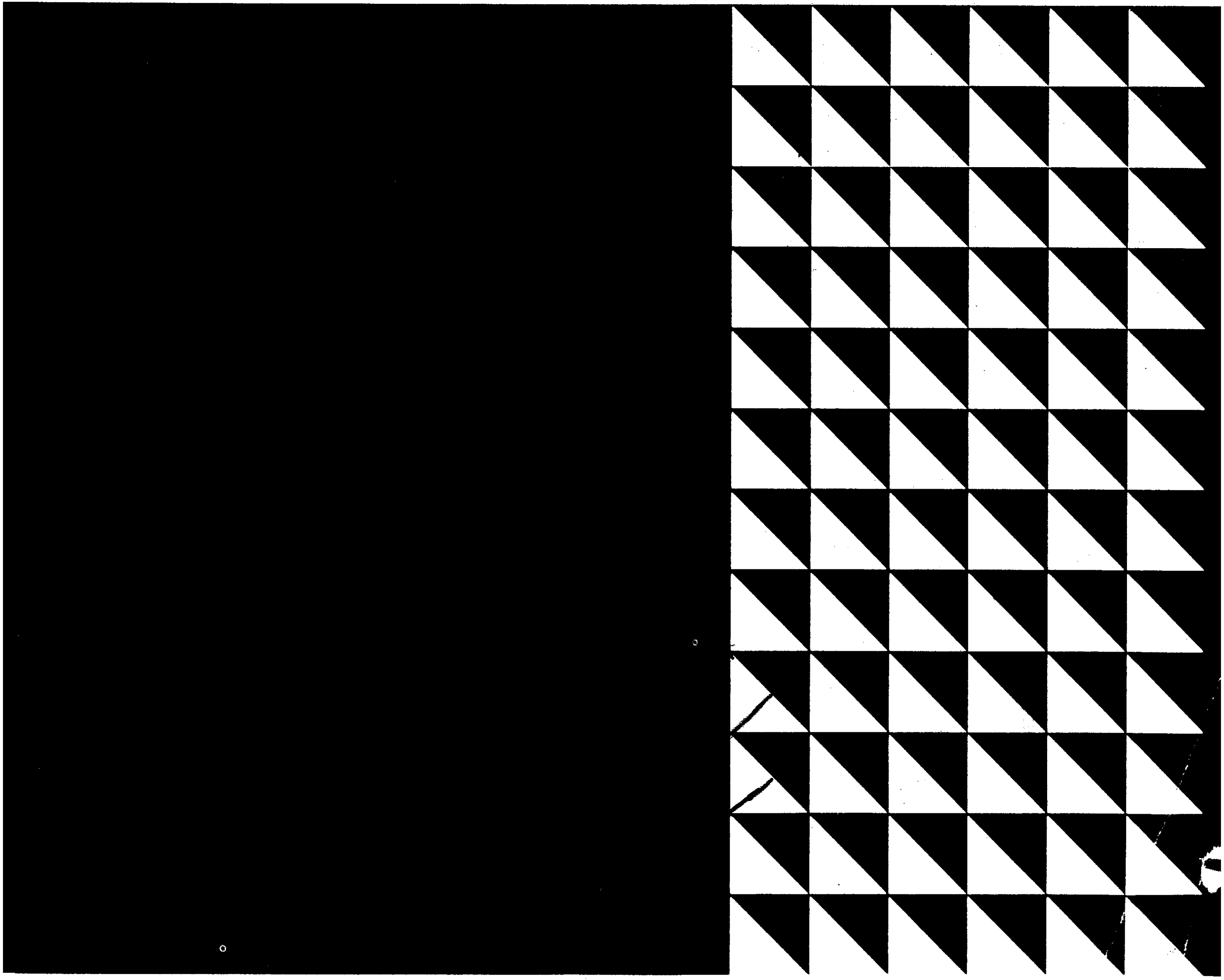
A Programmer's Introduction to IBM System/360 Assembler Language



Student Text

IBM

A Programmer's Introduction to IBM System/360 Assembler Language

Student Text

Minor Revision (August 1970)

This publication is a minor revision of Form SC20-1646-5 incorporating corrections made on pages dated (8/70). The original publication is not obsoleted.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. Address comments concerning the contents of this publication to IBM Corporation, DPD Education Development - Publications Services, Education Center, South Road, Poughkeepsie, New York 12602.

Preface

This student text is an introduction to System/360 assembler language coding. It provides many examples of short programs shown in assembled form. Some elementary programming techniques and the specific instructions illustrated in the programs are discussed in simple, relatively nontechnical terms. Much of the text is based on information in *IBM System/360 Principles of Operation* (GA22-6821). This includes a brief review of relevant System/360 concepts and descriptions of selected assembler language instructions for arithmetic, logical, and branching operations. Standard (fixed-point), decimal, and floating-point arithmetic are discussed. The book also includes an elementary introduction to assembler language and the assembler program, and chapters on base register addressing and on program linkages and relocation. The coding of many other common programming techniques, such as the use of branches, loops, and counters, is shown. The use of macro instructions is demonstrated, but not covered in detail. Program flowcharting and input/output operations are beyond the scope of the book.

The publication is a sampler rather than a comprehensive textbook. It is intended for supplementary reading for the student in a regular course of study on System/360 assembler language coding, and for the novice programmer. In general, the reader will find that the program examples are quite simple at the beginning of each chapter, or major subject division, and become progressively more complex. If the going seems difficult, it is suggested that he simply skip to the next subject and come back later.

The student should have access to two IBM System/360 System Reference Library (SRL) manuals for reference purposes: the *Principles of Operation* and the assembler specification manual for one of the System/360 operating systems. (All publications and their form numbers are listed at the end of the Preface.) He should also be familiar with fundamental concepts of data processing and the basic operating principles of System/360. Two IBM programmed instruction (P.I.) courses, or their equivalent, are prerequisite to a full understanding of this student text: *Computing System Fundamentals* and *Introduction to System/360*. The student who is not enrolled in a comprehensive programming course will find the P.I. book *Fundamentals of Programming* a valuable guide to problem analysis and program flowcharting.

The text and programs of this book have been revised throughout, mainly to reflect changes in programming conventions attributable to the development of System/360 operating systems. Chapter 1 is new, and several sections in other chapters have been entirely rewritten. The sample programs have been reassembled under the widely used Disk Operating System (DOS). As far as possible, usages limited to DOS have been avoided, and the programs and text in general are applicable to System/360 models 25, 30, 40, 50, 65, and 75, under any of the operating systems.

IBM publications that may be useful to the student are:

IBM System/360 Principles of Operation (SRL manual GA22-6821)

IBM System/360 Reference Data (card GX20-1703)

IBM System/360 System Summary (SRL manual GA22-6810)

Number Systems (Student Text GC20-1618)

Introduction to IBM System/360 Architecture (Student Text GC20-1667)

Introduction to System/360 (P.I. Course GR29-0256 through -0259)

Computing System Fundamentals (P.I. Course GR29-0280 through -0282)

Fundamentals of Programming P.I. Course SR29-0019)

System/360 Assembler Language Coding (P.I. Course SR29-0231 through -0235)

The form numbers of the assembler specification manuals for the various System/360 programming systems are:

Basic Programming Support (Tape System)–GC24-3335
Basic Operating System–GC24-3361
Tape Operating System } GC24-3414
Disk Operating System }
Operating System–GC28-6514

Contents

FIGURES

Assembly Listings of Programs

Other Figures Useful for Reference

TABLES

Chapter 1: Introduction

WHAT IS ASSEMBLER LANGUAGE?

Machine Language

A computer is a willing servant. It will invariably and reliably do exactly what it is told to do, as long as it is told in its own language. This is true of any computer. Let's take a quick look at the language that System/360–the machine itself–understands.

If an IBM System/360 computer is given the instruction 1B67, it will subtract whatever amount is in register 7 from the amount in register 6. When the operation is finished, the contents of register 7 will be the same as they were originally, but the contents of register 6 will be the difference between the two original quantities. The code 1B signifies to the computer (1) just what operation it is to perform, (2) what format it can expect the two quantities to be in, and (3) whether they are in registers or in main storage. Specifically, 1B indicates that the computer is to subtract two 32-bit binary numbers, both of which are in registers. The two quantities to be operated on are called *operands*. The one that is written first is called the first operand and in this case is in register 6. The second operand is in register 7.

The instruction 1B67 is in *machine language*. It is a representation in the hexadecimal number system (base of 16) of the actual binary arrangement in the computer. The computer responds to it in a particular way because its circuitry has been designed to do so whenever it senses this combination of signals.

Let's take another example of a machine language instruction, say 5A20B02A. The *operation code* 5A causes the computer to add two 32-bit binary numbers (the first in a register and the second in main storage) and to place the result in the first operand location. In this case, the first operand is in register 2, and the second operand is in main storage, beginning at the location designated by 0B02A.

Not many years ago all programs were written in machine language. The most valuable tool the programmer had was an eraser. He was concerned with an enormous amount of clerical detail. He had to remember dozens of numerical codes for the computer operations and try not to make a mistake when using them. He had to keep track of the storage space he used for instructions, data, and work areas, and actually calculate any addresses he needed to refer to in his program. Revising a program (a very frequent occurrence then, as it is now), often meant changing every address that followed the revisions. All this detail meant many errors and much time spent on checking, calculating, keeping tables, and other clerical tasks.

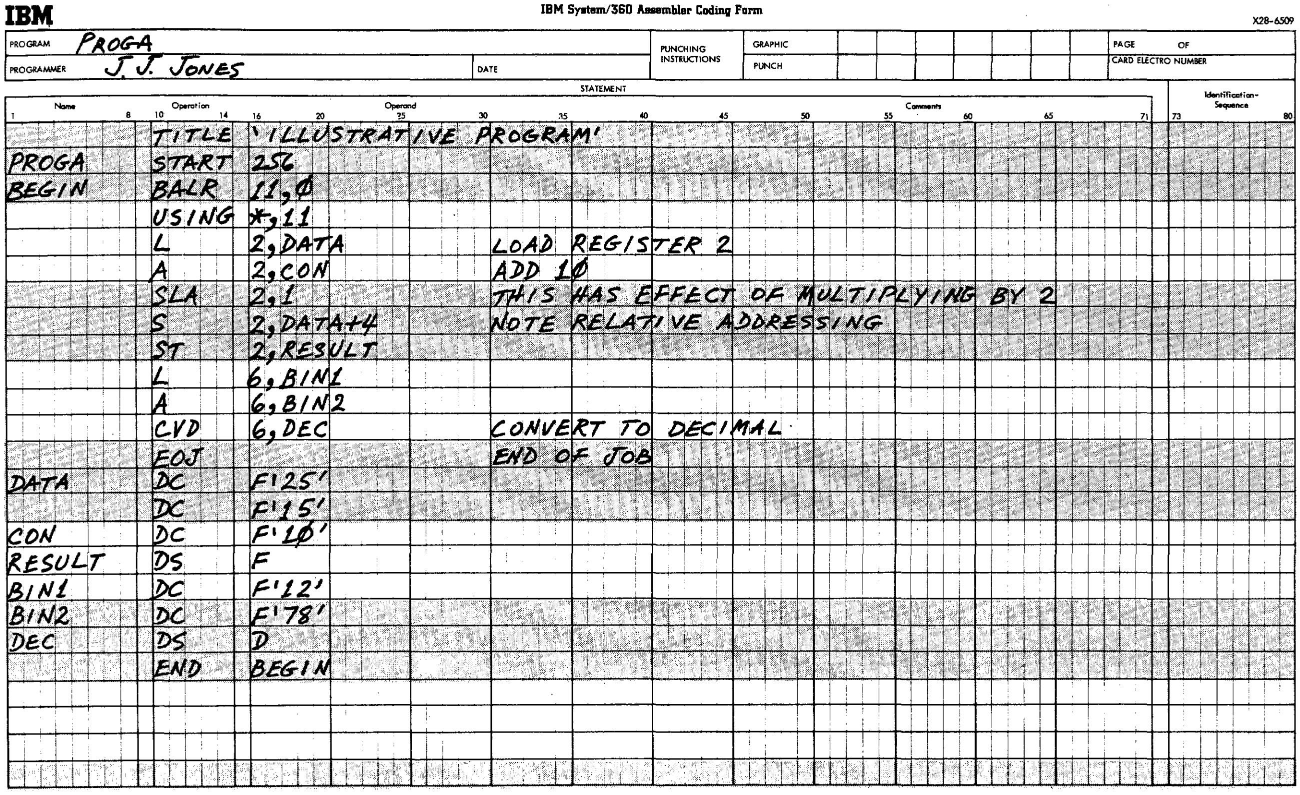
Assembler Language

The realization that the computer itself was better suited than man for doing this type of clerical work led to the development of assembler languages (each computer has its own assembler language). In System/360 assembler language, every operation code is written in alphabetic letters that are easy to remember, called mnemonics, and the addresses of locations in storage can be given symbolic names like PAY, HOURS, and RATE by the programmer. The machine language instruction 1B67 would be written in assembler language as SR 6,7 (SR for Subtract Register). The instruction 5A20B02A might be A 2,CON (A for Add), with another instruction to define CON as a certain value. We do not have to say where it is–the computer will take care of that. An assembler language program as prepared by a programmer is shown in Figure 1-1. The operations to be performed start in column 10, the operands in column 16.

As we said at the beginning, however, the computer cannot understand any language except its own machine language. Therefore, a *program* that translates our symbolic program into machine language or *object code* is needed. Such a program, actually a component part of an IBM System/360 operating system, is brought from the system "library" into a separate area in main storage when needed, and it does the job. This program is called an *assembler*. Besides translating the problem program statements into machine language, it calculates storage locations for all instructions, keeps track of any symbols like CON that are used, and performs a number of other necessary functions. The program written by the programmer is not executed during the assembly process; it will be executed later, after further processing. Figure 1-2 shows the listing produced by the assembler for our sample program.

Machine Instructions

All the columns to the left of the statement number (STMT) column are in machine language. The LOC, ADDR1, and ADDR2 columns have to do with address arithmetic handled by the assembler, and will be discussed later. The heart of our program has been translated into the code headed OBJECT CODE. The circled area at the left contains the code for every *executable* instruction in the entire program. What we mean by an executable instruction is one that, when the problem program is run, will tell the computer to perform an actual operation in the machine itself. Each of the executable instructions has a corresponding System/360 machine operation code; these operation codes



IBM IBM System/360 Assembler Coding Form X28-6509

PROGRAM PROGA
PROGRAMMER J. J. JONES
DATE
PUNCHING INSTRUCTIONS GRAPHIC PUNCH
PAGE OF
CARD ELECTRO NUMBER

STATEMENT
Name 1 8 | Operation 10 14 | Operand 16 20 25 30 35 40 45 50 55 | Comments 60 65 71 | Identification-Sequence 73 80

```
         TITLE 'ILLUSTRATIVE PROGRAM'
PROGA    START 256
BEGIN    BALR  11,0
         USING *,11
         L     2,DATA        LOAD REGISTER 2
         A     2,CON         ADD 10
         SLA   2,1           THIS HAS EFFECT OF MULTIPLYING BY 2
         S     2,DATA+4      NOTE RELATIVE ADDRESSING
         ST    2,RESULT
         L     6,BIN1
         A     6,BIN2
         CVD   6,DEC         CONVERT TO DECIMAL
         EOJ                 END OF JOB
DATA     DC    F'25'
         DC    F'15'
CON      DC    F'10'
RESULT   DS    F
BIN1     DC    F'12'
BIN2     DC    F'78'
DEC      DS    D
         END   BEGIN
```

Figure 1-1. An assembler language program as prepared by the programmer

are represented by the first two characters (the first two hexadecimal numbers, really) in the circled object code. In the example, the executable instructions include one of the branching instructions (BALR, op code 05), Load (L, op code 58), Add (A, op code 5A), one of the Shift Left instructions (SLA, op code 8B), Subtract (S, op code 5B), Store (ST, op code 50), and so on. In assembler language, the executable instructions are called *machine instructions*.

