length of Multiplicand field
L_I	= Length of Instruction
L_M	= Length of Multiplier field
L_Q	= Length of Quotient field
L_R	= Length of Divisor field
L_S	= Number of significant digits in Divisor (Excludes high-order 0's and blanks)
L_W	= Length of A- or B-field, whichever is shorter
L_X	= Number of characters to be cleared
L_Y	= Number of characters back to right-most "0" in control field
L_Z	= Number of 0's inserted in a field
I/O	= Timing for Input or Output cycle
F_m	= Forms movement times. Allow 20 ms for first space, plus 5 ms for each additional space
T_m	= Tape movement times
Σ	= Number of fields included in an operation

Figure 7.7. Abbreviations used in instruction timing formulas.

turn is the number of characters in the shorter field, in the MCW instruction.

The formulas for most of the instructions are equally simple. A few of the more complex instructions have correspondingly complex formulas. Multiplication, for instance, is a fairly long instruction and furthermore depends not only on the length of the field but also on what the digits in the field are. The formula that is shown is fairly complex and at that gives only an average. However, in most cases the computation of the time required for the instruction is not at all difficult.

It will be noted that the input and output instructions show the time in two parts. One part gives the time required in the central computer, to which must be added the time taken up in the actual input or output actions. The time for these actions cannot be given as fixed numbers because of such variables as the restricted number of points at which a card-reading or card-punching cycle can begin and the fact that the *effective* time of these operations depends on whether the processing time can be used for processing or whether the program is organized so that the computer will be interlocked during part of the processing time. Therefore, as we have noted, the estimation of input and output operations depends not only on the way the computer is built but also very strongly on the way the program is organized.

7.6 Subroutines and Utility Programs

A subroutine is a group of instructions that performs some well defined segment of a data processing operation. Subroutines are of two broad types and are used for two rather different reasons.

One frequent reason is that someone else has already written it and it can, therefore, be incorporated in a program with little effort. For instance, in the basic 1401 there is no multiply instruction. It is not unduly difficult to program multiplication, but it takes more effort than a programmer wants to expend every time he has to multiply. Fortunately, there is no need for him to do so: routines are already available for the purpose. All that the programmer has to do is to obtain the cards on which the subroutine is punched and insert them in his program deck before assembly. Knowing that after assembly his object program will contain the multiply subroutine, he can write instructions in his program to use the subroutine without having

to spend any time in programming the multiplication.

If multiplication is required only once or twice in a program, it is satisfactory to insert the subroutine right where it is needed in the program. The only extra operations required are those necessary to place the multiplier and multiplicand in standard locations where the subroutine can find them and to retrieve the product from a standard location where the subroutine puts it. Since the subroutine falls right in the sequence of instruction execution, there is obviously no need to branch to it. This is the essence of the *open subroutine*—that is, it is inserted in the main program where it is needed and appears in the program as many times as it is needed.

Suppose, on the other hand, that multiplication is required at a dozen different places in a program. Now we begin to worry about the storage space that is wasted by having the same subroutine at a dozen places in storage. Why not put it in just once and then branch to it whenever a multiplication must be performed? Now we have a *closed subroutine*. To summarize, a closed subroutine is placed in storage at *one* place; whenever the subroutine is needed, the main program branches to it, and the subroutine branches back to the main program when it is finished.

This does create one new problem: how does the subroutine know where to return when it is finished? This question is answered by a *linkage*. Before branching to the subroutine, one or two instructions in the main program store an address that the subroutine can use to compute the address to which it should return when it is finished. In most computers there is a special instruction that facilitates this storage of the return address. In the 1401 there is an optional instruction called Store B Address Register, which, in conjunction with a special aspect of the 1401 Address Registers, makes it a simple matter to obtain the address of the next instruction after a Branch. The first instruction of the subroutine can store this Branch instruction at the end of the subroutine. No matter where the subroutine was entered from, therefore, the subroutine will return to the next instruction after that. In the absence of this special feature, it is not difficult to do essentially the same thing with standard instructions.

We see now the contrast between an open subroutine, which is inserted where needed and as many times as needed, and a closed subroutine, which is inserted in the program once and to which

the main program branches whenever it is needed. Subroutines are used for two reasons: to save the trouble of writing routines that are already available (this applies to both open and closed subroutines) and to save storage space (this applies only to closed subroutines).

There is available for most computers a group of routines that come into this area of discussion, although they are not set up as subroutines. Examples are programs to load cards, clear storage, and print out specified areas of storage for help in program checkout. For machines where the primary input is through punched cards, these routines are prepared on small decks that are readily accessible at the computer. At least a few of them will generally find use in virtually every program that is run on the machine. The name *utility routines* is applied to a broad category of programs of this type.

In the case of the 1401, there are three heavily used utility programs to illustrate this concept. The *clear storage* program is a two-card routine that clears all storage to blanks, removes all word marks, and then sets a word mark in location 001. It is typically placed at the front of every program loaded into storage to insure that each will begin with a clean slate. It is therefore unnecessary for each programmer to write clear storage instructions at the beginning of his program. The *card loader* is also a two-card routine. It will accept cards of the type produced by the SPS assembly program and load the instructions or constants punched on them into the specified locations in storage. The program also sets all word marks required by the instructions or constants. Finally, the card loader recognizes the card in the object deck produced by the END card in the SPS assembly and branches to the location in the object program specified by this card.

A complete object program deck is typically organized as follows:

- Clear Storage
- Card Loader
- Object Program Deck
- Transition Card (produced from END card)
- Data Cards

The loading of the entire program is accomplished by pushing the *load button* on the 1401 console. This button automatically causes a word mark to be set in 001, the first card to be read into the read storage area, and the instruction at 001 to be executed. The clear storage routine is set up on its

two cards so that these actions will enable it to get started properly. From this point on, all card reading is initiated by instructions in the two utility programs and later in the object program. Thus the clear storage routine loads the card loader program after having cleared all of storage. The card loader program loads the Object Program deck. When the transition card is read, the loader program causes a branch to the object program which then reads and processes data cards.

The last utility program that we consider is one that prints a specified area of storage. This is typically used in checking out a new program when it is desired to see what the storage contents are after attempting to run it. The programmer punches on a *control card* the beginning and ending addresses of the region of storage that he wants printed. This control card is added at the end of the print storage deck and the deck is loaded. (It is, of course, impossible to print that part of storage used by the print storage program itself, but this is a matter of only 146 locations and these locations may be selected to be anywhere in storage.) The print storage program then prints out the contents of the specified storage locations, using an extra printing line to print 1's underneath the characters in which word marks are set. The whole operational sequence of punching a control card, loading the print storage deck and printing out the contents of all storage can be done in a matter of a few minutes, giving extremely valuable data for use in determining whether the program is operating correctly and in establishing what is wrong with it if it is not.

EXERCISES

*1. The following fields are in storage:

Field	Symbol	Length	Sample		
Date	DATE	5	05	22	1
			Month	Day	Year
Account Number	ACCT	7	0078405		
State	STATE	4	OREG		
Amount due	DUE	7	0164329		

These fields are to be printed as shown in the following sample:

bb78405	OREG	b5	22	1	b 1, 643.29
1-7	11-14	18-19	21-22	24	28-36

The four fields have word marks in their high-order positions only.

2. The fields in Table 7.3 are in storage. These fields are to be printed as shown in the following sample:

```
RbBbJOHNSONbbbbbbbbbb
      1-24
      535-22-1583      $861.89
      28-38           42-49
```

Write a program segment to set up this printing line. The three fields in storage have word marks in their high-order positions only.

*3. A deck of cards contains, among other things, an account number in columns 1 to 5 and a dollar amount in 23 to 28. The deck is in sequence on account number, and there are never more than 40 cards with the same account number. All the cards for one account number are to be printed on a separate page. The account number should be printed with zero suppression in positions 1 to 5, and the amount with a decimal point and zero suppression in 10 to 16. When all the cards for one account have been printed, a line should be skipped and the dollar total for the account printed with a decimal point and zero suppression in 8 to 16.

Draw a block diagram and write a program.

4. Two fields from each card in a deck are to be printed. Columns 1 to 7 contain an account number that is to be printed in positions 1 to 7 with zero suppression. Columns 8 to 14 contain a dollar amount that is to be printed in positions 11 to 20 with dollar sign, comma, decimal point, and zero suppression. A heading is to be printed at the top of each page, consisting of ACCOUNT in 1 to 7 and AMOUNT in 13 to 18. After 40 body lines a line is to be skipped and the total of all the amounts on the page printed in edited form in 9 to 20. When the last card is detected, print the total for the partial page and skip to the top of the next page. Draw a block diagram and write a program. Use either

a line counter or a page overflow punch in channel 12 to detect the end of each page. *Hint.* Be sure your program does not fall apart if the last page contains exactly 40 lines.

5. Compare the total input and output time, and the time available for processing, for

- a. reading a card, punching a card, and then printing a line (without read release, punch release, or print storage);
- b. executing the Write, Read, and Punch instruction (without read release, punch release, or print storage);
- c. executing the Write, Read, and Punch instruction (without read and punch release but with print storage);

6. Estimate the time required to execute the program in Figure 3.2, exclusive of reading and printing.

7. Estimate the time required to execute the program in Figure 4.1:

- a. *once*, exclusive of reading and printing;
- b. for 100 cards, including reading and printing (without print storage);
- c. for 100 cards, including reading and printing (with print storage).

*8. Estimate the time required to execute the program of Figure 5.3 for a deck of 10,000 cards including reading and printing (without print storage). Assume five cards per group. *Hint.* You might begin by deciding whether it is worth worrying about the time for housekeeping or about the alternative paths in the program, since they may or may not have any significant effect on total time. An estimate within 5 per cent is pretty good.

9. Estimate the time required to execute the program of Figure 6.4 for a deck of 2000 cards (with print storage). See hint in Exercise 8.

TABLE 7.3

Field	Symbol	Length	Sample
Name with two initials at right	NAME	22	JOHNSONbbbbbbbbbbRB
Social security number	SS	9	535221583
Amount	AMNT	6	86189

8. MAGNETIC TAPE OPERATIONS

Magnetic tape provides compact storage for much larger amounts of information than can be contained in core storage and allows for much faster reading and writing than with punched cards. Magnetic tape storage is available for virtually all large computers and can be installed on the IBM 1401. When the 1401 is used as a medium-sized computer by itself, magnetic tape provides large-capacity storage for files and input data. When the system is used as an auxiliary machine with a larger computer, magnetic tape is employed as a communication device between the larger machine and punch card input or printed output. The following discussion of the physical characteristics of magnetic tape is applicable to all IBM machines.

8.1 Physical Characteristics of Magnetic Tapes

Magnetic tape is wound on a 10½-in. diameter reel. The tape itself is ½ in. wide and 2400 ft long. It is coated with a magnetic oxide material on which information can be recorded in the form of magnetized areas. One reel of tape can contain as many as 16 million characters; the actual number depends on how the information on the tape is organized.

Each character on the tape is recorded in a seven-bit code very similar to that used within the computer. Characters are recorded in groups called *blocks*. A block may contain any number of characters. Blocks are separated from each other by about ¾ in. of blank unrecorded tape called an inter-

record gap. (The word *block* and *record* are occasionally used as synonyms; in this section we shall attempt to maintain a distinction between a physical block on the tape and the one or more problem records that may be contained in the block. Certain usages are so firmly entrenched, however, that we cannot be completely consistent on this point.)

Each tape character is composed of an *even* number of ones; this is contrasted with the representation within the computer in which each character has an *odd* number of ones. This difference is necessary to maintain compatibility with tapes from some of the large IBM computers.

The tape codes for the characters used in the 1401 are shown in Figure 8.1. It is necessary to qualify this statement as applying specifically to the 1401 because a number of special control characters shown at the right of Figure 8.1 do not apply to all IBM machines. The coding of the standard characters is the same in all, however.

In Figure 8.1 the seven-bit vertical groupings are the characters. We speak of the horizontal groupings, containing one bit from each character in the block, as being horizontal rows or, sometimes, channels. Rows are named in the same way as the seven bits of the character within the computer; that is, CBA 8421.

The reading and writing of information with magnetic tape is subject to a considerable amount of built-in checking in IBM tape systems. First, there is parity checking of each character. As each character is written onto tape, the parity bit is computed to make the total number of ones in the character

reading and a separate one for writing. The writing head is positioned in front of the reading head, in the sense of the direction of tape motion. Thus, when a character is written, it is automatically read a very short time later by the read head to determine that the information has been recorded readably on tape and that parity counts are correct. If there should happen to be something wrong with the tape or with the tape unit, then it is a fairly simple matter to correct the difficulty since the information is still available in core storage.

Magnetic tape is written and read in a *tape unit*, of which three types may be used with the 1401: the 7330, the 729 Model II, and the 729 Model IV. The number of tape units attached to a computer is variable at the discretion of the user; in this book we assume a machine with four.

The difference between the three tape units that can be used with the 1401 is in the speed at which the tape moves. In all three there may be either 200 or 556 characters per inch of tape, depending on the setting of a switch on the tape unit. In the 729 IV, the tape moves at 112.5 in. per sec making the transfer rate either 22,500 or 62,500 characters per sec. The corresponding figures for the 729 II are 75 in. per sec and either 15,000 or 41,667 characters per sec, and for the 7330 they are 36 in. per sec and either 7,200 or 20,016 characters per sec. These performance characteristics are summarized in Figure 8.2.

In figuring the time required to read or write a tape block, we must consider not only the time to transfer the characters in the blocks but also the time required to accelerate the tape to full speed before reading or writing. This start/stop time turns out to be about the same as the time required to read past the interrecord gap at full speed.

The total time required to read a tape depends

on the fraction of the time the tape is kept in motion. The average character transfer rate depends not only on this figure but on the size of the blocks, bearing in mind that there is about $\frac{3}{4}$ in. of blank tape between one block and the next. On the 729 tapes at low density, for instance, $\frac{3}{4}$ in. of tape will hold 150 characters. If the blocks are smaller than this size, we see that less than half the tape contains information. Therefore, even if the tape were kept moving all the time, the average character transfer rate would be only half the rate at which they are transferred during the reading or writing of a block. We shall see later that this consideration has a significant influence on the programming techniques used with magnetic tapes.

It is necessary to be able to detect the beginning and the ending of the tape, both for design reasons in the magnetic tape unit itself and because of programming considerations. For this purpose, *reflective spots*, also referred to as *photo-sensing markers*, are placed on the tape to enable the tape unit to sense where reading and writing are to begin and to stop. The reflective spot at the front end of the tape is called the *load point*. Appropriate button pushing on the tape unit when the tape is loaded will cause the tape to be positioned just beyond the load point. The reflective spot near the end of the tape is employed when writing to indicate that the physical end of the tape is about to be reached and that no further information should be written. Detection of the end-of-reel reflective spot during writing turns on an indicator in the computer.

When the end-of-reel spot is detected during a writing operation, we ordinarily write a final block on the tape consisting simply of a *tape mark*. This is a special character that will later be detectable

	7330	729-II	729-IV
Density, characters per inch	200 or 556	200 or 556	200 or 556
Tape Speed, inches per second	36	75	112.5
Inter-Record Gap, inches	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$
Start/Stop Time, Read/Write Operation, milliseconds	varies	10.8	7.3
Character Rate, characters per second	7,200 or 20,016	15,000 or 41,667	22,500 or 62,500
Rewind Time, full reel, minutes	13.3 or 2.2	1.2	0.9

Figure 8.2. Summary of magnetic tape characteristics.

upon reading to signal the end of the tape by turning on the same end-of-reel indicator that is turned on by the reflective spot when writing. The tape-mark technique of denoting the end of the tape is used for two reasons. First, the tape unit does not turn on the end-of-reel indicator when the reflective spot is detected on reading; we must therefore have some other technique for detection of this condition. The second reason is that we sometimes wish to put on tape an indication that no more information follows, even though the end of the physical tape has not been reached. The tape mark provides this capability.

Since the complete collection of valid data blocks is frequently called a file, the tape mark is also sometimes called an end-of-file mark. We prefer here, however, to reserve the word file for the meaning in which it was used in Chapter 1, in order to make a clear distinction between a tape *reel* and a *file* of information, which may consist of only a few blocks, a complete tape, or many tapes.

Each tape reel is provided with a removable plastic ring on the back side, that is, the side nearest the tape unit. This is called the *file protection ring*. It is not possible to write on a tape unless this ring is inserted in the tape reel. This feature is provided as a precaution against accidental destruction of permanent master files. The usual procedure is to remove the file protection rings from such master tapes after they have been initially written and then require the authorization of some responsible member of the organization before the file protection ring can be inserted in any tape reel.

Most people have at least slight difficulty in remembering whether the tape is protected by inserting or removing the file protection ring. We suggest the mnemonic phrase "no-ring-no-write."

REVIEW QUESTIONS

1. Outline the built-in error checking in IBM magnetic tape systems.
2. What is the maximum number of characters that can be written on a 2400 ft reel of tape at low density? At high density? Why would a reel never contain so many characters?
3. What is the purpose of the file protection ring?

8.2 Magnetic Tape Instructions

The operation of magnetic tapes in a computer system is controlled by the execution of suitable

instructions, as is everything else. We shall look into the tape instructions in the 1401 briefly in order to get a general idea of what the basic machine instructions are and what they do. We shall see in Section 8.3, however, that tape operations are seldom actually set up this way in normal programming; instead *macro-instructions* are used, which greatly simplify tape programming. For now, therefore, we wish merely to survey the actual machine instructions in order to understand better what the macro-instructions do.

Tapes are most commonly written and read in the 1401 with two instructions called Write Tape and Read Tape. The operation code for both instructions is M, which is also the operation code for Move; we therefore use the mnemonic operation code MCW. However, both instructions require a d-character, and the net result is that we have two entirely new instructions that are essentially unrelated to a Move. For reading tape the d-character must be R and for writing it must be W.

To write a block on tape in the 1401, we must specify three items of information to the computer.

1. Which of the tape units is to receive the block? This is specified by the A-address of the instruction, which must be of the form %Ux, where x is the number of one of the tape units attached to the system. The per cent sign and the U are required by the design of the system to specify magnetic tape. The numbers of the tape units may be set by a dial on each tape unit. An installation with five magnetic tapes, for instance, would most likely establish the convention that the tape units are dialed so as to run through the numbers 1 to 5.

2. Where in storage is the first character of the block to be written? This is answered by the B-address of the tape instruction, which specifies in regular three-character form the high-order character position of the first character. Note carefully that characters in a block are read from successively *higher* numbered locations.

3. What is the length of the block to be written? This question is answered by placing in core storage, after the last character of the block, a special symbol called a *group mark* with a word mark. The group mark consists of all ones in the zone and numerical portions of the character. When a group mark with word mark is detected in core storage during writing, the writing operation stops without having written the group mark with word mark on tape.

The actions on reading a tape are very similar, except that the operation is ended in a slightly different way. The operation code is M, the A-address is %Ux where x specifies a tape number, the B-address is the address of the first position into which a character should be read from the tape block, and the d-character is R. The operation is stopped by the occurrence of either of two things. If the interrecord gap is sensed in reading the tape, then a group mark is inserted in core storage following the last character of the block and the operation is stopped. If, on the other hand, a group mark with word mark is sensed in storage before reaching the end of the tape block, then the transmission of characters is stopped immediately, although the tape does move past any remaining characters on the tape until it reaches an interrecord gap.

Five somewhat related instructions for controlling the action of the tape unit without transmitting information complete the repertoire of tape instructions. All of the five instructions have the actual operation code U and the mnemonic CU, for Control Unit. They are distinguished by their d-characters.

The Back Space Tape instruction (d-character: B) causes the tape unit specified in the A-address to move backwards over one tape block. The first interrecord gap encountered in the backward direction stops the operation.

The Write Tape Mark (d-character: M) causes a tape mark to be recorded immediately following the last block on tape to indicate whatever the programmer wants it to indicate. Most commonly it denotes the end of valid information on the tape and/or that the physical end of the tape is about

Write Tape

FORMAT

Mnemonic	Op Code	A-address	B-address	d-character
MCW	<u>M</u>	%Ux	xxx	W

FUNCTION The tape unit designated in the A-address is started. The d-character specifies a tape write operation. The data from core storage is written in the tape record. The B-address specifies the high-order position of the record in storage. A group mark with a word mark in core storage stops the operation. The group mark with a word mark causes an interrecord gap to be created.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)ms + T_M$

Read Tape

FORMAT

Mnemonic	Op Code	A-address	B-address	d-character
MCW	<u>M</u>	%Ux	xxx	R

FUNCTION The tape unit specified in the A-address is started. The d-character specifies a tape read operation. The B-address specifies the high-order position of the tape read-in area of storage. The machine begins to read magnetic tape, and continues to read until either an interrecord gap in the tape record or a group mark with a word mark in core storage is sensed. The interrecord gap indicates the end of the tape record and a group mark (code CBA 8421) is inserted in 1401 core storage at this point.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)ms + T_M$. Time varies for type of tape unit and tape density used (see Figure 8.2).

to be reached. It is, therefore, sometimes called the end-of-reel indicator, but it can be used for several other purposes. The tape mark has zone bits of 00 and numerical bits of 1111. When a tape mark is later encountered in reading the tape, the end-of-reel indicator in the computer is turned on; it may be tested with a Branch If Indicator On instruction. It should be carefully noted that the Write Tape Mark instruction creates a separate block preceded by an interrecord gap. Thus, when the tape is read, the tape mark is *not* detected by reading the last block before the tape mark. It is detected only by reading the block that contains the tape mark. Therefore, after every Read Tape instruction there should ordinarily be a Branch If Indicator On instruction to determine whether data was read or the tape mark encountered. It should also be noted carefully that there is only the one indicator for this purpose in the entire system. Therefore, the Branch If Indicator On instruction will always test whether a tape mark was detected

on the tape most recently read. It is not possible to read from two tape units and then use a Branch If Indicator On instruction to determine whether there was a tape mark on the first one. The indicator is turned off by selecting a new tape unit or by testing it.

The Skip and Blank Tape instruction (d-character: E) is used to erase about $3\frac{3}{4}$ in. of tape. It is used when repeated attempts to write on an area of tape have shown that a readable tape record cannot be written there. By erasing the bad area of tape, we get the effect of an unusually long interrecord gap. The idea is that the bad section of tape may be limited to one small area and that tape farther along may be usable.

The Rewind Tape instruction (d-character: R) causes the selected tape unit to rewind its tape to the load point. If there is less than about 400 ft of tape to be rewound, the tape simply moves backward past the heads. In the 729 units, if there is more than about 400 ft, the heads are raised, the

Backspace Tape

FORMAT

Mnemonic	Op Code	A-address	d-character
CU	<u>U</u>	%Ux	B

FUNCTION The tape unit specified in the A-address backspaces over one tape record. The first interrecord gap encountered stops the operation.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)ms + T_M$

Write Tape Mark

FORMAT

Mnemonic	Op Code	A-address	d-character
CU	<u>U</u>	%Ux	M

FUNCTION This instruction causes a special character (8421) to be recorded immediately following the last record on tape to indicate an end-of-reel condition. When the tape mark is read back from a tape, the end-of-reel indicator is turned on. This signals the 1401 program that the end of the utilized tape has been reached.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)ms + T_M$

tape is lifted out of the vacuum columns, and the rewinding is done at a much higher speed. The total time to rewind the tape does not exceed the figure shown in Figure 8.2 for each tape unit, regardless of how much tape there is on the takeup

reel: the more tape there is, the faster the high speed rewind goes. Processing may continue during the rewinding.

The Rewind Tape and Unload (d-character: U) performs the same functions as the Rewind, but in

Skip and Blank Tape

FORMAT

Mnemonic	Op Code	A-address	d-character
CU	U	%Ux	E

FUNCTION The tape unit designated by the A-address spaces forward and erases $3\frac{3}{4}$ inches of tape. The actual skip occurs when the next Write Tape instruction is executed.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1) \text{ms}$. Processing can continue immediately after this operation. However, 47 ms for IBM 729 II and 27 ms for IBM 729 IV must be added to the next Write Tape instruction time.

Rewind Tape

FORMAT

Mnemonic	Op Code	A-address	d-character
CU	<u>U</u>	%Ux	R

FUNCTION The tape unit designated by the A-address is rewound to its load point.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1) \text{ms} + 10 \text{ms}$. Rewind time is 1.2 min per 2400-ft reel for the IBM 729 II, 0.9 min for the IBM 729 IV, and either 13.3 or 2.2 min for the IBM 7330, but it is not calculated with program time. Processing can continue approximately 10 ms after this instruction is interpreted.

Rewind Tape and Unload

FORMAT

Mnemonic	Op Code	A-address	d-character
CU	U	%Ux	U

FUNCTION This instruction causes the tape unit specified in the A-address to rewind its tape. At the end of the rewind the tape is out of the vacuum columns and the reading mechanism is disengaged. The unit is effectively disconnected from the system and is not available again until the operator restores it to a ready status.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1) \text{ms} + 10 \text{ms}$.

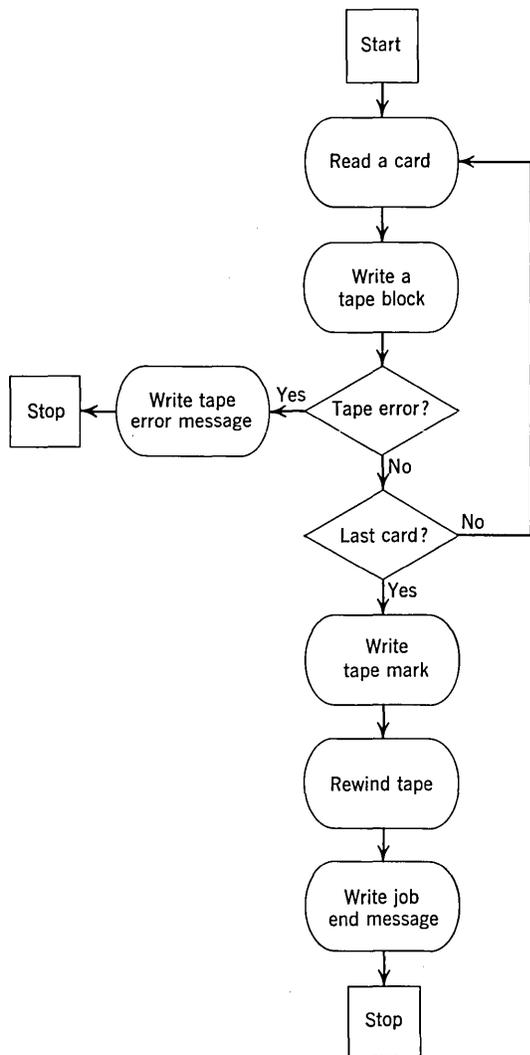


Figure 8.3. Block diagram of a program to write the information from a deck of cards onto magnetic tape.

addition it makes it impossible for the computer to use that tape unit until a switch on the tape unit is manually depressed. This is ordinarily used when a tape has been written that should be dismantled from the unit and a new tape mounted so that processing can continue with a new reel. Without this feature, there would be the constant danger that a recently written tape could be destroyed by writing new information on it, in the mistaken hope that the operator would have changed it by the time the new information was to be written.

For an elementary example of the use of some of these instructions, consider the first part of a job that involves putting the information from a deck of cards onto tape. The tape, once written, might

be used in later operations in the 1401 or it might be used as input to a larger computer. In order to illustrate the basic operations without unduly complicating the logic, we shall consider an extremely simplified and somewhat unrealistic approach. Each card is simply written onto the tape on tape unit 1, exactly as it is read into storage, with one card in each tape block.

The block diagram of this program is shown in Figure 8.3, and the program in Figure 8.4. We load a group mark with word mark as a constant into position 81, which is the next character position after the read area. At the beginning of the program we read a card and immediately write the information onto tape 1. Next, a test is made to see whether the information written was readable when it passed the read head after writing. If it was, we test to determine whether the card just read was the last one. If it was not the last card, we branch back to read another card. If the tape was not readable, we write on the printer a short message to the operator that we ran into difficulty and halt. In actual practice, using the Input/Output Control System, considerably more pains are taken to attempt to write the tape correctly and skip over the bad tape if it cannot be written in a few attempts. If the last-card test showed that the end of the deck had been reached, we write a tape mark, rewind the tape, print a message that the job is ended, and halt. This job-ended message is not too crucial here, since the operator would readily enough see that all the cards had been read, but it is good practice to be fairly free with such messages (or other signals) so that the operator need never be in doubt as to the status of the program. When appropriate, we also like to be able to write out messages specifying the action to take next.

This example is obviously vastly oversimplified. We shall see in Section 8.3 that the same general task can be done much more simply by using the Input/Output Control System; in spite of the simplification of the programming, the job will be done in a much more thorough fashion.

REVIEW QUESTIONS

1. How does the core storage addressing of information to be read from tape or written on tape differ from other data addressing on the 1401?
2. What are the two ways that tape reading may be stopped in the 1401?
3. When is a tape mark detected?

8.3 Tape Programming with Autocoder and IOCS

The effective use of magnetic tapes in a data processing system requires consideration of many factors. If every programmer had to take all these factors into account himself, and then write the detailed program properly to handle them, a great deal of time would be wasted: the same problems face anyone who ever writes a tape program. Fortunately, this is not the case; a standard tape programming system is available. This package, called the 1401 Input/Output Control System, or IOCS, handles all of the normal input and output programming considerations with a minimum of programmer effort.

IOCS is one of the major parts of an advanced coding language similar to SPS but considerably more powerful, called Autocoder. The basic ideas of Autocoder are generally the same as discussed in Chapter 4: symbolic addresses may be used in place of numerical addresses; mnemonic operation codes replace actual; the symbolic source program is translated into an actual object program by an assembly process. The major differences between SPS and Autocoder are these.

1. A *free-form* coding sheet is used, which means here that there are no fixed fields for the operands. Instead, the programmer uses as much space as

required for each operand and separates the operands by commas if there is more than one.

2. Augmented mnemonic operation codes are used. This relieves the programmer of writing the d-character in most instructions and, for instance, makes it unnecessary to write the %U in the A-address of a tape instruction. A complete list of Autocoder mnemonics appears in Appendix 2.

3. It is possible to use *literals*; that is, instead of writing the *address* of a constant in an instruction, we may write the constant *itself*. The Autocoder processor assigns a location to the constant and fills in the assigned address in the instruction.

4. *Macro-instructions* are provided. We are concerned here only with the IOCS macros, with which the programmer can specify in a very condensed form the tape operations that he wants to perform; the processor translates these into routines of dozens or hundreds of instructions. In short, *one* instruction in the source program is translated into *many* actual machine instructions; this characteristic makes Autocoder a *compiler* rather than an assembler as SPS is.