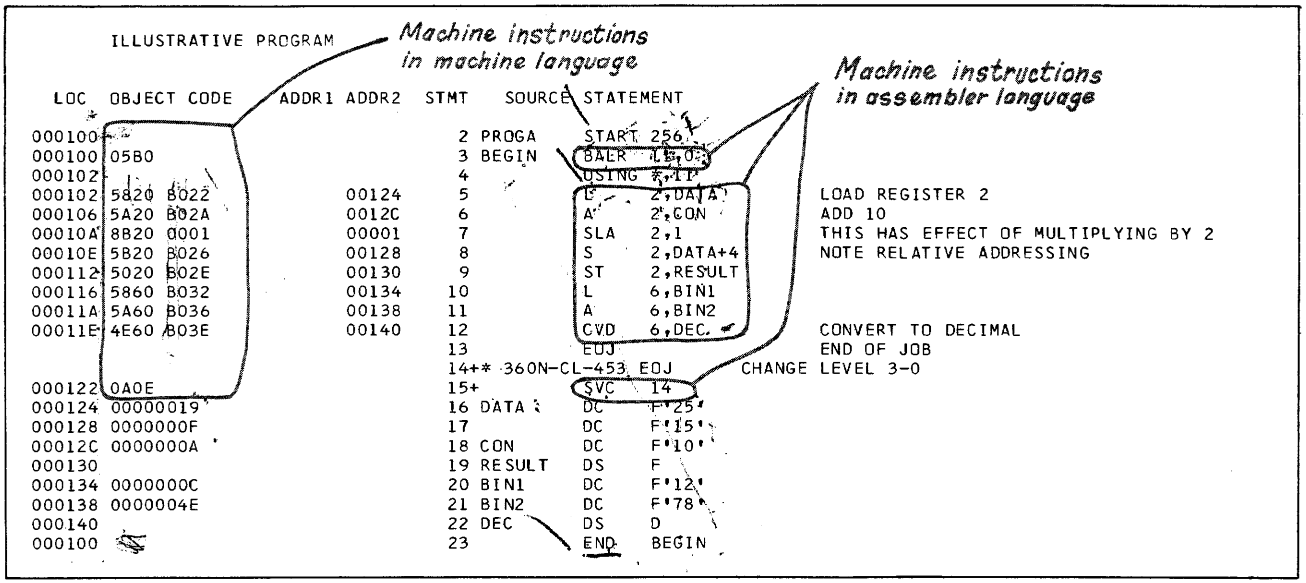
Not counting floating-point arithmetic instructions, System/360 assembler language has about 100 different machine instructions. It is fairly easy to recognize and remember all of the mnemonics for them—certainly easier than remembering the machine language operation codes. Some other examples are C for Compare, CVD for Convert to Decimal, SH for Subtract Halfword, STH for Store Halfword, M for Multiply, and BC for Branch on Condition. A full list of System/360 machine instructions appears in the Appendix; floating-point instructions are given in the chapter on that subject. Each machine instruction and what it does is described in complete detail in the IBM Systems Reference Library (SRL) manual *IBM System/360 Principles of Operation* (A22-6821). Many will be described in this bock in nontechnical language, but not in complete detail.

Assembler Instructions

What about the TITLE, START, and USING instructions that have not generated any object code in the assembly listing in Figure 1-2? The mnemonic TITLE does not even show up at all (it was in the source program), but we see that the assembly listing has the heading ILLUSTRATIVE PROGRAM. TITLE is an instruction to the assembler that tells it to print a heading or title at the top of each page in the listing. Similarly, START and USING are instructions to the assembler; these concern the addressing plan it is to follow. Although they will affect the way in which the assembler assigns addresses, they will have no direct function in the execution of the problem program. In contrast to machine instructions, they are called *assembler instructions*. They may be defined as instructions to the assembler program itself.

Skipping the EOJ for the moment, we see the mnemonics DC (Define Constant) and DS (Define Storage). These two instructions are also assembler instructions. DC's generate object code for the values they define, but no operation codes. DS's actually reserve storage space of a specific size, but they too do not generate operation codes. In other words, DC's cause the assembler to create object code for actual values and DS's reserve actual storage spaces, but they do not themselves give rise to any action during program execution. Instead, they are used for either information or space by other instructions in the program. If we look again at the assembly listing, we see that DATA, CON, RESULT, etc., are operands of some of the executable instructions.

Assembler-instruction mnemonics, which are also listed



```
                    ILLUSTRATIVE PROGRAM

  LOC   OBJECT CODE    ADDR1 ADDR2  STMT    SOURCE STATEMENT

000100                                 2 PROGA    START 256
000100 05B0                            3 BEGIN    BALR  11,0
000102                                 4          USING *,11
000102 5820 B022             00124     5          L     2,DATA            LOAD REGISTER 2
000106 5A20 B02A             0012C     6          A     2,CON             ADD 10
00010A 8B20 0001             00001     7          SLA   2,1               THIS HAS EFFECT OF MULTIPLYING BY 2
00010E 5B20 B026             00128     8          S     2,DATA+4          NOTE RELATIVE ADDRESSING
000112 5020 B02E             00130     9          ST    2,RESULT
000116 5860 B032             00134    10          L     6,BIN1
00011A 5A60 B036             00138    11          A     6,BIN2
00011E 4E60 B03E             00140    12          CVD   6,DEC             CONVERT TO DECIMAL
                                      13          EOJ                     END OF JOB
                                      14+* 360N-CL-453 EOJ          CHANGE LEVEL 3-0
000122 0A0E                           15+         SVC   14
000124 00000019                       16 DATA     DC    F'25'
000128 0000000F                       17          DC    F'15'
00012C 0000000A                       18 CON      DC    F'10'
000130                                19 RESULT   DS    F
000134 0000000C                       20 BIN1     DC    F'12'
000138 0000004E                       21 BIN2     DC    F'78'
000140                                22 DEC      DS    D
000100                                23          END   BEGIN
```

Machine instructions in machine language

Machine instructions in assembler language

Figure 1-2. Assembly listing of the program in Figure 1-1. The executable instructions (see text) are circled in both assembler language and the machine language translation.

in the Appendix, generally suggest their purpose. USING indicates a particular register to be used by the assembler for keeping track of storage addresses, EJECT tells the assembler to start a new page in the program listing, and END to terminate the assembly program. Assembler instructions and the functions of the assembler program are described fully in each of the SRL assembler language manuals for the various IBM operating or programming support systems (see Preface for list). It should be explained that variations of the System/360 assembler program are available for different operating systems and sizes of computers. Basically, they all work similarly, but some are more flexible and versatile than others. Many differences do exist, however, in the input/output (I/O) programming for different systems. Largely for this reason, the subject of I/O will not be covered in this book.

Macro Instructions

In an entirely different category, System/360 assembler language includes another type of instruction, called a *macro instruction* or macro. If a programmer writes a series of instructions for a routine that will be needed again and again during the program, he does not have to code the entire sequence each time. He can make up his own code name to represent the sequence, and, by using his code word in a single statement whenever it is needed, he can cause the sequence of instructions to be assembled and inserted. Incorporated in the system library, the sequence can also be used in entirely separate programs and by all programmers associated with a computer installation simply by writing one statement in the source program. The mnemonics used for macro instructions are often unique to an installation. Some macros are prepared and supplied by IBM; they have mnemonics like EOJ, READ, WRITE, OPEN, CLOSE, WAIT, WAITF, DTFCD, DTFIS, etc. The mnemonics for both the user-prepared and the IBM-supplied macros constitute an extension to System/360 assembler language.

The macros supplied by IBM are mainly for procedures that affect other components of the IBM operating system, like the supervisor and the input/output control system, and they ensure accuracy and consistency in maintaining the interrelations within the operating system. The EOJ (End of Job) in the program example is a supervisor macro instruction. It generates just two statements, which are indicated in the listing by plus signs. The first is simply for identification, and the second is the executable Supervisor Call instruction (SVC, op code 0A).

Most I/O routines are long and complicated, and for any particular device and operating system are programmed in exactly the same way in program after program. Most of the macros supplied by IBM are for these I/O routines. Some of the Disk Operating System (DOS) macro instructions we shall use in this book, besides EOJ, are CALL, SAVE, RETURN, and PDUMP. The book does not cover the preparation of new macros, but shows, in the chapter on subroutines, another method for reusing a sequence of instructions. However, the programmer can save much time and effort by using the macros that are already available in his system library. Their use will also ensure accuracy and standardization of frequently repeated procedures.

Summary

To summarize, these are the three kinds of instructions used in System/360 assembler language, and what each does:

1. *A machine instruction* specifies an actual operation

to be performed by the computer when the object program is executed. The operation may be arithmetic, or the comparison, movement, or conversion of data, or performing a branch. The instruction generates executable object code.

2. *An assembler instruction* specifies an instruction to the assembler program itself and is effective only at assembly time. It does not generate executable object code.

3. *A macro instruction* specifies a sequence of machine and assembler instructions to perform a frequently needed routine. The machine instructions generate executable object code.

Why Learn Assembler Language?

The most important single thing to realize about assembler language is that it enables the programmer to use all System/360 machine functions as if he were coding in System/360 machine language. Of all the programming languages, it is closest to machine language in form and content. The high-level languages such as FORTRAN, COBOL, and PL/I are problem-oriented rather than machine-oriented. Their languages are much like English or mathematical notation. Depending on what is involved, one statement in these languages may be compiled into a series of two or eight or fifty machine language instructions. The problem-oriented languages have the advantage of letting the programmer concentrate on what he wants to accomplish and not on how it is to be done by the computer, and they may save considerable time in programming, program modification, and program testing. Choice of a programming language in any given situation usually involves weighing the cost of programming time against the cost of machine time. A complex mathematical problem that can be run in a few minutes and will be run only once is a very different situation from a program that runs for several hours and will be repeated every week.

Here we can appreciate one of the important advantages of assembler language over the high-level languages: its efficient use, in the hands of a skillful programmer, of computer storage and time. High-level languages produce generalized routines so that a wide range of data processing needs can be met with a minimum of programming effort. A routine can be written in assembler language exactly to fit some particular data processing need, thus saving storage space and execution time.

As we shall see in the course of this book, there are often many ways of accomplishing the same data processing results. Sometimes the overall programming requirements of a computer installation strain its capacity. If the particular problem arises of either not enough main storage space or not enough processing time, the problem may be solved by assembler language. In such a situation, its flexibility permits the programmer to choose those programming techniques that will provide just the kind of economy needed—time or space.

A knowledge of assembler language has some important benefits for a programmer working in a high-level language. It can be helpful to him in analyzing and debugging programs. It also enables him to include certain assembler language routines in his program to meet special systems or other requirements.

THE ASSEMBLER PROGRAM

The System Environment

As a first step in the assembly process, the handwritten problem program has to be put into a form that can be read by the computer system. Punched cards are frequently used; they are convenient and easy to substitute in case of error. The program is punched by a keypunch operator, each line on a separate card. The original program and these cards are called the *source program*, or the cards may be called the *source deck*. The assembler program is loaded into main storage and executed, using the source deck as input.

It is important to realize that the basic function of the assembler is to translate the source program. It does not execute the program. The final output of the assembler program is called the *object program*. It contains the machine language equivalent of the source program, and is put on cards, tape, or disk by a system output device. It is this object program that will later be subjected to further processing and will itself be executed. The assembler output also includes several listings to aid the programmer, which are produced by a line printer. Figure 1-3 shows the assembly process in outline.

Before going into detail about the functions of the assembler, it may be helpful to look at the overall system environment into which a programmer-written problem program goes. As we already know, the assembler program is a component of the IBM operating system. It functions under the control of another, very important component, the control program. (To avoid confusion in terminology, perhaps it should be mentioned that the control program is often referred to as the control system. The supervisor is one element of the control program, and the most powerful. The job control program is another element.)

The System/360 control program is, in effect, a traffic director. It supervises the movement of data, the assignment of all the devices attached to the system, and the scheduling of jobs. Working under a set of priorities for various kinds of situations, it handles the flow of operations in the central processing unit (CPU), with the aim of keeping it constantly busy and the entire system at its most productive level. The control program sees to it that needed IBM processing programs, like the assembler program and the linkage editor program, are brought from the system library and loaded into main storage at the right time. These two kinds of programs combined—that is, the control program and the processing programs—make up what is called the IBM *operating system* (or, for smaller installations, the IBM *programming support system*). With an operating system at work, the programmer is relieved of practically all concern about having on hand for either processing or execution of his problem program the system resources available at his installation.

Functions of the Assembler

During execution of the assembler program, the assembler scans the source program statements a number of times. Its first activities are to process any macro instructions it finds, and to store the complete sequences of individual instructions generated by the macros. They are then standing by, ready to be inserted into the assembled problem program at the points indicated by the programmer. Afterwards, the assembler proceeds to translate the one-for-one assembler language statements into machine instructions.

① Programmer writes source program, named PROGA, on coding sheets.

② Keypunch operator copies PROGA source program on cards.

③ Assembler language translator program is loaded into main storage.

④ PROGA source program is read into a work area of the assembler program.

⑤ Assembler program is executed, using PROGA source program as input data.

⑥ Output of assembler program is PROGA object program and assembly listings. Object program may be on cards, tape, or disk.

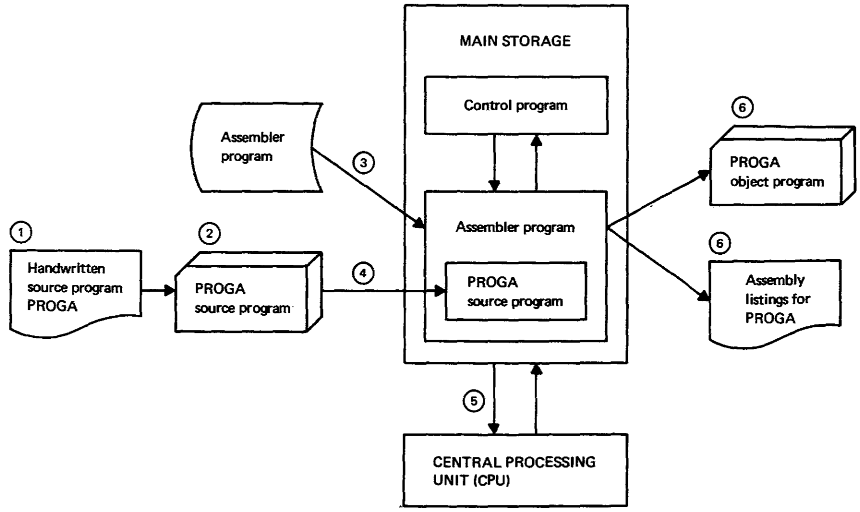


Figure 1-3. Assembly of a problem program, PROGA. Note that PROGA is not executed during the assembly process.

Briefly, here is how the assembler works. It reads source statements as input data, checking for errors and flagging them for further processing. At first, it translates the parts of the input (such as operation codes) that do not need further analysis or calculation. Meanwhile, it constructs a table of all the symbols used, in which it collects, as it goes along, such information as each symbol's length, its value or location, and the statements in which it is referred to. From this table and other analyses of the source statements, the assembler can then assign relative storage addresses to all instructions, constants, and storage areas. It uses a location counter for this purpose (see LOC column in Figure 1-2). It does all the clerical work involved in maintaining the base register addressing scheme of System/360 computers. During its operations, the assembler continues to note errors and to resolve any it can.

As shown in Figure 1-3, there are two kinds of output from the assembler program. The primary output is the object program in machine language; included with it is certain information tabulated by the assembler, which is needed for relocating the program to an actual main storage location and for establishing links with separate programs. (This information will later be passed on to the linkage editor for the next step in the processing.) The other output from the assembler is a series of printed listings that are valuable to the programmer for documentation and analysis of his program:

1. The listing of the program (samples of these will be shown throughout this book) includes the original source program statements side by side with the object program instructions created from them. Most programmers work from this assembly listing as soon as it is available, hardly ever referring to their coding sheets again.

2. Probably next in interest to the programmer is the diagnostics listing, which cites each statement in which an error condition is encountered and includes a message describing the type of error.

3. The cross-reference listing shows the symbol table compiled by the assembler.

4. The external symbol dictionary (ESD) describes any references in the problem program needed for establishing links with separate programs. It is possible for the programmer to combine his program with others, or to use portions of separate programs, or to make certain portions of his program available to other programs. The ESD is part of the tabular information passed on to the linkage editor. It always contains at least the name of the problem program, its total length, and its starting address on the assembler's location counter.