Autocoder also provides the programmer with the flexibility of making up macros for the purposes of his program. Thus some frequently used group of instructions can be called for in writing the source program simply by writing the macro, which is a much simpler matter than writing all the instruc-

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS			
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
010		START	R													READ CARD
020			M.C.W	%U,I						0001						WRITE ON TAPE
030			B	ERRHLT												TAPE ERROR Q
040			B	FINISH												ANØ. LAST CARD Q
050			B	START												NØ
060		FINISH	C.U	%U,I												WRITE TAPE MARK
070			C.U	%U,I												REWIND
080			L.C.A	M.S.S.G.1						0212						WRITE JOB
090			W													END MESSAGE
100			H	*						-003						
110		ERRHLT	L.C.A	M.S.S.G.2						0219						WRITE ERROR
120			W													HALT MESSAGE
130			H	*						-003						
140		Ø1	D.C.W	ØØØ1						£						GROUP MARK
150		12	M.S.S.G.1	D.C.W	*					JØB	FINISHED					
160		19	M.S.S.G.2	D.C.W	*					B.A.D	TAPE JOB HALTED					
170			E.N.D	START												
180																
190																

Figure 8.4. SPS program to write the information from a deck of cards onto magnetic tape.

tions themselves. Setting up a new macro is not appreciably more difficult than writing the detailed instructions to do the processing once; after that, all similar operations can be handled merely by writing the macro.

Autocoder represents a significant advance in programming sophistication over SPS. We shall not attempt to give a complete description of all the features of the language. A summary of the IOCS macros follows, and some of the other features are illustrated in the programs.

The IOCS macros are of three types. The first, DIOCS (for Define IOCS), is used by the programmer to define the machine configuration on which the object program will run, along with certain general information about the files and their processing. The second type, DTF (for Define Tape File), is used to describe in detail each of the files in a problem. The DIOCS and DTF are called *declarative* macros: they provide information to the Autocoder processor but do not result in any action in the object program.

The next four macros, on the other hand, do cause action in the object program and are therefore called *imperative* macros. The OPEN, CLOSE, GET, and PUT macros lead to the creation of object program instructions that actually carry out the desired processing of file information.

In order to see how these macros operate and to illustrate their use, we must investigate some of the considerations that affect tape programming.

Record blocking. It is usually quite inefficient of tape space, and therefore of time, to make tape blocks as short as they would be with normal tape records. Thus in the program of Figure 8.4, if we were using a 729 tape at high density, the card characters would take up only about 20 per cent of the total tape space. All the rest would be inter-record gaps. Therefore, it is common practice to write a number of what might be called "problem records" or "logical records" in one tape block. This is called *tape blocking*; the number of logical records in one tape block is called the *blocking factor*. In the special case where each block consists of one logical record, as in the preceding example, we speak of unblocked records or of a blocking factor of 1.

Tape blocking is a virtual necessity if the computer system is to be used effectively, but it does create certain programming problems. When a tape block is read, several problem records are

brought into core storage. The processing instructions must be arranged to pick up the records in succession from the storage area into which the block was read. This *deblocking* can be handled by moving the records from the block input area to a *working storage* area as needed or by using an index register to select the records in succession from the block input area.

In writing, the records must be assembled, or blocked, in the output area and then written when a complete block has been assembled. Once again, this can be done either with a working storage area or with an index register.

Variable versus fixed length records. It fairly often happens that the amount of information in a tape record varies greatly from one record to another, typically because a few of the file items require additional data not needed for the bulk of the file. For instance, a typical master record in an electric utility billing system contains a minimum of 200 characters, an average of 300, and a maximum of 600. Most customers have only one meter, but some have two; bills are generally sent to the same address as the meter address—but not always; if a customer has two meters, they are generally at the same address—but not always. It is clear that if the master records were set up to contain the maximum information that could ever be necessary—the simplest approach—a great deal of tape space would be wasted in the large majority of the records, which require only half as much information. The same sort of thing happens in many other types of applications.

A better choice is to let the length of the records vary according to the amount of information that must be recorded. Now, some means must be provided to indicate the length of each record; this is quite easily handled by placing in each record a number that specifies the total number of characters in the block.

Block counts. It is often very useful to know precisely how many blocks there are on a tape. This block count can readily be generated as the tape is written and recorded as part of a separate block at the end of the tape. (This is the *trailer label* described below.)

Record counts. It is also frequently useful to know how many records there are on the tape. This is usually not just the product of the number of blocks and the blocking factor because the num-

ber of records in the file may not be a multiple of the blocking factor. This count can also be generated by the program and written in the trailer block.

Control totals. To provide a check on the accuracy of programming and of machine operation, it is valuable to have in the trailer block the sum of some field in each record on the tape. This might be, for instance, the sum of the dollar amounts in all records. Such a control total can be accumulated as the tape is written; when the same tape is later processed, the control total can be developed again and compared with the control total in the trailer. This gives a fairly strong assurance that all records were processed. (Failure to process a record or two under unusual circumstances is a surprisingly common programming error.)

The field summed actually need not have any meaning as a number by itself, as a dollar total ordinarily does. Forming the sum of all the account numbers, or all the city codes, or almost anything else, gives just about the same degree of checking. When a control total has no meaning in itself, it is called a *hash total*.

Tape labels. Many computer installations have hundreds or even thousands of reels of tape, making it crucial that there be no mixups in tape identification. Running a major job with the wrong input tapes or writing over a tape that should not have been reused can be a minor catastrophe. Unfortunately, such mixups can happen all too easily, making it most desirable to have some sort of identification recorded on the tape itself, in addition to the paper label attached to the reel. This is the function of the *header label*. A normal header label contains the file name, a reel number, a reel sequence number within the file (if there is more than one reel to the file), the date of creation of the tape, and the retention cycle. These last two items serve important purposes: they prevent a tape from being reused when the information on it is still needed, and they prevent the information on a tape from being used after it is outdated.

A program to use labeled tapes must obviously provide for the creation of labels on new output tapes, for checking the labels of input tapes to determine that the right data is being used, and for checking labels of output tapes to be sure that valid data is not being destroyed. Label creation and label checking are two of the many functions provided by IOCS, with next to no effort required of the programmer.

Restart. It occasionally happens that a job must be stopped when it is partly completed. This can happen as a result of machine trouble, operator error, or because a higher priority job must be run. When the job is restarted, a number of problems arise. How far along was the job when it was stopped? What was in core storage? Which tapes were mounted on the tape units? Where was each tape positioned? The general idea of a restart procedure is to provide the answers to these questions at a number of points in the running of the job; these are called *checkpoints*. Anytime the job must be stopped, it is necessary only to return to the most recent checkpoint and start from there rather than going back to the beginning of the job.

Routines must be provided that will take care of all the problems of establishing checkpoints and restarting a job that was stopped before it was completed. At the completion of processing of every tape, and at any other point the programmer wishes to specify, the entire contents of core storage must be dumped onto a separate tape; this establishes where the program was at the time of the checkpoint. This dump must also contain the identification of every tape then mounted and block counters that tell the position of each tape. If it is necessary to restart, the special routines can be called into operation. They will print out instructions to the operator as to what tapes to mount; they will position each tape at the point where it was at the time of the selected checkpoint and call back into storage the exact storage contents at the time of the checkpoint. The program can now continue just as if nothing had happened.

Let us now return to the consideration of the IOCS package itself. As noted above, the programmer writes macro-instructions in his source program. The first of these is the Define Tape File (DTF) macro, of which there must be one for each tape file. We are not concerned here with the details of writing the DTF macros, which, although presenting no conceptual difficulties, would take too much space to describe completely. Therefore, the examples below are somewhat "schematic," in that we summarize the information that would have to be in the DTF macro without actually displaying the form in which it would be written.

The DTF macros *define* the files; the following macros call for action upon them.

OPEN. Before the processing of a file can be started, the file must be initialized by the use of the macro-instruction **OPEN**. This macro may

have any symbol in the label field. It has the code OPEN in the operation code field and the name of the file in the operand field. The name must be the same as that used in the DTF macro.

The OPEN macro-instruction performs the following operations on the file when the object program is run:

1. The file is made available for processing.
2. The tape is rewound, if desired.
3. The tape label is processed if the DTF indicates that the tape is labeled. For input files, the OPEN macro reads and checks the header tape label; for output files, OPEN checks the retention code of the mounted reel and writes a new label if the code indicates that the information is no longer valid.

The operations performed for the first reel of a multireel file are performed automatically for each succeeding reel within the file. The checks are made as the end of one reel is reached and before the use of records from the next reel. This is done as an automatic part of the GET and PUT macros; the programmer need write only one OPEN for all reels of a multireel file.

CLOSE. When a tape file is no longer needed, it is removed from use by executing the macro-instruction CLOSE. Like the other macros, this one may have a symbolic label; the operand field contains the names of the files being closed, with the names separated by commas if there is more than one. The following operations are performed on output files:

1. Any records still in the output area are written on tape, which takes care of partly filled blocks. The routine then writes a tape mark, followed by the trailer label, followed by another tape mark.
2. The tape is rewound, if desired.
3. The file is made unavailable for processing.

For input files, the operations are the following:

1. The trailer label block and/or record counts are checked if this action has been specified in the DTF macro.
2. The tape is rewound.

GET. This macro performs all the operations required to obtain another record and make it ready for processing. The programmer is thus relieved of the hours or days of programming required to accomplish all of the following:

1. If all the records in the preceding block have been processed, another tape block is read.

2. If all the records have not been used, a new record is made available.

3. If a tape error is detected in reading, the routine backspaces the tape and reads again. If the difficulty is nothing more serious than a speck of dust, which is often the case, the backspacing and rereading will often dislodge it and the second reading will be correct. However, if the tape is still not readable after several attempts to reread it, the routine branches to an error routine supplied by the programmer, which takes whatever action the programmer has decided should be taken in such a situation.

4. If the end-of-reel condition is detected in reading, the trailer label block and/or record counts are checked and a character in the trailer label is inspected to determine whether another reel of the same file follows. If the DTF macro specifies special routines for the end-of-reel and end-of-file conditions, a branch is made. If not, the tape is rewound and the tape on the alternate unit for this file is prepared for use.

The GET macro, which as usual may have a symbolic label, specifies in the operand field the file from which a record is to be obtained. All of the foregoing operations follow automatically (as far as the programmer is concerned).

PUT. This macro is analogous to GET, except that it refers to output files. It performs the following operations:

1. A record from an input area (or from a working storage area) is moved to an output area. If this record fills the output area, the block is written on tape.
2. If an error is detected in the writing, the tape is backspaced and rewritten. If the record is still bad, a section of tape is erased and the record is written again. If an extended section of tape is bad, the routine branches to the programmer's error routine.
3. If the end-of-reel reflective spot is detected during writing, the trailer label is written (with an indication that another reel follows), the reel is rewound, and another reel is used for further writing with this file.

It is realized that a quick sketch of this sort does not give the reader enough information to begin writing useful programs with IOCS. It is hoped, however, that if the general idea of how the system can be used has been grasped then the reader will have no particular difficulty in picking up the de-

tails. In order to get a little better feel for the use of the system, we may consider some examples.

Let us first rewrite the illustrative program in Section 8.2, with blocking of the tape. The block diagram of Figure 8.5 is now considerably simpler, even though a great deal more is being done. The OPEN box, for instance, takes care of all label checking. The PUT box takes care of all tape writing, blocking, and checking. The fact that the output tape is blocked would be specified in the DTF for this file, which we are not showing. Except for the DTF, all we have to do to handle blocking is to set up an output area large enough to hold a complete block. The CLOSE box takes care of writing a trailer label, writing tape marks, and rewinding the tape.

The Autocoder program shown in Figure 8.6 is not especially difficult either, although the new coding form makes it appear a little strange. Notice the free form in which the operands are written. About the only restriction in writing the operands is that there be not more than one blank space within the operand field, since two consecutive blanks indicate the end of the field. The remarks may begin anywhere after two blanks; it is common practice to start all remarks in the same column for ease of reading. Note also that the operation field is now five columns instead of three to allow for the augmented operation codes and the macro-operation codes.

This program uses two new Autocoder pseudo-

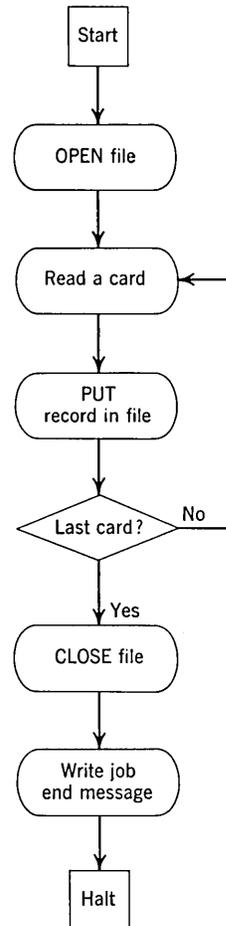


Figure 8.5. Block diagram of the Autocoder operations to do the job diagrammed in Figure 8.3.

Line	Label	Operation	OPERAND
0.1	START	OPEN OUTPUT	
0.2	READ	R	READ A CARD
0.3		PUT	CARD, OUTPUT
0.4		BLC	LASTCD
0.5		B	READ
0.6	LASTCD	CLOSE OUTPUT	
0.7		LCA	MESSG1, 2, 2
0.8		W	JOB END MESSAGE
0.9		H	*-3
1.0	MESSG1	DCW	@JOB FINISHED@
1.1	OUTPUT	DA	3X80, XI, 6
1.2	CARD	EQU	1
1.3		END	START
1.4			

Figure 8.6. Autocoder program to do the job diagrammed in Figure 8.5.

operations. The first is Define Area, for which the operation code is DA. It is used to set up an area of storage that can be referred to in the PUT macro. The 3 specifies that three groups of 80 characters are being set up for this area; the X1 specifies that index 1 is to be used by the object program in stepping through the records in the block; the G specifies that a group mark with a word mark is to be set in the character position to the right of the 240-position area. We shall see in Section 8.4 how the DA instruction can also be used to define fields within the area.

The second new pseudo-instruction is Equate, for which the operation code is EQU. It is used here to establish 0001 as the absolute equivalent of the symbol CARD. This would ordinarily be done with the DA operation, but it is always illegal to use a macro of this type to set up an area or a constant in the card read area, since this will ordinarily disturb program loading operations.

Note that there is no "count" field on the Autocoder form. Numerical constants are entered into the program with as many character positions as there are digits in the constant. Alphameric constants must be preceded and followed by the symbol @.

REVIEW QUESTIONS

1. Distinguish between a record count, control total, and hash total. List advantages and disadvantages of each, in terms of such considerations as simplicity of computation, degree of checking provided, and types of errors checked against.
2. Why is record blocking used?
3. It would seem that as many records as possible should be placed in a block—that is, that the blocking factor should be as large as possible. What sets a limit on the degree of blocking?
4. List the functions of a header label.
5. What does the programmer do to incorporate the desired IOCS routines in his program?
6. What are the principal differences between SPS and Autocoder?

8.4 Inventory Control Case Study

The following case study illustrates the use of many of the techniques and concepts that have been discussed in this section.

Inventory control, as used here, refers to the process of keeping an up-to-date record of the status of every part in the inventory of a manu-

facturing company. In the example to be studied here we are given a master inventory tape containing a record for each stock item. Each record contains the part number and the quantity on hand. The object of the inventory control application is to maintain this file so that it represents the status of the actual inventory as of the most recent updating of the file.

Changes in stock status are introduced into the data processing system in the form of a deck of cards. Each card shows the part number, a code to indicate whether the card represents a receipt of more stock items, a recount, or an issue to the manufacturing operation or to a customer. There can be more than one card per part number, such as when a shipment of stock items has been added to the inventory and a shipment has been issued to a customer, during the period represented by the update. Again, there could very well have been several shipments to customers during the period. Before the transaction deck reaches the computer, it has been sorted on part number and classification code in such a way that for any one part number recounts are first, then receipts, and then issues. This sequencing of the cards within one part number will guarantee that even though a large shipment is received and another large shipment issued during the transaction period, for instance, the data processing system will not erroneously indicate that the large issue created an out-of-stock condition.

Recounts are coded to sort at the front of the transactions. This is done on the assumption that the recount quantity is taken *before* any transactions. Another common way to handle adjustments is to enter them as changes, either plus or minus, rather than entering a complete new count as we have done here.

The master file is in ascending sequence on part number.

Our job is to use the transaction deck and the old master file to produce a new master file that shows the inventory status after the changes described by the transaction file.

A block diagram of the computer operations, including card reading and tape handling, is shown in Figure 8.7. The actions called for by this block diagram are best understood by considering several situations and seeing what the block diagram says to do for each. Suppose first that there is a single transaction card for the first item in the master file. We begin by setting a switch to what is called

the ON position. We read this first card and GET the first master record. We next ask if the part numbers in the transaction and the master are the same. They are, by assumption, so we use the transaction code to determine whether this is a receipt, receipt, or issue, and update the master record accordingly. We ask if this was the last card. The

answer is no, so we read another card and return to the comparison to determine whether the part numbers are equal. We assumed a one-card group for the first stock item, so the comparison this time will show that the master part number is less than the transaction part number. This will cause the updated master record to be PUT into the output

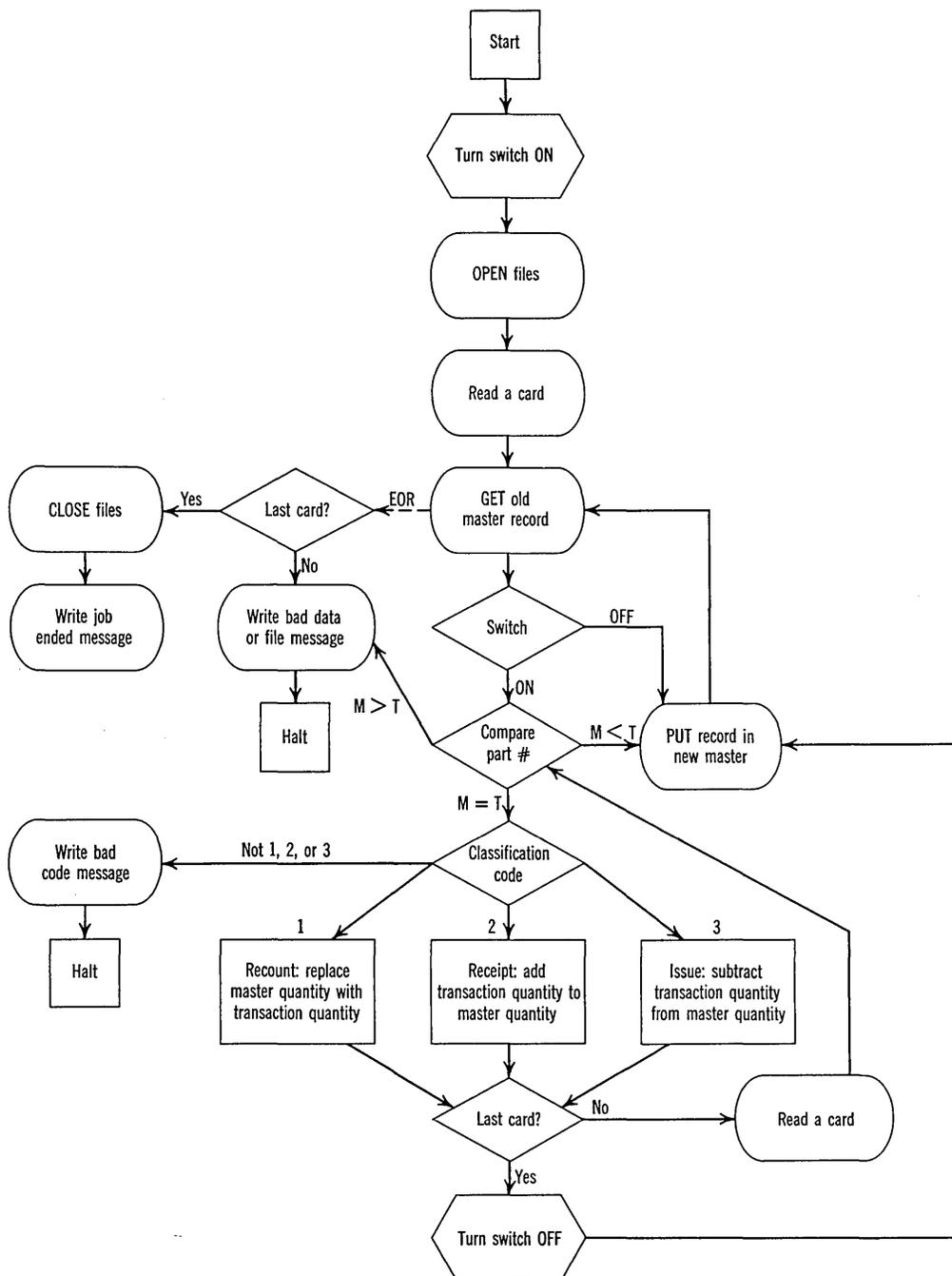


Figure 8.7. Block diagram of an inventory control procedure.

area, after which we return to GET another master record.

If there had been several cards for the first part number in the file, they would have been processed, without writing the new master record, by the repeated use of the loop containing the classification code test.

If the first transaction card is for some master file record other than the first, then all of the records prior to this one will be PUT into the output area before the part number comparison shows equal.

If at any time the part number of a master record turns out to be greater than the transaction part number, then an error is indicated. This could be caused either by an out-of-sequence file or by an incorrect transaction part number. It could happen, for instance, if the first transaction card had a part number smaller than that of the first master record. It is clear that in this case reading more master records is never going to find a matching part number, since both files are assumed to be in ascending sequence on part number. It could also happen if a transaction card had a part number that was not the same as the part number of any record in the master file. This test does not give an absolute guaranty that no transaction part numbers are incorrect but only that if an incorrect part number does not match we will catch it.

Suppose now that the last transaction card is a single-card group corresponding to the last master record. This time the last-card test will give a yes answer, which causes the switch to be set to the OFF position, after which we write this last new master and go back and try to GET another master record. This will be found instead to be the tape mark, and the GET routine will take us out of the processing loop. We shall see why another last card test is necessary in a moment.

If the last master record has corresponding to it several transaction records, then the analysis is the same as in the preceding paragraph, except that we will go around the updating loop the necessary times before finding the last card.

Suppose next that the last transaction card corresponds to a master record before the end of the tape. In this situation we must still copy the remaining master records in unchanged form from the old master to the new master. This is why the switch is necessary. After the master record corresponding to the last record in the transaction file has been PUT, then all following master records

are going to show larger than the final part number. But now this is not an error. Therefore, when the last card is detected, we set a switch to bypass the comparison and simply read and copy master records until the tape mark is sensed and the GET routine takes us out.

One final error possibility remains. Some transaction card might have an incorrect part number greater than that of the last master record. In this case we would go around the master record reading and writing loop looking for a matching master part number. Naturally, we would never find it but would instead finally detect the tape mark. Now, when we ask whether the card most recently read was the last one, the answer will be no, indicating that the end of the master file was reached without having reached the end of the transaction deck. This is, of course, an error and we so indicate.

This is a very standard type of block diagram in sequential file data processing. Its basic logic applies to many applications besides inventory control and it applies when both files are on tape. Even the reader who may not be directly concerned with magnetic tapes will profit by a careful study of the logic of this block diagram.

We may now consider the details of the actual program for this example. First assume that the transaction cards have a five-character part number in columns 1 to 5, a single-digit classification code in 6, and a four-digit quantity in 7 to 10. The classification code is a 1 if the transaction represents a recount quantity that should replace the master record quantity, a 2 if it represents a receipt, and a 3 if it represents an issue. The master records each consist of a five-character part number and a five-digit quantity. The master tape is blocked with 20 records in each block. If the last block is not full, it is padded out with blanks.

The program for this job is shown in Figure 8.8. We begin by opening the two files and reading a card. OLDMST and NEWMST are the names of the two files; these names would have to be in the DTF macros for the two files, which we are not showing. The DTF macros would also specify, among other things, the blocking factors for the two files and would name EOR as the routine to which the GET routine should branch when the end-of-reel is detected in reading the old master.

Next, an old master is placed in the work area. The switch is coded as a No Operation, so that it will have no effect until it is changed to a Branch.

After this we compare the part number of the transaction from the card, with the part number from the old master. The next two instructions branch out to the appropriate routines for the cases where the two are not equal. If they are the same, we reach the three branch instructions that determine whether the transaction is a recount, receipt or issue.

This testing of the classification code uses a new

variation of the Branch instruction, called Branch If Character Equal. This instruction says to branch to the instruction shown in the I-address if the single character at the B-address is the same as the d-character. If it is not, we go on in sequence. This is used here because we do not like to assume that the classification code will always be 1, 2, or 3. This is placing a little too much reliance on fallible human beings. Therefore, we

IBM FORM X24-1350-1
PRINTED IN U.S.A.

Program _____
 Programmed by _____
 Date _____

1401/1410 AUTOCODER CODING SHEET

Identification _____
 Page No. 1 of 3

Line	Label	Operation	OPERAND
5	6	15 16 20 21	25 30 35 40 45 50 55 60 65 70
0.1	START	OPEN	OLDMST, NEWMST
0.2		R	READ A CARD
0.3	READM	GET	OLDMST
0.4	SWITCH	NOP	B
0.5	COMP	C	TRANPN, MSTPN
0.6		BH	ERROR
0.7		BL	B
0.8		BCE	RECOUN, CODE, 1
0.9		BCE	RECPT, CODE, 2
1.0		BCE	ISSUE, CODE, 3
1.1		LCA	MESSG1, 225
1.2		W	
1.3		H	*-3
1.4	RECOUN	MCW	TRANQY, MSTQY
1.5		B	LCTEST
1.6	RECPT	A	TRANQY, MSTQY
1.7		B	LCTEST
1.8	ISSUE	S	TRANQY, MSTQY
1.9	LCTEST	BLC	SWSET
2.0		R	COMP
2.1	SWSET	MCW	SWBR, SWITCH
2.2			

Line	Label	Operation	OPERAND
5	6	15 16 20 21	25 30 35 40 45 50 55 60 65 70
0.1	B	PUT	OLDMST, NEWMST
0.2		B	READM
0.3	EOR	BLC	WRAPUP
0.4		LCA	MESSG2, 229
0.5		W	
0.6		H	*-3
0.7	WRAPUP	CLOSE	OLDMST, NEWMST
0.8		LCA	MESSG3, 210
0.9		W	
1.0		H	*-3
1.1		H	
1.2			

Figure 8.8. Autocoder program for the inventory control procedure diagrammed in Figure 8.7.

test explicitly against all three codes, branching to the correct instruction when one of them is found. If the code turns out to be none of the three, as it could through mispunching, then we write an error message and halt.

If the transaction is a recount, we replace the master quantity with the transaction quantity. If it is a receipt, we add the transaction quantity to the master quantity, and if it was an issue we subtract the transaction quantity from the master.

Setting the switch is a simple matter of moving a B to the operation code, thus changing it from a No Operation to a Branch.

The Define Area instructions perform a new function for us in this program. Notice that following each DA there are a few lines with no operation code. These are part of the DA, defining fields within it. The first location of the defined area is considered location 1. The high-order and low-order positions of the fields are punched, beginning with the leftmost column of the operand field. These two numbers are separated by a comma. The processor places a word mark over the leftmost location of each field defined in this way. The fields within the defined areas may now be referred to symbolically, without, of course, our knowing where they are located in storage. Both tapes have blocking factors of 20, which requires setting up sufficient space in the input and output areas for 200 characters.

The wrap-up for this job consists simply of closing both files and writing the "job finished" message to the operator.

The case study has been deliberately simplified to let us concentrate on the tape operations involved. It should be realized that a normal in-

ventory control task includes a great deal more than we have shown here. Furthermore, we have taken as straightforward an approach to the tape manipulation as possible in order to keep this first sample of work with blocked tapes as uncomplicated as possible. Chapter 10 is devoted entirely to a thorough study of a more nearly complete inventory control application, both from an application standpoint and from a programming standpoint.

REVIEW QUESTIONS

1. In the program of this section, when would the result of the last-card test following the tape mark test ever be yes?
2. Suppose that the last card of the transaction deck were inadvertently placed at the front of the deck. What would the program do?
3. What would the program as shown do if the old master file erroneously had two records with the same part number?
4. Why is it more convenient to have two Read a Card instructions than to have only one?

EXERCISES

- *1. Given a tape with unblocked records of 100 characters each, with a tape mark following the last record. Draw a block diagram and write an SPS program to print each record exactly as it appears on tape. Each page of the output is to have a maximum of 50 lines.
2. Given a tape with unblocked records of ten characters each, consisting of a four-digit account number and a six-digit dollar amount. The records are in ascending sequence on account number; there are duplicate account numbers. At the end of the tape there is a tape mark, followed by a trailer label, followed by another tape mark. Positions 6 to 10 of the trailer contain a count of the number of blocks in the tape.

Line	Label	Operation	OPERAND
3 5 6	15 16	20 21	25 30 35 40 45 50 55 60 65 70
0.1	TRANPM	EQU	5
0.2	CODE	EQU	6
0.3	TRANQY	EQU	10
0.4	OLDMST	DA	20X10, X1, G
0.5	MSTPN		1, 5
0.6	MSTQY		6, 10
0.7	NEWMST	DA	20X10, X2, G
0.8	MESSG1	DCW	@BAD CLASS CODE JOB HALTED@
0.9	MESSG2	DCW	@FILE OR DATA ERROR JOB HALTED@
1.0	MESSG3	DCW	@JOB FINISHED@
1.1	SWBR	DCW	@B@
1.2		END	START
1.3			

Figure 8.8 (Continued).

Draw a block diagram and write an SPS program to summarize the amounts by account number and make a block count check.

3. Estimate the time required to execute the program of Exercise 2, assuming that there are 10,000 blocks on the tape and that a 729 II tape at low density is used.

4. Draw a block diagram of the operations necessary to read a block, check for tape errors, and reread up to nine times if there is an error. If the error persists, an error message is to be written.

*5. Modify the block diagram and program of the inventory control case study as follows. Another type of transaction, with a code of zero, may be present in the deck: an addition. An addition record represents a new part number, the record for which is to be added to the master file. However, do not add the new record to the file if it has the same part number as a record already in the file. "Adding" the record to the file consists of getting the card record into the proper tape format and moving the part number and quantity to the output area.

6. Modify the block diagram and program of Exercise 5 as follows. A fifth type of transaction, with a code of 4, may be present in the deck: a deletion. A deletion record represents a part number the record of which is to be removed from the file, that is, not written into the new master. Write a short tape containing all deleted master records.

*7. Given two tapes having the same record format: a four-character employee number, followed by 90 characters of information about the employee. Both tapes are in ascending sequence on employee number, and there are no duplications. Draw a block diagram and write an Autocoder program to merge the two tapes—that is, produce a third tape containing in ascending sequence all records from the two input tapes.

8. Same as Exercise 7, except that a sequence check is to be made on both input tapes. This will require storage of the most recent employee number from each input tape to use in determining whether the record just read is greater than the preceding—for each tape.