5. The relocation dictionary (RLD) describes the address constants that will be affected by program relocation. This list is also passed on to the linkage editor.

We have now reached the end of the assembly process. What happens next? Our object program is in relocatable form, but it will not be executable until it has been processed by the linkage editor.

Final Processing

The *linkage editor* program is another component of the IBM operating system. Its functions, which will not be described fully here, can provide great flexibility and economy in the use of main storage. The linkage editor also makes it possible for a long and complicated program to be divided into separate sections, which can be programmed, assembled, and debugged by different programmers, and then linked together to be executed. The linkage editor is loaded into main storage and operates as a separate program under control of the control program, just as the assembler did. Input to the linkage editor may be a single assembled program or many separate programs. The linkage editor works on one after the other, building up composite dictionaries of ESD and RLD data to resolve all references between individual programs and to set up necessary linkages. It also searches the system library and retrieves any programs referred to. It relocates the individual programs as necessary in relation to each other, assigns the entire group to a specific area of main storage, and modifies all necessary address constants to the relocated values of their symbols.

After completion of the link-editing, our problem program can be loaded into main storage and executed under supervision of the control program. Unless specified otherwise, each machine instruction is executed in sequence, one after the other. If there is separate input data, it can be brought in by I/O instructions in the program. Output—the results of program execution—also requires I/O instructions.

The scope of this book does not go beyond the assembly process. For a clear understanding of the detailed program examples, however, it is essential for the reader to be able to visualize at just what stage in the entire process each action occurs. For this reason, the complete process from programmer-written program to its final execution has been outlined in this section.

USE OF THE CODING FORM

Assembler language programs are usually written on special coding forms like the one in Figure 1-1, which will be repeated here for convenience. Space is provided at the top for program identification and instructions to keypunch operators, but none of this information is punched into cards.

The body of the form is completely keypunched in corresponding columns of 80-column cards. Use of the Identification-Sequence field (columns 73 – 80) is optional and has no effect in the assembled program. Program identification and statement sequence numbers can be written in part or all of the field. They are helpful for keeping the source cards in order and will also appear on the assembly listing. Indeed, the programmer can use an assembler instruction (ISEQ) to request the assembler to check the input sequence of these numbers.

The statement field is for our program instructions and comments, which are normally limited to columns 1 – 71. Each statement can be continued on one or more lines, depending upon which assembler program is used. A statement consists of:

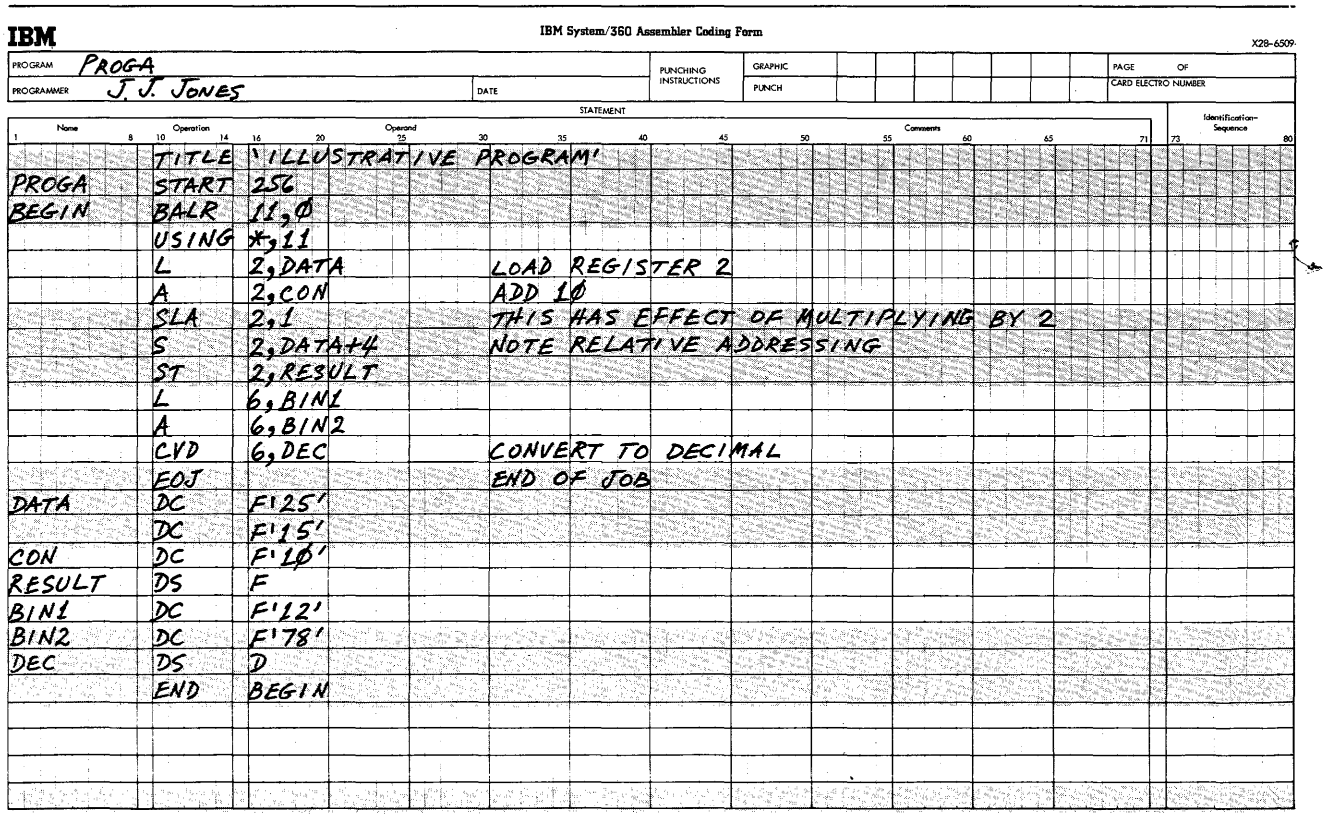
1. A name entry (sometimes)
2. An operation entry (always)
3. An operand entry (usually)
4. Any comment we wish to make

It isn't necessary to use the spacing shown on the form, since the assembler permits nearly complete freedom of format. However, lining up entries as shown makes it simpler to read a program, and following the form permits the programmer to painlessly observe the few essential rules required by the assembler.

Some of these rules are as follows. (1) The entries must be in proper sequence. (2) If a name is used, it must begin in column 1. (3) The entries must be separated by at least one space, because a space (except in a comment or in certain terms enclosed in single quotes) is the signal to the assembler that it has reached the end of an entry. (4) Spaces must not appear within an entry, except as noted. (5) A statement must not extend beyond the statement boundaries, normally columns 1 – 71.

We have been using that word "normally" because the programmer can override the specific column designations by an ICTL (Input Format Control) assembler instruction, which can specify entirely different begin, end, and continuation columns. A statement is normally continued on a new line in column 16, with some character (often an X) inserted in column 72 of the preceding line. Since the normal spacing is generally the most convenient and is easiest for a keypunch operator to follow, we shall use the spacing indicated on the form throughout this book.

The purpose of using a name in a statement is to be able to refer to it elsewhere. It is a symbol of eight characters or



IBM IBM System/360 Assembler Coding Form X28-6509

PROGRAM PROGA

PROGRAMMER J. J. JONES DATE

PUNCHING INSTRUCTIONS GRAPHIC PUNCH

PAGE OF CARD ELECTRO NUMBER

STATEMENT

Name 1 8 | Operation 10 14 | Operand 16 20 25 30 35 40 45 50 | Comments 55 60 65 71 | Identification-Sequence 73 80

```
         TITLE 'ILLUSTRATIVE PROGRAM'
PROGA    START 256
BEGIN    BALR  11,0
         USING *,11
         L     2,DATA         LOAD REGISTER 2
         A     2,CON          ADD 10
         SLA   2,1            THIS HAS EFFECT OF MULTIPLYING BY 2
         S     2,DATA+4       NOTE RELATIVE ADDRESSING
         ST    2,RESULT
         L     6,BIN1
         A     6,BIN2
         CVD   6,DEC          CONVERT TO DECIMAL
         EOJ                  END OF JOB
DATA     DC    F'25'
         DC    F'15'
CON      DC    F'10'
RESULT   DS    F
BIN1     DC    F'12'
BIN2     DC    F'78'
DEC      DS    D
         END   BEGIN
```

Figure 1-1. An assembler language program as prepared by the programmer less, created by the programmer. It may identify a program, a location in storage, a specific value, or a point in the program to which the programmer may plan to branch. As we know, the assembler compiles a symbol table, keeping track of where each name is defined and where each reference to it appears. These references occur when the name is used as an operand in an instruction.

Each instruction must include an operation entry, which may be a machine, assembler, or macro mnemonic. They are limited to five characters in length (some systems allow longer macro mnemonics) and begin in column 10 of the form.

Operand entries are always required for machine instructions and usually for assembler instructions. They begin in column 16 and may be as long as necessary, up to the maximum statement size the assembler can handle. An operand entry is the coding that identifies and describes the data to be acted upon by the instruction. All operands in a statement must be separated from each other by commas, without blank spaces.

Comments may be used freely, at the programmer's discretion, to document the purpose of coding or the approach used in the programming. These notes can be helpful during debugging and other phases of program checkout and also during later maintenance of a program. They have no effect in the assembled program, but are only printed in the assembly listing. If a programmer wishes to include extensive notes in the printed record, he can use entire lines just for comments by inserting an asterisk in column 1 of each line. A comment that is part of an instruction statement may begin anywhere beyond the operand entry, provided there is at least one blank space after the operand. Most programmers like to line up all comments in some convenient column for easier reading.

A word of caution may be in order about leaving "illegal" blanks in operand entries. If, in our sample program, we were to write:

```
L    2, DATA    LOAD REGISTER 2
```

the assembler, on finding a blank after the comma, would interpret DATA as the first word of the comment and give us an error message MISSING OPERAND.

AN ASSEMBLER LANGUAGE PROGRAM

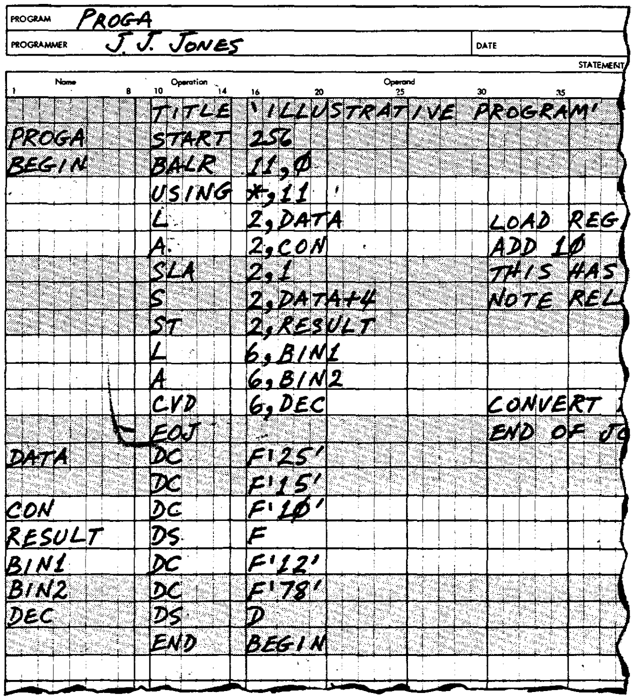
Writing the Program

Let's look at some of the actual instructions in the program in Figure 1-1. This program does not have any particular task to accomplish; it merely demonstrates the use of some serviceable assembler language instructions. In later chapters, each program example will be prefaced by a clear statement of the problem to be solved, which is good practice, but for now let's just get started.

The TITLE assembler instruction in the first line will cause a heading to be printed on every page of the assembly listing. The heading will be ILLUSTRATIVE PROGRAM, which is written within single quotes as the operand entry.

The START instruction specifies to the assembler what the initial value of the location counter for this program should be. Although zero is the usual practice, we specify decimal 256, which is the equivalent of hexadecimal 100. The assembler assumes in most cases that any numeral we use in an operand is a decimal number, unless specified otherwise. We are also using the START statement to give our program a name, PROGA, which is another good programming practice.

The next two instructions are important ones that will appear in every program. To understand their effect, we had better look at these two statements in the order in which they will actually take effect. During assembly, the USING statement will tell the assembler: (1) that it should use register 11 for address calculations and (2) that the address of the *next* machine instruction, which is L 2,DATA, will be in register 11 when PROGA is finally executed. To fulfill this promise, the Branch and Link (BALR) will, when PROGA is executed, actually put the address of the L 2,DATA instruction into register 11. The BALR and USING combination is generally the most efficient way of setting up a register for use as a base register in the System/360 addressing scheme. This subject will be discussed in detail in a separate chapter.



PROGRAM PROGA
PROGRAMMER J. J. JONES
DATE
STATEMENT

```
         TITLE 'ILLUSTRATIVE PROGRAM'
PROGA    START 256
BEGIN    BALR  11,0
         USING *,11
         L     2,DATA         LOAD REG
         A     2,CON          ADD 10
         SLA   2,1            THIS HAS
         S     2,DATA+4       NOTE REL
         ST    2,RESULT
         L     6,BIN1
         A     6,BIN2
         CVD   6,DEC          CONVERT
         EOJ                  END OF J
DATA     DC    F'25'
         DC    F'15'
CON      DC    F'10'
RESULT   DS    F
BIN1     DC    F'12'
BIN2     DC    F'78'
DEC      DS    D
         END   BEGIN
```

So much for the preliminaries. The body of the program starts with the L 2,DATA instruction. L is the mnemonic for the machine instruction Load, which in this case will place in register 2 the contents of a location in storage that has the symbolic address DATA. Looking down the coding sheet, we see that DATA is in the name field of a DC assembler instruction that defines a constant value of 25, occupying four bytes. The name DATA refers to the address of the first byte; the length is implied by the F, for fullword.

The A 2,CON is a similar type of instruction. It adds to register 2 the contents (that is, the constant value 10) of a fullword that has its first byte at the symbolic location CON.

The next instruction (SLA 2,1) is quite different. SLA stands for the algebraic Shift Left Single. The contents of register 2 are to be shifted left one binary place. There is no symbolic address in this case; the second operand simply indicates the extent of the shift.

The Subtract instruction that comes next (S 2,DATA+4) includes an example of *relative addressing*: the address is given relative to another address. This address is specified as four bytes beyond DATA. Looking at the constant area of the program, we see that four bytes (one fullword) beyond DATA there is indeed another fullword constant, the number 15.

The Store instruction (ST 2,RESULT) specifies that the contents of register 2 are to be placed in a storage area with the symbolic address RESULT. Looking below again, we see RESULT in the name field of a DS for a fullword area. As a machine operation, Store has one somewhat unusual feature. In most System/360 machine instructions, the result of an operation replaces the first operand. In Store, however, the result is stored in the second operand location. The same is also true of CVD, which we shall come to shortly.

The following two statements, the Load and the Add, present no new assembler language concepts. They will form a sum in register 6, in the same way as before.

The Convert to Decimal (CVD) converts the contents of register 6, which are binary, to a decimal number, and stores the result in the eight-byte area beginning at DEC. The operation of the machine instruction CVD requires that this location be a doubleword, aligned on a doubleword boundary. More on this later.

The next instruction, EOJ, is a macro instruction that will, after PROGA has been executed, return control to the supervisor, so that the computer can immediately go on with other jobs. EOJ is the last executable (or machine) instruction in our program example.

The DC's and DS's follow the executable part of the program in a group, as is customary. These assembler instructions were discussed earlier in this chapter. Define Storage (DS) is used to define and reserve an area of storage, which may be used during execution of the program for work areas or for storing a varying value. Define Constant (DC) allows us to introduce specific data into a program (a constant simply means an unchanging value).

Each DC and DS must have a *type specification* that designates the particular data format in which it is to be entered into internal machine storage. Some of the data formats are the eight-bit character code (type C), the four-bit hexadecimal code (type X), zoned decimal numbers (type Z), packed decimal numbers (type P), and fixed-point binary numbers (type F or H). A more complete list appears in the Appendix and in the assembler language specification manuals listed in the Preface. In the program at hand and in Chapter 3, where we shall be studying System/360 fixed-point binary operations, however, all the constants are type F or H (the F is for fullword, H for halfword, implying length as well as giving the type).

Fixed-point operations work on fixed-length operands and in most systems require that they be located in storage on halfword, fullword, or doubleword boundaries. In other words, the addresses must be multiples of 2, 4, or 8. When F or H is used to signify the length of a DC or DS (D for doubleword may also be used in a DS), the assembler will perform the necessary alignment, skipping a few bytes if necessary. In our program all the F-type constants and areas will be on four-byte boundaries. The DS at DEC will reserve an eight-byte space, aligned on a doubleword boundary. If the programmer modifies these terms, for example, by specifying 2F instead of D, the assembler will not perform alignment, and it becomes the programmer's responsibility.

The END assembler instruction specifies that nothing further follows, and it terminates the assembly process. The END instruction must always be the last statement in a source program.

The Assembly Listing

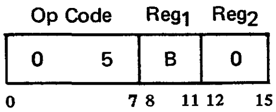
Let's inspect the assembly listing, repeated here as Figure 1-4, to see how the assembler handled things. We see that, except for the TITLE statement, the original source program has been reproduced without change on the righthand side of the listing. The object code created from the source instructions is listed under that heading. The location counter setting of each statement is shown in the leftmost column. The address of the second operand in each instruction is under the heading ADDR2. (All first operands here happen to be in registers.) All entries to the left of the statement number column are in the hexadecimal number system, which is the alphabet, so to speak, of System/360 machine language.

The assembler instructions TITLE, START, USING, and END did not produce any object code, and, as we can see from an inspection of the location counter readings, do not use any space in the object program. The location shown on each of these lines is simply the current setting of the location counter, which, after assembly of each instruction that will use storage space, was updated to show the next available byte.

The START 256 sets the assembler's location counter to hexadecimal 000100, or 100. The object code that is actually at location 100 (in bytes 100 and 101) and will be at the equivalent location in core storage is 05B0, the machine language translation of the BALR instruction. Hex 05 is the BALR operation code, B is register 11 (B is the

```
        ILLUSTRATIVE PROGRAM

  LOC  OBJECT CODE        ADDR1 ADDR2 STMT  SOURCE STATEMENT

000100                                   2 PROGA    START 256
000100 05B0                              3 BEGIN    BALR  11,0
000102                                   4          USING *,11
000102 5820 B022                00124    5          L     2,DATA              LOAD REGISTER 2
000106 5A20 B02A                0012C    6          A     2,CON               ADD 10
00010A 8B20 0001                00001    7          SLA   2,1                 THIS HAS EFFECT OF MULTIPLYING BY 2
00010E 5B20 B026                00128    8          S     2,DATA+4            NOTE RELATIVE ADDRESSING
000112 5020 B02E                00130    9          ST    2,RESULT
000116 5860 B032                00134   10          L     6,BIN1
00011A 5A60 B036                00138   11          A     6,BIN2
00011E 4E60 B03E                00140   12          CVD   6,DEC               CONVERT TO DECIMAL
                                        13          EOJ                       END OF JOB
                                        14+* 360N-CL-453 EOJ           CHANGE LEVEL 3-0
000122 0A0E                             15+         SVC   14
000124 00000019                         16 DATA     DC    F'25'
000128 0000000F                         17          DC    F'15'
00012C 0000000A                         18 CON      DC    F'10'
000130                                  19 RESULT   DS    F
000134 0000000C                         20 BIN1     DC    F'12'
000138 0000004E                         21 BIN2     DC    F'78'
000140                                  22 DEC      DS    D
000100                                  23          END   BEGIN
```

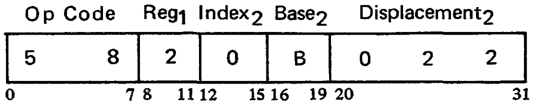
Figure 1-4. Assembly listing of the program in Figure 1-1 hex equivalent of 11), and 0 is register zero (which means, in effect, *no* register and *no* branching). This instruction is in the RR (register-to-register) machine format, which has a length of two bytes and looks like this in storage (contents are shown here in hex rather than binary):



The subscripts 1 and 2 refer, both here and in other instruction formats, to the first and second operands. In the RR format both operands are in registers.

After the BALR was assembled, the location counter read 102, which was the next available byte, and stayed that way until additional object code was generated. USING did not generate object code, so 102 was the setting when the L 2,DATA was assembled. The asterisk in the USING means the *current*, updated location counter setting, which at that point was 102.

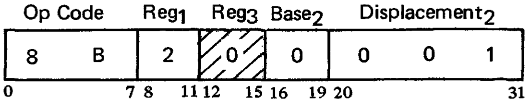
The next instruction, Load, is the first that will actually process program data. It is an RX (register-and-indexed-storage) instruction, which has a machine format of four bytes. It occupies bytes 102 to 105:



In this format, the first operand is in a register, the second in main storage. Reading the assembled bytes from left to right, we have the op code 58 for Load and register 2 for the register to be loaded, and the remaining code gives the address of the second operand. Zero means there is no index register, B (hex for 11) is the base register, and 022 is the displacement in bytes. The effective address, formed by the assembler, is the sum of the contents of the base register (102), the contents of the index register (0 or no register), and the displacement (022). These add up to hexadecimal 124. Looking down to the assembled location of DATA, we see that it is 124, as it should be.

The Add instruction that follows is also in the RX format, and again no index register is used. The base register contents of 102 (258_{10}) plus the displacement of 02A (42_{10}) gives a sum of 12C (300_{10}), which is the location of CON. (The subscript 10 is used to indicate a number in the decimal system. A subscript of 16 is used for hexadecimal, and 2 for binary.)

SLA is in the RS (register-and-storage) machine instruction format, also four bytes in length, and it is in bytes 10A, 10B, 10C, and 10D.



The op code is 8B and the first operand is in register 2. The next four bits are never used in a shift operation, the next four could be used for a base register for the second operand but are not in this case, and the final 001 merely indicates a shift of one binary place. No provision is made for using an index register in this format. As we shall see later, some RS instructions, like Branch on Index High (BXH) and Store Multiple (STM), have a third operand.

The next five instructions are all in the RX format and offer no new concepts. The reader may wish to brush up on hexadecimal numbers and check that the displacements have been computed correctly, taking into account the relative address in the Subtract. We can see even in this simple example how much of the clerical burden the assembler takes over by automatically assigning base registers and calculating displacements.

The assembled entries for the DC's are simply the requested constants, in hexadecimal. We note that the DS entered nothing, but simply reserved space. A study of the address for the doubleword constant at DEC shows that boundary alignment was performed. The fullword constant BIN2 was placed at 138. Counting in hexadecimal, BIN2 occupies four bytes: 138, 139, 13A, and 13B. Although 13C was available for DEC, it is not on a doubleword boundary, nor is 13D, 13E, or 13F. So the assembler skipped these four bytes and assigned DEC to 140.

The END assembler instruction terminates the assembly of the program. The operand indicates the point to which we wish control to be transferred when the program is loaded. In this case, it is to our first instruction in the object program, named BEGIN, where actual execution of the program is to begin. Note that the location counter shows the value 100 at the END statement.

ERROR ANALYSIS BY THE ASSEMBLER

Certain kinds of programming errors can be detected rather simply by the assembler. In fact, some errors make it impossible for the assembler to generate an instruction and complete the assembly. The assembler carries out the assembly as completely as possible, regardless of the number of errors, even if the first error detected makes it impossible for the object program to be executed. The idea is that, if there are more errors, the programmer needs to know about all of them, not just the first one the assembler encounters.

Figure 1-5 is the assembly listing of a program written deliberately with a number of errors in it, to demonstrate what the assembler can do and how it announces its findings. The first announcement is made on the program listing itself, where every statement with a discernable error is followed by a line prominently reading

*** ERROR ***

When the programmer is warned of the existence of an error, he can often see rather quickly what is wrong. Looking over the listing in Figure 1-5, he would probably notice at once that the comma between operands in statement 9 is omitted, and that statement 19 is, from his point of view (but not the assembler's), a bundle of typographical keypunching errors.

Some errors may not be so obvious. To help the programmer analyze them, the assembler prints a separate listing of diagnostic messages. This is part of the output from the assembler program, which was described earlier in this chapter. The diagnostics listing for our program example is shown in Figure 1-6. The assembler always gives a summary message, shown at the bottom, of the total number of statements in error. If no errors are found, the happy message NO STATEMENTS FLAGGED IN THIS ASSEMBLY is printed at the end of the symbol cross-reference table, and no diagnostic listing is printed.

Let's see what the assembler has to tell us about statement 6. The message is on the first line of the diagnostics listing: UNDEFINED OPERATION CODE.

We check the mnemonic for Shift Left Single and find of course that we should have written SLA instead of SLS. The assembler program cannot assume SLA was meant; we might have meant SL, SLR, SLL, or any other valid operation code. Since it cannot tell what was intended, it flags the statement and does not assemble the object code or even assign space to the instruction.

The diagnostic message for statement 7 is UNDEFINED SYMBOL. The undefined symbol is DATA4. This is accepted as a valid symbol, since it follows all the rules governing the writing of symbols. That is, it begins with a letter, uses only letters and numbers, does not contain special characters or blanks, and isn't more than eight characters. Looking at the symbols or names listed in the source statements, we see we have defined DATA and remember that we intended DATA+4 as the address of the

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

 000100                              1 PROGC    START 256
 000100 05B0                         2 BEGIN    BALR  11,0
 000102                              3          USING *,11
 000102 5820 B01E          00120     4          L     2,DATA
 000106 5A20 B026          00128     5          A     2,CON
                                     6          SLS   2,1
          *** ERROR ***
 00010A 0000 0000          00000     7          S     2,DATA4
          *** ERROR ***
 00010E 5020 B02A          0012C     8          ST    2,RESULT
 000112 0000 0000          00000     9          L     6BIN1
          *** ERROR ***
 000116 5A60 B02E          00130    10          A     6,BIN2
 00011A 0000 0000          00000    11          CVD   6,BIN1
          *** ERROR ***
                                    12          EOJ
                                    13+* 360N-CL-453 EOJ       CHANGE LEVEL 3-0
 00011E 0A0E                        14+         SVC   14
 000120 00000019                    15 DATA     DC    F'25'
 000124 4CB016EA                    16          DC    F'9876543210'
          *** ERROR ***
 000128 0000000A                    17 CON      DC    F'10'
 00012C                             18 RESULT   DS
          *** ERROR ***
 00012C 0000 0000          00000    19 IN1      C     '12'
          *** ERROR ***
 000130 0000004E                    20 BIN2     DC    F'78'
 000138                             21 DEC      DS    D
 000140 00000019                    22 DATA     DC    F'25'
          *** ERROR ***
 000100                             23          END   BEGIN
```

Figure 1-5. Assembly listing of the program rewritten with deliberate errors

```
                                   DIAGNOSTICS

STMT   ERROR CODE    MESSAGE

   6   IJQ088        UNDEFINED OPERATION CODE
   7   IJQ024        UNDEFINED SYMBOL
   9   IJQ039        INVALID DELIMITER
   9   IJQ039        INVALID DELIMITER
  11   IJQ024        UNDEFINED SYMBOL
  16   IJQ017        DATA ITEM TOO LARGE
  18   IJQ031        UNKNOWN TYPE
  18   IJQ009        MISSING OPERAND
  19   IJQ039        INVALID DELIMITER
  19   IJQ018        INVALID SYMBOL
  22   IJQ023        PREVIOUSLY DEFINED NAME


   8 STATEMENTS FLAGGED IN THIS ASSEMBLY
```

Figure 1-6. Assembly listing of diagnostic error messages for the program in Figure 1-5

next constant. To the assembler there is no relationship at all between DATA4 and DATA; they are simply different symbols. But if we write DATA+4, the assembler program will recognize the plus sign as a special character that, among other things, delimits the symbol DATA. Confronted with DATA4, the assembler does not assemble the object code. This time, however, the valid mnemonic S indicates that this instruction will be in RX format. So the assembler assigns four bytes to the instruction.

In statement 9, the Load instruction, we already know that our error was the omission of the comma in 6,BIN1. This made the assembler give two identical diagnostic messages: INVALID DELIMITER. From the mnemonic L, the assembler anticipates an RX format, the L to be followed by a register number, a comma, and a storage operand. Finding a B instead of a comma probably led to the first message. What about the second message? What does it mean?

Here the error code in the second column of the diagnostic listing may help. The meaning of each message is given in expanded form in a table of error codes in the assembler manuals. (The letters IJQ here simply stand for a particular assembler program, the Disk Operating System D assembler.) If we were to look up IJQ039 in the table, we would find that it means "*any* syntax error". About a dozen possibilities are listed. An invalid delimiter is the usual error in assembler language syntax, hence the wording of the message. Some other possibilities are (1) an unpaired parenthesis, (2) an embedded blank, (3) a missing delimiter, (4) a missing operand, and (5) a symbol beginning with other than an alphabetic character. Well, the first two obviously don't apply to 6BIN1, and it would be difficult and unrewarding to make a choice among the others, especially considering the compounded error in the symbol in statement 19. What the two messages signify is that there is no reliable evidence of what was intended or just which specification was really violated. The programmer is amply warned that an error exists. It is his job to make his intentions known.

UNDEFINED SYMBOL appears again for statement 11. From the programmer's viewpoint, a reverse situation exists from the one in statement 7. This time the instruction statement is as it should be, but the DC defining the symbol shows IN1 instead of BIN1. There is no indication that these are related in any way or that one is not correct.

Statement 16 elicits the message DATA ITEM TOO LARGE. This is perfectly clear. The decimal value 9,876,543,210 cannot be contained in a 32-bit binary fullword, and the hexadecimal value shown as four bytes has evidently been truncated.

Statement 18 was awarded two error messages: UNKNOWN TYPE when the assembler program found no type designation, and MISSING OPERAND when it scanned further on. Jumping ahead for a moment, we find that statement 22 has the message PREVIOUSLY DEFINED NAME, and we see that DATA has already been given in statement 15.

In statement 19 the first letter of each entry is omitted. The messages are INVALID DELIMITER, which may mean almost any error of syntax, and INVALID SYMBOL, which apparently applies to the name IN1. What's the matter with IN1? It begins with a letter and violates no rules we know of. It should be perfectly acceptable to the assembler. We are the only ones who know it is misspelled. Also, when the message UNKNOWN TYPE is available, why single out the operand with its missing F as a syntax error? Four bytes of zeros have been generated. Why did the assembler assign a specific length? Also, apparently no fault was found with the mnemonic C. How is that? The point is precisely that C *is* a valid operation code. So the assembler, being given this definite "fact" (the most important single fact in any instruction), performs its syntax scan and other operations as if it were dealing with a Compare. The mnemonic C indicates that the instruction is in the RX format requiring four bytes, that the first operand must be a number between 0 and 15 followed by a comma, and that the second operand may be a symbol. But the operand field of this Compare instruction contains simply the characters " '12' ". *This* then is the "symbol" the second message refers to. Indeed, both messages evidently apply to the operand field. To the assembler program nothing is wrong with the name or the operation code mnemonic.

For such reasons as these, the diagnostic messages given by the assembler may often seem quite inaccurate from the programmer's point of view. In many cases, the assembler simply does not have enough clues to pin down the precise error, and the messages should not be taken literally. The assembler program was designed to be as helpful as it can be, and the messages are an effort to help the programmer diagnose the trouble. Usually the error flag on the program listing is enough. The programmer will be interested in the message itself only when he cannot identify the mistake.

This review of how the assembler analyzes programming errors should also make it clear that many errors are beyond the power of the assembler even to recognize. When we incorrectly write DATA4 for DATA+4, the assembler can detect it, but not if DATA4 itself is a legitimate symbol. If we write SLL for SLA, the assembler will assume that SLL is what we mean; both are valid operation codes with the same format. The ability of the assembler to detect and analyze errors can be very helpful to the programmer. However, the message NO STATEMENTS FLAGGED IN THIS ASSEMBLY cannot be taken to mean that a program has no errors or that it will necessarily produce the right answers when it is executed.

MODIFYING AN ASSEMBLER LANGUAGE PROGRAM

After a program has been written, assembled, and completely debugged, it frequently happens that some change must be made later. Many types of revisions are simple to make in an assembler language program. But let us see what happens to the locations of instructions and data when even a minor change is made. We shall base the example on the correct version of the program, as it appeared assembled in Figure 1-4.

Let us suppose that for some unspecified reason it is necessary to store the sum of BIN1 and BIN2 in binary before converting it to decimal. We must insert an instruction:

ST 6,BINANS

just before the CVD.

This is a rather simple sort of change and one that is representative of the kind of modification made with routine frequency on many programs. Yet it can have the effect of changing almost every effective address in the program! The insertion of the four-byte instruction "pushes down" the storage spaces for the DC's and DS's, requiring a change in the displacements of all the instructions that refer to the constants.

Figure 1-7 is the assembly listing of the modified program. Scanning down the assembled instructions, we see that the displacements have been computed to reflect the change in locations. Continuing the comparison, however, we see that ADDR2 and the displacement in the Convert to Decimal instruction are the same as in the earlier version. Has there been a mistake?

The answer is the boundary alignment of the doubleword constants. In the earlier version, it was necessary to skip four bytes to provide an address for DEC that was on a doubleword boundary. The inserted instruction, in effect, filled that skipped space. The reassembly therefore left the assembled address for DEC unchanged.