9. RANDOM ACCESS FILE STORAGE

9.1 Basic Concepts

So far in this text we have concentrated on programs built around files on punched cards or magnetic tape. These file storage media have the advantage that they are relatively inexpensive. They have the partially offsetting disadvantage that a record within the file can be accessed only sequentially, since it is not possible to get to one record without passing over all of the records in front of it. This characteristic forces us to sequence all files and transactions according to the keys of the records, which leads to considerable amounts of time spent in sorting. It also forces us to batch the transactions to be processed against the file, since it is not economical to read the entire master file to process a few transactions.

In many applications these requirements of sequenced files and batch processing are not serious handicaps; in fact, batch processing is in many cases a natural mode of operation. In other applications, however, it would be much more desirable to be able to process transactions immediately as they arise, rather than waiting for batches of them to accumulate, and without sorting. To do so requires a master file storage medium that permits any record in the file to be obtained about as quickly as any other. (This is most definitely not true of magnetic tape.) Such a device is called a *random access file storage medium*.

It must be realized that the time to obtain a record from the currently available random access file storage devices does depend somewhat on its location relative to

the location of the record most recently read. Thus the devices are not *truly* random. However, the maximum time to obtain a record is so much less than the time to obtain a randomly placed record from a reel of magnetic tape that we are justified in using the word "random" in comparison with tape file storage. (The only *completely* random access storage device widely used at present is magnetic core storage; the time to obtain a word is absolutely independent of the location of the preceding word. In comparison with magnetic core storage, "random access" file storage media are decidedly *not* random, but this is not a relevant comparison, since the much more expensive core storage is restricted to smaller sizes than are needed for file storage.)

With any of the various random access file storage devices, it is possible and desirable to organize programs in entirely different ways from those employed with magnetic tape file storage. No sorting of the transactions is ordinarily required; transactions can be processed as soon as they reach the data processing center, if desired; the program can be organized to refer to many small files if it is convenient to do so. Furthermore, certain types of applications become feasible that are simply not practical with magnetic tapes.

9.2 The IBM 1405 and 1301 Disk Storage Units

The random access file storage devices available for IBM computers are built around

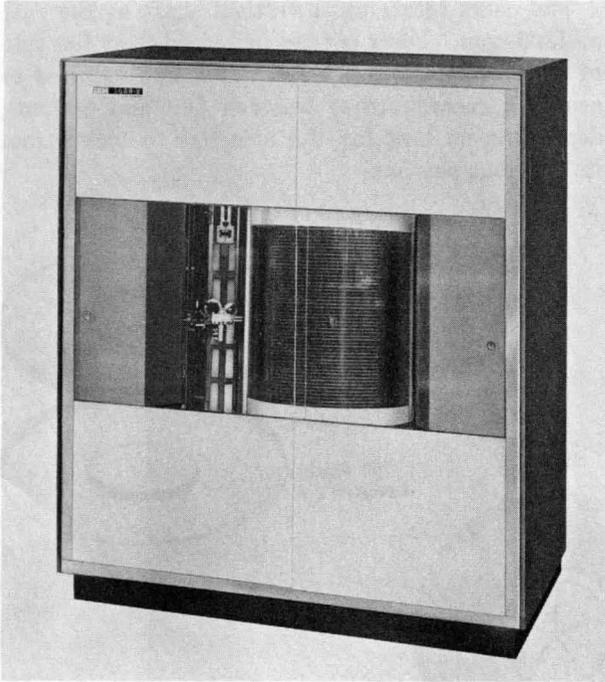


Figure 9.1. The IBM 1405 Disk Storage Unit.

a set of rotating metal disks on which information is recorded. In the IBM 1405, pictured in Figure 9.1, information is written or read by one or more *access arms* that are able to move to the desired disk and to the desired position on a disk. A 1401 system including the 1405 is called an IBM RAMAC® 1401 System, where RAMAC stands for Random Access Method of Accounting and Control. In the IBM 1301, pictured in Figure 9.2, information is written and read by a complete set of access arms, one for each disk surface.

The IBM 1405. The IBM 1405 Disk Storage Unit can contain 25 disks (Model 1) or 50 disks (Model 2), storing either 10 or 20 million characters. Each disk has 200 concentric *tracks* on which data can be recorded. A track has two sides, one on each surface of the disk. Each track is further divided into 10 *sectors*, five on each side: the upper side of each track contains sectors zero through 4, and the lower side sectors 5 through 9. A track sector, which contains 200 characters, is the smallest unit of disk information that can be addressed.

A 1405 unit normally has one access arm, with a second available as an optional special feature. The fork-shaped arm has two read-write heads

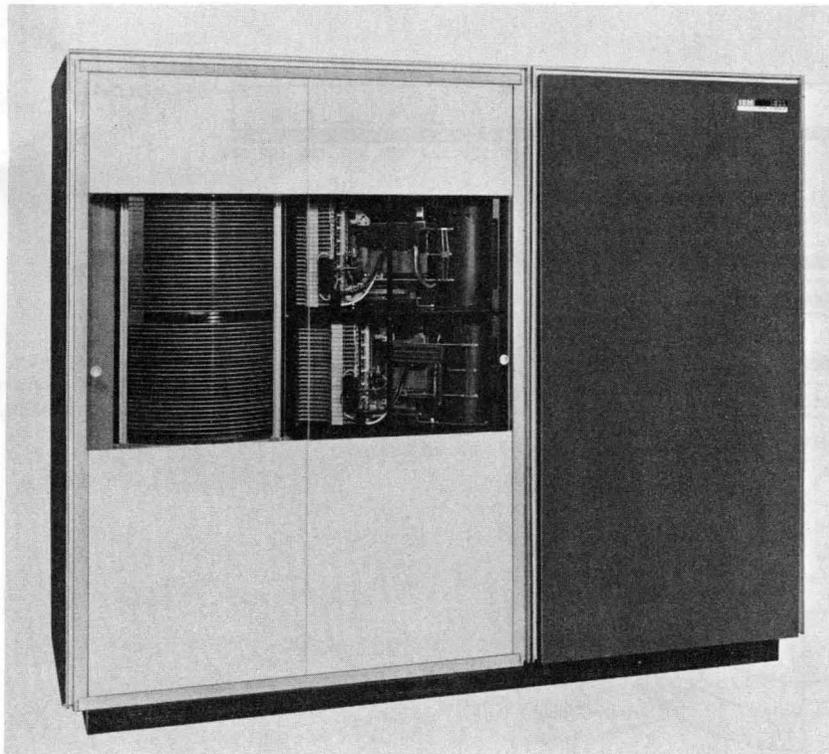


Figure 9.2. The IBM 1301 Disk Storage Unit.

that read and record data on the disks. One read-write head is for the top disk face and the other is for the bottom disk face. An access arm moves to the position specified by an instruction by moving vertically to the correct disk and horizontally to the specified track on that disk.

The disks rotate on a vertical shaft at the rate of 1200 rpm. Data is read or recorded at the rate of 22,500 characters per sec. The time required to access a record varies between 100 and 800 ms, depending on how far the arm has to move from its previous position.

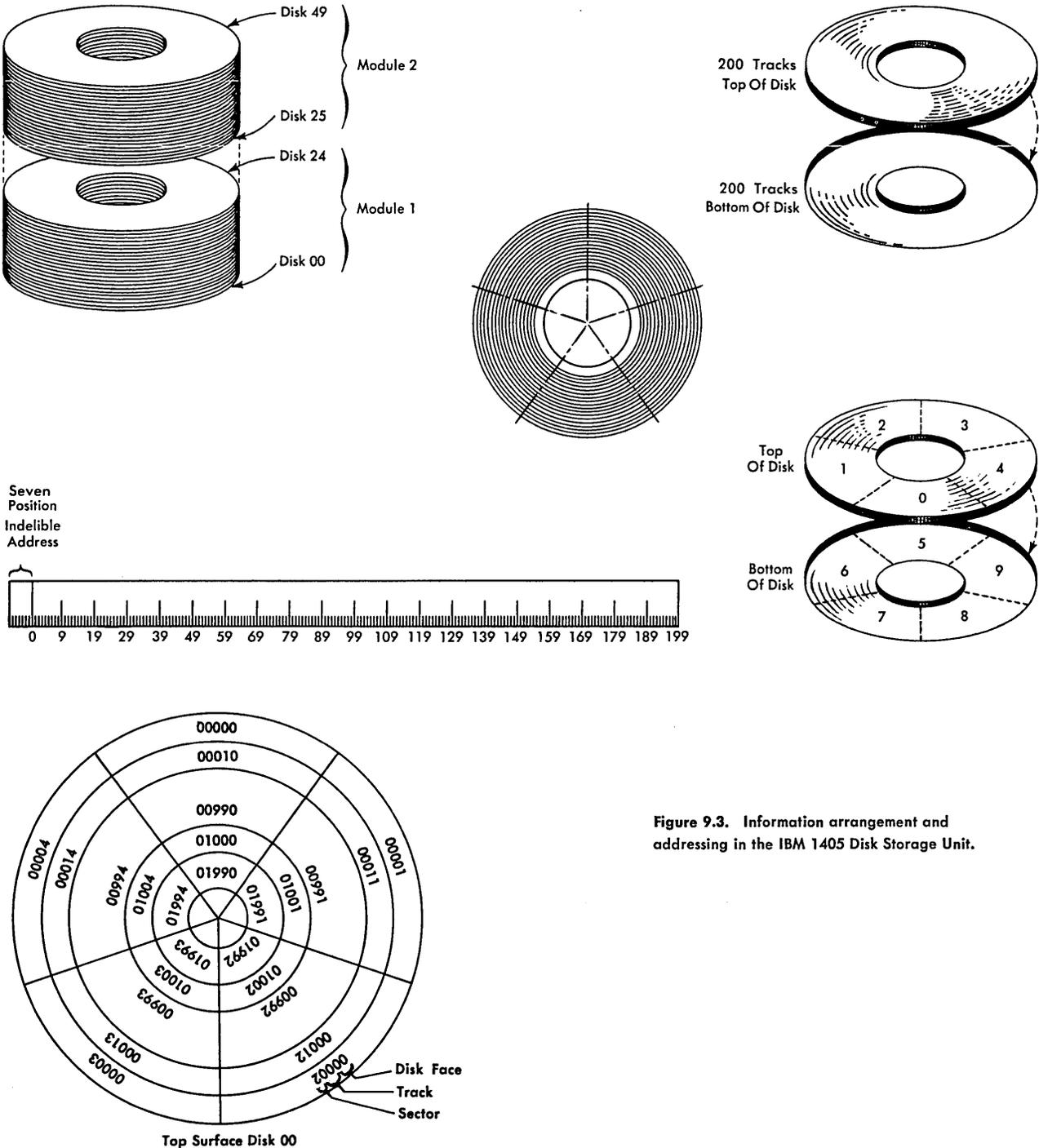


Figure 9.3. Information arrangement and addressing in the IBM 1405 Disk Storage Unit.

As with magnetic tapes, the representation of a character on a disk consists of seven bits, including a parity bit. When information is read from the disk, parity is checked and an indicator is turned on if parity is not correct. An instruction is provided with which it is possible to determine whether the information recorded on the disk is actually the same as the information in core storage from which the disk data was written.

Each sector on a disk has a seven-digit address. The first digit specifies the disk storage unit; for a 1401 system, this is always zero. The next four digits specify the track. A Model 1 unit contains 5000 tracks (0000–4999); a Model 2 unit contains 10,000 tracks (0000–9999). The outermost track of the bottom disk has the address 0000 and the innermost track of the bottom disk has the address 0199. The outermost track of the second disk has the address 0200 and the innermost track of the second disk has the address 0399. The tracks on the twenty-fifth disk have addresses running from 4800 at the outside to 4999 at the inside. The tracks on the fiftieth disk have addresses running from 9800 at the outside to 9999 at the inside. The sixth digit specifies the sector. The seventh digit of a disk storage address must always be zero. When a sector is addressed from the computer, it is preceded by a digit specifying the desired access arm; this digit is either zero or one. The arrangement and addressing of disk information is shown schematically in Figure 9.3. It may be noted from this figure that each record actually contains its address indelibly recorded in seven additional positions at the beginning of the record.

The IBM 1301. The IBM 1301 Disk Storage Unit is similar to the 1405, but in several particulars it is much more powerful. The major change is that instead of having one arm that travels to the specified disk, it has a complete set of arms, one for each disk, arranged in a comblike array. This means that once the arms have been positioned to a track location, that track on *all* disks is available with no further delay. It is convenient to think of all the tracks accessed from one setting of the arms as composing a *cylinder*.

Record length in the 1301 is flexible, instead of being restricted to 200 characters. This obviously simplifies working with records that are too large or too small to fit conveniently and efficiently in 200 characters.

Besides the radically different arm arrangement

and the flexible record length, the 1301 also has greater speed and capacity. The disks turn at 1800 rpm instead of 1200, which shortens all delays based on disk rotation by a third. This increase in speed of rotation, coupled with a greater character density on the disks, raises the transfer rate from 22,500 characters per second to 75,000. Finally, the capacity of a 1301 module is 25 million characters instead of the maximum of 20 million in a 1405 Model 2.

The unique powers of the 1301 lead to program organization and processing techniques that are sometimes markedly different from those used with the 1405. Since we shall not have space to cover the application of both systems, we limit the following discussions to the 1405, which is more widely distributed, at least at present.

REVIEW QUESTIONS

1. How many characters can be stored on one 1405 disk?
2. Which disk in a 1405 Unit contains sector 33862? How many disks must the access arm go past in moving from sector 08330 to sector 41045?
3. True or false: in the 1405 the greatest distance (both horizontally and vertically) that an arm can move is from track 00000 to track 99999.

9.3 Disk Storage Programming for the IBM 1405

There are five instructions in the 1401 that are used in working with disk storage.

The first of these is the Seek Disk instruction. Before any reading or writing can occur, the access arm must be moved to the desired track. This movement is initiated with a Seek Disk instruction. After the correct track has been located, a separate Read or Write instruction is used to move data.

As with the tape instructions, these instructions have an operation code of M, which is the same as that for Move Characters to a Word Mark. The fact that this is a disk instruction is specified by the first two characters of the A-address, which must be %F. The nature of the disk instruction is specified by the low-order character of the A-address and by the d-character, if there is one. In a Seek Disk instruction the A-address sign must be %F0 and there is no d-character. The B-address specifies the high-order position in core storage of the address of the desired sector.

The use of Autocoder considerably simplifies the writing of disk instructions, as it does tape instructions. The augmented operation codes make it unnecessary to write an A-address or the d-character. When the Autocoder processor translates the operation code SD for Seek Disk, for instance, it fills in the %F0 automatically.

It should be emphasized that the Seek Disk instruction does not move any data; it simply positions the access arm at the correct disk and correct track. The computer may continue processing with other instructions while the arm is in motion. If a disk Read or Write instruction is encountered before the arm motion is completed, the read or write operation is delayed until the access arm is in correct position.

Once the access arm is positioned at the correct track, a Read or Write Disk instruction can be used to transfer data. Reading is initiated with an instruction that once again has an operation code of M. The A-address must be either %F1 or %F2 and the d-character must be R for read. If the A-address is %F1, then a single 200-character record will be read. If the A-address is %F2, then we have specified what is called a full track read—that is, the specified record and the four following on the same side of the track will be read. The B-address specifies the high-order position in core storage of the disk record address, which might be thought to be redundant, for the disk position has already been established by the Seek instruction.

However, the presence of the address in the Read instruction allows the accuracy of the machine's functioning to be checked, since at the beginning of each sector the address of the sector is recorded in nonerasable form. Furthermore, the Read instruction is not able to "remember" from the Seek instruction which sector was specified.

The data from disk storage is read into core storage beginning immediately after the disk address, the position of which is specified by the B-address. In other words, the B-address specifies where in core storage the disk address is located, and the record is read into core storage immediately after the disk address. The transfer of information stops at the end of the sector or on encountering a group mark with a word mark in core storage. (Under normal conditions the program is organized so that the two occur at the same time.) The group mark should be one position to the right of the space reserved for the information from the disk. If it is encountered earlier than that, the transmission stops and an indicator called Wrong-Length Record is turned on.

Writing of information from core storage to disk storage is carried out with a Write Disk instruction, which is very similar to Read Disk, except that the d-character is W.

A Write Disk instruction must *always* be followed by a Write Disk Check instruction. This instruction causes a character-by-character comparison of data in core storage with the data just

Seek Disk

FORMAT

Mnemonic	Op Code	A-address	B-address
MCW	<u>M</u>	%F0	xxx

AUTOCODER FORMAT SD Address

FUNCTION The A-address specifies that a seek operation is to be performed by the access arm. The B-address specifies the high-order position in core storage of the disk record address. The selected access arm seeks the disk and track specified in the disk record address. Processing can continue while the access arm is in motion.

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 9) \text{ms} + \text{access time.}$

Note If the access arm is already at the disk track (not necessarily at the correct sector) that is to be used, a Seek Disk instruction is not needed.

written on the disk and turns on an error indicator if there are any differences. The actual machine instruction is just like a Write Disk instruction except that the A-address must be %F3 and there is no d-character.

In the instructions that we have described, word marks in core storage are not written on the disk and when disk data is read word marks are not affected in core storage. There is, as in the case of tapes, a separate instruction for converting core storage word marks into separate characters when the record is written on the disk. When such a record is read back with a suitable instruction, the special characters are again reconverted to word marks. Beyond this note of their existence we shall not consider instructions for reading and writing word marks.

The final type of disk instruction is simply a variation of one that we met much earlier, Branch If Indicator On. For use with disk storage, we have five additional indicators and corresponding d-characters. These are shown in the summary box for the instruction. "Any Disk Unit Error

Condition" comes on if any one of the first three is on. This makes it possible to make one check for any disk errors and, if the indicator is off, to proceed with the normal program. If the indicator is on, only then is it necessary to test the other three indicators to find out which of the errors is present.

For a simple illustration of the way these instructions may be used, we may consider the inventory control application of Section 8.4. We assume as before that the transaction cards contain a part number in columns 1 to 5, a code in column 6 indicating an adjustment, a receipt or an issue, and a quantity in columns 7 to 10. We assume that we have a 1405 Model 2, which has exactly 100,000 sectors in it, and that each master inventory record contains 200 characters. We assume that the part numbers are numerical and shall, therefore, be able to use the part number directly as the address of the corresponding master record. The master record is assumed to contain in positions 1 to 5 the part number and in positions 6 to 10 a quantity.

Read Disk Single-Record

Read Disk Full-Track

FORMAT

Mnemonic	Op Code	A-address	B-address	d-character
MCW	<u>M</u>	%Fx	xxx	R

AUTOCODER FORMAT Single-Record: RD Address
 Full-Track: RDF Address

FUNCTION This instruction causes data to be read from disk storage into core storage. The digit 1 in the A-address (%F1) specifies that a single record is to be read. The reading of the disk is stopped by a group mark with a word mark in core storage and the end of the sector. If the digit 2 is present in the A-address (%F2) a full-track read occurs. That is, five 200-character records are read from disk storage into core storage. Reading stops at the end of the fifth sector.

The B-address specifies the high-order position in core storage of the disk-record address and is followed by the area in storage reserved for the data read from the disk.

The R in the d-character position signifies that this is a read operation.

WORD MARKS A group mark with a word mark must appear one position to the right of the last position reserved in core storage for the disk record. If a group mark with a word mark is detected before reading of the record is completed, the wrong-length record indicator turns on and reading stops.

TIMING $T = 0.0115 (L_T + 9) + 10$ ms + disk rotation.
 60.196 ms is maximum time for single-record read.
 10.196 ms is minimum time for single-record read.

We suppose that transactions are entered in the card reader as small groups of them accumulate. If the job were large enough to occupy a 1401 Disk System fully, small groups might be entered almost continuously. If this application were only one of many things being done with the computer, the groups would be entered occasionally, and between times the system could be used for other work.

Note that it is completely unnecessary to sort the transactions into sequence on any key, for the master file is not in any such sequence, which in turn is possible because it is not necessary to access the file sequentially. For the same reason it is not necessary to batch the transactions, although this might be done as a matter of convenience if the transactions do not need immediate action.

The procedure is now so simple that the block

diagram in Figure 9.4 is hardly needed. As each transaction card is read, the corresponding master record is obtained from disk storage, updated, and replaced in disk storage. This simple process is repeated for as many cards as there are.

The program shown in Figure 9.5 presents no special problems. We begin by reading a card and setting up the disk address, which is done by moving the part number into a constant that will have zeros as the first and last digits. Then we seek the disk record having this address and, when it is found, read it into storage. This is followed by a test for reading errors. Note in the constants in this program that immediately following the disk address we have defined an area of 200 characters to hold the record. In this Define Area instruction, note the G; this will cause a group mark with a

Write Disk Single-Record

Write Disk Full-Track

FORMAT

Mnemonic	Op Code	A-address	B-address	d-character
MCW	<u>M</u>	%Fx	xxx	W

AUTOCODER FORMAT Single-Record: WD Address
 Full-Track: WDF Address

FUNCTION This instruction causes a single record (or full-track) in core storage to be written on a disk record. The digit 1 in the A-address (%F1) specifies that a single record is to be written. If a 2 is present in the A-address (%F2), five 200-character records are written on a disk track. Writing stops at the end of the fifth sector.

The B-address specifies the high-order position of the disk-record address and is followed by the data to be written on the disk.

The W in the d-character position signifies that this is a write operation.

Before writing starts, an automatic check of the record address in storage, with the record address on the disk, is made. If they are not the same, the unequal-address compare indicator is turned on, and the data in storage is not written on the disk.

WORD MARKS The writing of data stops when the end of a record is reached or when a group mark with a word mark is sensed in core storage. If the group mark with word mark is sensed before the end of a record, the remainder of the disk record is filled with blanks and the wrong-length record indicator is turned on.

TIMING $T = 0.0115 (L_I + 9) + 10 \text{ ms} + \text{rotation time.}$
 60.196 ms is maximum time for a single record write.
 10.196 ms is minimum time for a single record write.

Note. A Write Disk Check instruction must be performed following a write disk operation. No other disk storage operation can be performed until the check of data written on the disk is completed.

Write Disk Check

FORMAT

Mnemonic	Op Code	A-address	B-address
MCW	<u>M</u>	%F3	xxx

AUTOCODER FORMAT WDC Address

FUNCTION This instruction causes a character-by-character comparison of the data in core storage with the data just recorded on the disk. The system automatically reads the disk record that was most recently addressed. This instruction *must* follow a Write Disk instruction.

The digit 3 in the A-address specifies that a checking operation is to be performed. Either a single record or a full track is checked, depending on how the data was recorded by the most recent Write Disk instruction.

The B-address specifies the area in core storage where the record address and the data recorded on the disk are located.

WORD MARKS A group mark with a word mark must appear one position to the right of the disk data in core storage.

TIMING $T = 0.0115 (L_I + 9)ms + 50 ms$.

Note. If the disk address in core storage is not the same as the address in the record, the unequal-address compare indicator is turned on. If any of the characters in the disk record are not the same as the characters in core storage, the read-back-check error indicator is turned on.

Branch If Indicator On

FORMAT

Mnemonic	Op Code	I-address	d-character
B	<u>B</u>	xxx	x

FUNCTION The d-character specifies the indicator tested. If the indicator is on, the next instruction is taken from the I-address. If the indicator is off, the next sequential instruction is taken. The valid d-characters and the indicators they test are as shown below.

d-character	Indicator
V	Read-or-Write Parity or Read-Back Check Error
W	Wrong-Length Record
X	Unequal-Address Compare
Y	Any Disk-Unit Error Condition
N	Access Inoperable

WORD MARKS Not affected.

TIMING $T = 0.0115 (L_I + 1)ms$.

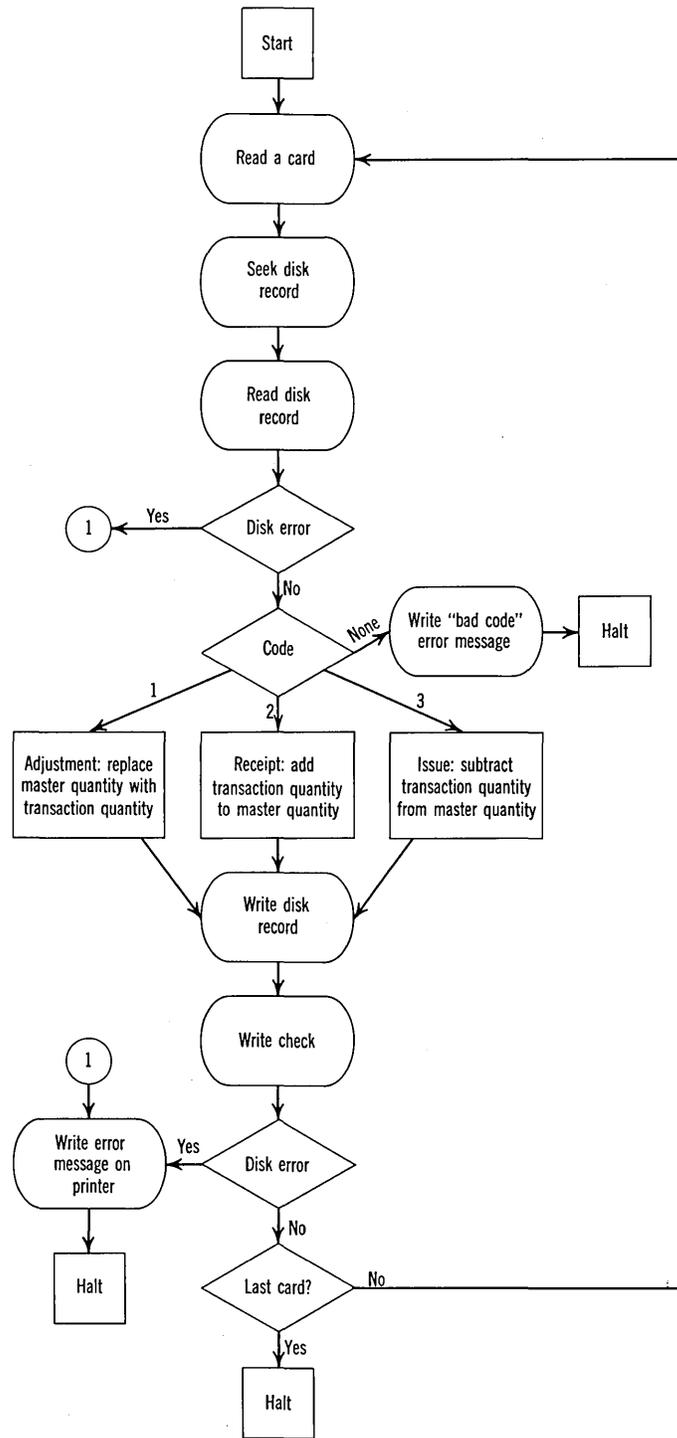


Figure 9.4. Block diagram of the disk storage approach to the inventory control application of Section 8.4.

word mark to be entered following the 200-character area. Next we inspect the classification code in the card record to determine whether it is a recount, receipt, or issue, just as in the magnetic tape version. Once again, if it is none of these three, we write an error message. Next, the transaction

quantity is used to update the master record and the master record is written back in disk storage. Note that there is no Seek instruction here: it is not necessary to seek the correct track if we are already positioned in it. Immediately following the Write, there is the Write Check and a branch

IBM FORM X24-1350-1
PRINTED IN U.S.A.

Program _____
 Programmed by _____ 1401/1410 AUTOCODER CODING SHEET
 Date _____ Identification _____
 Page No. 1 of 2

Line	Label	Operation	OPERAND
3	5	15	20 25 30 35 40 45 50 55 60 65 70
0.1	START	R	READ A CARD
0.2		MCW	TRANPN, DSKADD-1 SET UP DISK RECORD ADDRESS
0.3		SD	DSKADD-6 SEEK
0.4		RD	DSKADD-6 READ
0.5		BIN	ERRØR, Y BRANCH IF ANY ERRØR
0.6		BCE	RECØUN, CØDE, 1 DETERMINE
0.7		BCE	RECPT, CØDE, 2 TYPE OF
0.8		BCE	ISSUE, CØDE, 3 TRANSACTION
0.9		LCA	MESSG1, 226 BAD CØDE -
1.0		W	WRITE ERRØR MESSAGE
1.1		H	*-3 AND HALT
1.2	RECØUN	MCW	TRANQY, MSTQY
1.3		B	WRITE
1.4	RECPT	A	TRANQY, MSTQY
1.5		B	WRITE
1.6	ISSUE	S	TRANQY, MSTQY
1.7	WRITE	WD	DSKADD-6 WRITE DISK RECORD
1.8		WDC	DSKADD-6 CHECK WRITING
1.9		BIN	ERRØR, Y BRANCH IF ANY ERRØR
2.0		BLC	HALT LAST CARD Q
2.1		B	START NØ
2.2	ERRØR	LCA	MESSG2, 222
2.3		W	
2.4		H	*-3
2.5	HALT	H	START BEGIN AGAIN IF PUSH START

Line	Label	Operation	OPERAND
3	5	15	20 25 30 35 40 45 50 55 60 65 70
0.1	DSKADD	DCW	0000000
0.2	FILE	DA	1X200, G
0.3	MSTPN		1, 5
0.4	MSTQY		6, 10
0.5	CARD	0001 DA	1X80
0.6	TRANPN		1, 5
0.7	CØDE		6, 6
0.8	TRANQY		7, 10
0.9	MESSG1	DCW	@BAD CLASS CØDE JØB HALTED@
1.0	MESSG2	DCW	@FILE ERRØR JØB HALTED@
1.1		END	START
1.2			

Figure 9.5. Autocoder program of the procedure diagrammed in Figure 9.4.

to an error routine if there is any file error. Finally, we test the last card switch and go back to the beginning if there are more cards.

Note in the constants for this program that the alphabetic constants are entered in Autocoder preceded and followed by the character @.

REVIEW QUESTIONS

1. Must a new Seek instruction be executed if a different sector in the same track is to be read or written?
2. Is there any circumstance in which it is not necessary to follow a Write Disk instruction with a Write Disk Check instruction?

9.4 Disk Organization and Addressing

It may be well to consider precisely how the preceding example is not typical, in order to develop a few of the standard programming techniques in using disk storage.

First of all, we assumed that the entire disk storage was taken up with the master file. This is seldom the case. Usually, space is reserved for programs and for the files of other applications.

The most unrealistic thing about the preceding example is the assumption that the part numbers run from zero to 99999 in an unbroken numerical sequence. Such a simple correspondence between the key of the records and the record addresses is extremely uncommon. For one thing, part numbers are often not purely numerical; they often have letters and symbols in them. Second, even if they are numerical, they are often longer than five digits. Third, whether they are numerical or alphabetic, there are usually many unused numbers in the sequence, so that if we organize the disk storage as in the preceding example, a large part of it would never be used, which is obviously uneconomical. Our task for the rest of this section is to consider some of the commonly used ways of deriving from the key of a record the disk storage address of that record.