```
  LOC  OBJECT CODE    ADDR1 ADDR2   STMT   SOURCE STATEMENT

000100                                 1 PROGB    START 256
000100 05B0                            2 BEGIN    BALR  11,0
000102                                 3          USING *,11
000102 5820 B026            00128      4          L     2,DATA         LOAD REGISTER 2
000106 5A20 B02E            00130      5          A     2,CON          ADD 10
00010A 8B20 0001            00001      6          SLA   2,1            THIS HAS EFFECT OF MULTIPLYING BY 2
00010E 5B20 B02A            0012C      7          S     2,DATA+4       NOTE RELATIVE ADDRESSING
000112 5020 B032            00134      8          ST    2,RESULT
000116 5860 B036            00138      9          L     6,BIN1
00011A 5A60 B03A            0013C     10          A     6,BIN2
00011E 5060 B046            00148     11          ST    6,BINANS
000122 4E60 B03E            00140     12          CVD   6,DEC
                                      13          EOJ                  END OF JOB
                                      14+* 360N-CL-453 EOJ      CHANGE LEVEL 3-0
000126 0A0E                           15+         SVC   14
000128 00000019                       16 DATA     DC    F'25'
00012C 0000000F                       17          DC    F'15'
000130 0000000A                       18 CON      DC    F'10'
000134                                19 RESULT   DS    F
000138 0000000C                       20 BIN1     DC    F'12'
00013C 0000004E                       21 BIN2     DC    F'78'
000140                                22 DEC      DS    D
000148                                23 BINANS   DS    F
000100                                24          END   BEGIN
```

Figure 1-7. Assembly listing of the same program modified to store the binary contents of register 6

Chapter 2: System/360 Review

The reader may find it helpful at this point to review some basic facts about System/360 that are directly relevant to assembler language programming. These are stated as briefly as possible in this chapter and will serve mainly as a reminder. A student who is familiar with the material may skip any or all of the sections without loss. A student who needs more than a reminder is urged to go back to the textbook or course materials he originally studied for an introduction to System/360.

The basic structure of a System/360 consists of main storage, a central processing unit (CPU), the selector and multiplexor channels, and the input/output (I/O) devices attached to the channels through control units. For basic information that applies to the material in this book, we are concerned principally with the CPU and main storage. In this chapter, discussion will essentially be limited to these machine units and their basic operating principles.

Since a knowledge of hexadecimal numbers is necessary in assembler language programming, these will also be explained.

MAIN STORAGE

Main storage is also called core or processor storage to distinguish it from storage on tape, disk, or other auxiliary devices. It is closely involved in the operation of the CPU, although it may be either physically integrated with it or constructed as a stand-alone unit. Capacity may be from 8,192 bytes to several million bytes, depending on the system model. Protection features are available that make it possible to protect the contents of main storage from access or alteration.

In general, instructions and data are stored along with each other in whatever order they are presented to the machine. Particular areas of storage may be used over and over again by a succession of programs or groups of programs being executed. Each group overlays, or replaces, the instructions and data of the one preceding. The programmer must therefore specify blanks or zeros where he needs them; he can never assume he is writing on a clean slate. During execution of his program, he can obtain a printout or "dump" of an area of storage at any point in the program by use of suitable instructions.

Bytes and Data Field Lengths

The system transmits information between main storage and the CPU in units of eight bits, or a multiple of eight bits at a time. Each eight-bit unit of information is called a *byte*, the basic building block of all formats. A ninth bit, the parity or check bit, is transmitted with each byte and carries odd parity on the byte. The parity bit cannot be affected by the program; its only purpose is to cause an interruption when a parity error is detected. References in this book to the size of data fields and registers exclude the mention of the associated parity bits.

Bytes may be handled separately or grouped together in fields. A *halfword* is a group of two consecutive bytes and is the basic building block of instructions. A *word* is a group of four consecutive bytes; a *doubleword* is a field consisting of two words (Figure 2-1). The location of any field or group of bytes is specified by the address of its leftmost byte.

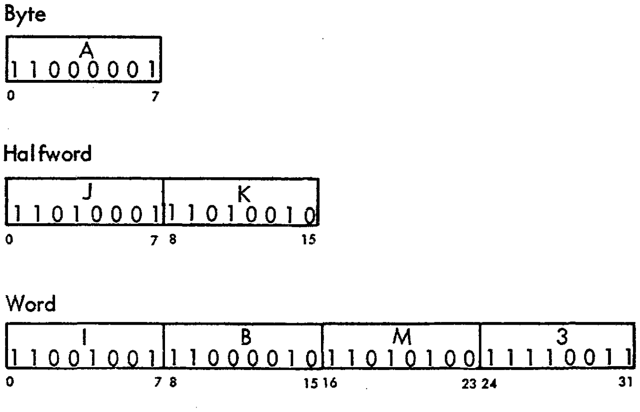


Figure 2-1. Sample data formats

The length of fields is either implied by the operation to be performed or stated explicitly as part of the instruction. When the length is implied, the information is said to have a fixed length, which can be either one, two, four, or eight bytes.

When the length of a field is not implied by the operation code, but is stated explicitly, the information is said to have variable field length. This length can be varied in one-byte increments.

Within any program format or any fixed-length operand format, the bits making up the format are consecutively numbered from left to right starting with the number 0.

This general information on data formats and field lengths will be supplemented later by further details. Lengths and the specific form of the *contents* of the fields are discussed in the section on the arithmetic and logical unit, under the headings for logical operations and the specific types of arithmetic.

Addressing

Byte locations in storage are consecutively numbered starting with 0; each number is considered the address of that byte. A group of bytes in storage is addressed by the leftmost byte of the group. The number of bytes in the group is either implied or explicitly defined by the operation. The addressing arrangement uses a 24-bit binary address to accommodate a maximum of 16,777,216 byte addresses. This set of main-storage addresses includes some locations reserved for the supervisor and other special purposes. How storage addresses are generated is described in the section on program execution.

The available storage is normally contiguously addressable, starting at address 0. An addressing exception is recognized when any part of an operand is located beyond the maximum available capacity of an installation. Except for a few instructions, the addressing exception is recognized only when the data are actually used and not when the operation is completed before using the data. The addressing exception causes a program interruption.

Positioning on Integral Boundaries

Fixed-length fields, such as halfwords and doublewords, must be located in main storage on an *integral boundary* for that unit of information. A boundary is called integral for a unit of information when its storage address is a multiple of the length of the unit in bytes. For example, words (four bytes) must be located in storage so that their address is a multiple of the number 4. A halfword (two bytes) must have an address that is a multiple of the number 2, and doublewords (eight bytes) must have an address that is a multiple of the number 8.

For greatest efficiency in storage addressing, address arithmetic is done exclusively in binary. In binary, integral boundaries for halfwords, words, and doublewords can be specified only by the binary addresses in which one, two, or three of the low-order bits, respectively, are zero (Figure 2-2). For example, the integral boundary for a word is a binary address in which the two low-order positions are zero.

Variable-length fields are not limited to integral boundaries, and may start on any byte location.

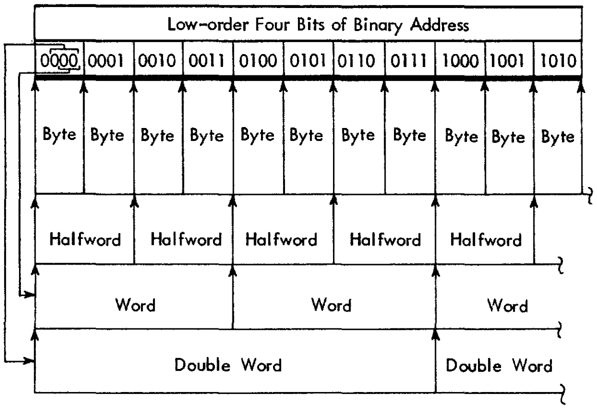


Figure 2-2. Integral boundaries for halfwords, words, and doublewords

CENTRAL PROCESSING UNIT

The central processing unit (Figure 2-3) contains the facilities for addressing main storage, for fetching or storing information, for arithmetic and logical processing of data, for sequencing instructions in the desired order, and for initiating the communication between storage and external devices.

The system control section provides the normal CPU control that guides the CPU through the functions necessary to execute the instructions. The programmer-trainee will probably be glad to know that the result of executing a valid instruction is the same for each model of System/360.

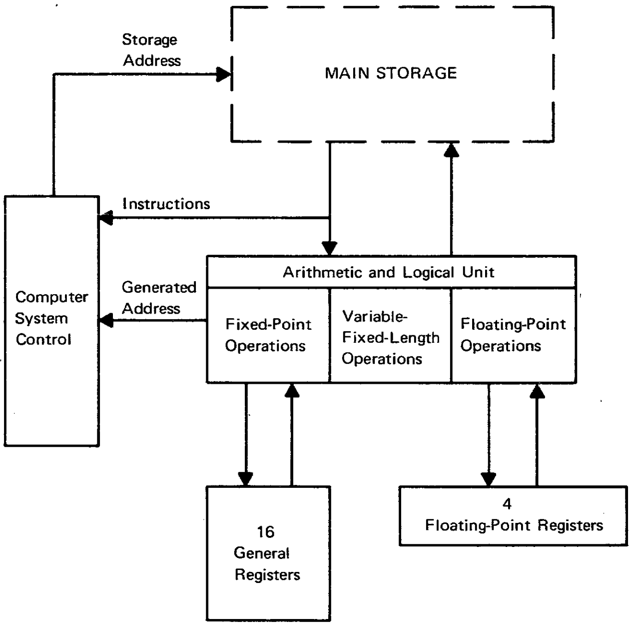


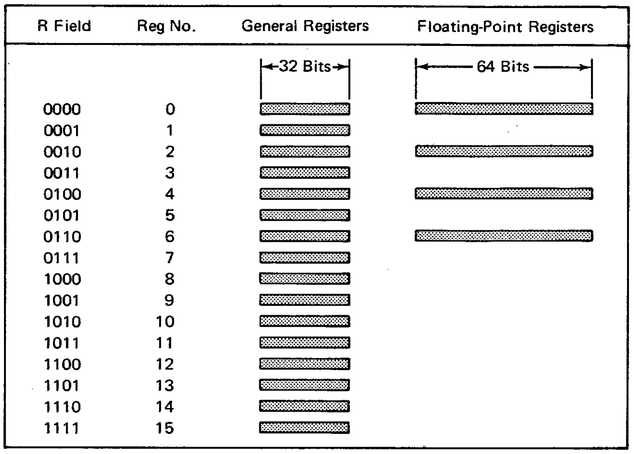
Figure 2-3. Functions of the central processing unit

General and Floating-Point Registers

The CPU provides 16 *general registers* for fixed-point operands and four *floating-point registers* for floating-point operands. Physically, these registers may be in special circuitry, in a local storage unit, or in a separate area of main storage. In each case, the address and functions of these registers are identical.

The CPU can address information in 16 general registers. The general registers can be used as index registers, in address arithmetic and indexing, and as accumulators in fixed-point arithmetic and logical operations. The registers have a capacity of one word (32 bits). The general registers are identified by numbers 0–15 and are specified by a four-bit R field in an instruction (Figure 2-4). Some instructions provide for addressing multiple general registers by having several R fields.

For some operations, two adjacent general registers are coupled together, providing a two-word capacity. In these operations, the addressed register contains the high-order operand bits and must have an even address, and the implied register, containing the low-order operand bits, has the next higher address.



R Field	Reg No.	General Registers	Floating-Point Registers
		32 Bits	64 Bits
0000	0	▨	▨
0001	1	▨	
0010	2	▨	▨
0011	3	▨	
0100	4	▨	▨
0101	5	▨	
0110	6	▨	▨
0111	7	▨	
1000	8	▨	
1001	9	▨	
1010	10	▨	
1011	11	▨	
1100	12	▨	
1101	13	▨	
1110	14	▨	
1111	15	▨	

Figure 2-4. General and floating-point registers

Four floating-point registers are available for floating-point operations. They are identified by the numbers 0, 2, 4, and 6 (Figure 2-4). These floating-point registers are two words (64 bits) in length and can contain either a short (one word) or a long (two words) floating-point operand. A short operand occupies the high-order bits of a floating-point register. The low-order portion of the register is ignored and remains unchanged in short-precision arithmetic. The instruction operation code determines which type of register (general or floating-point) is to be used in an operation, and if floating-point whether short or long precision.

Arithmetic and Logical Unit

The arithmetic and logical unit can process binary integers and floating-point fractions of fixed length, decimal integers of variable length, and logical information of either fixed or variable length.

Arithmetic and logical operations performed by the CPU fall into four classes: fixed-point arithmetic, decimal arithmetic, floating-point arithmetic, and logical operations. These classes differ in the data formats used, the registers involved, the operations provided, and the way the field length is stated. Data formats are discussed under each of the headings in this section. General information on field lengths was given in the section on main storage.

Fixed-Point Arithmetic

The basic arithmetic operand is the 32-bit fixed-point binary number. Sixteen-bit halfword operands may be specified in most operations for improved performance or storage utilization (see Figure 2-5). To preserve precision, some products and all dividends are 64 bits long. A fixed-point number is a signed value, recorded as a binary integer. It is called fixed point because the programmer determines the fixed positioning of the binary point.

In both halfword (16 bits) and word (32 bits) lengths, the first bit position (0) holds the sign of the number. The remaining bit positions (1–15 for halfwords and 1–31 for fullwords) are used to designate the value of the number.

Positive fixed-point numbers are represented in true binary form with a zero sign bit. Negative fixed-point numbers are represented in two's complement notation with a one bit in the sign position. In all cases, the bits between the sign bit and the leftmost significant bit of the integer are the same as the sign bit (i. e. all zeros for positive numbers, all ones for negative numbers). The filled-in examples in Figure 2-5 show the equivalent of decimal +62 and -62 in fixed-point halfwords.

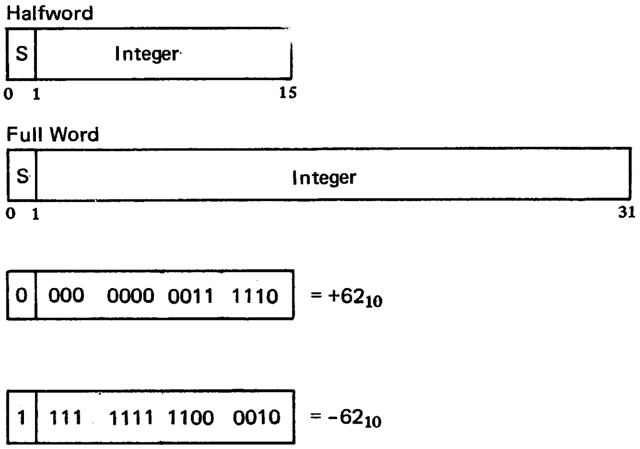


Figure 2-5. Fixed-point number formats. In the example the negative number is in two's complement notation

Because the 32-bit word size readily accommodates a 24-bit address, fixed-point arithmetic can be used both for integer operand arithmetic and for address arithmetic. This combined usage provides economy and permits the entire fixed-point instruction set and several logical operations to be used in address computation. Thus, multiplication, shifting, and logical manipulation of address components are possible.

Additions, subtractions, multiplications, divisions, and comparisons are performed upon one operand in a register and another operand either in a register or from storage. Multiple-precision operation is made convenient by the two's-complement notation and by recognition of the carry from one word to another. A word in one register or a double word in a pair of adjacent registers may be shifted left or right. A pair of conversion instructions–Convert to Binary and Convert to Decimal–provides transition between decimal and binary number bases without the use of tables. Multiple-register loading and storing instructions facilitate subroutine switching.

Decimal Arithmetic

Decimal arithmetic lends itself to data processing procedures that require few computational steps between the source input and the documented output. This type of processing is frequently found in commercial applications. Because of the limited number of arithmetic operations performed on each item of data, conversion from decimal to binary and back to decimal is not justified, and the use of registers for intermediate results yields no advantage over storage-to-storage processing. Hence, decimal arithmetic is provided, and both operands and results are located in storage. Decimal arithmetic includes addition, subtraction, multiplication, division, and comparison.

Decimal numbers are treated as signed integers with a variable-field-length format from one to 16 bytes long. Negative numbers are carried in true form.

The decimal digits 0–9 are represented in the four-bit binary-coded-decimal (BCD) form by 0000–1001, respectively, as follows.

Digit	Binary Code	Digit	Binary Code
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

The codes 1010–1111 are not valid as digits and are reserved for sign codes. The sign codes generated in decimal arithmetic depend upon the character set code used. When the extended binary coded decimal interchange code (EBCDIC) is used, the codes are 1100 for a plus sign and 1101 for a minus. (When the USASCII set, expanded to eight bits, is preferred, the sign codes are 1010 and 1011. The choice between the two code sets is determined by a mode bit.)

Decimal operands and results are represented by four-bit BCD digits packed two to a byte (see Figure 2-6). They appear in fields of variable length and are accompanied by a sign in the rightmost four bits of the low-order byte. Operand fields may be located on any byte boundary, and may have length up to 31 digits and sign (16 bytes). Operands participating in an operation may have different lengths. Packing of digits within a byte and use of variable-length fields within storage results in efficient use of storage, in

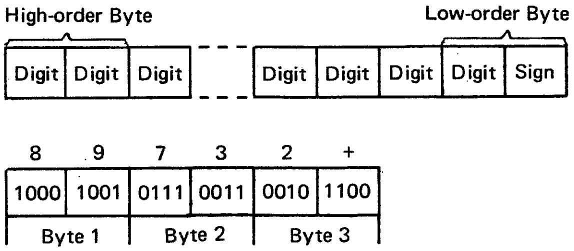


Figure 2-6. Packed decimal number format. The three-byte example shows decimal value +89,732

increased arithmetic performance, and in an improved rate of data transmission between storage and files.

Decimal numbers may also appear in a zoned format in the regular EBCDIC eight-bit alphameric character format (Figure 2-7). This representation is required for I/O devices that are character-set sensitive. A zoned format number carries its sign in the leftmost four bits of the low-order byte. The zoned format is not used in decimal arithmetic operations. Instructions are provided for packing and unpacking decimal numbers so that they may be changed from the zoned to the packed format and vice versa.

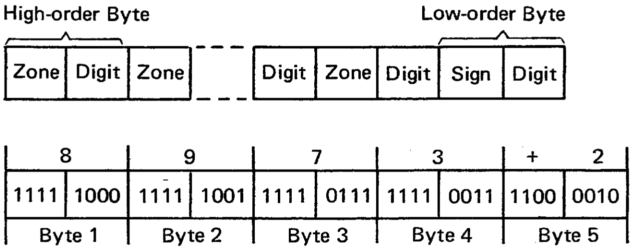


Figure 2-7. Zoned decimal number format. The decimal number +89,732 requires five bytes

Floating-Point Arithmetic

Floating-point numbers occur in either of two fixed-length formats—short or long. These formats differ only in the length of the fractions (Figure 2-8). They are described in detail in the chapter on floating-point arithmetic.

Floating-point operands are either 32 or 64 bits long. The short length permits a maximum number of operands to be placed in storage and gives the shortest execution times. The long length, used when higher precision is desired, more than doubles the number of digits in each operand.

Four 64-bit floating-point registers are provided. Arithmetic operations are performed with one operand in a register and another either in a register or from storage. The result, developed in a register, is generally of the same length as the operands. The availability of several floating-point registers eliminates much storing and loading of intermediate results.

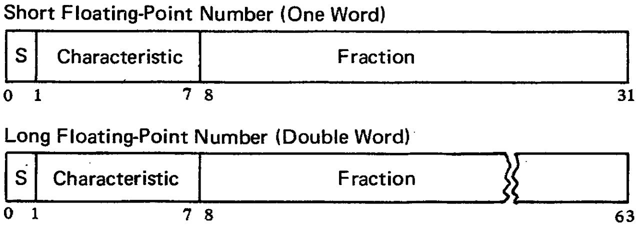


Figure 2-8. Short and long floating-point number formats

Logical Operations and the EBCDIC Character Set

Logical information is handled as fixed- or variable-length data. It is subject to such operations as comparison, translation, editing, bit testing, and bit setting.

When used as a fixed-length operand, logical information can consist of either one, four, or eight bytes and is processed in the general registers (Figure 2-9).

A large portion of logical information consists of alphabetic or numeric character codes, called *alphameric data*, and is used for communication with character-set sensitive I/O devices. This information has the variable-field-length format and can consist of up to 256 bytes (Figure 2-9). It is processed storage to storage, left to right, an eight-bit byte at a time.

The CPU can handle any eight-bit character set, although certain restrictions are assumed in the decimal arithmetic and editing operations. However, all character-set sensitive I/O equipment will assume either the extended binary coded decimal interchange code (EBCDIC) or the USA Standard Code for Information Interchange (USASCII) extended to eight bits. Use of EBCDIC is assumed throughout this book.

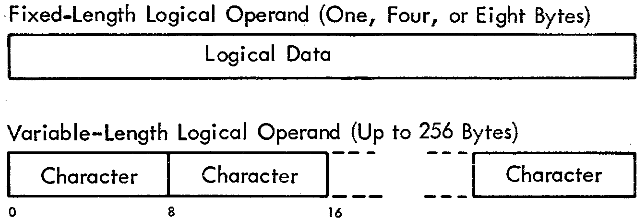


Figure 2-9. Fixed- and variable-length logical information

EBCDIC does not have a printed symbol, or graphic, defined for all 256 eight-bit codes. When it is desirable to represent all possible bit patterns, a hexadecimal representation may be used instead of the preferred eight-bit code. The hexadecimal representation uses one graphic for a four-bit code, and therefore, two graphics for an eight-bit byte. The graphics 0–9 are used for codes 0000–1001; the graphics A–F are used for codes 1010–1111. EBCDIC eight-bit code for characters that can be represented by well-known symbols is shown in Table 2-1. The hexadecimal equivalents and punched card code are also shown. For other symbols, System/360 control characters, and unassigned codes, see the complete 256-position EBCDIC chart in the Appendix. It may be observed from the table that the EBCDIC collating sequence for alphameric characters, from lower to higher binary values, is (1) special characters, (2) lower case letters, (3) capital letters, and (4) digits, with each group in its usual order.

Table 2-1. Extended Binary Coded Decimal Interchange Code (EBCDIC) for Graphic Characters

Graphic character	EBCDIC 8-bit code Bit Positions 0123 4567	Hex equivalent	Punched card code
blank	0100 0000	40	no punches
¢	0100 1010	4A	12-8-2
.	0100 1011	4B	12-8-3
(	0100 1101	4D	12-8-5
+	0100 1110	4E	12-8-6
&	0101 0000	50	12
!	0101 1010	5A	11-8-2
$	0101 1011	5B	11-8-3
*	0101 1100	5C	11-8-4
)	0101 1101	5D	11-8-5
;	0101 1110	5E	11-8-6
-	0110 0000	60	11
,	0110 1011	6B	0-8-3
%	0110 1100	6C	0-8-4
?	0110 1111	6F	0-8-7
:	0111 1010	7A	8-2
#	0111 1011	7B	8-3
@	0111 1100	7C	8-4
’	0111 1101	7D	8-5
=	0111 1110	7E	8-6
”	0111 1111	7F	8-7
a	1000 0001	81	12-0-1
b	1000 0010	82	12-0-2
c	1000 0011	83	12-0-3
d	1000 0100	84	12-0-4
e	1000 0101	85	12-0-5
f	1000 0110	86	12-0-6
g	1000 0111	87	12-0-7
h	1000 1000	88	12-0-8
i	1000 1001	89	12-0-9
j	1001 0001	91	12-11-1
k	1001 0010	92	12-11-2
l	1001 0011	93	12-11-3
m	1001 0100	94	12-11-4
n	1001 0101	95	12-11-5
o	1001 0110	96	12-11-6
p	1001 0111	97	12-11-7
q	1001 1000	98	12-11-8
r	1001 1001	99	12-11-9
s	1010 0010	A2	11-0-2
t	1010 0011	A3	11-0-3
u	1010 0100	A4	11-0-4
v	1010 0101	A5	11-0-5
w	1010 0110	A6	11-0-6
x	1010 0111	A7	11-0-7
y	1010 1000	A8	11-0-8
z	1010 1001	A9	11-0-9
A	1100 0001	C1	12-1
B	1100 0010	C2	12-2
C	1100 0011	C3	12-3
D	1100 0100	C4	12-4
E	1100 0101	C5	12-5
F	1100 0110	C6	12-6
G	1100 0111	C7	12-7
H	1100 1000	C8	12-8
I	1100 1001	C9	12-9
J	1101 0001	D1	11-1
K	1101 0010	D2	11-2
L	1101 0011	D3	11-3
M	1101 0100	D4	11-4
N	1101 0101	D5	11-5
O	1101 0110	D6	11-6
P	1101 0111	D7	11-7
Q	1101 1000	D8	11-8
R	1101 1001	D9	11-9
S	1110 0010	E2	0-2
T	1110 0011	E3	0-3
U	1110 0100	E4	0-4
V	1110 0101	E5	0-5
W	1110 0110	E6	0-6
X	1110 0111	E7	0-7
Y	1110 1000	E8	0-8
Z	1110 1001	E9	0-9
0	1111 0000	F0	0
1	1111 0001	F1	1
2	1111 0010	F2	2
3	1111 0011	F3	3
4	1111 0100	F4	4
5	1111 0101	F5	5
6	1111 0110	F6	6
7	1111 0111	F7	7
8	1111 1000	F8	8
9	1111 1001	F9	9

PROGRAM EXECUTION

Interplay of equipment and program is an essential consideration in System/360. The system is designed to operate with a control program that coordinates and executes all I/O instructions, handles exceptional conditions, and supervises scheduling and execution of multiple programs. System/360 provides for efficient switching from one program to another, as well as for the relocation of programs in storage. To the problem programmer, the control program and the equipment are indistinguishable.

The CPU program consists of instructions, index words, and control words that specify the operations to be performed. Some of its functions will be discussed here. The format of the machine instructions is basic to an understanding of how the CPU executes them and how it forms addresses of operands in main storage. A doubleword called the program status word (PSW) contains detailed information required by the CPU for proper program execution: the instruction address, the condition code setting, etc. It is stored at a fixed location. If a problem program aborts and the contents of storage are printed out, the PSW can be inspected by the programmer. He will find much information to help him analyze the trouble, including a code that identifies the cause of the interruption.

The interruption system permits the CPU to respond automatically to conditions arising outside of the system, in I/O units, or in the CPU itself. Interruption switches the CPU from one program to another by changing not only the instruction address but all essential machine-status information.

Programs are checked for correctness of instructions and data as the instructions are executed. (The types of errors involved are not detectable during assembly.) This policing action distinguishes and identifies program errors and machine errors. Thus, program errors cannot cause machine checks: each of these types of error causes a different type of interruption.

Sequential Instruction Execution

Normally, the operation of the CPU is controlled by instructions taken in sequence. An instruction is fetched from a location specified by the instruction address in the current PSW. The instruction address is then increased by the number of bytes in the instruction fetched to address the next instruction in sequence. The instruction is then executed and the same steps are repeated using the new value of the instruction address.

A change from sequential operation may be caused by branching, interruptions, etc.

Branching

The normal sequential execution of instructions is changed when reference is made to a subroutine, when a two-way choice is encountered, or when a segment of coding, such as a loop, is to be repeated. All these tasks can be accomplished with branching instructions. Provision is made for subroutine linkage, permitting not only the introduction of a new instruction address but also the preservation of the return address and associated information.

Decision-making is generally and symmetrically provided by the Branch on Condition instruction. This instruction inspects a two-bit *condition code* in the PSW, that reflects the result of a majority of the arithmetic, logical, and I/O operations. Each of these operations can set the code in any one of four ways, and the conditional branch can specify any of these four settings, or any combination of them, as the criterion for branching.

Loop control can be performed by the conditional branch when it tests the outcome of address arithmetic and counting operations. For some particularly frequent combinations of arithmetic and tests, the instructions Branch on Count and Branch on Index are provided. These branches, being specialized, provide increased performance for these tasks.

Instruction Format

The length of an instruction format can be one, two, or three halfwords. It is related to the number of storage addresses necessary to specify the location of all operands in the operation. Operands may be located in registers or in main storage, or may be a part of an instruction. An instruction consisting of only one halfword causes no reference to main storage. A two-halfword instruction provides one storage-address specification; a three-halfword instruction provides two storage-address specifications. All instructions must be located in storage on integral boundaries for halfwords. Figure 2-10 shows the five basic instruction formats, called RR, RX, RS, SI, and SS.

These format codes express, in general terms, the operation to be performed. RR denotes a register-to-register operation; RX, a register-and-indexed-storage operation; RS, a register-and-storage operation; SI, a storage and immediate-operand operation; and SS, a storage-to-storage operation. An immediate operand is one contained within the instruction.

For purposes of describing the execution of instructions in the SRL manual *IBM System/360 Principles of Operation* (A22-6821), operands are designated as first and second operands and, in the case of branch-on-index instructions, third operands. These names refer to the manner in which the operands participate. The operand to which a field in an instruction format applies is generally denoted by the number following the code name of the field, for example, R_1, B_1, L_2, D_2.

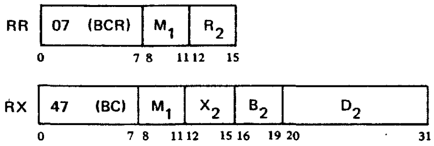
In each format, the first instruction halfword consists of two parts. The first byte contains the operation code. The length and format of an instruction are specified by the first two bits of the operation code:

Bit Positions (0-1)	Instruction Length	Instruction Format
00	One halfword	RR
01	Two halfwords	RX
10	Two halfwords	RS or SI
11	Three halfwords	SS

The second byte is used either as two 4-bit fields or as a single 8-bit field. As shown in Figure 2-10, this byte can contain the following information:

Four-bit operand register specification (R_1, R_2, or R_3)
Four-bit index register specification (X_2)
Four-bit operand length specification (L_1 or L_2)
Eight-bit operand length specification (L)
Eight-bit byte of immediate data (I_2)

In some instructions a four-bit field or the whole second byte of the first halfword is ignored. In the Branch on Condition instruction, which may be used in either the RR or RX format, the first four bits of the second byte are used as a 4-bit mask field (M_1 in the following diagram). This mask tests the four settings of the condition code and is used to determine whether a branch will or will not be made.



In all instructions, the second and third halfwords always have the same format: four-bit base register designation (B_1 or B_2), followed by a 12-bit displacement (D_1 or D_2).

Generation of Main Storage Addresses

To permit the ready relocation of program segments and to provide for the flexible specifications of input, output, and working areas, all instructions referring to main storage have been given the capacity of employing a full address.

The address used to refer to main storage is generated from the following numbers, all binary:

Base Address (B) is a 24-bit number contained in a general register specified by the program in the B field of the instruction. (One way to insert a base address into a register is to specify a BALR operation at the beginning of a program. The BALR operation does just that, getting the address of the next sequential instruction from the current program status word, no matter where the program may have been relocated.) The B field is included in every address specification. The base address can be used as a means of relocation of programs and data. It provides for addressing the entire main storage. The base address may also be used for indexing purposes.

Index (X) is a 24-bit binary number contained in a general register specified by the program in the X field of the instruction. It is included only in the address specified by the RX instruction format; or it may simply be omitted in an RX instruction. The RX format instructions permit double indexing.

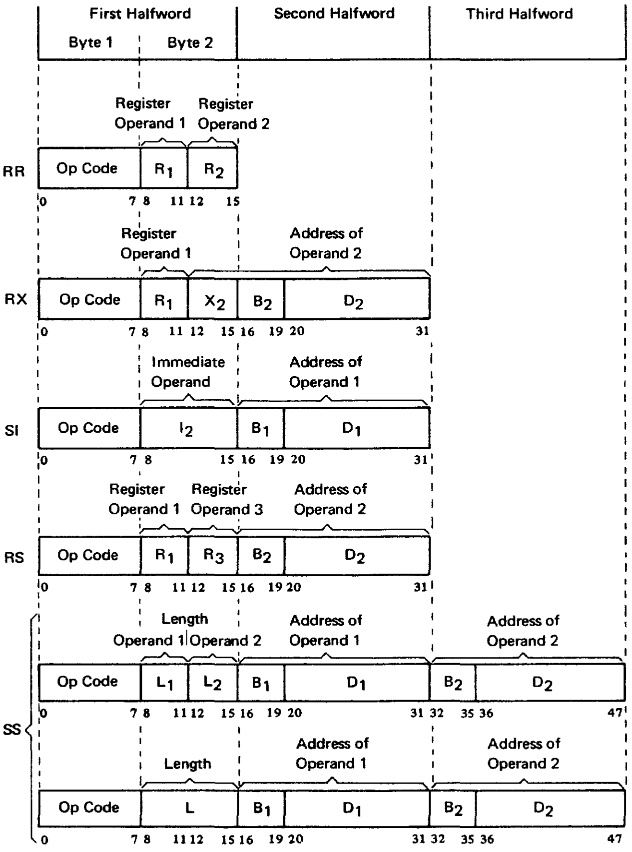


Figure 2-10. Machine instruction formats

Displacement (D) is a 12-bit binary number contained in the instruction format. It is included in every address computation. The displacement provides for relative addressing up to 4095 bytes beyond the base address, which is the limit that can be expressed by 12 binary bits. In Chapter 1 we saw how the displacements were calculated by the assembler from symbolic addresses written by the programmer.

We also saw that the three binary numbers are added together to form the actual address. This sum is a 24-bit number, which can be represented by six hexadecimal digits.

The program may have zeros in the base address, index, or displacement fields. A zero is used to indicate the absence of the corresponding address component. A base or index of zero implies that a zero quantity is to be used in forming the address, regardless of the contents of general register 0. Initialization, modification, and testing of base addresses and indexes can be carried out by fixed-point instructions, or by Branch and Link, Branch on Count, or Branch on Index instructions.

Interruptions and the Program Status Word

To make maximum use of a modern data processing system, some automatic procedure must be made available to alert the system to an exceptional condition, the end of an I/O operation, program errors, machine errors, etc., and send the system to the appropriate routine following the detection of such an event. The system must have, in effect, the ability to pause to answer the telephone and then to resume the interrupted work. This automatic procedure is called an interruption system.

It makes possible the operation of a system in a non-stop environment and greatly aids the efficient use of I/O equipment. The desire to make the interruption procedure as short and simple as possible means that the method of switching between the interrupted program and the program that services the interruption must be quite efficient. It operates as follows:

The complete status of the System/360 is held in eight bytes of information. This status information, which consists of the instruction address, condition code, storage protection key, etc., is saved when an interruption occurs, and is restored when the interruption has been serviced.

As soon as the interruption occurs, all the status information, together with an identification of the cause of the interruption, is combined into a doubleword called the program status word (PSW).

The PSW is stored at a fixed location, the address of which depends on the type of interruption. The system then automatically fetches a new PSW from a different fixed location, the address of which is also dependent on the type of interruption. Each class of interruption has two fixed locations in main storage: one to receive the old PSW when the interruption occurs, and the other to supply the new PSW that governs the servicing of that class of interruption.

After the interruption has been serviced, a single instruction uses the stored PSW to reset the processing unit to the status it had before the interruption.

Types of Interruptions

The interruption system separates interruptions into five classes:

Supervisor Call interruptions are caused when the processing program issues an instruction to turn over control to the supervisor in the control program. The exact reason for the call is shown in the old PSW.

External interruptions are caused by either an external device requiring attention or by the system timer going past zero.

Machine Check interruptions are caused by the machine-checking circuits detecting a machine error.

I/O interruptions are caused by an I/O unit ending an operation or otherwise needing attention. Identification of the device and channel causing the interruption is stored in the old PSW; in addition, the status of the device and channel is stored in a fixed location.

Program interruptions are caused by various kinds of programming errors or unusual conditions resulting from improper specification or use of instructions or data. The exact type of error is shown in an interruption code in the PSW.

Finding the Source of a Program Interruption

When a program interruption occurs, provision is always made to locate the instruction that was being interpreted and to identify the exact type of error involved, so that the programmer can make the necessary corrections. For this information he must go to the PSW in a printout of storage contents.

Fifteen interruption codes are used for the different types of program interruptions, as follows.

Interruption Code		Program Interruption Cause
1	00000001	Operation
2	00000010	Privileged operation
3	00000011	Execute
4	00000100	Protection
5	00000101	Addressing
6	00000110	Specification
7	00000111	Data
8	00001000	Fixed-point overflow
9	00001001	Fixed-point divide
10	00001010	Decimal overflow
11	00001011	Decimal divide
12	00001100	Exponent overflow
13	00001101	Exponent underflow
14	00001110	Significance
15	00001111	Floating-point divide

To take an example, one of the conditions that causes a "data exception" to be recognized is an incorrect sign or digit code in an operand used in decimal arithmetic. In this case, the operation would be terminated, and all, part, or none of the arithmetic result would be stored. Since the result is unpredictable, it should not be used for further computation. The interruption code, binary 0000 0111, or hexadecimal 07, for a data exception would be recorded in bit positions 24–31 of the *program* old PSW (always at main storage location 40_{10}).

The location of the instruction that was being interpreted when the interrupt occurred can also be determined from an inspection of the old PSW. The instruction address, which is found in bit positions 40–63 of the PSW, is for the instruction *to be executed next.* To locate the preceding instruction, all that is needed is to subtract its length in bytes. This instruction length can be found in bit positions 32 and 33 of the PSW, recorded there in binary as 1, 2, or 3 halfwords.

HEXADECIMAL NUMBERS

Hexadecimal Code

Hexadecimal numbers have been mentioned a number of times. In Chapter 1 we used them to represent machine language instructions, and we saw that the assembler listed object code, location counter settings, and addresses in hexadecimal numbers. In System/360 hexadecimal code is a shorthand method of representing the internal binary zeros and ones, one hex digit for each four binary bits.

Hex numbers are a convenient way for the assembler language programmer to specify masks in testing and branching operations, and to specify hexadecimal constants (type X). Principally, he uses hexadecimal code to locate and interpret the contents of storage, which may be printed out when a program must be analyzed and debugged. In a later chapter, we shall see some "dumps" of storage and attempt to locate information in them.

Converting from binary to hex, or from hex to binary, is simple. There are only 16 hex symbols, and their value is based on the numerical value of four bits. We recall that four bits in the binary number system can express all values from zero to 15_{10}. We also recall that the position of each bit determines its value:

Binary	Decimal
0001	1
0010	2
0100	4
1000	8

Some people find it easier to remember these binary positional values this way:

8	4	2	1

If we try the four bit values in various combinations, we find that we can rather quickly discover how to count from zero to the equivalent of decimal 15 in sequence. In order to be able to represent these 16 values by a single symbol, the letters A, B, C, D, E, and F are used for 10, 11, 12, 13, 14, and 15, respectively. The numbers 0–9 stand for themselves. The entire four bit code is shown in Table 2-2.

Table 2-2. Hexadecimal Code

Binary	Hexadecimal	Decimal	Binary	Hexadecimal	Decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

All kinds of information, data, instructions, etc., in System/360 can be represented in hexadecimal code, two graphic hex symbols per byte. The same hex coding system is used regardless of the code in which the information is recorded internally. The internal information may be EBCDIC characters, zoned decimal numbers, signed binary numbers, the eight-bit code used for System/360 operation codes, or any of the other codes and formats in use. All are coded in some form of binary coding, and, since the eight-bit byte is the basic unit of System/360, they can readily be taken four bits at a time.

Let's look at some examples. Each "box" represents a byte. Binary bits are shown in groups of four for convenience.

1. EBCDIC characters

Characters	I	B	M	3
Internal form	1100 1001	1100 0010	1101 0100	1111 0011
Hex code	C 9	C 2	D 4	F 3

2. Zoned decimal number

Decimal	8	9	7	3	+ 2
Internal form	1111 1000	1111 1001	1111 0111	1111 0011	1100 0010
Hex code	F 8	F 9	F 7	F 3	C 2

3. Packed decimal number

Decimal	8 9	7 3	2 +
Internal form	1000 1001	0111 0011	0010 1100
Hex code	8 9	7 3	2 C

4. Signed binary number

(This fixed-point fullword is equivalent to decimal +89,732)

Internal form	0000 0000	0000 0001	0101 1110	1000 0100
Hex code	0 0	0 1	5 E	8 4

The reader may wonder how, when he sees a hexadecimal printout of storage contents, he will be able to interpret the different formats correctly. This is not a problem, but does require care. The programmer can refer to the assembly listing of the program. By tracing the assembler addresses, he can calculate just where in main storage each instruction or data item is. In some cases, the format will have been specified explicitly. In others, he must know which format is implied by use of particular instructions or types of data.

Hexadecimal Number System

Turning back to the examples, we notice that internally the characters and decimal numbers are, generally speaking, coded separately in either four- or eight-bit binary *codes*. The binary number in example 4, however, is recorded internally in its actual value as an integer in a number system with a base of 2. The 0's and 1's are the only digits in this number system. Similarly, the hexadecimal equivalent 15E84 is an integer in a valid number system with a base of 16. Considering the binary and hex numbers in this example in their entirety, they have exactly the same total value. Each hex digit also equals the value of the four bits it represents. We see from this that hex numbers can be used in two different ways: (1) simply as a four-bit code into which each internal half-byte is translated, and (2) both as a four-bit code and as a valid number system with a base related in a definite way to the base of the binary number system.

In the familiar decimal number system, the base is 10, and there are ten digits, 0–9. In the decimal number 234, we know that the 2, because of its position, equals 2 x 100, or 200; the 3 equals 3 x 10, or 30; and the 4 equals 4 x 1, or 4. The three values are in effect added together. We may represent the place value of each digit in a whole number (not a fractional or mixed number) in this way:

Power of base 10	10^4	10^3	10^2	10^1	10^0
Value	10,000	1000	100	10	1

In the same way, the binary number system has a base of 2 and has two digits, 0 and 1. Its place values are:

Power of base 2	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value in decimal	256	128	64	32	16	8	4	2	1

The hexadecimal number system has a base of 16 and has 16 digits, 0–9 and A–F. Its place values are:

Power of base 16	16^4	16^3	16^2	16^1	16^0
Value in decimal	65,536	4096	256	16	1

We may notice that there is a relationship between binary and hexadecimal place values. Beyond the zeroth power (this always equals 1), hex place values are exactly four times greater than binary. This becomes clear when we compare them up to $2^{12}=16^3$:

Power of base 2	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Power of base 16	16^3				16^2				16^1				16^0
Value in decimal	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

It is this relationship that makes one hex digit equal arithmetically to four binary bits, two hex digits equal to two groups of four bits each, etc. All hex and binary digits must of course be kept in correct place order.

Now we are ready to figure out some actual hexadecimal values. Hex numbers are especially useful for calculating main storage addresses and displacements. A storage address, we may remember, is a 24-bit true binary number internally, always represented externally by the machine as six hexadecimal digits.

We shall use Table 2-3 for converting hex numbers to decimal, and decimal to hex. It is for integers only. The table shows eight places, each place being the position of a hex digit, starting from the right.

The table shows the equivalent decimal value of each hexadecimal digit in each hex position from 1 to 8. To convert a hex number to decimal, it is necessary only to find the value of each hex digit in the column corresponding to its position, and to add them together. To convert $D34_{16}$ to decimal, we start in column 3 because this is a three-digit number. We find (1) $D00_{16} = 3328_{10}$ in column 3, (2) $30_{16} = 48_{10}$ in column 2, and (3) $4_{16} = 4_{10}$ in column 1; then (4) summing the decimal values, we get

$$\begin{array}{r} 3328 \\ 48 \\ \underline{4} \\ 3380_{10} \end{array} = D34_{16}$$

To convert the five-digit number $B60A6_{16}$ to decimal, we follow the same procedure, beginning in column 5:

Hex		Decimal
B0000	=	720 896
6000	=	24 576
000	=	0
A0	=	160
6	=	6
B60A6		745 638

Using the same table to convert a decimal number to hexadecimal requires a rather different procedure. Let's

Table 2-3. Hexadecimal and Decimal Integer Conversion Table

HALF WORD								HALF WORD							
BYTE				BYTE				BYTE				BYTE			
BITS 0123		BITS 4567		BITS 0123		BITS 4567		BITS 0123		BITS 4567		BITS 0123		BITS 4567	
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
8		7		6		5		4		3		2		1	

X20-8047

take 3380_{10} as an example. We look for the highest decimal value in the table that will fit into 3380. The closest is 3328_{10} in column 3, equal to the D. We make a note that this corresponds to $D00_{16}$ and subtract, as shown below. The closest value below the remainder (52_{10}) is 48_{10} in column 2, and we note it is equal to 30_{16}. Subtracting again, we look for the best fit into the remainder of 4_{10}, and find 4_{10} in column 1, equal to 4_{16}. Adding the hex values, we get the result 3380 = D34, which we know from our first conversion example is correct. (The best way to check the result of a conversion is to reconvert. Any lost zeros are likely to be found in the process.)

Decimal		Hex
3380		
3328	=	D00
52		
48	=	30
4	=	4
		D34

Without looking back, let's convert $745{,}638_{10}$ to hexadecimal:

Decimal		Hex
745 638		
720 896	=	B0000
24 742		
24 576	=	6000
166		
160	=	A0
6	=	6
		B60A6

The easiest way to find the decimal value of a long binary number is to convert it to hex, and from hex to decimal. Similarly, to find the binary value of a decimal number, the decimal number should be converted to hex, and from hex to binary. To get the binary equivalent of $745{,}638_{10}$, we would convert it to hex as in the last example and merely substitute the four-bit code for each hex digit in the result:

0	B	6	0	A	6
0000	1011	0110	0000	1010	0110

It is entirely feasible to perform all kinds of arithmetic calculations in hexadecimal arithmetic. The rules are the same as in decimal arithmetic. Most programmers prefer to convert hexadecimal values to decimal, however, do their calculations in decimal, and then convert back to hex. This can be done easily and quickly with the use of a conversion table.

On the other hand, computer personnel often find it useful to be able to do simple addition in hexadecimal. Until they become proficient, they can simply count on their fingers. The rules for carrying are the same as in decimal addition. In decimal, the highest digit value is 9. When 1 is added to 9, the result is 0 and a carry of 1. Or, as we usually see it:

$$\begin{array}{r} 9\\ +1\\ \hline 10 \end{array} \qquad \begin{array}{r} 99\\ +1\\ \hline 100 \end{array} \qquad \begin{array}{r} 999\\ +1\\ \hline 1000 \end{array}$$

In hex, when 1 is added to the highest digit F, the result is also 0 and a carry of 1:

$$\begin{array}{l} F\\ +1\\ \hline 10\ (=16_{10}) \end{array} \qquad \begin{array}{l} FF\\ +1\\ \hline 100\ (=256_{10}) \end{array} \qquad \begin{array}{l} FFF\\ +1\\ \hline 1000\ (=4096_{10}) \end{array}$$

The following list of equivalent values may help to crystallize the concepts of hexadecimal notation. Hex numbers that end in zero are always multiples of 16. To avoid confusion hex numbers like 10, 11, 12, etc., should be read as "one zero, one one, one two," and not as "ten, eleven, twelve."

Dec.	Hex	Dec.	Hex	Dec.	Hex	Dec.	Hex
1	1	22	16	43	2B	80	50
2	2	23	17	44	2C	81	51
3	3	24	18	45	2D	.	.
4	4	25	19	46	2E	.	.
5	5	26	1A	47	2F	.	.
6	6	27	1B	48	30	94	5E
7	7	28	1C	49	31	95	5F
8	8	29	1D	50	32	96	60
9	9	30	1E	51	33	97	61
10	A	31	1F	52	34	98	62
11	B	32	20	.	.	99	63
12	C	33	21	.	.	100	64
13	D	34	22	.	.	.	.
14	E	35	23	62	3E	.	.
15	F	36	24	63	3F	.	.
16	10	37	25	64	40	240	F0
17	11	38	26	65	41	.	.
18	12	39	27	.	.	.	.
19	13	40	28	.	.	254	FE
20	14	41	29	78	4E	255	FF
21	15	42	2A	79	4F	256	100

Chapter 3: Fixed-Point Arithmetic

This chapter introduces and discusses some of the fixed-point operations of the standard instruction set in the System/360. These include the arithmetic and shifting instructions as the central topic, with important consideration also of certain logical operations (comparison, branching), and loop methods.

Fixed-point instructions perform binary arithmetic on fixed-length data of either a fullword or a halfword. The use of registers for arithmetic and other operations is thus most convenient. As might be expected, the fixed-point instruction set uses only these three instruction formats: RR, RX, and RS.

In the course of presenting the instructions and considering programming methods used with the System/360, we shall review the basic ideas of the machine organization and operation.

The presentation will be almost entirely through the medium of eight examples and a final extended case study.

ADDITION AND SUBTRACTION

For a first example we shall consider a simple inventory calculation. We begin the calculation with an on-hand quantity, a receipt quantity, and an issue quantity. We are required to compute the new on-hand, according to the formula:

new on-hand = old on-hand + receipts - issues

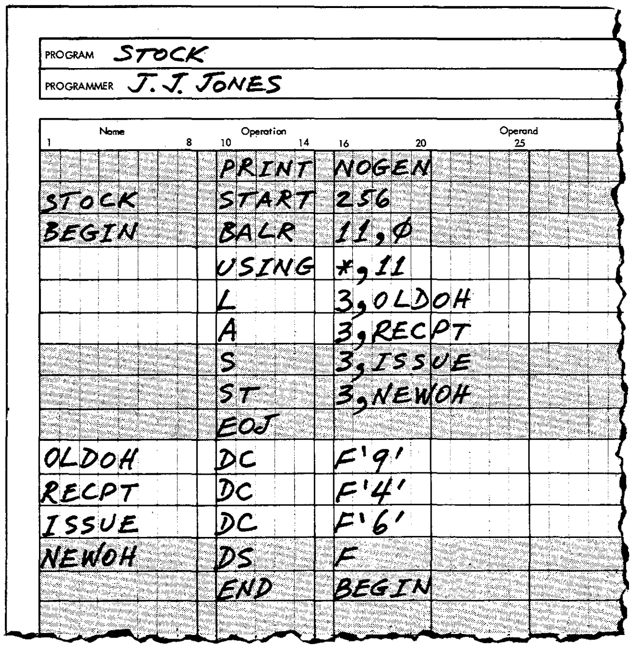
Using fairly obvious symbols for the four quantities, this becomes:

NEWOH = OLDOH + RECPT - ISSUE

A program to carry out this calculation is shown in Figure 3-1. We shall be concentrating on the four actual processing instructions, but at the outset we shall display all programs in logically complete form.

The assembler instruction PRINT NOGEN is used simply to suppress printing of statements generated by macro instructions such as the EOJ macro. These statements and their storage locations and displacements will still be part of the object program; they will be omitted only from the printed listing.

The next three lines of coding are rather standard preliminaries; instructions of this character will appear at the beginning of all but highly specialized programs. To review briefly, the START establishes a reference point for the assembly: the assembly listing (shown later) will assume that the first byte is to be loaded into 256 as shown. The BALR (Branch and Link Register) and the USING, as written here, together direct that register 11 shall be used as a base register wherever one is needed, and inform the assembler that the base register at execution time will contain the location of the first byte after the USING.



PROGRAM STOCK

PROGRAMMER J. J. JONES