The simplest method is based on deriving the address from the key by simple arithmetic. Suppose, for example, that the keys are purely numerical and seven digits long. Assume, further, that 20,000 disk storage records have been assigned to this file, with record addresses from 50,000 to 69,999. What sort of a scheme could we set up to obtain from such a key an address in the spe-

cified range? One way is to proceed as follows: multiply the seven digit number by 2, drop the last three digits of the product, and add 5 to the high-order position of the remaining digits. A little experimentation will show that for any seven-digit number this yields an address between 50,000 and 69,999.

This method does create a new problem, however. It is very likely that in some cases several keys will convert to the same address. For instance, the keys 1234567 and 1234568 both convert to disk address 52469. This situation is handled by a technique that is known as *chaining*—which has no relation to 1401 address chaining. To understand this technique, we must discuss how disk storage is initially loaded and how the records are obtained when disk storage is later read.

Suppose we are loading storage with records whose addresses are derived by the simple computation described above. The section of storage that is to be loaded is initially cleared to blanks, using a utility program that is described later. Then, as each record is about to be loaded into storage, the record address is computed from its key. Before storing the record at this address, however, we must first check to make sure that the space really is free. If the space still contains blanks, we go ahead and load the record into the sector address as computed. If, however, the space already contains a record because some previous key had converted to the same address, then we store this record in an *overflow* location. This might be the next consecutive record, or it might be in a separate section of storage set up for overflows. Then, in the record having the address computed from this key, we place an overflow address that specifies where the second record having this same computed address is located. When two or more records have the same disk storage address, we speak of the one that is placed in the computed address location as the *home record* and all of the others as *overflow records*. Each record is said to be *chained* to the one following.

We naturally hope that the characteristics of the keys of the source records, together with the method of computing the addresses, will lead to a minimum of such overflows. This, in fact, is one of the primary considerations in choosing an address computation method.

When a file that has been loaded in this fashion is to be read, we go through the same address com-

putation scheme on the key. We seek and read the record at this computed address and then check to see whether it contains the record that we desire by comparing the key of the record that has been read with the key from which the address was computed. If the two are the same, we are able to proceed immediately with processing. If they are not the same, then we must obtain the address of the first overflow record from the record that has been read. When it has been read into core storage, we can similarly inspect its key and find out whether it is the desired one. This process is continued until the proper record has been brought into core storage.

Under unfavorable circumstances, the address computation method suggested above could lead to very long chains of records. This, in turn, would lead to long processing times to search through the chains to find the desired record. For instance, suppose that in one range of the keys there was an unbroken sequence of part numbers, running from 1200000 to 1200499. Every one of these keys would convert to the same address, namely 52400. This would lead to a chain 500 records long, which would obviously be highly undesirable. It appears, therefore, that this method of arriving at a record address applies only if the keys are fairly uniformly distributed over the entire range of possible values. This is frequently not the case, and we must therefore attempt to find address computation schemes that will create a fairly uniform distribution of the addresses, even when the incoming keys are tightly bunched together in some regions. A good deal of effort has been put into finding such schemes, and the subject is still under development.

It is not possible to state any one method that will *always* lead to a sufficiently uniform pattern. Two methods that *often* work, however, are the following:

1. Split the key into sections of four or five digits and add them. Multiply by a compression factor that will bring the final product into the desired range of address and add the base address.

Example. Given eight-digit keys that must be changed into sector addresses between 32000 and 36999, which is 5000 sectors. Take the key 82145369 as a sample.

Add the first four digits to the second four, giving 13583. Multiply by 0.25, which is required to "compress" a number that could be as large as

19998 into a number no larger than 5000, giving 3395. Add the base address, giving the final result: 35395.

2. Split the key into two parts, multiply the two parts and extract the middle five digits. Multiply by a suitable compression factor and add the base address.

Example. Given nine-digit keys that must be changed into sector addresses between 24000 and 38999, which is 15,000 sectors. Take the key 298154726 as a sample.

Multiply the first five digits by the last four, giving 140905690. Extract the middle five digits, giving 09056. Multiply by the compression factor 0.15, giving 01358. Add the base address, giving the final result: 25358.

Another approach is to use some address computation scheme to reach a specified track, without going on to compute a sector within the track. Sector zero within this track is then used as an index to the other nine sectors. That is to say, sector zero contains the keys of all the records stored in that track, together with their addresses. Now, to obtain a record, we compute the track address, seek sector zero on that track, read the index into storage, and from that obtain the address of the proper record location. Since the record will be in the same track as the index, only one Seek is required and the method is not too time consuming. Overflow records become necessary only if the keys of more than nine records convert to the same track address. A disadvantage of this method is that it does require two Read Disk instructions to obtain a record.

The same general idea can be extended even further. An index to the entire file can be set up so that we first locate the proper disk, then go to the outside track on that disk to find an index to the desired track, and from there go to the desired record. This has the advantage that little or no address computation is required and that if suitably set up there are no overflow records.

REVIEW QUESTIONS

1. Using the address computation scheme outlined at the beginning of this subsection, to what disk sector address does 0085692 convert? How about 8882450?
2. What is the basic idea of chaining?
3. What is the most important factor in choosing a randomizing formula for computing the address of the home record in a chained file?

9.5 Disk Storage Utility Routines

A number of utility programs are available for simplifying work with files stored in random access storage. A brief description of some of these routines also allows us to introduce a few more ideas about how disk storage may be used.

Clear disk storage. The clear disk storage program erases all data in areas of disk storage specified by the user and fills these areas with blanks. The program can clear disk storage completely or only in selected areas.

Disk to tape. Each time a disk storage transaction is processed, the previous contents of the master file are no longer available. This raises the possibility that by machine error, program error, or improper data, parts of the file could be destroyed. This problem is not nearly so serious when magnetic tapes are used because we can save the tapes from previous cycles and, if necessary, rerun the job. With disk storage, of course, we no longer have the previous contents of the file unless steps have been taken to make a copy at periodic intervals. This capability is provided by the disk-to-tape routine, whereby the entire file or selected portions of it can be written on magnetic tape. This dumping of file storage can be done in a reasonably short time and in most applications would be done periodically, perhaps weekly. Now, if through some sort of error the file contents were destroyed, it is necessary only to reload the file from the most recent tape and reprocess all of the transactions that have occurred since then. It is, of course, necessary to save the transactions for this purpose.

Tape to disk. This routine is the exact analogy of the disk to tape, making it possible to reload the entire disk file or selected portions of it from magnetic tape.

Disk to card. This program also makes it possible to preserve the contents of disk storage. It is normally used only in systems that do not include magnetic tapes because the time required to punch cards is much greater than the time required to write on magnetic tape.

Card to disk. This program is the exact opposite of the disk to card.

In all of these loading and unloading programs

the smallest unit of information that can be moved is a single track (2000 characters).

Chain loading program. This program simplifies the initial loading of a disk file when the file is being created. In order to use this system, the programmer must provide an address computation routine that can be used by the loading program. The program loads the master records into disk storage under control of this addressing routine and establishes chains for master records converting to the same disk storage address. Each record in a master chain is located as close to the preceding chain record as possible, thus minimizing access time during disk storage operations. Input records can be on cards or tape.

Chain additions program. This program adds new records to a chained file, once again under control of an addressing routine that must be provided by the programmer. The format of the added records must be consistent with that of the records already in the file.

Chain maintenance program. This program carries out a number of operations that are required in using a random access file storage system. For instance, when a record is to be deleted from the file the simplest thing to do is to *tag* it by placing a character somewhere in the record to indicate that it is to be deleted. Then the chain maintenance program can be used to remove the record from the file and make the record storage locations available for later additions, modifying chains as may be necessary.

The chain maintenance program makes it possible to take advantage of a characteristic of most files. Analysis of many typical files shows that a relatively small fraction of the items account for a relatively large fraction of the total activity. This is sometimes described approximately as the 80-20 rule: 80 per cent of the activity comes from 20 per cent of the records. This being the case, it obviously saves disk access time to place the records having the highest activity at the front of their chains. A simple way to accomplish this sequence is to allow space in the records for a count of the number of times each record is referred to; this count is kept by the application program. The chain maintenance program can then inspect the count and reorganize the chains so that the records most frequently referred to appear early in the chains.

The chain maintenance program can be run periodically whenever time is available, since it keeps track of a portion of the file remaining to be processed.

In use, the chain loading program, the chain additions program, and the chain maintenance program are all stored in one part of the disk storage unit so that they can be loaded into core storage in a simple calling procedure. A common plan is to allot some of the lowest numbered tracks to these service routines.

9.6 Case Study: Wholesale Grocery

The following example, based on the data processing requirements of the chain or wholesale grocery operation, serves several purposes. It provides an example of how an IBM 1401 RAMAC System can be used. At the same time, it provides an example of how several related data processing activities are frequently combined into one program. Finally, it illustrates how an ingenious programmer can take full advantage of the equipment available to him by tailoring the machine methods to fit both the equipment and the application. (In this last respect the case study is, in certain details, slightly atypical of disk file methods.) We describe the business situation in which this program would be applied, discuss the organization of the program itself, and show a block diagram of the processing. We shall not write a program.

A certain wholesale grocery distributor has an inventory of 5000 merchandise items which he trucks to 30 stores. An order must be shipped not later than the next working day after he receives it. The order must be accompanied by an invoice.

The order, as received, shows the items in the sequence in which they appear in the catalog. They are arranged for convenience in making up the order, with similar items grouped together and with the broad classes arranged in the order in which they appear on the shelves in a typical store. There is no way to change this general scheme of catalog arrangement. In the warehouse, however, the merchandise is arranged for ease in making up the order, with the most active items, for instance, located close to the loading dock. The invoice used by the warehouse to make up the shipment must show the merchandise items in the sequence in which they should be "picked." Thus

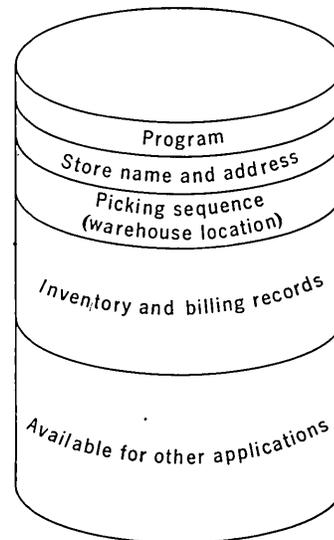


Figure 9.6. Disk storage organization for the wholesale grocery application described in the text.

the order sequence must be transformed into the picking sequence for printing the invoice. Furthermore, each page of the invoice must show the store name and address.

It is necessary to maintain records of the shipments to each store for billing purposes. It is also necessary to maintain inventory records on all of the items in the warehouse and to print low-stock notices when the balance on hand falls below a minimum point.

The order is fed into the computer on a deck of cards that shows the desired merchandise in terms of page and line numbers in the catalog, along with the quantity desired, store number, and date. Each card refers to one catalog page and shows the quantity desired for each of as many as 50 items. The order will require as many cards as there are catalog pages from which merchandise is ordered. By a special method of card coding it is possible to specify a quantity as high as 79 for each item.

The disk storage is divided into five sections for this application, as shown in Figure 9.6.

Store name and address file. This file contains the number, name, and address of each store customer, along with billing information. As each store order is processed during invoice preparation, the proper store record is selected from the disk file and placed with the order number and the date in core storage. This information is printed on each page of the invoice.

Page	Line	Quantity	Picking Sequence Number
1	13	20	4
1	34	5	7
2	02	15	2
6	41	2	9
8	12	30	1
8	13	8	5

Figure 9.7. Illustrative grocery order, with picking sequence numbers for the items ordered.

Picking sequence table. A picking sequence table is set up in disk storage with an entry for every stock item. This table is in order by page and line number and shows the picking sequence for each item in the warehouse stock.

Billing and inventory record. For each item of stock, a billing and inventory record is stored in the disk storage as one 200-character record. These records are arranged in warehouse location order, that is, in picking sequence. The record contains the warehouse location, the picking sequence, catalog page and line number, size, alphabetic description of the item, minimum balance, total sales to date, unit price, balance on hand, and any other information required by the individual customer.

These three files, together with the program for the application, will not completely fill the disk file. The remaining space is available for other applications.

Store orders are processed as they arrive or perhaps in small batches. The store order cards are fed into the 1401 grouped by store and in sequence by page number—the same order in which they were received. An entire order is read and the quantity stored before printing of the invoice begins. This is made necessary by the fact that the catalog sequence and the picking sequence are essentially unrelated. As each order card is read, the picking sequence table for that page is obtained from disk storage. Each line of the order is scanned, and, whenever a quantity appears, that quantity is stored at a core storage location indicated by the table. If no quantity was punched a zero is stored.

The power of the program organization for this problem depends very much on the use of the core storage picking sequence table. This table must have one character position for each item in the stock. In our case we assumed 5000 items, and,

therefore, 5000 core storage locations would have to be allocated to the table. (This would, of course, require a larger core storage than that assumed for the rest of this text.) Each item of stock is associated with one character position in this table. The first position in the table is associated with the stock item that should be picked up first if it is present in the order. The second position is associated with the stock item that should be picked up second, and so on through the 5000 positions.

As each order card is read, the picking sequence table in disk storage is used to determine where in core storage the quantity for that stock item should be stored. When all the order cards have been read, the core storage picking sequence table will contain as many nonzero entries as there are items ordered by the store but no identification of the items; this is inherent in the position of each quantity within the table. After all the order cards have been read, it is necessary only to scan through the 5000-position table looking for nonzero entries and keeping a count of which position of the table is being inspected. Whenever a nonzero character is found, the counter can then be used to compute the address of the corresponding record in the billing and inventory section of the disk storage, which we said was also in picking sequence order.

An example may help to clarify this procedure. Suppose that the warehouse stocked only 10 items, to keep the example simple, and that a certain order lists six items. As each item is processed, its picking sequence number is obtained from the disk file. Assume that the items ordered, their quantities, and their picking sequence numbers are as shown in Figure 9.7. The essence of the scheme is to store

Core Storage Location	Quantity
3001	30
3002	15
3003	0
3004	20
3005	8
3006	0
3007	5
3008	0
3009	2
3010	0

Figure 9.8. Illustrative core storage picking sequence table for the order of Table 9.1.

the quantity for each item in the position in the core storage picking sequence table corresponding to its picking sequence number. Assuming that the entire core storage picking sequence table is cleared to zeros before the order is processed, our example would produce a table similar to that shown in Figure 9.8, which is taken (arbitrarily) to start at 3001. The 30 in position 3001 is now associated with the item shown on page 8, line 12 only by the relative location of the 30 in the table—but this is enough to identify it, since the billing and inventory records are in the same sequence.

What has been done here amounts to sorting the items in the order into picking sequence by a method known as distribution sorting. It is not typical of disk file applications to do this, but the programmer should always be alert for unconventional ways to do things, if time and expense can be saved.

We may note briefly how it is possible to store a quantity up to 79 in one core storage position. This merely requires coding the quantity in terms not only of numerical bits but also the zone bits and the word mark bit. One possible system would be to specify that a word mark bit of 1 stands for a quantity of 40, the B-bit stands for a quantity of 20, and the A-bit thus stands for a quantity of 10. Numerical bits are used in the normal manner to stand for quantities of zero to nine. Figure 9.9 shows how a few representative quantities would be coded in this scheme.

A moderately simple program can be used to create these codes as the quantities are read from the order cards, and another program can convert the codes back to normal two-digit quantities when the invoices are prepared.

This use of the word mark bits is definitely not typical, but there is nothing wrong with it. In

Quantity	Coding			
	WM	B	A	Numerical
0	0	0	0	0000
10	0	0	1	0000
15	0	0	1	0101
23	0	1	0	0011
39	0	1	1	1001
61	1	1	0	0001
79	1	1	1	1001

Figure 9.9

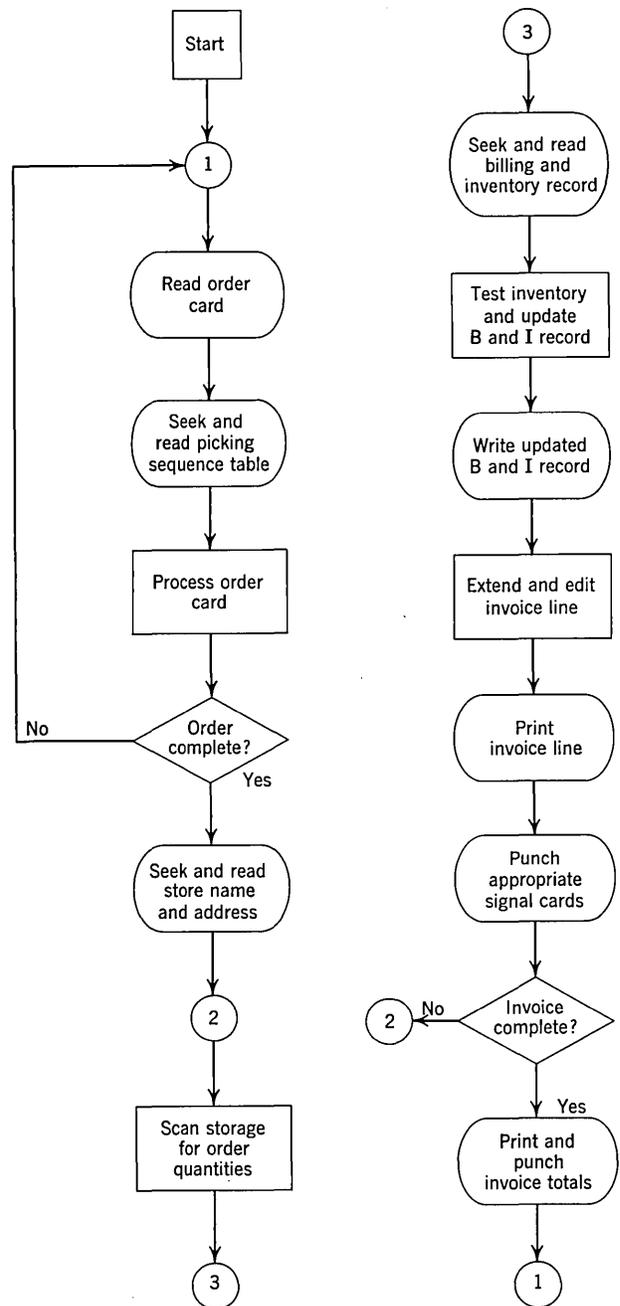


Figure 9.10. Block diagram of the procedure for the wholesale grocery application.

this case it brings about a saving of 5000 characters of storage, which, in effect, makes the whole approach feasible.

To prepare the invoice, we search through the core storage picking sequence table as outlined above. Each time a nonzero item is detected the corresponding billing and inventory record is obtained from disk storage. The inventory balance

is tested for availability. If stock is available, the inventory and sales-to-date balances are updated by the quantity ordered and the updated billing and inventory record is returned to disk storage. The quantity ordered is multiplied by the price, and a billing line is printed on the invoice. If an item is out of stock or if a minimum balance has been reached, an appropriate card is punched for information to the buyers. After all items have been recorded on the invoice, a card is punched for the invoice total.

The items now appear on the invoice in picking sequence. All card-sorting operations required by unit record methods have been eliminated by recording the entire order in core storage in picking sequence as the initial step.

A block diagram of the operations in this application is shown in Figure 9.10.

EXERCISES

*1. Write a routine to compute a disk address from a seven-digit key by the method outlined in the beginning of Section 9.4, then read that record into core storage.

2. Using the routine written for Exercise 1, write a routine to handle chaining. Assume that if the transaction key does not match the key in positions 1 to 10

of the record, the sector address of the next record in the chain appears in positions 180-184 of the record. (Chains may be any number of records long.)

*3. Set up a routine to read a record from an indexed file. The input key is nine digits long and purely numerical. Obtain a track address by forming the sum of the left three digits, the middle three digits, and the right three digits, then retaining only the last three digits of the sum. This gives the address of a *track*; obtain sector zero of this track, which contains an index of the records stored in the other nine sectors in that track. The index consists of ten-character groups, each group containing a nine-digit key and a one-digit sector number. Write a loop to search through the index, once it is in storage, to find the key in the index that matches the desired key; then use the corresponding sector number from the index to get the address and to read the desired record.

4. A labor distribution problem begins with a deck of cards, each containing an employee number, a number of hours worked, and a job code. You are required to compute the labor cost for each labor voucher, assuming the existence of a file giving the pay rate for each man and assuming no overtime (for this problem). There is also a file containing a record for each job code. You are required to print a line for each job represented in the input deck, showing the total labor cost for the week, and to update the job record to reflect this week's costs.

Outline the method you would follow to carry out these operations, including block diagram.

10. PLANNING AND INSTALLING A COMPUTER APPLICATION

In this section we consider an inventory control problem that is a somewhat more realistic version of the case study at the end of Chapter 8. This case study still does not show the entire complexity of a normal inventory control job, but it is close enough to give a fair indication of the work that must be done in setting up a data processing application. We use this application as the framework for considering the various steps in going from a problem statement to a running computer program.

10.1 Problem Statement

A certain company employs about a thousand workers in building small to medium electric motors. The company has an inventory of 20,000 stock items. Four hundred of these are motors and related items built to stock; the remainder are raw materials and subassemblies. The company also manufactures equipment to special order, but since it is never stocked this type of finished product is not included in the inventory control system.

Inventory records in the past have been kept on ledger cards. As the company's business has expanded, this method has become more and more cumbersome, expensive, and time consuming. The company has decided to obtain an IBM 1401 Tape System for this problem and for a variety of other work, such as payroll, production scheduling,

cost accounting, and a small amount of engineering calculation. The company anticipates that keeping inventory control records with the computer will reduce costs slightly, provide more accurate inventory control information, reduce clerical delay, and eventually provide the basis for a more thorough management control of inventory position.

The master file on magnetic tape contains the following information:

Part Number
Abbreviated Alphabetic Description
Quantity on Hand
Quantity on Order
Reorder Point (the point at which more stock is ordered)
Reorder Quantity (the size of the order that is placed)
Code Character (indicates whether this is a finished item for sale, or a raw material or subassembly)
Unit Price for Finished Goods
Year-to-Date Sales for Finished Goods

In this case study there are four types of transactions against this master file:

1. *Issues.* These refer to stock that has been issued by the stock room either to purchasers or to the manufacturing operation.
2. *Receipts.* These refer to inventory items received in the stock room.
3. *Orders.* These are orders placed by purchasing or production control for materials, subassemblies, or finished stock.
4. *Adjustments in the quantity on hand.*

These are the result of such things as recounts, loss, and spoilage.

The processing required may be summarized as follows. The inventory tape is to be updated daily with the transactions being processed against the master file in much the same way as in the case study of Chapter 8. Adjustments replace the quantity on hand in the master file record. Receipts are added to the master file quantity on hand. Issues are subtracted from the master file quantity on hand. Orders are added to the quantity on order in the master file. Since the system is being set up so that nothing is ever received in the stockroom that was not ordered either by purchasing or production control, receipts also represent a fulfillment of an order that was placed previously. Therefore, receipt quantities are subtracted from quantity on order.

Before an issue quantity is subtracted from quantity on hand, a test is made to determine whether there is enough stock on hand to supply the amount specified. If there is not, we reduce the quantity on hand to zero and print an out-of-stock notice. If the quantity on hand plus the quantity on order falls below the reorder point, we print a recommendation to purchasing or production control that more of this item be ordered. We do not add this quantity to the quantity on order until we subsequently receive an order transaction—that is, until purchasing or production control responds to the order recommendation.

For issues of finished goods, the transaction quantity is to be multiplied by the unit price and the product added to the year-to-date sales.

We may summarize how this application relates to the work of other departments within the company. The sales department sends to the data processing center notices of sales of standard items. These become issue transactions. Sales and engineering together determine the raw materials and subassemblies required to manufacture special orders, which, in turn, become issue transactions when these items go out to the manufacturing floor. When production control receives a notice that the level of a standard item has fallen below the reorder point, it is production control's responsibility to schedule the production of more of this item. In doing so, the need is created for raw materials from the stockroom, which once again become issue transactions. When items go into the stockroom after being completed in manufacturing, they be-

come receipt transactions. When an order recommendation for raw materials goes to purchasing, a purchase order is normally placed with a vendor, taking into account any special considerations such as the combination of orders and quantity discounts. As soon as purchasing places the order, an order transaction goes to the data processor to indicate that the quantity has been ordered.

We see that virtually every part of the company is affected in one way or another by inventory control. One of the first things that would normally be considered in designing the data processing system would be to put much of the interdepartment communication into a series of related computer programs. This, however, would get us deeply in the area of systems design, which is beyond the scope of a book on programming.

10.2 Problem Analysis

The decisions described in Section 10.1 fall in the category of what might be called over-all systems design. There is still considerable systems work that must be done before programming can begin; we might call the following area detailed systems design. Both areas have a bearing on the choice of machine to be used and the precise machine configuration to be ordered. We are ignoring this subject entirely.

Detailed systems design in this application would include things like the following.

Card, report, and file formats. Decision must be made on the information formats to be used in the system.

Human readability: Spacing of information within a line and of lines on a report.

Readability of cards after they have been punched.

Whether preprinted forms are to be used for output or whether all headings are to be produced by the computer.

How many copies of each report are required for the various groups that need them.

Card punching simplicity. Cards are often used as the original source document, where the user writes his order or receipt or issue notice on a card, from which the same card is punched. If this is to be done, it must be ascertained that the necessary por-

tions of the card are visible when required on the card punch.

Whether any of the card or tape file formats must be compatible with those of other applications.

This entire area of forms design is a specialized one, best undertaken by someone with experience and training. The job is not so simple as it may appear, and an inexperienced person can create sizable difficulties with poorly designed forms.

Time schedules. Considerable attention must be given to the scheduling of computer time and data arrival. When is the computer available in relation to other work? What is the deadline on the arrival of input and what is done with late data? When must the reports be available to users? How do any peak loads such as at month or year end affect other applications being run on the computer?

Volume data. What is the size of the master file in terms of number of records and number of characters per record? How many transactions of each type may be expected daily? Are there peak periods when the transactions pile up? Can the entire master file be contained on one reel of tape or must a multireel file be set up?

Controls and error checking. How much and what type of error checking is to be done? Some things will be done as a matter of course: tape read checking, parity checking throughout the computer, and, in most well-run installations, label checking and record or block counts. Beyond these obvious and simple possibilities, there are many other things that can be done.

1. Batch control totals on transactions.
2. Balancing equations of the general sort:

Old Balance + Receipts - Issues = New Balance

3. Testing for invalid conditions such as non-existent classification codes, negative quantities on hand, and impossible dates.

The cost of checking for errors must be weighed against the cost of not checking for errors. It is clear that errors can cost money. What is sometimes overlooked is the fact that more money can be spent on error checking than it is worth. Unfortunately, no satisfactory formulas can be given for arriving at a decision, since it is usually very difficult to establish what the cost of an error is or even to enumerate all the possibilities of errors. At

the present state of the art of data processing we can only suggest that experience is the best guide.

Master file creation. At some point it is necessary to create the master file. If the job has previously been done with manual methods, it will be necessary to punch cards from the present records. If the job has previously been done with punched card techniques, it may be possible to convert the present file to magnetic tape with a separate computer program written for the purpose. Often the data in the cards is rearranged and new information added.

In some applications this problem of file conversion can be a major undertaking all by itself, requiring not only a good deal of personnel and computer time but careful planning in the scheduling of the conversion. This planning is made necessary by the fact that the business has to go on running while the computer is put into operation. Often the master file is much too large to permit a complete shutdown of this area of the company's clerical operation while the file is being converted. The usual technique is to prepare the master file as of a specific date and continue with the manual or punched card methods until the computer system is ready to go into operation. During the period between conversion of the file and the termination of the previous methods, all transactions are saved. When the system is finally in operation, the first step is to process all the accumulated transactions against the new master file. If the file is extremely large, it is necessary, furthermore, to make the conversion gradually so that for a certain period of time part of the file will be processed by manual methods and part by electronic methods. We shall return to this point later and see that a period of parallel operation is usually desirable in any case.

There is insufficient space in this book to provide enough background information about our assumed company to permit a realistic discussion of the bases of the decisions that must be made in this area of detailed systems design. We must instead be content with a statement of the following decisions.

Formats. The master file tape consists of blocks of eight records of 54 characters each, as assigned in Table 10.1.

The transaction cards have the format shown in Table 10.2.

An order recommendation card is punched when

TABLE 10.1

Information	Number of Characters	Character Position Within Record
Part Number	7	1-7
Description	12	8-19
Quantity on hand (QOH)	5	20-24
Quantity on order (QOO)	5	25-29
Reorder point (RP)	5	30-34
Reorder quantity (RQ)	5	35-39
Unit price	6	40-45
Year-to-date sales	8	46-53
Code: 0 = raw material or sub-assembly		
1 = finished goods	1	54

TABLE 10.2

Information	Number of Characters	Character Position Within Record
Part number	7	1-7
Transaction code	1	8
1 = adjustment		
2 = receipt		
3 = order		
4 = issue		
Transaction quantity	5	9-13

the inventory falls below the reorder point (see Table 10.3).

Out of stock notices are printed with a format dictated only by normal readability requirements.

The printer is used to print notices to the operator about the problem.

The effect of schedules on our planning of this program is ignored, since this subject gets us too deeply into the interrelationships in the company, interrelationships between different applications on the computer, and the detailed requirements placed on this application by management.

Volumes are similarly largely ignored except to note that the master file is small enough to fit on one reel of tape and small enough to permit creation of the master file in one step. (These are poor assumptions in many actual applications.)

Programmed error checking includes certain tests based on the organization of the file, such as check-

ing for nonexistent part numbers and invalid transaction codes. These are in addition to the label checking and tape checking that are automatically handled by the Input Output Control System.

Before the transactions can be processed against the master file, they must be sorted into the same sequence as the master file. This is done here with a standard tape sorting package that is designed for the 1401, as for most computers. The sorting program is available in generalized form, requiring only that the programmer supply a few items of information about the file and machine configuration in order to produce a sort program tailored to his needs. A generalized sorting routine of this type relieves each programmer of the large amount of effort required to write a specific sort for each job.

It is a good idea to analyze each application to determine whether tape sorting or card sorting is more effective. Tape sorting is considerably faster, but it does tie up the relatively expensive computer. Card sorting is sometimes much less expensive, if the control field is short, but the total job time is greater. This is a decision that must be made for each application.

10.3 Block Diagram and Program for Inventory Control Processing

A block diagram of the processing to carry out the operations described in Section 10.2 is shown in Figure 10.1. The bulk of the file processing logic is the same as in the corresponding block diagram in Figure 8.6, but there are a few differences.

In any file processing logic we must handle the situation where master file items remain after the end of the transactions has been reached. In the preceding block diagram and program (Figure 8.6) we set a switch for this purpose. Here we have

TABLE 10.3

Information	Number of Characters	Card Columns
Part number	7	1-7
Code: 0 = raw material or subassembly		
1 = finished goods	1	8
Recommended quantity	5	9-13

The logic of the processing of the transactions is a little more complex now because we are required to do more. Recounts are handled as before by replacing the master quantity on hand with the transaction quantity. A receipt must be added to the quantity on hand but also subtracted from the quantity on order—since we assume that nothing is ever received unless it has been ordered. (A situation not satisfying this assumption could be handled with a recount entry.) The transaction quantity of an order is simply added to the quantity on order.

Before, we merely subtracted an issue quantity from the quantity on hand. Now, we are required to determine that there is sufficient stock to fill the order before doing this. If the quantity on hand is not great enough to fill the issue request, we write a stock shortage notice and replace the transaction quantity with the quantity on hand. This will make certain that when the year-to-date sales are updated we will not have credited ourselves with selling more than we had. It also means that when the transaction quantity is subtracted from the

quantity on hand the quantity on hand will be reduced to zero, as it should be. (This could, of course, be handled in other ways, but this way simplifies coding.)

If the item represents finished goods that are being sold to a customer, which we can determine by inspecting the code in the master record, we update sales.

In any case, we next determine whether the quantity on hand plus the quantity on order has fallen to or below the reorder point. If it has, we punch an order recommendation. We do *not* immediately add the reorder quantity to the quantity on order; this is done only when an order transaction is entered, indicating that purchasing or production control has responded to the recommendation.

Note the use of connectors (small circles enclosing numbers) to avoid long lines on the block diagram.

The program in Figure 10.2 follows the block diagram fairly closely. It provides a review of 1401 coding, with no new concepts being introduced.

Line	Label	Operation	OPERAND
3	5,6	15,16 20,21	25 30 35 40 45 50 55 60 65 70
0.1	START	OPEN	OLDMST, NEWMST
0.2		R	TRANSACTION CARD
0.3	READM	GET	OLDMST, WORK
0.4	COMP	C	TRANPN, MSTPN
0.5		BH	ERROR
0.6		BL	LOW TRANS, SEQUENCE ERROR
0.7		BCE	LOW MASTER - NO ACTIVITY
0.8		BCE	TEST
0.9		BCE	TRANSACTION
1.0		BCE	CLASSIFICATION
1.1		LCA	CODE
1.2		W	CODE NO GOOD
1.3		H	*-3
1.4	RECOUN	MCW	TRANQY, MSTQOH
1.5		B	REORD
1.6	RECPT	A	TRANQY, MSTQOH
1.7		S	TRANQY, MSTQOH
1.8		B	REORD
1.9	ORDER	A	TRANQY, MSTQOH
2.0		B	REORD
2.1	ISSUE	C	TRANQY, MSTQOH
2.2		BL	ENOUGH ON HAND Q
2.3		B	SHORT
2.4		OK	NO
			YES

Figure 10.2. Autocoder program of the procedure diagrammed in Figure 10.1.

IBM FORM X24-1350-1
PRINTED IN U.S.A.

Program _____

Programmed by _____ 1401/1410 AUTOCODER CODING SHEET

Date _____ Identification _____
Page No. 2⁷⁶ of 4⁸⁰

Line	Label	Operation	OPERAND
3	5	15	20 21 25 30 35 40 45 50 55 60 65 70
0.1	SHORT	LCA	MESSG2, 249 WRITE
0.2		MCW	TRANPN, 210 STOCK
0.3		MCW	TRANQY, 223 SHORTAGE
0.4		MCW	MSTQOH, 239 MESSAGE
0.5		W	
0.6		MCW	MSTQOH, TRANQY PUT QTY AVAIL IN TRANS
0.7	OK	S	TRANQY, MSTQOH
0.8		BCE	REORD, MCODE, 0 BRANCH IF RAW MATL, SUBASSY
0.9		MCW	UNITPR, MULT FINISHED GOOD -
1.0		M	TRANQY, MULT EXTEND PRICE
1.1		A	MULT, YTDS ADD TO YEAR TO DATE SALES
1.2	REORD	ZA	MSTQOH, TEMP COMPARE QTY ON HAND PLUS
1.3		A	MSTQOH, TEMP QTY ON ORDER
1.4		C	TEMP, RP WITH REORDER POINT
1.5		BL	LCTEST NO REORDER
1.6		MCW	TRANPN, 107 YES REORDER - PUNCH PART
1.7		MCW	MCODE, 108 NUMBER, CODE, AND
1.8		MCW	RQ, 113 REORDER
1.9		P	QUANTITY
2.0	LCTEST	BLC	LC LAST CARD 0
2.1		R	COMP READ A CARD AND BRANCH
2.2	LC	MCW	NINES, TRANPN FORCE HIGH TRANS PART NO
2.3	WRITNM	PUT	WORK TO NEWMST
2.4		B	READM
2.5			

IBM FORM X24-1350-1
PRINTED IN U.S.A.

Program _____

Programmed by _____ 1401/1410 AUTOCODER CODING SHEET

Date _____ Identification _____
Page No. 3⁷⁶ of 4⁸⁰

Line	Label	Operation	OPERAND
3	5	15	20 21 25 30 35 40 45 50 55 60 65 70
0.1	EOR	BLC	WRAPUP HERE FROM GET ROUTINE
0.2	ERROR	CS	0299 CLEAR STORAGE
0.3		LCA	MESSG3, 229 WRITE
0.4		W	ERROR MESSAGE
0.5		H	*-3
0.6	WRAPUP	CL	OLDMST, NEWMST
0.7		CS	0299 CLEAR STORAGE
0.8		LCA	MESSG4, 212 WRITE JOB
0.9		W	END MESSAGE
1.0		H	*-3 AND HALT
1.1		H	
1.2			

Figure 10.2 (Continued).

10.4 Program Checkout

Any sizable data processing application provides hundreds of opportunities to make mistakes that will completely invalidate the program. These may range from simple slips of the pencil that result in nonexistent addresses, to block diagramming errors that destroy the logic of the program, to misunderstandings of the intended procedures. A program is not finished until it has been thoroughly tested to remove all programming mistakes and to establish that the program properly carries out the intentions of the person who defined the problem. This process of detecting and correcting errors and proving the correctness of a program can easily take weeks.

There is no single way of solving all the problems of checkout. We must rely on a combination of procedures on the part of the programmer and on specialized checkout programs that are available to help him. Above all, the programmer must exercise good judgment in determining how to go about the checkout process. This is one of the primary

areas in which experience is essential to a satisfactory result. We can only point out some of the standard procedures and tool programs and encourage the reader to get as much practice as he can in this area, if possible under the guidance of an experienced programmer.

One practice that is most strongly recommended is that all programs be very thoroughly checked before they are assembled. The programmer should plan to spend as much as several days prechecking a major program. This should be a character-by-character check to make sure that every symbol and every operation code is correct, that O's and zeros are properly distinguished, that core storage information is never destroyed prematurely, that loops are counted correctly, that decimal points are properly handled, that all utility programs have been used in accordance with specifications, that timing problems are properly handled, etc.

There seems to be an almost overwhelming temptation when a program is finished to put it on a machine to see whether it will work. We cannot emphasize too strongly that all experience indicates

IBM ○

Program _____

Programmed by _____

Date _____

1401/1410 AUTOCODER CODING SHEET

FORM X24-1350-1
PRINTED IN U.S.A.

Identification _____

Page No. 4 of 4

Line	Label	Operation	OPERAND													
			3	5	6	15	16	20	21	25	30	35	40	45	50	55
0.1	MESSG1	DCW	@BAD CLASS CODE JOB HALTED@													
0.2	MESSG2	DCW	@ONLY PART AVAILABLE REQUESTED@													
0.3	MESSG3	DCW	@FILE OR DATA ERROR JOB HALTED@													
0.4	MESSG4	DCW	@JOB FINISHED@													
0.5		0001 DA	1X80													
0.6	TRANPN		1, 7													
0.7	TCODE		8, 8													
0.8	TRANQY		9, 13													
0.9	WORK	DA	1X54													
1.0	MSTPN		1, 7													
1.1	DESC		8, 19													
1.2	MSTQPH		20, 24													
1.3	MSTQPH		25, 29													
1.4	RP		30, 34													
1.5	RQ		35, 39													
1.6	UNITPR		40, 45													
1.7	YIDS		46, 53													
1.8	MCODE		54, 54													
1.9	MULT	DCW	000000000000													
2.0	TEMP	DCW	00000													
2.1	NINES	DCW	9999999													
2.2		END	START													
2.3																

Figure 10.2 (Continued).

that time spent in desk checking at this point will save much more time later. It can easily happen that half a dozen errors of an easily found variety can waste weeks of time and cause several unnecessary reassemblies. If possible, it is an excellent idea to have someone else go over the program, on the theory that the original programmer may have overlooked errors because he knows what the program *ought* to do and has not checked to see that it *actually will do* what it is supposed to do.

A listing of the program as punched is a valuable tool in desk checking, since it allows the programmer to scan for errors in punching or in interpreting his handwriting at the same time that he is checking the program as he wrote it.

It is recommended that a copy of the listing of the assembled absolute program be obtained; this is called the post listing in the SPS and Autocoder systems. This listing provides not only a diagnosis of certain errors that can be detected by the processor but also is a very useful document for all of the following stages of program checkout. This usefulness is based on the fact that the original symbolic program and the assembled absolute program are shown on the same piece of paper. This allows the programmer to correlate the program that is actually running in the machine with the way that he wrote it to begin with. Most programmers file their original coding sheets at this point and rarely refer to them again.

The basic idea of program checkout is to put the program in the machine with suitable test data to see whether it will compute correct results. At the outset, this test data may be very simple; the idea is to find out whether the program will produce any answers at all. The most common experience when the program is first put on the machine is that it stops before producing any results. This can be caused by a variety of errors. If the difficulty results in some sort of error indication on the computer console, then it is fairly easy to get back to the source of the trouble, correct it, and try again. If the problem runs nearly to completion and then hangs up, or if it does produce problem answers but they are incorrect, then we have the problem of determining where in a program of many hundreds of instructions something went wrong. Two powerful techniques come into play at this point.

The first is the use of a storage print, or, as it is more commonly called, a memory dump. This provides a printed listing of exactly what was in

core storage at the time the program was stopped. In the 1401, and in most computer systems, the listing is printed with line identifications that make it simple to associate storage addresses with the printed contents. The memory dump allows the programmer to inspect the results of program modification, it provides a complete listing of all intermediate and final results computed up to the point of the dump, it shows exactly what test data is being used, and it allows the programmer to determine whether anything in storage has been modified that should not have been.

Usually only a part of storage is printed; there is seldom any need to see the *entire* contents. The starting and ending addresses of the regions to be printed are entered from the computer console or from the card reader.

The entire procedure is so simple, and the assistance it provides so valuable, that it is strongly recommended that a memory dump be obtained at every checkout session. The alternative practice, followed by some inexperienced programmers, of copying down from the console a few supposedly critical numbers cannot be defended; it is inaccurate, time consuming, and generally completely ineffective.

The second important checkout technique is to check out the program in sections by providing for the printing of intermediate results. This allows the source of errors to be narrowed down successively. Furthermore, it assists in proving the correctness of a program. The purpose of checkout, after all, is not only to locate and correct errors but to guaranty that no undetected errors remain. The only way to do this is to verify every partial and final result against known test cases; printing out the partial or intermediate answers helps in this process.

The intermediate answers may be obtained with memory dumps taken at various intermediate points, or they may be printed by instructions inserted in the program for the purpose. These may be left in the program until a final assembly, at which time they are removed, or they may be made conditional on the setting of a sense switch and left in the program.

This brings up a point that should be kept in mind: checkout is such an important part of the whole process of getting a computer application into operation that it must be planned for in writing the program in the first place. The provision of

instructions to print intermediate results is only one example of this sort of planning. Others that may apply in various situations: placing all results in one area of storage to simplify printing; avoiding "tricks" in coding that may be difficult to test; including comments that make it easy to correlate the program with the block diagram; etc.

It is occasionally desirable to use a technique called *tracing*, which provides a listing of the instructions and data as each instruction in a section of the program is executed. This technique should ordinarily be used only when all else fails. When used indiscriminately, it can waste large amounts of computer time and still not provide the needed information.

A variety of specialized checkout programs have been produced for most computers. These may do such things as supplying printouts of selected sections of storage during the execution of the program, analyzing storage after the program has been run to print all areas that have changed since the program was loaded, and inserting halt instructions in all unused sections of storage so that the program will stop if it reaches an unintended location. The availability and operation of these programs vary considerably from one machine to the next.

The final check that should be made on any data processing program is called *pilot operation*. This consists in using a large quantity of actual transactions from a previous period. Use of actual transactions gives a better test for possibilities that may not have been considered in making up test cases. The results produced by the computer system can be checked against the results produced by the previous manual or unit record methods. And since the transactions have already been processed, there are no business pressures to get the results out, pressures that would prevent careful analysis of the operation of the program.

One of the major secondary benefits of pilot operation is that it gives personnel in the department for which the work is being done an opportunity to see whether the program actually does what they intended. It is all too easy for the problem originator to say one thing and mean another, for the programmer to misunderstand what is meant by terms in an area with which he is not familiar, or for either one to overlook special conditions. The program is not actually finished until it has been proven to produce precisely correct results using real problem data in volume.

10.5 Going Into Operation

After a new program has been properly tested, some problems still remain in going into full-scale operation. The transition from previous manual or punched-card methods involves a sufficient number of people and enough changes in procedures that careful plans must be laid to insure that the change-over is smooth.

The first thing that must normally be done is to catch up with the transactions that have occurred since the new master file was created. This is also an excellent chance to give the program one further shakedown before it is required to do all the data processing itself. During this process a fair number of errors in the master file will be discovered. These may be the result of incorrect conversion or they may represent errors that were in the previous file all along and had not been detected. Actually, this process of file cleanup ordinarily continues into the first few weeks or months of operation of the system.

A common practice is to continue the use of the the previous manual or punched-card system for a few weeks in parallel with the operation of the new electronic system. This parallel operation provides a backup in case the electronic system develops difficulties that require it to be taken out of operation to correct. The parallel operation also provides an excellent test of the accuracy and adequacy of electronic processing, since it is a simple matter to compare the results produced by the two systems. The processing of the transactions accumulated since the creation of the master file also provides a means of checking one system against the other.

As we noted before, many master files are so large that they must be converted in segments, since an attempt to convert the entire master file at one time would result in such an accumulation of transactions that it would be difficult to catch up. One common way to effect a partial conversion is to break the file into segments arbitrarily on the basis of the keys of the records. Another possibility is to convert the records for which there are transactions as the transactions arise. Any such partial conversion scheme must obviously be carefully planned in advance so that there will be a minimum of confusion between the data processing center and those responsible for the previous methods. The difficulty, of course, is that the people who

have been using the old methods will in most cases still be heavily involved in the new system. If the transition is not properly carried out, these clerical personnel may be overloaded by having to deal with two systems at once.

It is perhaps obvious that the introduction of various applications to be done with the computer should be spaced out. Any attempt to put several major applications in operation at one time would create peak loads both at the computer and in the rest of the organization that could very well cause failure of the whole system.

10.6 Documentation

A computer system is next to worthless if it is not adequately documented. To provide all of the information that is required for the many people in various parts of the company organization that must use the system, several different types of documentation are needed.

One that is obviously necessary is a complete writeup of the program itself. This document, which is often called a *run manual*, normally contains some or all of the following information:

1. A run number and title.
2. The name of the programmer.
3. The date of completion of the program and of the last modification.
4. A sheet summarizing the computer operating instructions for ready reference, including labels and descriptions of tapes and their disposition, error or special procedures, rerun instructions, average run time, and switch settings.
5. A one- or two-paragraph description of the purpose of the run.
6. A complete set of flow charts and block diagrams.
7. Completed forms where applicable, for storage allocation, record designs, and operating instructions for any off-line equipment.
8. An assembly listing of the program. Changes in the coding after the program has been checked out should be entered in red pencil, initialed, and dated.
9. A sample of each report produced by the program.
10. Suggestions for future changes and warnings about making changes.

A document of this sort is obviously necessary if intelligent use is to be made of the program and if modifications and revisions in the program are to be made with minimum effort.

The programmer should prepare instructions for the computer operator covering machine setup, console switch settings, tape units required, carriage control tapes, error correction procedures, etc. This, of course, duplicates some of the material in the run book, but it should be remembered that the run book is too big a document to be kept at the computer console. Operating instructions for all programs are generally kept in a single notebook at the console.

A third type of document might be called a procedures manual. This is used by the people who make use of the computer, that is, the personnel in other departments of the company who originate data and use the results. Typical contents of such a manual are these:

1. Exhibits of input documents, with instructions for their preparation and transmittal.
2. Exhibits of output forms and reports and an explanation of their contents, discussion of the frequency of preparation, etc.
3. Timing schedules for data submission and receipt of reports.
4. Handling of special circumstances.

10.7 Summary

Putting a data processing application on a computer involves a number of steps, carried out at various levels of the organization by many different people. It begins with a study of what the data processing needs are and of alternative ways of solving them. After it has been established that a particular computer is to be ordered, much work must be done in deciding just how to go about splitting up the company's data processing requirements into manageable pieces that can be set up on the computer. After this has been done, the general characteristics of each computer run, including file and record formats, must be planned. Only at this point is it possible to write computer instructions. When instructions have been written, the accuracy of the program must be verified. The file conversion and the start-up of the application both require considerable planning. To this list

should be added such activities as planning for the physical installation of the computer, the training of the people to program and operate it, indoctrination of the people in other departments of the company who will make use of the services of the computer, and an education program to introduce the computer to the entire company in a way that will minimize ever-present fears about job security.

It must be admitted that in this complete list the subject of coding, which we have discussed in this book, is only one part, representing less than a majority of the time required to get into operation. The person who expects to be working closely with computers nevertheless needs to start his education with the subject of computer coding or programming, for without this knowledge a proper grasp of the more advanced subjects cannot be gained. Still, it should be realized that the area we have introduced in this book is only the beginning and that the person who expects to be a truly professional computer expert has a number of years of apprenticeship and study before him.

EXERCISES

1. The block diagram and program of this section contain a questionable procedure: the reorder calculation

is made after every transaction. If there are several issues for one part, it could easily happen that we will punch many order recommendations, which is at least pointless and perhaps confusing. Modify the block diagram so that the reorder calculation is made only when all transactions for a part number have been processed.

2. You are given a master payroll tape for a payroll of 4000 hourly workers. Each record contains a payroll number in positions 1 to 5, an hourly pay rate of the form x.xxx in positions 22 to 25, and other information totaling 200 characters. You are also given a cost accounting tape containing one record of 80 characters for each of 800 active jobs in the company. Each record contains a job number in 6 to 9, a total-to-date cost in dollars in 49 to 54, and other information.

Each week there are about 40,000 labor voucher cards giving payroll number in 1 to 5, hours worked to tenths of an hour in 6 to 8, and a job number in 9 to 12.

You are required to prepare a weekly labor distribution report showing, for each job on which work was performed this week, the total direct labor cost for the week; the cost accounting tape must be updated to include this week's direct labor cost; for each man, a card must be punched to show gross pay for the week.

a. Draw a flow chart of the computer and punched-card operations necessary to satisfy these requirements.

b. Draw a block diagram of the two computer runs required to get gross pay and to produce the labor distribution report.

c. Write programs corresponding to the block diagrams. State any additional assumptions that must be made to carry out these operations.

11. ADDITIONAL PROGRAMMING METHODS

11.1 Introduction

The presentation of computer coding and programming in this book has been based primarily on the use of the Symbolic Programming System and Autocoder, in the interest of a unified presentation. The reader should realize, however, that there are many other programming techniques and systems that serve many of the same purposes as SPS and Autocoder, namely, to reduce the work of initial programming and to simplify program modification. These other systems also sometimes help to take advantage of special conditions and to handle other types of problems than have been illustrated here.

In other words, SPS and Autocoder are good and are heavily used, but they are not the whole story. In this rather brief section we shall investigate a few additional methods of which the reader should be aware. The discussion gives only a quick introduction to the various topics; various other publications are available for the reader who wishes more complete information.

The topics to be considered are the use of decision tables in system design, the FORTRAN coding system for scientific and engineering problems, the Report Program Generator for the production of programs to write reports, and the COBOL coding system, which is a sophisticated coding language for expressing data processing procedures. No attempt will be made to evaluate these advanced programming languages because the whole subject is under such intensive de-

velopment at the present time that any evaluation would soon be out of date.

11.2 Decision Tables

Before any coding can be done on a problem, there must be a precise definition of the procedure to be followed. As we have seen, this definition of the problem and procedure can easily require more effort than the coding and checkout which follow, and the effort is usually at a more sophisticated level. As with any other activity, this work, which is called systems design, requires methods of representing the actions to be carried out and the conditions under which they are to be done. In this book we have used two techniques for this purpose: narrative description and flow charts. Various other methods of describing the work are employed occasionally.

Another method that is coming into prominence is the use of *decision tables*. A decision table is a rectangular array of boxes, organized to describe a decision system involving many variables and many results. The basic arrangement of a decision table is shown in Figure 11.1, where we see that a horizontal double line separates *conditions* above from *actions* below and a vertical double line separates the *stub* on the left from the *entries* on the right. The stub contains the descriptions of the conditions and actions for each row of the table. Figure 11.2

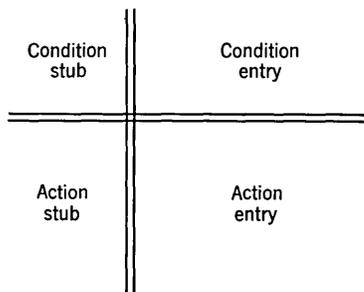


Figure 11.1. Schematic diagram of the parts of a decision table.

is a simple example of a decision table which describes the decisions in one section of the inventory control case study of Chapter 10.

The basic idea of using a table is to inspect the decision parameters, one column at a time, until a column is found in which all the conditions are satisfied. When this occurs, the actions contained in that column are to be carried out. If a condition entry is blank, then that condition has no bearing on deciding which *rule* (column) is to be followed. For instance, if the transaction is an order, we do not care whether the transaction quantity is more or less than the quantity on hand. If an action entry is blank, then no action is required for the operation named in the action stub for that row. For instance, there is no quantity shipped in this example except on an issue.

At the end of the table is ordinarily a line stating where to go if the table cannot be solved, that is,

if no column's conditions are satisfied by the input. Following this is the name of the table which normally should be solved next. If the choice of the next table depends on decisions in the body of the table, as in this example, a separate row can be set up to determine this branching.

Even in a simple example like this it may be seen that a decision table provides a clear picture of the relationships between conditions and actions and of the interrelationships of combinations of conditions. The technique has the added advantage that omissions are explicitly indicated, leading to a complete statement of the procedure early in the planning.

The decision table concept is much broader than this example might indicate, extending to such diverse areas as tape processing logic, accounting procedures, manufacturing operations, routine engineering decisions, and the writing of utility routines. Furthermore, the decision table concept is not limited to providing insights into the logic of a system. A suitably standardized form of decision table is also feasible as a source language for describing data processing procedures to a computer.

11.3 The FORTRAN Coding System

FORTRAN is a source program language for expressing problems in science and engineering, together with a processor that translates the source program statements into an object program that can be run on a computer. In contrast with SPS, the source program language has virtually no relation

Transaction Type	Issue	Issue	Receipt	Order	Adjustment
Transaction Quantity	\leq QOH	$>$ QOH			
QOH =	$QOH - TQ$	0	$QOH + TQ$	QOH	TQ
QOO =	QOO	QOO	$QOO - TQ$	$QOO + TQ$	QOO
Quantity Shipped =	TQ	QOH			
GO TO	Extend	Shortage	Reorder	Reorder	Reorder

If unsolvable, go to CODE-ERROR

Figure 11.2. Decision table to describe an inventory control procedure.

establishes 1000 as the initial value of the frequency. Statement 39, which is given a statement number because it will be necessary to return to it, calls for the actual computation. We note here the use of symbols to specify arithmetic operations, according to the convention:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**

We see also that the square root is called for simply by writing the name SQRTF. When the FORTRAN processor encounters this *function* name, it will incorporate into the object program a routine to take a square root. The FORTRAN programmer never has to know how to write the 15 machine language instructions by which square roots are usually computed or even to know what the method is.

The PRINT statement leads to object program instructions to print the input data and the results. The next statement is the FORTRAN equivalent of a conditional branch. The effect, in this case, is this: if the frequency is less than 2000, go to statement 50 where we set up the next value of the frequency, but if the frequency is equal to or greater than 2000 go to the STOP statement. Statement 50 is another example of an arithmetic formula statement. This is not an equation, obviously, but rather a command to FORTRAN: replace the value of the variable named on the left with the value of the expression on the right. The GO TO 39 creates in the object program a simple unconditional branch.

FORTRAN is an example of a *procedure-oriented* language; that is, the language is used to write a problem-solving procedure in terms of the method to be followed. As we shall see, COBOL is also a procedure-oriented language. This is in contrast to programming systems such as SPS, where the procedure must be described more in terms of the machine operations to be executed and which are therefore called *machine-oriented* languages. Procedure-oriented languages have a number of advantages.

1. They are generally somewhat easier to learn and use than machine-language coding systems. The beginner does not have to know anything about how the machine operates in order, for instance, to

take a square root: he simply writes SQRTF. This advantage should be kept in perspective, however. It must be remembered that getting a problem solved with a computer involves many activities besides coding. The use of a language like FORTRAN or COBOL in no way reduces the careful planning that must go into getting a correct problem statement, determining the best approach to the solution, planning a thorough set of test cases, or documenting the program. What FORTRAN does is to simplify the job of coding so the programmer can concentrate on these other things.

2. Program modifications are easier to make because of features designed into the language. In the case of COBOL modifications are easier because the description of the procedure is kept rigidly separate from the description of the data; this means that one can be changed without having to rewrite the other.

3. The procedure statements are to a large extent independent of the machine on which the object program will be run. The FORTRAN program in Figure 11.2, with a few modifications, can be compiled and run on any one of a dozen or more different computers. The object programs for the various machines would be very different, but the source program is largely independent of this fact.

FORTRAN has the desirable characteristic that it is attractive both to beginner and expert. To the beginner, FORTRAN offers the advantage of ease of learning and the quick solution of simple problems. To the expert, it offers faster coding, ease of modification, and machine independence of the procedure statements.

11.4 The Report Program Generator

The desirability of simpler ways of programming is, of course, not restricted to scientific computations. The Report Program Generator is one of several systems that provide much the same advantages for commercial data processing that FORTRAN does for scientific work. Using this system, the source language user once again does not have to know much about machine language coding. The procedure is stated on four types of forms which provide answers to the following questions:

1. What are the characteristics of the file from which the data to appear in the report is obtained?

guage for stating procedures in business data processing. As such, it shares the advantages of procedure-oriented languages stated above in connection with FORTRAN.

A COBOL source program is composed of three sections:

1. Procedure division: the procedure statements that specify how the data is to be processed.
2. Data division: description of the format and organization of the data and results.
3. Environment division: a description of the equipment to be used by the object program.

These three divisions are separated in writing the source program, which leads to one of the major advantages of COBOL and languages similar to it: changing the procedure does not require changing the data descriptions and vice versa. Considering that in most programs for commercial data processing the object program and the data arrangement are strongly interrelated, this becomes a sizable advantage. In the frequent situation where the data arrangement must be changed slightly, it is necessary only to modify the data division and recompile. This is in contrast to the situation with actual machine language coding, for instance, where in extreme cases a single added digit in the data can force reprogramming most of the problem.

The independence of the procedure and data divisions leads to another major advantage of COBOL, which is also shared with FORTRAN and a number of other systems: machine independence. With rather minor changes, a COBOL source program can be compiled for running on any computer for which a COBOL processor exists. The data division does depend somewhat on the object machine, to take account of such machine characteristics as variable versus fixed word length, the handling of signs, and tape formats. It usually turns out, however, that changing the data division is far less work than rewriting the whole program,

and the relative machine independence is in fact achieved.

Procedure statements in COBOL are written in a form closely paralleling ordinary English construction. In fact, a completed COBOL program can in some cases be read like an English description of the procedure. There is, however, very little flexibility in the way the "sentences" can be written. English language readability is an advantage of COBOL, but a secondary one.

11.6 Fundamentals of COBOL Programming

To give a little better appreciation of what programming is like when these advanced tools are used, we shall take COBOL as a representative example and consider it a little more fully. This is done first with a series of short examples that will introduce the fundamental ideas, and then the inventory control of Chapter 10 will be rewritten in COBOL.

The central feature of a billing procedure is the multiplication of unit price by the quantity sold. To do this, a sentence in a COBOL program could be as shown in Figure 11.8. In this sentence the word MULTIPLY is a verb. The sentence, furthermore, is *imperative*. When the COBOL processor translates it into actual machine instructions, it creates the instructions necessary to bring about the action specified by the verb.

As can be seen in the example, a COBOL sentence is very similar to an ordinary English statement in construction and format. Actually, the parallel extends to some aspects of punctuation. A COBOL sentence must always end with a period. Data is referred to by *name*, such as "TOTAL-PRICE." We note immediately an exception, however, in that words which are to be considered as combined in one name must be hyphenated because in COBOL

```
01 MULTIPLY UNIT-PRICE BY QUANTITY GIVING TOTAL-PRICE.
```

Figure 11.8

```
01 REORDER-ROUTINE. IF QUANTITY-ON-HAND IS LESS THAN
02 MINIMUM THEN MOVE ORDER-QUANTITY TO PURCHASE-AMOUNT.
```

Figure 11.9

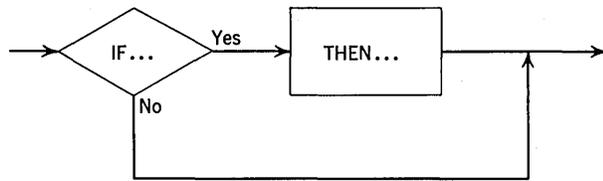


Figure 11.10

a blank space always indicates the beginning of a new word. Furthermore, although it is not apparent from this one example, the structure of a COBOL sentence must follow very precisely the rules laid down in the COBOL manual. It would not be possible to rewrite this sentence as THE TOTAL PRICE IS COMPUTED BY MULTIPLYING UNIT PRICE AND QUANTITY. This would be the English language equivalent, but COBOL would not accept such a sentence. We shall not attempt to give the precise rules for forming each type of COBOL sentence; this information may be obtained readily enough from a COBOL manual.

For a second example, consider a part of an inventory calculation. One sentence of the COBOL procedure for determining whether to place an order might be as shown in Figure 11.9. The first word "REORDER-ROUTINE" is called a *procedure name*. A procedure name provides a way of referring to the sentences that follow.

The sentence illustrates a *conditional expression* involving a simple relation between two quantities. If the quantity on hand is less than the reorder point, the action specified following THEN is carried out. If the quantity on hand is not less than the reorder point, the action following THEN is not carried out. Figure 11.10 shows in schematic form the structure of this sentence. A *conditional clause*, which is introduced by the word IF and concluded by THEN, in effect asks a question to which the answer must be yes or no. We shall speak of each relation involved in a conditional expression as being true or false, or satisfied or not satisfied. This example uses the IS LESS THAN relation. The allowable relations in COBOL language are shown in Figure 11.11.