```
Name     Operation Operand
1      8 10   14   16  20   25
         PRINT     NOGEN
STOCK    START     256
BEGIN    BALR      11,0
         USING     *,11
         L         3,OLDOH
         A         3,RECPT
         S         3,ISSUE
         ST        3,NEWOH
         EOJ
OLDOH    DC        F'9'
RECPT    DC        F'4'
ISSUE    DC        F'6'
NEWOH    DS        F
         END       BEGIN
```

Figure 3-1. A program, written in assembler language, to perform a simple computation in binary arithmetic

Now we reach the first processing instruction, where we wish to concentrate our attention.

The Load instruction is classified as an RX format instruction, which implies a number of facts about it:

1. The instruction itself takes up four bytes of storage.
2. The fields within the instruction are, from left to right: the operation code (eight bits), the number of the register to be loaded from storage (four bits), the number of the register used as an index register (four bits), the

number of the register used as a base register (four bits), and the displacement (twelve bits).

3. The instruction involves a transfer of information between storage and a general register.

4. The effective address of a byte in storage is formed by adding the contents of the base register, the contents of the index register, and the displacement. If register zero is specified for an index register or a base register, a zero value is used in the address computation, rather than whatever register zero may contain.

The operation of the Load instruction is straightforward: obtain a fullword (four bytes) from storage at the effective address specified, and place the word in the general register indicated. The effective address must refer to a fullword boundary, which means that the address must be a multiple of 4.

Let us consider the complete line of coding for the Load instruction to see what each part does.

The letter L is the mnemonic operation code for Load; this is converted by the assembler into the actual machine operation code for Load, 58. The 3 is the number of the general register we wish loaded with a word from storage. OLDOH is the symbolic address of the word in storage to be copied into general register 3. By writing the address in this fashion, we have indicated that the assembler should supply the base register and the displacement, and that we do not wish indexing.

The assembly listing for this program is shown in Figure 3-2. Looking at the machine instruction assembled from this symbolic instruction, and remembering that all numbers are shown in hexadecimal, we see that the operation code is 58, the general register is 3, the index register is zero, the base register is B (= 11_{10}), and the displacement is 012_{16}. Since the base register contains 102, the effective address is 114, which is shown in the assembly listing under ADDR2 as the address of the second operand and which we see *is* the location of OLDOH.

The Add instruction is also of the RX format. The operation is to add the fullword at the storage address specified, to the general register named. In our case, we have, of course, named the same general register as in the Load instruction, since the intent is to add OLDOH and RECPT together. Looking at the assembled instruction, we see that things have been handled much as they were with the Load. Base register 11 has been assigned, there is no index register, and the displacement has been computed to give the effective address of the storage location associated with RECPT (118).

After the execution of this instruction, register 3 will contain the sum of the storage quantities identified in our program by OLDOH and RECPT.

The Subtract instruction (S) in the next line subtracts the quantity identified by the symbol ISSUE from the quantity now standing in register 3. The format and general operation of the instruction are very similar to Add.

Now we have the desired result in register 3. The problem statement required the result to be placed in another location in storage, that is identified by the symbol NEWOH. Placing the contents of a general register in storage is the function of the Store instruction (operation code ST). The general register contents are unchanged by the operation. The format is again RX, so address formation is as before.

This completes the actions required by the problem statement, but we must now indicate what we want done next. The System/360 forces a program organization that keeps the machine in operation as much of the time as possible. What we have shown here is an End of Job macro instruction, which is used in the Disk Operating System environment. As we saw in the preceding chapter, the EOJ macro generates a Supervisor Call instruction, SVC 14. The use of this instruction assumes that there is in storage, at the time of execution of this program, a control program that runs the machine between jobs. We here indicate to the supervisor that this program has no further need for the machine.

The program in Figure 3-2 does not include any instructions for reading in data from an input device such as a card reader or magnetic tape unit, or for printing out or punching out the results of our calculations. Input and

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      1          PRINT NOGEN
000100                                2 STOCK    START 256
000100 05B0                           3 BEGIN    BALR  11,0
000102                                4          USING *,11
000102 5830 B012            00114     5          L     3,OLDOH
000106 5A30 B016            00118     6          A     3,RECPT
00010A 5B30 B01A            0011C     7          S     3,ISSUE
00010E 5030 B01E            00120     8          ST    3,NEWOH
                                      9          EOJ
000114 00000009                      12 OLDOH    DC    F'9'
000118 00000004                      13 RECPT    DC    F'4'
00011C 00000006                      14 ISSUE    DC    F'6'
000120                               15 NEWOH    DS    F
000100                               16          END   BEGIN
```