The example in Figure 11.9 also shows a different command, MOVE TO. The action called for is the copying of information within storage. The information named ORDER-QUANTITY is to be copied and called PURCHASE-AMOUNT.

It is probably apparent by now that certain words have special meanings in COBOL language: in the present examples, the words IF, THEN, MOVE, and TO and the phrase IS LESS THAN all have special meaning, and confusion would result if we tried to interpret these words in any other way. Such words are a fixed part of the language and are called *reserved words*. A complete list of the reserved words in COBOL language may be found in a COBOL manual. Reserved words must not be used to mean anything but what COBOL defines them to mean.

To introduce a few more features of COBOL language, we may use a common payroll example, as shown in Figure 11.12.

The conditional clause in this example is different from what we have seen previously. The first part of the clause consists of just the word HOURLY, which is called a *condition name*. HOURLY is one of the possible values that can be taken on by the implied data name PAYROLL TYPE, the other values being EXEMPT, SALARIED, and TEMPORARY. Since there are only a few of these conditions, it is convenient for the programmer to use his normal terminology. The actual machine instructions are set up to work with the coded representation of these values; for instance, the numbers 1, 2, 3 and 4. Some way must be provided to correlate the condition names with

Long Form	Short Form
IS EQUAL TO	EQUAL TO
IS NOT EQUAL TO	NOT EQUAL TO
IS LESS THAN	LESS
IS NOT LESS THAN	NOT LESS
IS GREATER THAN	GREATER
IS NOT GREATER THAN	NOT GREATER

Figure 11.11

```

01  IF HOURLY AND HOURS-WORKED IS LESS THAN 40
02  THEN GO TO GROSS-PAY, OTHERWISE GO TO NET-PAY.
  
```

Figure 11.12

the corresponding values. Establishing this correspondence is one of the many functions of the data division. Figure 11.13 shows the appropriate part of the data division to establish this correspondence.

PAYROLL TYPE is defined as a level 3 entry, which indicates its relative importance with respect to other elements of data. EXEMPT, SALARIED, HOURLY, and TEMPORARY are named as the four conditions, and the code number used for each is given. Thus in this example the value of PAYROLL TYPE is 3 whenever HOURLY is meant.

The second part of the conditional clause is

HOURS WORKED IS LESS THAN 40

In this case the value of the data name HOURS WORKED is compared with the number 40; 40 is not to be interpreted as a data name but literally is the value of 40 itself. We speak of 40 as a *numeric literal*.

The second part of the conditional clause is joined to the first part by the reserved word AND, which specifies that both the first and the second parts of the clause must be satisfied before carrying out the operations that follow THEN. This is shown schematically in Figure 11.14, which emphasizes that *both* parts of the conditional clause must be true before carrying out the action specified after THEN. If either or both of the parts is false, the

```

01 3 PAYROLL-TYPE
02 88 EXEMPT VALUE IS 1
03 88 SALARIED VALUE IS 2
04 88 HOURLY VALUE IS 3
05 88 TEMPORARY VALUE IS 4
    
```

Figure 11.13

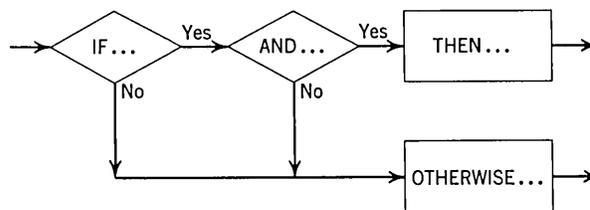


Figure 11.14

action specified after OTHERWISE is executed. If the sentence is written without the word OTHERWISE the program continues with the following sentence.

In the COBOL sentence under consideration the action specified after both THEN and OTHERWISE is a new command, GO TO. The GO TO command makes it possible to get out of the one-after-the-other sequential execution of sentences and instead execute next the sentence named by the GO TO.

The fourth example is based on a part of an inventory control calculation. Assume as we have previously that there are just four types of transactions—recounts, receipts, orders, and issues. The part of the job that we wish to consider is how to take action appropriate to the type of transaction. The program shown in Figure 11.15 is a little longer than those preceding but most of the ideas are already familiar.

The first line of this program brings into play a processor command:

NOTE INVENTORY RECORD
MAINTENANCE

NOTE indicates that what appears in the rest of the sentence is information for the reader of the program; it is not for the COBOL processor, which ignores it. The programmer is permitted and encouraged to use notes freely in order to make the

```

01 NOTE INVENTORY RECORD MAINTENANCE
02 GO TO (RECOUNT-ROUTINE, ORDER-ROUTINE, RECEIPT-PROCEDURE,
03 ISSUE-PROCEDURE) ON TRANSACTION-CODE.
04 ISSUE-PROCEDURE. SUBTRACT TRANSACTION-QUANTITY,
05 QUANTITY-ON-HAND. PERFORM REORDER-CALCULATION.
06 GO TO NEXT ITEM.
    
```

Figure 11.15

program more intelligible to the reader. These play the same part as the comments in SPS and Auto-coder programs.

The GO TO shown on the second line is a more powerful form of the command than we have seen before. This is called an *assigned* GO TO. For any one transaction, only one of the four procedures named in parentheses is performed. The one selected depends on the current value of the transaction code, which can be 1 to 4. These numbers correspond to the names within the parentheses. If the value is 1, the first name is selected, etc. This is summarized in Figure 11.16.

An assigned GO TO provides a multiple branch or switching point. The last two lines of Figure 11.15 illustrate another COBOL verb, namely, SUBTRACT. The SUBTRACT verb may also be used in this form:

```
SUBTRACT TRANSACTION-QUANTITY
FROM QUANTITY-ON-HAND GIVING
QUANTITY-ON-HAND
```

In the condensed form used in Figure 11.15 the meaning is exactly the same; that is, the value corresponding to the first data name is subtracted from the value corresponding to the second data name. These are the only two ways the SUBTRACT verb may be used.

The last line in Figure 11.15 shows another type of transfer of control:

```
PERFORM REORDER-CALCULATION
```

The PERFORM verb may be thought of as meaning "go to the place named, do whatever it says to do, and come back." In our case it is used to transfer to a procedure named REORDER-CALCULATION and to set up a return path so that, after executing the procedure, control will return to the sentence immediately following PERFORM. PERFORM thus provides the facilities for a subroutine linkage. After performing the reorder calculation, control will return to and execute the GO TO NEXT ITEM sentence in Line 6.

As another illustration of the use of COBOL we may use a savings bank procedure: updating an account record to indicate interest payment. The program might be as shown in Figure 11.17. Line 1 again shows the use of a procedure name to provide a named point to which the program can transfer. In this case it precedes a slightly different type of conditional clause. Instead of simply comparing two values as we have before, the programmer has indicated that he wishes to see if the value of an arithmetic expression (.03* PRINCIPAL) is less than a numeric literal (1.00). The asterisk is used to indicate multiplication. It is quite valid to incorporate an arithmetic expression within a conditional expression. For clarity it may often be desirable to use parentheses to denote the beginning and end of such an arithmetic expression.

There is one other new point to notice in this example. On Lines 4 and 5 we have

```
MOVE "INTEREST" TO ACTION
```

If the current value of TRANSACTION-CODE is	Then GO TO:
1	RECOUNT-ROUTINE
2	ORDER-ROUTINE
3	RECEIPT-PROCEDURE
4	ISSUE-PROCEDURE

Figure 11.16

```
01 INTEREST-CALCULATION. IF .03* PRINCIPAL IS LESS
02 THAN 1.00 THEN GO TO END OTHERWISE MULTIPLY
03 PRINCIPAL BY 1.03 GIVING ACCOUNT BALANCE,
04 MOVE 'INTEREST' TO ACTION.
```

Figure 11.17

The quotes indicate that the word INTEREST *itself* is to be moved to the area name ACTION. Thus INTEREST is identified by the quotes as being an *alphameric literal*.

There are six verbs that handle all of our input-output problems. In terms of the 1401, the verb ACCEPT calls for data to be read from cards. This would normally be written in the form ACCEPT data-name FROM CARD READER, where we would write the name of the card record for the data-name. Printing and punching are handled by the DISPLAY verb. Here, we write the word DISPLAY, followed by the names of the data to be printed, followed by the word UPON, followed by the word PRINTER or CARD PUNCH.

The verbs OPEN and CLOSE have the same meanings that they have in the Autocoder Input-Output Control System. That is, OPEN checks labels and positions tape to read the first tape block. CLOSE handles the trailer label and rewinds the tape.

The COBOL equivalent of the IOCS GET is called READ and performs exactly the same functions; that is, it makes available an input record either by advancing to a new record if records in the block remain to be processed or by actually reading a tape block if all records in the input area have been processed. The routine compiled from the READ verb performs the same error checking

and checks for the end of the file. We specify in the source program what should be done if the end of the file is reached by writing the words AT END, followed by any imperative statement.

The COBOL equivalent of the IOCS PUT is called WRITE and performs the same function. We are not required in using READ and WRITE to determine whether the object program will use indexing or a work area and similar matters. All such considerations are handled by the COBOL processor.

Examples of these verbs appear in the program of the next subsection.

This has by no means been a complete exposition of the COBOL language. It is hoped, however, that this discussion, together with the extended example that follows, will provide some insight into the nature of COBOL programming and its advantages.

11.7 COBOL Program for Inventory Control Case Study

Figure 11.18 is a COBOL program to carry out the operation described in the block diagram of Figure 10.1. Since the program is written in English it is largely self-explanatory.

```
01 PROCEDURE DIVISION.
02   OPEN INPUT OLD-MASTER-FILE, OUTPUT NEW-MASTER-FILE.
03   ACCEPT TRANSACTION-CARD FROM CARD-READER.
04   MASTER-READING. READ OLD-MASTER-FILE RECORD AT END GO TO
05   WRAPUP-TEST.
06   COMPARISON. IF PART-NUMBER OF OLD-MASTER IS GREATER THAN
07   PART-NUMBER OF TRANSACTION-CARD GO TO WRAPUP-TEST.
08   IF PART-NUMBER OF OLD-MASTER IS EQUAL TO PART-NUMBER OF
09   TRANSACTION-CARD GO TO CODE-TESTING-ROUTINE OTHERWISE
10   GO TO MASTER-WRITING.
11   CODE-TESTING-ROUTINE. GO TO RECOUNT, RECEIPT, ORDER, ISSUE
12   DEPENDING ON TRANSACTION-CODE. MOVE 'BAD CLASS
13   CODE JOB HALTED' TO MESSAGE.
14   DISPLAY MESSAGE ON PRINTER. STOP 1.
15   RECOUNT. MOVE TRANSACTION-QUANTITY TO QUANTITY-ON-HAND.
16   GO TO REORDER-ROUTINE.
17   RECEIPT. ADD TRANSACTION-QUANTITY, QUANTITY-ON-HAND.
18   SUBTRACT TRANSACTION-QUANTITY, QUANTITY-ON-ORDER.
19   GO TO REORDER-ROUTINE.
20   ORDER. ADD TRANSACTION-QUANTITY, QUANTITY-ON-ORDER.
21   GO TO REORDER-ROUTINE.
22   ISSUE. IF QUANTITY-ON-HAND IS NOT LESS THAN TRANSACTION-
23   QUANTITY GO TO SUBTRACTION-ROUTINE. MOVE
24   QUANTITY-ON-HAND TO MESSAGE-A. MOVE PART-NUMBER OF
25   TRANSACTION-CARD TO MESSAGE-B. MOVE TRANSACTION-QUANTITY
26   TO MESSAGE-C. DISPLAY MESSAGE ON PRINTER. MOVE
27   TRANSACTION-QUANTITY TO QUANTITY-ON-HAND.
28   SUBTRACTION-ROUTINE. SUBTRACT TRANSACTION-QUANTITY,
29   QUANTITY-ON-HAND. MULTIPLY UNIT-PRICE BY TRANSACTION-
30   QUANTITY GIVING TOTAL-PRICE. ADD TOTAL-PRICE,
31   YEAR-TO-DATE-SALES.
32   REORDER-ROUTINE. IF QUANTITY-ON-HAND + QUANTITY-ON-ORDER
33   IS GREATER THAN REORDER-POINT GO TO LAST-CARD-ROUTINE.
34   MOVE PART-NUMBER OF TRANSACTION-CARD TO CARD-1. MOVE
35   MASTER-CODE TO CARD-2. MOVE REORDER-QUANTITY TO CARD-3.
36   DISPLAY CARD ON CARD-PUNCH.
37   LAST-CARD-ROUTINE. IF LAST-CARD GO TO REPLACEMENT-ROUTINE.
38   ACCEPT TRANSACTION-CARD FROM CARD-READER. GO TO
39   COMPARISON.
40   REPLACEMENT-ROUTINE. MOVE HIGH-VALUE TO PART-NUMBER OF
41   TRANSACTION-CARD.
42   MASTER-WRITING. WRITE NEW-MASTER FROM OLD-MASTER. GO TO
43   MASTER-READING.
44   WRAPUP-TEST. IF LAST-CARD GO TO CLOSEOUT. MOVE
45   'FILE OR DATA ERROR JOB HALTED' TO MESSAGE.
46   DISPLAY MESSAGE ON PRINTER. STOP 2.
47   CLOSEOUT. CLOSE OLD-MASTER-FILE, NEW-MASTER-FILE. MOVE 'JOB
48   FINISHED' TO MESSAGE. DISPLAY
49   MESSAGE ON PRINTER. STOP 3.
```

Figure 11.18. COBOL program for inventory control case study.

APPENDIX 1. IBM 1401 INSTRUCTIONS WITH SYMBOLIC PROGRAMMING SYSTEM MNEMONICS

Not all of the instructions listed here have been discussed in the text. For those which have been introduced, the page number of the detailed description is given.

Where no d-modifier is shown, none is required;

where it is shown as "d," consult the appropriate table below for meanings.

Instructions involving special features not found on a basic 1401 are marked with *.

INPUT-OUTPUT INSTRUCTIONS

Instruction	Actual Op Code	d	SPS Op Code	d	Page
Read a Card	1		R		36
Write a Line	2		W		36
Write Word Marks	2	□	W	□	
Write and Read	3		WR		96
Punch a Card	4		P		36
* Read Punch Feed	4	R	P	R	
Read and Punch	5		RP		97
Write and Punch	6		WP		97
* Write and Read Punch Feed	6	R	WP	R	97
Write, Read, and Punch	7		WRP		
* Start Read Feed	8		SRF		
* Start Punch Feed	9		SPF		

ARITHMETIC INSTRUCTIONS

Instruction	Actual Op Code	SPS Op Code	Page
Add	A	A	43
Subtract	S	S	44
Zero and Add	0	ZA	80
Zero and Subtract	0	ZS	
* Multiply	@	M	56
* Divide	%	D	

LOGIC INSTRUCTIONS

Instruction	Actual Op Code	d	SPS Op Code	d	Page
Branch	B		B		63
Branch If Indicator On	B	d	B	d	64
Branch If Character Equal (contents of B address compared with d-modifier)	B	d	B	d	119
Branch If Word Mark and/or Zone	V	d	BWZ	d	67
Compare	C		C		67

d-modifier for Branch If Indicator On Instruction

	Blank	Unconditional
Comparison	/	Unequal compare: $B \neq A$
	S	* Equal compare: $B = A$
	T	* Low compare: $B < A$
	H U	* High compare: $B > A$
Overflow	Z	Overflow
Last card	A	Last card switch
Sense switches	B	* Sense switch B
	C	* Sense switch C
	D	* Sense switch D
	E	* Sense switch E
	F	* Sense switch F
	G	* Sense switch G
	Printer	@
9		Carriage channel 9
P		* Printer busy
R		* Carriage busy
I/O errors	+	Reader error with I/O check stop switch off
	-	Punch error with I/O check stop switch off
	0	Punch error with I/O check stop switch off
Processor error	+	Printer error with I/O check stop switch off
	%	Processing error with process check stop switch off
Magnetic tape	K	* End of reel
	L	* Tape error
Disk file	V	* Read/Write parity-check or read back check error
	W	* Wrong-length record
	X	* Unequal address compare
	Y	* Any disk storage error
	N	* Access inoperable

DATA CONTROL INSTRUCTIONS

Instruction	Actual Op Code	SPS Op Code	Page
Move Characters to A or B Word Mark	M	MCW	34
Load Characters to A Word Mark	L	LCA	55
Move Characters and Suppress Zeros	Z	MCS	65
Move Characters and Edit	E	MCE	58
Move Numerical	D	MN	
Move Zone	Y	MZ	
Set Word Mark	,	SW	35
Clear Word Mark	□	CW	35

MISCELLANEOUS INSTRUCTIONS

Instruction	Actual Op Code	d	SPS Op Code	d	Page
Control Carriage	F	d	CC	d	93
Select Stacker	K	d	SS	d	72
No Operation	N		NOP		78
Clear Storage	/		CS		37
Halt	.		H		66
* Store A-address Register	Q		SAR		
* Store B-address Register	H		SBR		
* Modify Address	#		MA		

MAGNETIC TAPE INSTRUCTIONS

A-address is of the form %Ux where x is the tape unit number

Instruction	Actual Op Code	d	SPS Op Code	d	Page
Read Tape	M	R	MCW	R	107
Write Tape	M	W	MCW	W	107
Read Tape with Word Marks	L	R	LCA	R	
Write Tape with Word Marks	L	W	LCA	W	
Control Unit	U	d	CU	d	108
* Move Characters to Record or Group Mark	P		MCM		
* Move and Insert Zeros	X		MIZ		

d-modifiers for Control Unit Instruction

B	Backspace Tape Record
E	Skip and Blank Tape
M	Write Tape Mark
R	Rewind Tape
U	Rewind Tape and Unload

DISK STORAGE INSTRUCTIONS

A-address is of the form %FX where

- X = 0 is used with operation code M for Seek Disk
- X = 1 specifies single record
- X = 2 specifies full track
- X = 3 is used with operation code M for Write Disk Check

Instruction	Actual Op Code	d	SPS Op Code	d	Page
Seek Disk	M		MCW		126
Read Disk	M	R	MCW	R	127
Write Disk	M	W	MCW	W	128
Read Disk with Word Marks	L	R	LCA	R	
Write Disk with Word Marks	L	W	LCA	W	

APPENDIX 2. AUTOCODER OPERATION CODES

DECLARATIVE OPERATIONS

Mnemonic Op Code	Description
DA	Define Area
DC	Define Constant (No Word Mark)
DCW	Define Constant With Word Mark
DS	Define Symbol
DSA	Define Symbol Address
EQU	Equate

IMPERATIVE OPERATIONS

Type	Mnemonic Op Code	Description	Machine Language	
			Op Code	d-char.
Arithmetic	A	Add	A	
	D	Divide	%	
	M	Multiply	@	
	S	Subtract	S	
	ZA	Zero and Add	&	
	ZS	Zero and Subtract	—	
Data Control	MBC	Move and Binary Code	M	B
	MBD	Move and Binary Decode	M	A
	MCE	Move Characters and Edit	E	
	MCS	Move Characters and Suppress Zeros	Z	
	MIZ	Move and Insert Zeros	X	
	MLC } MCW }	Move Characters to Word Mark	M	
	MLCWA } LCA }	Move Characters and Word Marks to Word Mark in A-Field	L	
	MLNS } MN }	Move Single Numerical Character	D	
	MLZS } MZ }	Move Single Zone	Y	
	MRCM } MCM }	Move Characters to Record Mark or Group Mark-Word Mark	P	

Type	Mnemonic Op Code	Description	Machine Language		
			Op Code	d-char.	
Logic	B	Branch Unconditional	B		
	BAV	Branch on Arithmetic Overflow	B	Z	
	BBE	* Branch if Bit Equal	W	d	
	BC9	Branch on Carriage Channel 9	B	9	
	BCV	Branch on Carriage Overflow (12)	B	@	
	BE	Branch on Equal Compare (B = A)	B	S	
	BEF	Branch on End of File or End of Reel	B	K	
	BER	Branch on Tape Transmission Error	B	L	
	BH	Branch on High Compare (B > A)	B	U	
	BIN	* Branch on Indicator	B	d	
	BL	Branch on Low Compare (B < A)	B	T	
	BLC	Branch on Last Card (Sense Switch A)	B	A	
	BM	Branch on Minus (11-zone)	V	K	
	BPCB	Branch Printer Carriage Busy	B	R	
	BPB	Branch Printer Busy	B	P	
	BU	Branch on Unequal Compare (B ≠ A)	B	/	
	BW	Branch on Word Mark	V	1	
	BWZ	* Branch on Word Mark or Zone	V	d	
	BCE	* Branch if Character Equal	B	d	
	BSS	* Branch if Sense Switch On	B	A-G	
	C	Compare	C		
	I/O Commands	BSP	Backspace Tape	U	B
		CU	* Control Unit	U	d
		DCR	Disengage Character Reader	U	D
		ECR	Engage Character Reader	U	E
		LU	* Load Unit	L	d
		MU	* Move Unit	M	d
P		Punch	4		
PCB		Punch Column Binary	4	C	
R		Read	1		
RCB		Read Column Binary	1	C	
RD		Read Disk Single Record	M	R	
RDT		Read Disk Full Track	M	R	
RDW		Read Disk Single Record With Word Marks	L	R	
RDTW		Read Disk Full Track With Word Marks	L	R	
RF		Read Punch Feed	4	R	
RP		Read and Punch	5		
RT		Read Tape	M	R	
RTB		Read Tape Binary	M	R	
RTW		Read Tape With Word Marks	L	W	
RWD		Rewind Tape	U	R	
RWU		Rewind and Unload Tape	U	U	
SD		Seek Disk	M		
SKP		Skip and Blank Tape	U	E	
SPF		Start Punch Feed	9		
SRF		Start Read Feed	8		
W		Write	2		

* d-character must be coded in the operand of the instruction.

Type	Mnemonic Op Code	Description	Machine Language	
			Op Code	d-char.
	WD	Write Disk Single Record	M	W
	WDC	Write Disk Check	M	
	WDCW	Write Disk Check With Word Marks	L	
	WDT	Write Disk Full Track	M	W
	WDTW	Write Disk Full Track With Word Marks	L	W
	WDW	Write Disk Single Record With Word Marks	L	W
	WM	Write Word Marks	2	□
	WP	Write and Punch	6	
	WR	Write and Read	3	
	WRF	Write and Read Punch Feed	5	R
	WRP	Write, Read and Punch	7	
	WT	Write Tape	M	W
	WTB	Write Tape Binary	M	W
	WTM	Write Tape Mark	U	M
	WTW	Write Tape With Word Marks	L	W
Mis- cellaneous	CC	* Carriage Control	F	d
	CCB	* Carriage Control and Branch	F	d
	CS	Clear Storage	/	
	CW	Clear Word Mark	□	
	H	Halt	.	
	MA	Modify Address	#	
	NOP	No Operation	N	
	SAR	Store A-Address Register	Q	
	SBR	Store B-Address Register	H	
	SS	* Select Stacker	K	1, 2, 4, 8
	SSB	* Select Stacker and Branch	K	1, 2, 4, 8
	SW	Set Word Mark	,	

* d-character must be coded in the operand of the instruction.

CONTROL OPERATIONS

Mnemonic Op Code	Description
CTL	Control
END	End
ENT	Enter New Coding Mode
EX	Execute
LTORG	Literal Origin
ORG	Origin

APPENDIX 3. CARD AND COMPUTER CHARACTER CODES

Prints As	Defined Character	Card Code	BCD Code	Prints As	Defined Character	Card Code	BCD Code
.	BLANK		C	G	G	1-27	BA 421
□	□	12-3-8	BA8 21	H	H	12-8	BA8
(Left Parenthesis (Special Character)	12-4-8	CBA84	I	I	12-9	CBA8 1
<	Less Than (Special Character)	12-5-8	BA84 1	—	! (Minus Zero)	11-0	B 8 2
‡	Group Mark (Note 1)	12-6-8	BA842	J	J	11-1	CB 1
&	&	12-7-8	CBA8421	K	K	11-2	CB 2
\$	\$	12	CBA	L	L	11-3	B 21
*	*	11-3-8	CB 8 21	M	M	11-4	CB 4
)	Right Parenthesis (Special Char.)	11-4-8	B 84	N	N	11-5	CB 4 1
;	Semicolon (Special Character)	11-5-8	CB 84 1	O	O	11-6	B 42
Δ	Delta (Mode Change)	11-6-8	CB 842	P	P	11-7	CB 421
—	—	11-7-8	B 8421	Q	Q	11-8	CB 8
/	/	11	B	R	R	11-9	B 8 1
,	,	0-1	C A 1	#	# Record Mark	0-2-8	A8 2
%	%	0-3-8	C A8 21	S	S	0-2	C A 2
=	Word Separator	0-4-8	A84	T	T	0-3	A 21
'	Apostrophe (Special Character)	0-5-8	C A84 1	U	U	0-4	C A 4
”	Tape Segment Mark	0-6-8	C A842	V	V	0-5	C A 4 1
¢	Cent (Special Character Note 2)	0-7-8	A8421	W	W	0-6	A 42
#	#	3-8	A	X	X	0-7	C A 421
@	@	4-8	C 84	Y	Y	0-8	C A8
:	Colon (Special Character)	5-8	841	Z	Z	0-9	A8 1
>	Greater Than (Special Character)	6-8	842	0	0	0	C 8 2
√	Tape Mark	7-8	C 8421	1	1	1	1
?	(Plus Zero)	12-0	CBA8 2	2	2	2	2
A	A	12-1	BA 1	3	3	3	C 21
B	B	12-2	BA 2	4	4	4	4
C	C	12-3	CBA 21	5	5	5	C 4 1
D	D	12-4	BA 4	6	6	6	C 42
E	E	12-5	CBA 4 1	7	7	7	421
F	F	12-6	CBA 42	8	8	8	8
				9	9	9	C 8 1

The IBM 1401 has the ability to read MLP card codes in the read feed only. The 1401 ignores the 8-9 punches when they appear in the same column. The 1401 does not punch out MLP card codes.

Note 1. If specified, this code can be made compatible with 705 Group Mark Code (12-5-8).

Note 2. The A-bit coding must be program generated in the IBM 1401 (it cannot be read from a card; it can be punched as a zero). It is used in conjunction with the C-bit to indicate a blank position on tape that was written in even-bit parity.

APPENDIX 4. INSTRUCTION TIMING DATA

System Timings		System Timings		
Key to abbreviations used in formulas L_A = Length of the A-field L_B = Length of the B-field L_C = Length of Multiplicand field L_I = Length of Instruction L_M = Length of Multiplier field L_Q = Length of Quotient field L_R = Length of Divisor field L_S = Number of significant digits in Divisor (Excludes high-order 0's and blanks) L_W = Length of A- or B-field, whichever is shorter L_X = Number of characters to be cleared L_Y = Number of characters back to right-most "0" in control field L_Z = Number of 0's inserted in a field I/O = Timing for Input or Output cycle F_m = Forms movement times. Allow 20 ms for first space, plus 5 ms for each additional space T_m = Tape movement times Z = Number of fields included in an operation		Operation	OP Code	Formula
		Punch a Card	4	.0115 ($L_I + 1$) + I/O
		Read a Card	1	.0115 ($L_I + 1$) + I/O
		Read and Punch	5	.0115 ($L_I + 1$) + I/O
		Select Stacker	K	.0115 ($L_I + 1$)
		Set Word Mark	,	.0115 ($L_I + 3$)
		Start Punch Feed *	9	.0115 ($L_I + 1$)
		Start Read Feed *	8	.0115 ($L_I + 1$)
		Store A-address Register *	Q	.0115 ($L_I + 4$)
		Store B-address Register *	H	.0115 ($L_I + 4$)
		Subtract (no recomplement)	S	.0115 ($L_I + 3 + L_A + L_B$)
		Subtract (recomplement)	S	.0115 ($L_I + 3 + L_A + 4L_B$)
		Write a Line	2	.0115 ($L_I + 1$) + I/O
		Write and Punch	6	.0115 ($L_I + 1$) + I/O
		Write and Read	3	.0115 ($L_I + 1$) + I/O
		Write, Read and Punch	7	.0115 ($L_I + 1$) + I/O
		Zero and Add	?	.0115 ($L_I + 1 + L_A + L_B$)
		Zero and Subtract	!	.0115 ($L_I + 1 + L_A + L_B$)

Operation	OP Code	Formula	Tape Operations
Add (no recomplement)	A	.0115 ($L_I + 3 + L_A + L_B$)	T_m - Tape movement can be determined from the following: N = Number of Characters C = Character Rate 729 II at 200 cpi = .067 ms at 556 cp = .024 ms 729 IV at 200 cpi = .044 ms at 556 cpi = .016 ms 7330 at 200 cpi = .139 ms at 556 cpi = .050 ms 729 Model II, Read 10.7 + CN ms = TAU interlocked 10.5 + CN ms = Processing interlocked Write 11.7 + CN ms = TAU interlocked 7.5 + CN ms = Processing interlocked 729 Model IV, Read 6.8 + CN ms = TAU interlocked 6.7 + CN ms = Processing interlocked Write 7.8 + CN ms = TAU interlocked 5 + CN ms = Processing interlocked 7330 Read 20.5 + CN ms = TAU interlocked 7.7 + CN ms = Processing interlocked Write 20.3 + CN ms = TAU interlocked 5 + CN ms = Processing interlocked Rewind 729 Model II = 1.2 minutes/reel 729 Model IV = .9 minutes/reel 7330 (High Speed) = 2.2 minutes/reel Skip and Blank Tape (add to subsequent write time) 729 Model II = 40.5 ms 729 Model IV = 27 ms 7330 = 103 ms Backspace (after Read) Backspace (after Write) 729 Model II = 46 + CN ms 729 Model II = 52 + CN ms 729 Model IV = 33 + CN ms 729 Model IV = 37 + CN ms 7330 = 428 + CN ms 7330 = 435 + CN ms
Add (recomplement)	A	.0115 ($L_I + 3 + L_A + 4L_B$)	
Branch	B	.0115 ($L_I + 1$)	
Branch if Bit Equal *	W	.0115 ($L_I + 2$)	
Branch if Character Equal	B	.0115 ($L_I + 2$)	
Branch if Indicator On	B	.0115 ($L_I + 1$)	
Branch if Word Mark and/or Zone	V	.0115 ($L_I + 2$)	
Clear Storage	/	.0115 ($L_I + 1 + L_X$)	
Clear Word Mark	□	.0115 ($L_I + 3$)	
Compare	C	.0115 ($L_I + 1 + L_A + L_R$)	
Control Carriage	F	.0115 ($L_I + 1$) + F	
Control Unit	U	.0115 ($L_I + 1$) + T	
Divide (aver.) *	%	.0115 ($L_I + 2 + 7L_R L_Q + 8L_Q$)	
Halt	°	.0115 ($L_I + 1$)	
Load Characters to A	L	.0115 ($L_I + 1 + 2L_A$)	
Word Mark	#	.0115 ($L_I + 9$)	
Modify Address *			
Move Characters to A or B Word Mark	M	.0115 ($L_I + 1 + 2L_W$)	
Move Characters and Edit	E	.0115 ($L_I + 1 + L_A + L_B + L_Y$)	
Move Characters to Record or Word Mark *	P	.0115 ($L_I + 1 + 2L_A$)	
Move Characters and Suppress Zeros	Z	.0115 ($L_I + 1 + 3L_A$)	
Move and Insert Zeros *	X	.0115 ($L_I + 1 + 2L_A + 2L_Z$)	
Move Numeric	D	.0115 ($L_I + 3$)	
Move Zone	Y	.0115 ($L_I + 3$)	
Multiply (aver.) *	@	.0115 ($L_I + 3 + 2L_C + 5L_C L_M + 7L_M$)	
No Operation	N	.0115 ($L_I + 1$)	

APPENDIX 5. IBM 1401: CONFIGURATION ASSUMED IN TEXT

The operations described in the text would require a machine with the equipment listed below.

IBM 1401 Processing Unit with 4000 characters of storage.

Sense Switches

Multiply-Divide feature

Print storage

High-Low-Equal Compare

Indexing Feature

IBM 1402 Card Read Punch

IBM 1403 Printer with 132 printing positions

IBM 1405-1 Disk Storage Unit

IBM 1407 Console Inquiry Station

4 IBM 729-IV Magnetic Tape Units

GLOSSARY

Terminology in the computing field is not yet fully standardized. Every attempt has been made in this book to use terms in their most common meaning, but it should be realized that variations do exist. Furthermore, this listing is not intended to be complete or rigorous; it is intended simply to provide a basic vocabulary.

Absolute Coding. Coding in which instructions are written in the basic machine language, that is, with absolute addresses and actual operation codes.

Accumulator. A storage register where results are accumulated.

Addition Record. A record that results in the creation of a new record in a master file being updated.

Address. A label, name, or number that designates a register, a location, or a device where information is stored; the part of an instruction that specifies the location of an operand.

Address Computation. Computer operations that result in the creation or modification of the address parts of instructions.

Alphameric. Characters that may be either letters of the alphabet, numerical digits, or certain special symbols.

Analog Computer. A computer that represents variables by physical analogies in continuous form, such as amount of rotation of a shaft and amount of voltage. Contrasted to digital computer: the difference is sometimes expressed by saying that an analog computer *measures*, whereas a digital computer *counts*.

Arithmetic Unit. That component of a computer in which arithmetic and logical operations are performed.

Assemble. To translate a routine coded in a symbolic machine language into absolute machine instructions and to assign machine storage for those instructions and for data; usually done by the computer under control of an assembly routine. Distinguished from *compile* by the fact that assembly produces one machine instruction from one symbolic instruction, whereas compiling produces (in general) *many* machine instructions from one pseudo instruction.

Batch Processing. The system of processing in which a number of similar input items are grouped for processing during the same machine run.

Binary Digit. One of the symbols 0 or 1. A digit in the binary scale of notation, usually called a *bit*.

Blank. The character that results in storage from reading an input record such as a card column which contains no punches; the character code in storage that will result in not printing in a given position.

Block. A group of records, words, or characters handled as one unit. Used in this book primarily to denote a group of records on magnetic tape.

Block Diagram. A graphic representation of the logical sequence of procedural steps for processing data. More detailed than a flow chart; a flow chart shows the over-all steps to be performed, whereas a block diagram shows the details of *how* to perform each step.

Blocking. Combining two or more records into one block; usually refers to tape operations.

Branch. A point in a routine where one of two or more choices is selected under control of the routine, that is, a conditional transfer.

Buffer Storage. Any device that temporarily stores information during a transfer of information. From a programming standpoint, refers to a device for matching the speeds of internal computation and an input or output device, thereby permitting simultaneous computation and input or output.

Card Field. A fixed number of consecutive card columns assigned to a unit of information.

Cell. See **Location**.

Chaining. (1) 1401 instruction addresses: a technique of omitting one or both addresses of an instruction with the omitted address being supplied by the previous contents of the corresponding address register. (2) Disk storage: a system of storing records in a disk file in which each record belongs to a chain (group of records) and has a linking field for tracing the chain.

Character. One of a set of elementary symbols which may be arranged in ordered groups to express informa-

- tion; these symbols may include the decimal digits 0 through 9, the letters A through Z, punctuation symbols, special input and output symbols, and any other symbols that a computer may accept.
- Checkout.** The process of determining the correctness of a computer routine, locating any errors in it, and correcting them. Also the detection and correction of malfunction in the computer itself.
- Closed Subroutine.** A subroutine not stored in the main path of the routine. Such a subroutine is entered by a branch operation and provision is made to return control to the main routine at the end of the subroutine.
- Code.** To write instructions for a computer either in absolute or in some other language.
- Collate.** To merge items from two or more similarly sequenced files into one sequenced file without necessarily including all items from the original files.
- Collating Sequence.** The sequence into which the allowable characters of a particular computer are ranked.
- Compare.** To examine the representation of two groups of characters to discover identity or relative magnitude.
- Compile.** To produce a machine language routine by translation from a program written in some non-machine language. See also **Assemble**.
- Compiler.** A special machine language routine used to perform compiling operations.
- Complement.** Usually a complement represents the negative of a quantity. For example, the three-digit tens complement of 026 is 974.
- Computer.** Any device capable of accepting information, processing it and providing the results of these processes in acceptable form. In this text the term is always meant to imply a *stored program digital computer*.
- Console.** A part of the computer where most of the external controls for a computer operation are exercised and where most of the indicators of internal operation are located.
- Control Card.** A card which contains input data or parameters for a specific application of a general routine.
- Control Field.** The field of information by which a record in a file is identified and/or controlled.
- Control Panel.** The panel that uses removable wires to direct the operation of some computers and punched card equipment. Not used in the 1401.
- Control Total.** A sum formed by adding together some field from each record in an arbitrary grouping of records; may have some significance as a number; used for checking machine, program, and data reliability.
- Control Unit.** The portion of the hardware of the computer that directs the sequence of automatic operations, interprets the coded instructions, and initiates the proper signals to the computer circuits to execute the instructions.
- Core Storage.** A form of high speed storage in which information is represented by the magnetization of ferromagnetic cores.
- Data Processing.** A generic term for all operations carried out on data according to precise rules of procedure; a generic term for computing in general as applied to business situations.
- Debugging.** See **Checkout**.
- Deletion Record.** A record that results in the deletion of some corresponding record from a master file.
- Detail File.** A file to be processed against a master file.
- Digital Computer.** A computer in which information is represented in discrete form, such as by one of two directions of magnetization of a magnetic core or by the presence or absence of an electric pulse at a certain point in time. Contrasted with analog computer.
- Document.** Any representation of information that is readable by human beings; usually on paper.
- Edit.** To rearrange information for machine output or input. To prepare for publication, that is, delete, rearrange, select, or insert data as needed.
- Execute.** To carry out an instruction or perform a routine.
- Field.** A set of one or more characters which is treated as a whole; a unit of information.
- File.** A collection of records; an organized collection of information directed toward some purpose.
- File Maintenance.** The processing of a master file required to handle changes in it. Examples: changes in number of dependents in a payroll file, the addition of new checking accounts in a banking application.
- Fixed Word Length.** Refers to a computer in which a computer word always contains the same number of characters. Contrasted with variable word length.
- Flow Chart.** A graphic representation of the sequence of processing operations required to carry out data processing. More general than a block diagram; a flow chart shows the sequence of processing steps, whereas a block diagram shows in detail how to carry out each step.
- Form.** A printed or typed document that usually has blank spaces for the insertion of information.
- Format.** The predetermined arrangement of characters, fields, lines, page number, punctuation marks, etc. Refers to input, output, and file information.
- Generate.** To produce a complete routine from one which is in skeleton form under control of parameters supplied to the generator routine.
- Hardware.** The mechanical, magnetic, electric, and electronic devices from which a computer is constructed.
- Hash Total.** A control total that has no meaning in itself as a number.
- Header Label.** A magnetic tape block at the beginning of a tape, which identifies and describes the information on the tape.
- Home Record.** The first record in a chain of records, using the chaining method of disk file organization.

- Housekeeping.** Operations in a routine which do not directly contribute to the solution of the problem at hand but which are made necessary by the method of operation of the computer. Examples: Loop testing, setting of word marks.
- Index Register.** A register that contains a quantity that may be used to modify addresses automatically (and for other purposes) under direction of the control section of the computer.
- Initialize.** To execute the instructions immediately prior to a loop, which set addresses, counters, data, etc., to their desired initial values.
- Input.** Information transferred from auxiliary or external storage into the internal storage of a computer.
- Instruction.** A set of characters which as a unit causes the computer to perform one of its operations. An instruction may contain one or more addresses according to the number of references to operands in storage contained in the instruction.
- Internal Storage.** Computer storage for data and instructions, from which instructions can be moved directly to the control unit for execution.
- Interpret.** (1) To print on a punched card the information punched in that card. (2) To translate non-machine language to machine language.
- Interpretive Routine.** A routine that decodes instructions written in nonmachine language and immediately executes those instructions, as contrasted with a *compiler*, which decodes the nonmachine language and produces a machine language routine to be executed *at a later time*.
- Key.** See **Control Field**.
- Label.** (1) In SPS programming the symbolic location of a word. (2) In magnetic tape operations a record magnetically recorded on a tape to identify its contents to a computer routine.
- Library.** An organized collection of standard and proven routines and subroutines which may be incorporated in larger routines.
- Linkage.** A technique for providing interconnections between a main routine and a subroutine.
- Location.** A place in storage where a unit of data or of an instruction may be stored.
- Loop.** A coding technique whereby a group of instructions is repeated with modification of some of the instructions within the group and/or with modification of the data being operated on. Usually consists of initialization, computing, modification, and testing, although not necessarily in that order.
- Machine Language.** A language for writing instructions in a form to be executed directly by the computer. Contrasted to symbolic coding languages and to procedure-oriented languages.
- Macro-Instruction.** A machinelike source language statement which can produce a number of machine instructions when compiled.
- Magnetic Disk.** A storage device in which information is recorded on the magnetizable surface of a rotating disk. A magnetic disk storage system is an array of such devices with associated reading and writing heads that are mounted on movable arms.
- Magnetic Drum.** A storage device in which information is recorded on the magnetizable surface of a rotating cylinder.
- Magnetic Tape.** A storage system in which information is recorded on the magnetizable surface of a strip of plastic tape.
- Master File.** A file of semipermanent reference information which is usually updated periodically.
- Memory.** See **Internal Storage**.
- Merge.** To combine items from two or more similarly sequenced files into one sequenced file, including all items from the original files.
- Microsecond.** One millionth of a second.
- Millisecond.** One thousandth of a second.
- Mnemonic Operation Code.** An operation code written in a symbolic notation that is easier to remember than the actual operation code of the machine. Must be converted to an actual operation code before execution, which is done as part of an assembly, interpretive, or compiling routine.
- Object Routine.** The machine language routine which is the output after translation from the source language. The running routine.
- Off-line.** Pertaining to the operation of input or output devices or auxiliary equipment not under direct control of the central processing unit.
- On-line.** Pertaining to the operation of input or output devices under direct control of the computer.
- Open Subroutine.** A subroutine that is inserted directly into a larger routine where needed.
- Origin.** The absolute storage address of the beginning of a program.
- Output.** Information transferred from the internal storage of the computer to output devices or external storage.
- Overflow.** (1) The generation of a quantity beyond the capacity of a register. (2) A record linked to a home record in the chaining method of disk file organization.
- Parameter.** A quantity to which arbitrary values may be assigned; used in subroutines and generators specifying such things as record size, decimal point location, and record format.
- Parity Check.** A checking technique based on making the total numbers of ones in some grouping of binary digits odd (or even). Whenever such a group is read, it is presumed to be correct if the number of ones is still odd (or even).
- Procedure-Oriented Language.** A source language oriented to the description of procedural steps in machine computing.

Processor. A program of instructions that carries out the translation from a source language program to an object program. Includes compilers, assemblers, report program generators, etc.

Program (verb). To plan the method of attack on a specified and defined problem for computer solution. Distinguished from coding by the fact that coding is writing instructions, whereas programming is characterized by the drawing of flow charts.

Program (noun). A group of related routines which solve a given problem.

Pseudo-Instruction. A symbolic representation of information to an assembler or a compiler; not an instruction to the computer, although for convenience it is often written in the same general format as a computer instruction.

Random Access Storage. Storage in which the time required to obtain information is relatively independent of the location of the information most recently obtained.

Read. To transfer information from an input device to internal storage.

Real Time Computation. A data processing arrangement in which the computer is required to be able to supply information to a physical or business activity whenever the information is demanded.

Record. A collection of fields; the information related to one area of activity in a data processing activity; files are made up of records.

Register. A device that can hold information while or until it is used. May consist of core storage.

Report Generation. A technique for producing complete machine reports from information that describes the input file and the format and contents of the output report.

Rewind. To return a tape to its beginning.

Routine. A set of computer instructions that carries out some well-defined function.

Run. One routine or several routines automatically linked so that they form an operating unit during which manual interruptions are not normally required of the computer operator.

Software. All the programming systems required for an effective data processing operation, in addition to the hardware of the computer system itself. Includes assemblers, compilers, and utility routines.

Source Language. The language used to specify computer processing; translated into object language by an assembler or compiler.

Storage. Any device into which information can be transferred, that will hold information, and from which the information can be obtained at a later time.

Stored Program Computer. A computer that can alter its own instructions in storage as though they were data and later execute the altered instructions.

Subroutine. A routine that may be incorporated into a larger routine.

Switch. A symbol used to indicate a branch point or a set of instructions to condition a branch for later execution.

Symbolic Coding. Coding in which instructions are written in nonmachine language. That is, coding using symbolic notation for operators and operands instead of actual machine instruction codes and addresses.

Systems Analysis. The analysis of a business activity to determine precisely what must be accomplished and how to accomplish it.

Trailer Label Block. A block that follows one or more other blocks and contains data pertinent to the preceding blocks.

Transaction File. A file containing current information related to a data processing activity; it is usually used to update a master file.

Update. To modify a master file according to current information, often that contained in a transaction file, according to a procedure specified as part of a data processing activity.

Utility Routine. A standard routine used to assist in the operation of a computer. For instance, a conversion routine, a print out routine, a tape reading routine, etc.

Variable Word Length. Refers to a machine in which the number of characters comprising a computer word is not fixed.

Word. A set of characters which is treated as one unit and has one addressable location.

Working Storage. A portion of internal storage used for input data, immediate results, or output.

Write. To transfer information from internal storage to an output device or to auxiliary storage.

Zero Elimination. The process of eliminating nonsignificant zeroes to the left of significant digits, usually before printing.

BIBLIOGRAPHY

There are many good books available on computing, some of which are listed here. A more complete listing may be found in the IBM Data Processing Bibliography, Form J20-8014-2.

- Gotlieb, C. C., and J. N. P. Hume. *High Speed Data Processing*. McGraw-Hill Book Company, New York, 1958, 11 + 338 pp. Begins with an introduction to data processing and programming ideas, followed by a number of chapters on typical commercial applications.
- Grabbe, Eugene M., Simon Ramo, and Dean E. Wooldridge, editors. *Handbook of Automation, Computation, and Control*. Vol. 2: Computers and Data Processing. John Wiley and Sons, New York, 1959, 1100 pp. Contains a long chapter on the theory of programming, several chapters on typical applications, and long sections on computer design and analog computers.
- Grabbe, Eugene M., editor. *Automation in Business and Industry*. John Wiley and Sons, New York, 1957, 611 pp. Contains information on a wide variety of computer applications as of 1956.
- Gregory, Robert H., and Richard L. VanHorn. *Automatic Data-Processing Systems: Principles and Procedures*. Wadsworth Publishing Company, San Francisco, 1960, 705 pp. Introduces concepts in data processing, computing equipment, programming, and systems design. The coverage is thorough and thoughtful.
- Hein, Leonard W. *Introduction to Electronic Data Processing for Business*. D. Van Nostrand Company, Princeton, 1961. 14 + 320 pp. A general introduction to computer programming, based on the IBM 650. All of the examples are based on commercial applications. Includes chapters on file maintenance, merging and collating, sorting, and report writing.
- Kaufman, Felix. *Electronic Data Processing and Auditing*. Ronald Press, New York, 1961, 180 pp. The title of this book is somewhat misleading. Although the emphasis is on auditing problems, there is a great deal of valuable general information on applications and systems design. Contains many flow charts and block diagrams, many examples of the flow of information through a business organization, and discussions of breaking an application into runs, as well as the treatment of questions of error-checking and proving accuracy.
- Leeds, Herbert D., and Gerald M. Weinberg. *Computer Programming Fundamentals*. McGraw-Hill Book Company, New York, 1961, 368 pp. Designed both for individual and classroom training in computer programming. The IBM 7090 is used as the illustrative computer, but the understanding of its functions and purpose may easily be adapted to any computer course. Heavy emphasis is laid on the basic concepts, ideas, and techniques required for the purpose of communication with computers.
- Martin, E. W., Jr. *Electronic Data Processing: An Introduction*. Richard D. Irwin, Homewood, Illinois, 1961, 423 pp. Ranges from punched-card methods to management problems in introducing and using a computer. The programming section is based on the IBM 650. Contains much valuable information on applications and systems design.
- McCormick, E. M. *Digital Computer Primer*. McGraw-Hill Book Company, New York, 1959, 214 pp. Discusses the components, operation, and organization of a digital computer. Does not discuss programming or applications, except incidentally. The treatment is elementary, requires little mathematics.
- McCracken, D. D. *A Guide to FORTRAN Programming*. John Wiley and Sons, New York, 1961, 88 pp. An introduction to the FORTRAN system, requiring little mathematics.
- McCracken, D. D., H. Weiss, and T. H. Lee: *Programming Business Computers*. John Wiley and Sons, New York, 1959, 510 pp. Discusses computer coding and programming in terms of a hypothetical computer quite different from the IBM 1401. Includes chapters on tape programming methods, program checkout procedures, and sorting methods.
- Murphy, J. S. *Basics of Digital Computers*. John F. Rider Publishers, New York, 1958, Vol. I: 116 pp.; Vol. II: 133 pp.; Vol. III: 136 pp. An extremely readable and profusely illustrated introduction to computer design. Introduces basic programming concepts, but in no depth. For a quick picture of how a computer works, this is a good book.
- Phister, Montgomery, Jr. *Logical Design of Digital Computers*. John Wiley and Sons, New York, 1958, 408 pp. A standard engineering text on the design and operation of computers.
- Richards, R. K. *Arithmetic Operations in Digital Computers*. D. Van Nostrand Co., Princeton, 1955, 397 pp.

Clear explanation of the design and workings of a computer. Considerably more technical than Murphy but less so than Phister.

Wrubel, Marshall H. *A Primer of Programming for Digital Computers*. McGraw-Hill Book Company, New York, 1959, 230 pp. A general introduction to computer programming based on the IBM 650 and several programming systems for it. Many of the examples involve scientific applications, although the mathematical preparation required is not extensive. Discussions of symbolic and automatic coding techniques are included.

IBM Publications

These and other IBM publications are available from local IBM branch offices.

1401 Data Processing System. General Information Manual, form D24-1401-2. A brief introduction to the 1401, with a sketch of some basic computer concepts. Not a complete description of the machine and not intended to teach programming.

1401 Data Processing System. Reference Manual, form A-24-1403-4. A complete description of the programming and operating characteristics of the 1401. Contains few examples; not intended as a teaching text.

IBM 1401 Data Processing System. From Control Panel to Stored Program. General Information Manual, form F20-0208. A brief introduction to the 1401 and to programming concepts, followed by a description of how punched card operations can be implemented on the 1401. Good introduction to programming for the person with punched card experience.

Introduction to IBM Data Processing Systems. General Information Manual, form F22-6517. A manual prepared to provide a basic understanding of computer systems and programmed functions. Used as an introductory text in IBM education programs.

Flow Charting and Block Diagramming Techniques. Reference Manual, form C20-8008-0. Discusses the need for flow charting and block diagramming, introduces the accepted IBM notation, and gives a number of examples.

IBM Charting and Diagramming Template. Form X24-5884-5. Simplifies the drawing of flow charts and block diagrams, as well as providing a convenient source of various reference information on printing and card punching.

IBM 1401 Symbolic Programming System: Preliminary Specifications. Bulletin, form J29-0200-2.

IBM Magnetic Tape Record Characteristics. Form X22-6785.

Report Program Generator for IBM 1401 Card and Tape Systems. Bulletin, form J24-0215.

Autocoder for the IBM 1401: Preliminary Specifications. Bulletin, form J24-1434-1.

Utility Programs for IBM 1401 Tape Systems: Preliminary Specifications. Bulletin, form J29-1411-0.

Utility Programs for IBM RAMAC 1401 Systems: Preliminary Specifications. Bulletin, form J29-1426-0.

COBOL. General Information Manual, form F28-8053-1. A brief introduction to data processing and the COBOL language, followed by a complete definition of the language and several examples.

An Introduction to Information Retrieval. General Information Manual, form E20-8044.

FORTRAN. General Information Manual, form F28-8074.

IBM Data Processing Bibliography. Form J20-8014-2.

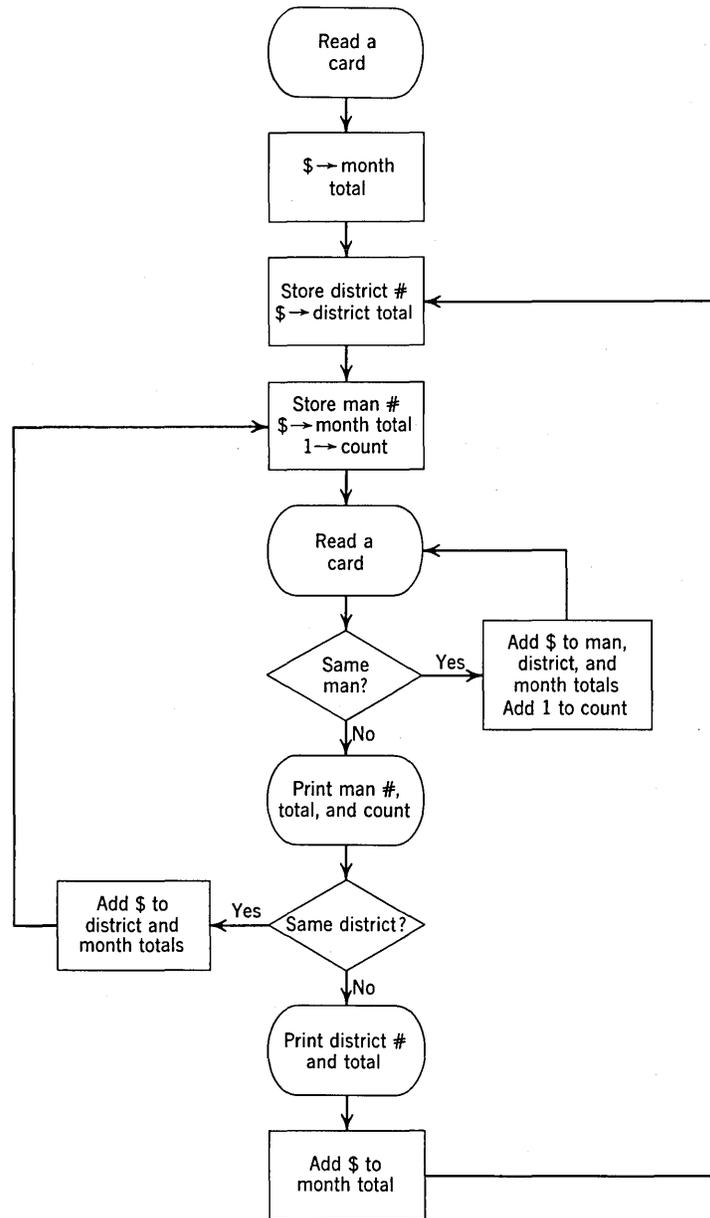
ANSWERS TO SELECTED EXERCISES

It is seldom that there is only one correct answer to a problem in programming. The answers shown here are correct in the sense that they will carry out the

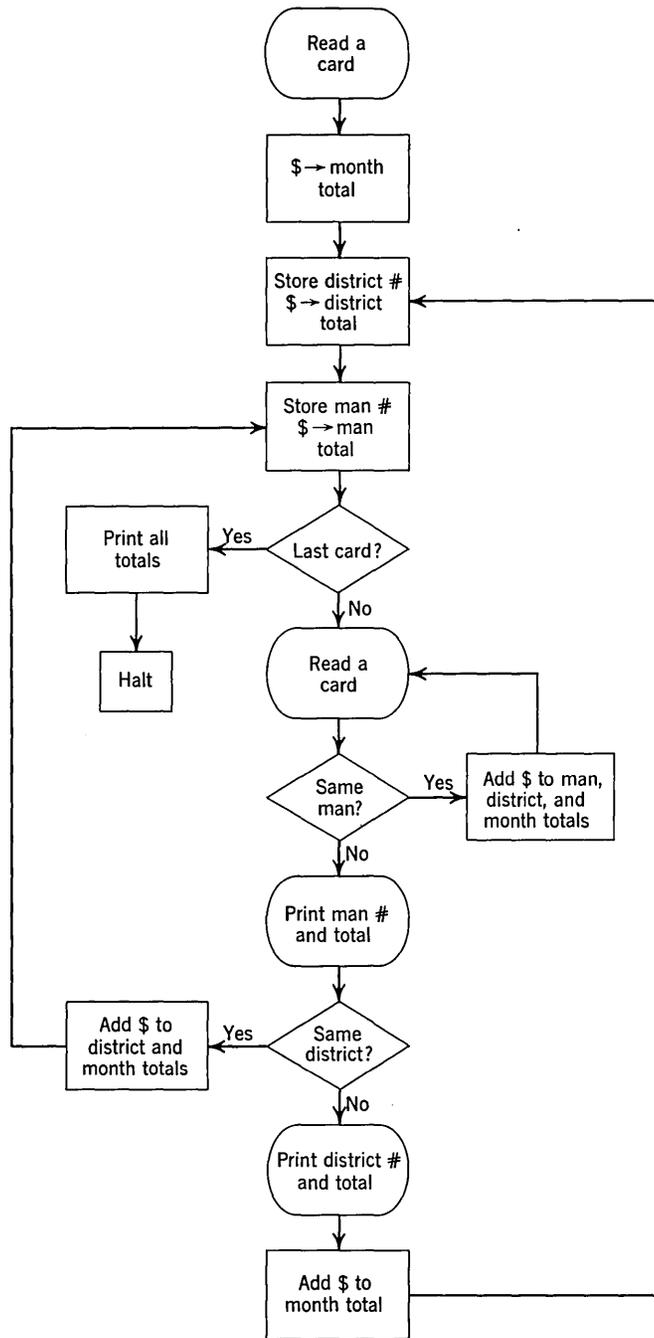
processing required by the problem statement, but there are in most cases many other acceptable solutions.

CHAPTER 1

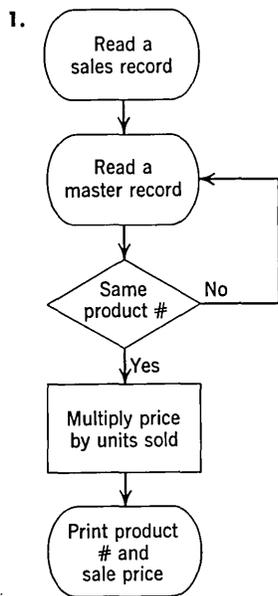
1.



3.



CHAPTER 2



CHAPTER 3

1. 1 2 3 4 5 6 7 2 3 7 6 5 6 3
 ↑ ↑
 800 200

4. 5 0 2 8 6 2 3 2 0
 ↑ ↑
 497 508

6. **IBM** 1401 PROGRAM CHART FORM X24-6437-0
PRINTED IN U.S.A.

Program: _____ Date: _____

Programmer: _____

Step No.	Inst. Address	O P	Instruction			Remarks	Effective No. of Characters		
			A/I	B	d		Inst.	Data	Total
		/	080						
		/	180						
		,	001						
		/							
		M	080	180					
		4							

8. **IBM** 1401 PROGRAM CHART FORM X24-6437-0
PRINTED IN U.S.A.

Program: _____ Date: _____

Programmer: _____

Step No.	Inst. Address	O P	Instruction			Remarks	Effective No. of Characters		
			A/I	B	d		Inst.	Data	Total
		/	080						
		/	299						
		/	332						
		,	001	010					
		,	015	019					
		,	023	230					
		M	008	208					
		M	012	222					
		A	018	235					
		A	022	235					
		A	026	235					
		2							

CHAPTER 4

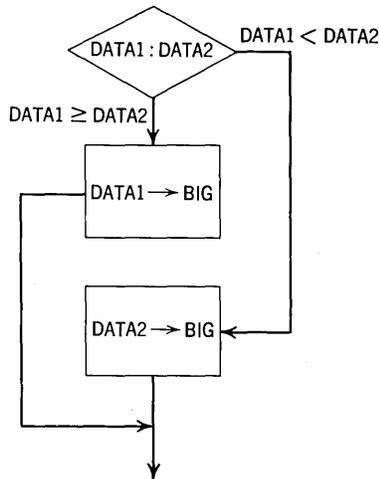
1. START 0333
 REPEAT 0349
 RØUND 0400
 TØTAL 0411
 READI 0010
 PRINTI 0209

3.

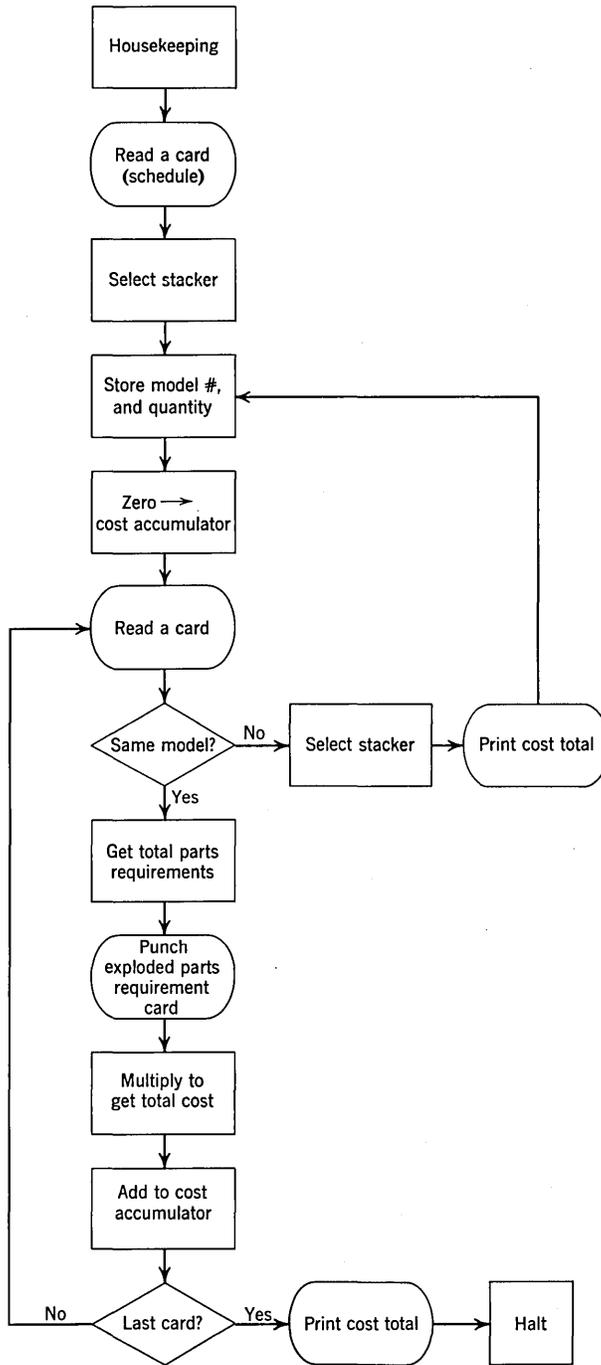
PG	LIN	CT	LABEL	OP	A OPERAND	B OPERAND	D	LOC	INSTRUCTION
1	010			ORG	0500				
1	020	4	ABC	CS	0080			0500	/ 080
1	030	4		CS	0299			0504	/ 299
1	040	4		CS	0332			0508	/
1	050	7		SW	A1 -004	A2 -004		0512	, 001 006
1	060	7		SW	A2 -004	B1 -003		0519	, 911 207
1	070	1	BCD	R				0526	1
1	080	7		A	A1	TOT		0527	A 005 571
1	090	7		A	A2	TOT		0534	A 010 571
1	100	7		S	A3	TOT		0541	S 015 571
1	110	7		A	HALFD	TOT -001		0548	A 572 570
1	120	7		MCS	TOT -002	B1		0555	Z 569 210
1	130	4		W	BCD			0562	2 526
1	140		A1	DS	0005			0005	
1	150		A2	DS	0010			0010	
1	160		A3	DS	0015			0015	
1	170		B1	DS	0210			0210	
1	180	6	TOT	DCW	*		000000	0571	
1	190	1	HALFD	DCW	*		5	0572	
1	200			END	ABC				/ 500 080

CHAPTER 5

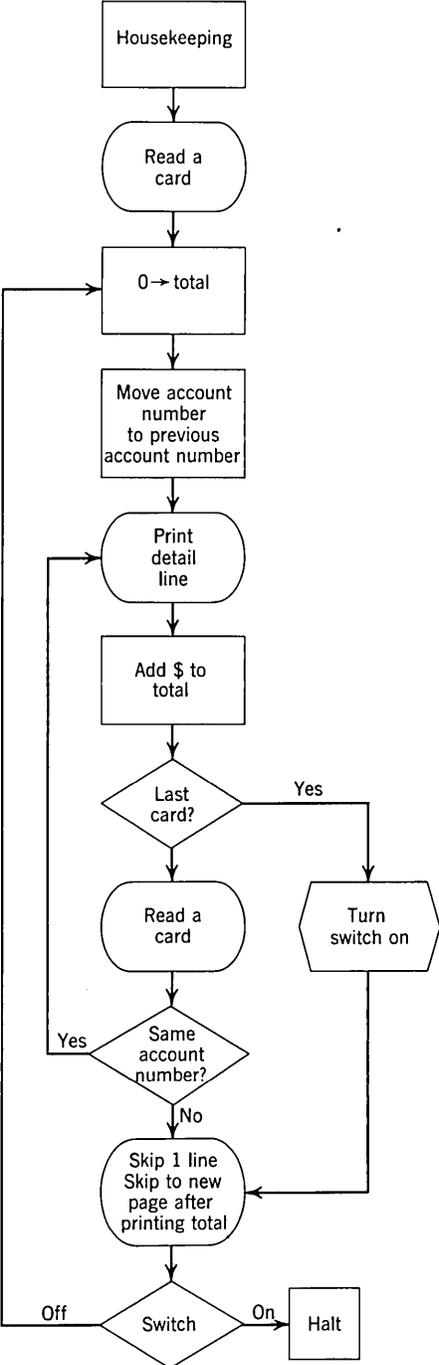
1.



6.



3.



3. (Continued)

LINE	COUNT	LABEL	OPERATION	(A) OPERAND			(B) OPERAND			COMMENTS						
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±		CHAR. ADJ.	IND.				
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0 1 0		C	MCW	SWON							SWITCH				SET SWITCH ON	
0 2 0			B	D												
0 3 0	06	ZEROS	DCW*		0.00	0.00										
0 4 0	06	TOTAL	DCW*													
0 5 0	10	EDIT	DCW*			0										
0 6 0	05	PRACTNØ	DCW*													
0 7 0		SWON	DCW*				N									
0 8 0			END	START												

LINE	COUNT	LABEL	OPERATION	(A) OPERAND			(B) OPERAND			COMMENTS						
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±		CHAR. ADJ.	IND.				
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0 1 0		START	SW	0.001							0.023				HOUSEKEEPING	
0 2 0			R													
0 3 0	A		MCW	ZEROS							TOTAL				INITIALIZE ACCUM	
0 4 0			MCW	0.005							PRACTNØ				PREV. ACCT. NØ.	
0 5 0	B		MC	S 0.005							0.205				ACCT. NØ.	
0 6 0			LCA	EDIT							0.216				SET UP EDIT	
0 7 0			MC	E 0.023							0.216					
0 8 0			W													
0 9 0		A		0.028							TOTAL				ACCUM. AMOUNT	
1 0 0		B		C											ALAST CARD Q	
1 1 0			R												NØ	
1 2 0			C	0.005							PRACTNØ				SAME ACCT. NØ. Q	
1 3 0			B	B											YES	
1 4 0	D		CC												JNØ - SPACE 1 LINE	
1 5 0			LCA	EDIT							0.216				SET UP EDIT	
1 6 0			MC	E TOTAL							0.216					
1 7 0			CC												ASET UP SKIP	
1 8 0			W												PRINT AND SKIP	
1 9 0		SWITCH	B	A												
2 0 0			H	*							-0.03					

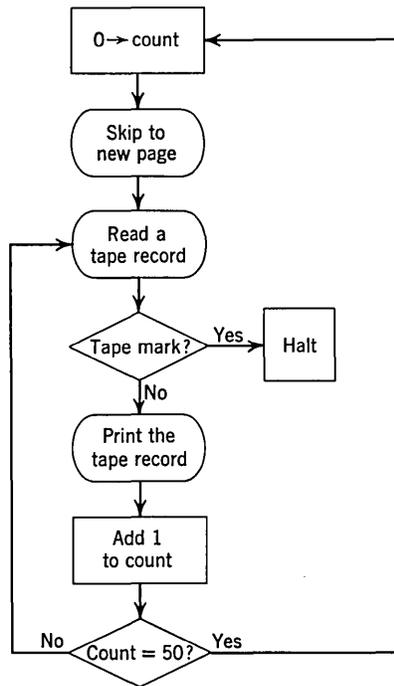
8. The housekeeping is so small a fraction of the total job time as to be completely negligible. The most important consideration in estimating this job is whether the processing between cards can be done within the 10 ms of processing time; if not, the card reader will slow down to 400 cards/min. In this case, however, the processing requires less than 1 ms, so there is no problem. The reading of a five-card group will take $5 \times 75 = 375$ ms. The printing of a summary line for a group requires 100 ms, but to this must be added the waiting time until another starting

point in the card-reading cycles is reached. Since one starting point will have been passed, we must wait for another, so that the reading of another card after printing will not begin until 150 ms after the end of reading the card before printing. The effective time for reading five cards and printing the summary line is therefore $375 + 150 = 525$ ms. With 2000 groups of five cards each, the total time to do the job is thus

$$2000 \times 525 \text{ ms} = 1,050,000 \text{ ms} = 1050 \text{ sec} = 17.5 \text{ min}$$

CHAPTER 8

1.



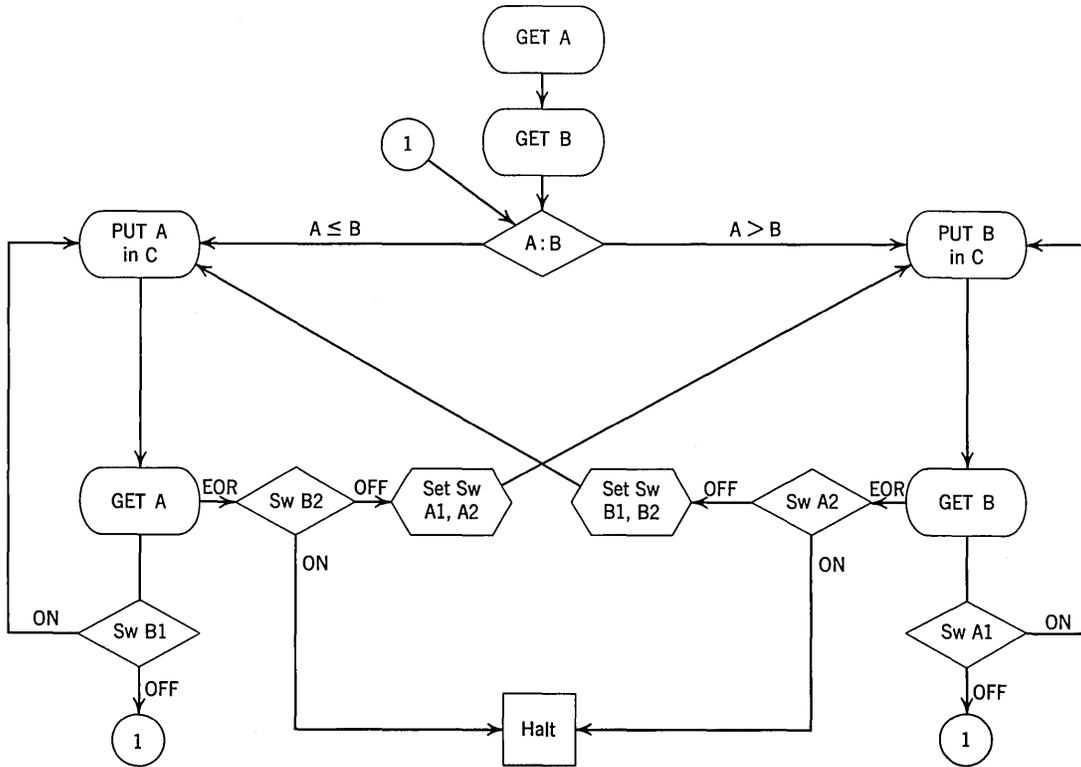
Page No. 1 of 2

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d	COMMENTS			
				ADDRESS	±	CHAR. ADJ.	IND.	ADDRESS	±	CHAR. ADJ.	IND.					
3	5	6	7	8	13	14	16	17	23	27	28	34	38	39	40	55
0	1	0	START	MCW	ZEROS					COUNT						INIT. COUNTER
0	2	0		CC												SKIP TO NEW PAGE
0	3	0	READ	MCW	%UI					PRINT						READ FROM TAPE I
0	4	0		B	HALT											END OF REEL Q
0	5	0		MCW	BLANK					PRINT	+0.50					NO. ERASE GROUPM
0	6	0		W												PRINT
0	7	0		ADD	ONE					COUNT						
0	8	0		C	COUNT					FIFTY						PAGE FULL Q
0	9	0		B	READ											NO
1	0	0		B	START											YES
1	1	0	HALT	H	HALT											
1	2	0	02 ZEROS	DCW	*				00							
1	3	0	02 COUNT	DCW	*											
1	4	0	PRINT	DS	0201											
1	5	0	01 BLANK	DCW	*											
1	6	0	01 ONE	DCW	*				1							
1	7	0	02 FIFTY	DCW	*				50							
1	8	0		END	START											
1	9	0														

5. In the path coming out of the box "Compare Part Numbers" labeled $M > T$, insert a test to insure that the transaction code is a zero. If it is, set up the transaction information in master-record format, PUT the record in the new master tape, read another card, and return to the comparison. If the code is not zero,

write a bad code message. If such an addition record should happen to have the same part number as a master record in the master file, the classification code test will prevent writing it. Modifying the program should not be difficult; in the interest of space, it is not shown.

7.



Line	Label	Operation	OPERAND
0.1	START	OPEN A, B, C	
0.2		GET A	
0.3		GET B	
0.4	COMP	C AEMPNO, BEMPNO	COMPARE EMP NUMBERS
0.5		BL BPUT	BRANCH IF B IS LOW
0.6	A PUT	PUT A, C	
0.7		GET A	
0.8	B1	NO P A PUT	SWITCH B1
0.9		B COMP	
1.0	B2	NO P HALT	END OF REEL ENTRY FOR A
1.1		MCW BRANCH, A1	SET SWITCHES A1
1.2		MCW BRANCH, A2	AND A2, GO TO BPUT
1.3	B PUT	PUT B, C	
1.4		GET B	
1.5	A1	NO P B PUT	SWITCH A1
1.6		B COMP	
1.7	A2	NO P HALT	END OF REEL ENTRY FOR B
1.8		MCW BRANCH, B1	SET SWITCHES B1
1.9		MCW BRANCH, B2	AND B2
2.0		B A PUT	
2.1	HALT	H HALT	
2.2	BRANCH	DCW @B@	
2.3		END START	
2.4			

CHAPTER 9

1.

IBM FORM X24-1350-1
PRINTED IN U.S.A.

Program _____

Programmed by _____ 1401/1410 AUTOCODER CODING SHEET Identification 76 of 80

Date _____ Page No. 1 of 2

Line	Label	Operation	OPERAND
3	5	15	20 25 30 35 40 45 50 55 60 65 70
0.1		MCW	TWØ, MULT+6 SET UP AND
0.2		MPY	KEY, MULT MULTIPLY KEY BY 0.2
0.3		ADD	BASE, MULT-1 ADD BASE ADDRESS
0.4		MCW	MULT-1, DISKAD-1 SET UP SEEK ADDRESS
0.5		SD	DISKAD-7 SEEK TRACK
0.6		RD	DISKAD-7 READ DISK
0.7		?	
0.8		?	
0.9		?	
1.0	TWØ	DCW	2
1.1	BASE	DCW	50000
1.2	MULT	DCW	0000000
1.3	DISKAD	DCW	00000000. GROUP MARK TO RIGHT
1.4			

3.

IBM FORM X24-1350-1
PRINTED IN U.S.A.

Program _____

Programmed by _____ 1401/1410 AUTOCODER CODING SHEET Identification 76 of 80

Date _____ Page No. 1 of 2

Line	Label	Operation	OPERAND
3	5	15	20 25 30 35 40 45 50 55 60 65 70
0.1		SW	KEY-2, KEY-5 COMPUTE
0.2		MCW	KEY-6, WORK TRACK
0.3		ADD	KEY-3, WORK ADDRESS
0.4		ADD	KEY, WORK
0.5		SW	WORK-2 3 DIGITS ONLY
0.6		MCW	WORK, DISKAD-2 MOVE TO DISK ADDRESS
0.7		CW	WORK-2
0.8		SD	DISKAD-7 SEEK TRACK
0.9		RD	DISKAD-7 READ TRACK ZERO
1.0		CW	KEY-2, KEY-5
1.1		MCW	FIRST, ABC INITIALIZE LOOP ADDRESS
1.2	ARC	C	0000, KEY FIND MATCHING KEY
1.3		BE	FIND
1.4		ADD	TEN, ABC+3 MODIFY ADDRESS
1.5		B	ABC NO TEST-ASSUME WILL BE FIND
1.6	FIND	SW	ABC+1 SET UP ADDRESS
1.7		MCW	ABC+3, DEF+3 OF SECTOR
1.8		CW	ABC+1 NUMBER
1.9		SW	DISKAD-1 SET UP
2.0		MCW	0000, DISKAD-1 TRACK
2.1		CW	DISKAD-1 ADDRESS
2.2		RD	DISKAD-5 READ CORRECT TRACK
2.3			

INDEX

- A-address, 33
- A-address register, 40, 41
- A-register, 41
- Absolute address, 47, 48
- Access arm, 123
- Accumulator, 10, 68
- Actual address, *see* Absolute address
- Add instruction, 43
- Address, 31, 32, 39, 47, 111
 - disk storage, 125, 132
- Address modification, 75, 79
- Address part of instruction, 33
- Algebraic operations, 44, 68
- Ampersand, 90
- Assembly, 50
- Assembly listing, *see* Post listing
- Augmented operation code, 111
- Autocoder, 47, 111, 126, 151, 167

- B-address, 33
- B-address register, 40, 41
- B-box, 84
- B-register, 40, 41
- Backspace Tape instruction, 107, 108
- Batch processing, 10, 122
- Binary digit, 29
- Bit, 29, 103
- Blank (character), 34, 36, 90
- Block, magnetic tape, 103, 112
- Block count, 112
- Block diagram, 3, 7, 8, 69
- Blocking, of tape records, 112
- Blocking factor, 112, 120
- Body of control word, 90
- Branch instruction, 48, 49, 57, 62, 63
- Branch If Character Equal instruction, 119
- Branch If Indicator On instruction, 44, 62, 64, 66, 94, 108, 129
- Branch If Word Mark and/or Zone instruction, 67, 68
- Brush, 23
- Buffering, 97

- Calculation, 2
- Card codes, 170

- Card column, 13
- Card loader, 101
- Card punch, 7
- Card punching, 18, 95, 161
- Card reading, 17, 33, 95, 153, 161
- Card to disk routine, 134
- Card to tape, 20
- Card verifier, 7
- Carriage control, 64, 92
- Central processing unit, 16
- Chain Addition Program, 134
- Chaining, disk storage, 132
 - of instruction addresses, 42
- Chain Loading Program, 134
- Chain Maintenance Program, 134
- Chain printing mechanism, 18, 19
- Channel, carriage tape, 92
- Character, 14, 29, 32
- Character adjustment, 49
- Character coding, 29, 103, 104, 170
- Character density, magnetic tape, 105
- Character rate, magnetic tape, 105
- Checking, *see* Error checking
- Checkout, program, 3, 146
- Clear Disk Storage routine, 134
- Clear Storage instruction, 36, 37
- Clear storage program, 101
- Clear Word Mark instruction, 34, 35
- Closed subroutine, 100
- CLOSE macro-instruction, 112, 114, 161
- COBOL, 154, 156
- Coding, 3, 150, 154
- Collating, 3, 71
- Collator, 5, 24
- Color stripe, 14
- Column, card, 13
- Comma, 90
- Comments, 59, 159, 160
- Compare instruction, 66, 67, 68
- Comparison, 64, 66
- Compatibility, 21
- Compiler, 111
- Computation section of loop, 79
- Conditional branch, 62, 154, 158
- Console, 1401 Processing Unit, 17, 101
- Control card, 101

- Control Carriage instruction, 92, 93, 96
- Control field, 2, 10
- Control level, 10
- Control panel, 15, 26
- Control section, 23, 40
- Control total, 8, 20, 113, 141
- Control Unit instruction, 107
- Conversion, file, 4, 141, 148
- Core, 29
- Corner cut card, 14
- Count field, 49
- Credit amount, 90

- D-character, 33, 41, 62, 111
- DA macro-instruction, 116, 120
- Data division, 154, 157, 159
- Data processing, 2
- DC pseudo-instruction, 52
- DCW pseudo-instruction, 49
- Deblocking, 112
- Decimal point, 90
- Decision, 3, 33, 151
- Decision table, 151
- Define Area macro-instruction, 116, 120
- Define Constant, 52
- Define Constant with a Word Mark, 49
- Define IOCS macro-instruction, 112
- Define Symbol, 50
- Define Tape File macro-instruction, 112, 113, 118
- Density, magnetic tape, 105
- Desk checking, 146
- Detail file, 5
- DIOCS macro-instruction, 112
- Disk file utility routines, 134
- Disk storage, 122
- Disk storage addressing, 125, 132
- Disk storage instructions, 125
- Disk-to-card routine, 132
- Disk-to-tape routine, 134
- Documentation, 149
- Dollar sign, 90
- DS pseudo-instruction, 50
- DTF macro-instruction, 112, 113, 118
- Dual level sensing, 21

- E-phase, 41, 75
- Early card read, 95
- Editing, 2, 20, 89
- Effective address, 86
- End-of-reel spot, 64, 105
- END pseudo-instruction, 50, 54, 101
- Environment division, COBOL, 157
- Equal compare, 64, 66
- EQU instruction, 116
- Error checking, 17, 18, 28, 64, 103, 141
- Error routine, 114
- Execute phase, 41, 75
- Execution of instructions, 41, 75
- Expanded print edit, 91
- Extension, of an amount, 6, 11, 26
- External storage, 31

- Field, 14, 31
- File, 2
 - magnetic tape, 106
- File conversion, 4, 141, 148
- File protection ring, 106
- Fixed length record, 112
- Fixed word length, 31
- Flow chart, 3, 7, 151
- Format, 89
- Form control, 92
- Form design, 89, 146
- FORTRAN, 152
- Full track read, 126
- Function, FORTRAN, 154

- GET macro-instruction, 112, 114, 161
- Grocery wholesaler example, 135
- Group mark, 106, 110, 111, 116, 126

- Halt instruction, 66, 70
- Hardware, 13
- Hash total, 113
- Head, magnetic tape, 104
- Header label, 113
- Heading line, 94
- High compare, 64, 66
- High-Low-Equal compare device, 66
- High-Order position, 39, 51
- Hole count, 18
- Home record, in disk storage, 132
- Horizontal check character, 104

- I-address register, 41
- I-phase, 41, 75
- IBM 83 Card Sorter, 23
- IBM 85 Collator, 24
- IBM 729 Tape Unit, 21, 105
- IBM 1301 Disk Storage Unit, 122, 125
- IBM 1401 Data Processing System, 1, 16
- IBM 1402 Card Read Punch, 17, 94, 95
- IBM 1403 Printer, 18, 92, 96
- IBM 1405 Disk Storage Unit, 21, 122
- IBM 1407 Console Inquiry Station, 22, 23

- IBM 7330 Tape Unit, 21, 105
- IBM Charting and Diagramming Template, 7
- IBM punched card, 13
- Indexed disk storage, 133
- Indexing accumulator, *see* Index register
- Index register, 84
- Initialization Section of loop, 79
- Input/Output Control System, 110, 111, 113, 161
- Instruction, 31, 33, 38, 63, 75, 163
- Instruction phase, 41, 75
- Internal storage, 31
- Interpreter, 14
- Interrecord gap, 103, 105
- Inventory control example, 116, 127, 139, 152, 161
- IOCS, *see* Input/Output Control System

- Key, *see* Control field
- Keypunch, 7

- Label, tape, 113
- Label table, 53
- Last card switch, 63, 64, 68
- Line number, 51
- Linkage, subroutine, 100, 160
- Listing, 2
- Literals, 111, 159, 160, 161
- Load button, 101
- Load Characters to A Word Mark, 54
- Load point, magnetic tape, 105
- Location, 31
- Location counter, 53
- Loop, 75, 78
- Low compare, 64, 66
- Low-order position, 32, 51

- Machine-oriented language, 154
- Macro-instruction, 106, 111, 113
- Magnetic core, 29, 32
- Magnetic tape, 18, 23, 103
- Master file, 2, 19, 116
- Master file creation, 141, 148
- Memory, 31
- Memory dump routine, 101, 147
- Merging, 3
- Minus sign, 30, 90
- Mnemonic Operation code, 48, 111, 163, 167
- Modification section of loop, 79
- Move Characters and Edit instruction, 55, 58, 90
- Move Characters and Suppress Zeros instruction, 63, 65, 89
- Move Characters to A or B Word Mark instruction, 33, 34
- Multiply instruction, 55, 56

- Negative quantity, 90
- No-Operation instruction, 78

- Normal punch pocket, 18
- Normal read pocket, 18
- Numerical bits, 29
- Numerical punch, 13

- Object program, 47, 50, 52, 112, 154
- OP-register, 40
- OPEN macro-instruction, 112, 113, 161
- Open subroutine, 100
- Operating instructions, 149
- Operation code, 33, 48
- ORG pseudo-instruction, 53
- Origin, 53
- Overflow, 64
- Overflow record, in disk storage, 132

- Page heading, 94
- Page number, 51
- Parallel operation, 148
- Parity bit, 29
- Parity checking, 21, 30, 103
- Partial chaining, 42
- Parts explosion and summary example, 70
- Pass, 24, 53
- Payroll example, 54
- Phase, of instruction execution, 41
- Photo-sensing markers, magnetic tape, 105
- Picking sequence, warehouse, 135, 136
- Pilot operation, 148
- Plus sign, 30
- Post listing, 50, 147
- Print area, 36
- Printer carriage control, 92
- Printing, 18, 96, 161, 154
- Print storage, 64, 98
- Procedure design, 1
- Procedure division, COBOL, 154, 157
- Procedure-oriented language, 154
- Procedures manual, 149
- Processing time, 95
- Processing unit, 16, 23, 40
- Processor, 47, 50, 52
- Program, 3, 31
- Program checkout, 146
- Program identification, 51
- Program loading, 101
- Programming, 3, 151
- Program switch, 77, 116, 117
- Pseudo-instruction, 52
- Punch a Card instruction, 35, 36, 95
- Punch area, 35
- Punch feed read, 18
- Punch release, 95
- Punch start time, 95
- PUT macro-instruction, 112, 114, 161

- RAMAC, 123
- Random access file processing, 10
- Random access storage, 4, 10, 21, 23, 122

- Read a Card instruction, 35, 36, 93, 94
- Read and Punch instruction, 97
- Read area, 35
- Read Disk instruction, 126, 127
- Read release, 95
- Read start time, 95
- Read Tape instruction, 107
- Record, 2
 - magnetic tape, 103, 112
- Record count, 112
- Reel, magnetic tape, 103, 106
- Reflective spot, magnetic tape, 105
- Register, 40
- Relative sensitivity level, 104
- Report, 89, 154, 156
- Report file, 2
- Report Program Generator, 154
- Restart procedure, 113
- Rewind, magnetic tape, 105, 108
- Rewind Tape and Unload instruction, 109
- Rewind Tape instruction, 108, 109
- Rounding, 49
- Routine, *see* Program
- Run manual, 149

- Sales statistics example, 4, 11
- Sector, disk storage, 123
- Seek Disk instruction, 125, 126
- Select Stacker instruction, 72
- Sense switch, 64
- Sequential access file, 4
- Sequential file processing, 4, 10, 122, 143
- Set Word Mark instruction, 34, 35
- Sign, 30, 43, 68
- Skip and Blank Tape instruction, 108, 109
- Skipping, of form, 92, 96
- Sorter, 5, 23
- Sorting, 2, 5, 10, 19, 23, 71, 116, 122, 142
- Source data, 2, 7
- Source program, 47, 50, 52, 112, 152, 154, 157

- Spacing, of form, 92, 96
- Speed, *see* Timing
- SPS, *see* Symbolic Programming System
- Stacker selection, 18, 28, 71
- Stacking, 7, 18
- Start and Stop time, magnetic tape, 105
- Start button, 63
- Status portion of edit word, 90
- Storage, 31
- Storage of instructions, 38
- Storage print routine, 101, 147
- Store B-Address Register instruction, 100
- Stored program computer, 75
- Subroutine, 100, 160
- Subtract instruction, 44
- Summarization, 2
- Summary line, 94
- Switch, program, 77, 116, 117
- Symbolic address, 47
- Symbolic instruction, 52
- Symbolic Programming System, 47, 52, 101, 151, 154, 163
- System analysis and design, 1, 28, 151

- Tape, carriage, 92
- Tape blocking, 112
- Tape-controlled carriage, 64, 92
- Tape head, 104
- Tape label, 113
- Tape mark, 105, 108
- Tape to card, 20
- Tape-to-disk routine, 134
- Tape unit, 105
- Template, charting and diagramming, 7
- Test data, 147
- Testing, program, 3, 146
- Testing section of loop, 79
- Three-character address, 32, 48, 53
- Timing, input and output, 17, 94, 99, 171
 - instruction, 99, 171

- Timing, magnetic tape, 105, 171
 - program, 98
- Tracing, 148
- Track, disk storage, 123
- Trailer label, 112
- Transaction file, 2, 4
- Transition card, 54, 101
- Two-gap head, 21, 104

- Unconditional branch, 49, 62, 154, 159
- Unequal compare, 64, 66
- Unit record equipment, 10
- Unmatched detail, 5, 27
- Unmatched master, 5, 71
- Updating, of files, 2
- Utility programs, 100, 101, 134

- Variable instruction length, 33
- Variable length record, 112
- Variable word length, 31
- Verifier, 7

- Wholesale grocery example, 135
- Word, 31
- Word mark, 29, 30, 32, 33, 37, 39, 62, 70, 101, 106, 127, 137
- Work-flow chart, 7
- Working storage, 112
- Write a Line instruction, 36, 96, 98
- Write and Branch instruction, 49
- Write and Punch instruction, 97
- Write and Read instruction, 96
- Write Disk Check instruction, 126, 129
- Write Disk instruction, 126, 128
- Write, Read, and Punch instruction, 97
- Write Tape instruction, 106, 107
- Write Tape Mark instruction, 107, 108

- Zero and Add instruction, 80
- Zero suppression, 55, 62, 89
- Zero zone, 13
- Zone bits, 29, 43
- Zone punch, 13

A GUIDE TO FORTRAN PROGRAMMING

DANIEL D. McCracken, McCracken Associates, Inc.

This book provides a quick, effective way to master Fortran, an efficient and widely used computer language. It provides a grasp of Fortran programming that will enable you to solve scientific or engineering problems with a computer—no matter how limited your understanding of the computer itself and its operation. 1961. 88 pages.

PROGRAMMING BUSINESS COMPUTERS

DANIEL D. McCracken, McCracken Associates, Inc., and

HAROLD WEISS and TSAI-HWA LEE,

both of General Electric Company, Phoenix.

An introductory book that covers programming from analysis to coding with an emphasis on business data processing, written by people who have had first hand experience in procedures, analysis, programming, coding, machine operation, teaching, and machine design. While suitable for teaching basic programming, it has a wealth of organized material on advanced techniques. 1959. 510 pages.

DIGITAL COMPUTER PROGRAMMING

DANIEL D. McCracken

Here Mr. McCracken gives a general introduction to computers plus the practical details necessary to work with specific machines. Carefully avoiding technical jargon, he clears up many of the points that are especially troublesome to newcomers to automatic computing and gives the basic elements necessary to a sound understanding of current and future computer programming. 1957. 253 pages.

JOHN WILEY & SONS, INC., 440 Park Avenue South, New York 16, N. Y.

Kroch's & Brentano's

5.75 PM06

Chicago • Evanston • Old Orchard