Figure 3-2. The assembly listing of the program in Figure 3-1 output instructions vary considerably in different systems, depending upon the operating (or programming support) system in use and the particular pieces of input/output equipment available at an installation.

In normal commercial practice, a computer program would be used, not for calculations on just one set of data, but on large series of data that require similar treatment. An example would be a program to calculate weekly pay for several hundred employees of a company, and the data would include the hours worked, pay rate, withholding amounts, etc., for each of them. In our program examples, our principal interest lies in how the assembler language instructions work, and so we will generally use only one set of specific values for the purpose of illustrating what happens in each step.

We have simply entered the illustrative values for the input data with DC instructions, and reserved space for the output with a DS. The F's in the DC's and the DS specify fullwords of four bytes. The Load, Add, Subtract, and Store instructions all operate on fullwords. As we shall see in later examples, there are corresponding halfword instructions.

The END instruction informs the assembler that the termination of the program has been reached and specifies in this case that the first instruction to be executed after the program is loaded is the one with the name BEGIN, that is, the BALR instruction.

By using either a suitable assembler language routine or macro instruction, it is possible to get a "dump" of the contents of the registers and selected areas of storage, and get our data and results out of the machine. Such a routine produced the numbers, converted to decimal, that are shown in Figure 3-3. The four items, in sequence, are OLDOH, RECPT, ISSUE, and NEWOH.

```
0000009+ 0000004+ 0000006+ 0000007+
```

Figure 3-3. Output of the program of Figure 3-2. The four numbers are OLDOH, RECPT, ISSUE, and NEWOH, in that order.

It might be interesting to run this program again with a value of, say, 16 for ISSUE. We know that negative fixed-point numbers are represented in two's complement form. Our output routine will make a conversion to true numbers and sign, as shown in the first line of Figure 3-4. In the second line, the same numbers are shown in hexadecimal as they normally appear in a dump.

We see that the first three numbers, which are positive, have zeros before the significant digits. The last number, which is negative, has 1's to the left of the significant digit (hexadecimal F equals binary 1111). If we were to write out this hexadecimal number, FFFFFFFD, in the binary form actually in storage, we would have thirty 1's followed by 01. Recalling how two's complement numbers are formed, we see that the complement of this number is binary 11, which equals decimal 3. Checking with the given data and the formula, we see that this is the correct answer, and, of course, the decimal value was printed out as a minus 3.

Naturally, if a negative result were actually obtained in an inventory control program, it would indicate some kind of trouble, probably bad data; it is not possible to issue more than there are on hand plus what was received. A realistic program would include a test for the possibility of a negative result and the corrective action to be taken.

```
0000009+ 0000004+ 0000016+ 0000003-


00000009 00000004 00000010 FFFFFFFD
```

Figure 3-4. Output of the same program with a value for ISSUE that causes NEWOH to be negative. Values are shown in decimal in the first line, hexadecimal in the second; the value for NEWOH is in complement form.

MULTIPLICATION

For a simple example of fixed-point multiplication in the System/360, consider the following problem. We are to multiply an ISSUE quantity by a PRICE to get TOTAL. We shall assume that PRICE is an integer, expressed in pennies. The product will therefore also be in pennies. For instance, an ISSUE of 5 and a PRICE of 25 would give a TOTAL of 125.

The program to do this multiplication is shown in Figure 3-5. The first four lines are the same as before. The Load places the multiplicand in general register 5. The Multiply (M) forms the product of what is in 5 and what is in the full word identified by PRICE, and places the result, which could of course be much longer than either of the factors, in registers 4 and 5 combined. It is required that the general register named in the Multiply be even numbered; if it is not, a specification exception and an interrupt occur. The multplicand must always be in the odd-numbered register of an even-odd pair, such as 4 and 5 here. The multiplicand in the odd register, and whatever may have been in the even register, are both destroyed by the operation of the Multiply.

After the product has been formed, we store it in TOTAL on the assumption that the result does not exceed the length of one register. The validity of such an assumption, of course, is the responsibility of the programmer; if in fact the product extended over into register 4, there would be no automatic signal of the fact that the result in TOTAL is not the complete product. If a product extending into the even register could be a legitimate outcome, we would naturally have to arrange to store both parts of the product.

Let us try this program with several sets of sample factors in order to see precisely how the operation works. Figure 3-6 shows the values of ISSUE, PRICE, TOTAL, and the contents of register 4 and 5 after the completion of the program. These were obtained by a dump routine and converted to decimal. We see that the product of 7 and 23 is indeed 161, as we might expect. This number is shown as the contents of register 5, while register 4 is zero; the product was not long enough to extend into 4.

```
ISSUE          PRICE          TOTAL
0000007+       0000023+       0000161+

REG 4          REG 5
0000000+       0000161+
```

Figure 3-6. Output of the program of Figure 3-5

In Figure 3-7 the numbers are the same except that the 7 is negative. (This makes no sense in terms of the problem, of course.) We see that TOTAL and register 5 are negative, as expected, but what has happened to register 4? The answer is that the product is a full 64 bits long; a negative number has 1's to the left of the leftmost significant digits. Register 4 properly contained all 1's which, considered as part of the 64-bit product, are merely sign bits. But printed as a separate number (which is pointless, in reality), a word of all 1's represents -1 as shown. A printout not reproduced here substantiates what we have said: register 4 printed in hexadecimal form appears as eight F's.

```
ISSUE          PRICE          TOTAL
0000007-       0000023+       0000161-

REG 4          REG 5
0000001-       0000161-
```

Figure 3-7. Output of the program with a negative value for ISSUE

In Figure 3-8 we see an example of what can happen when the numbers entering the machine do not conform to the assumptions made in setting up the program (that is, the product would never extend into register 4). With both factors being 87654, the product, in decimal, should be 7,683,223,716. This is too long to fit into register 5, so we would expect TOTAL to contain only the equivalent of the part of the product that appeared there. But we would hardly have expected it to be negative! What happened?

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE  STATEMENT

                                     1            PRINT  NOGEN
000100                               2  GROSS     START  256
000100 05B0                          3  BEGIN     BALR   11,0
000102                               4            USING  *,11
000102 5850 B00E            00110    5            L      5,ISSUE
000106 5C40 B012            00114    6            M      4,PRICE
00010A 5050 B016            00118    7            ST     5,TOTAL
                                     8            EOJ
000110 00000007                     11  ISSUE     DC     F'7'
000114 00000017                     12  PRICE     DC     F'23'
000118                              13  TOTAL     DS     F
000100                              14            END    BEGIN
```

Figure 3-5. Assembly listing of a program to perform binary (fixed-point) multiplication

The answer becomes apparent if we look at the product as a hexadecimal number and note the part of it that would appear in register 5. The complete product is 1C9F4B0A4, that is, nine hexadecimal digits – a register can hold eight. So the 1, preceded by seven hexadecimal zeros, would be the contents of register 4, as shown. The part in register 5 begins with the hexadecimal digit C, which is 1100 in binary. This means that the leftmost bit is 1, which, when C9F4B0A4 is taken as a number by itself, indicates a negative number that is in two's-complement notation! Thus, in converting to decimal for Figure 3-8, System/360 performed as it was designed to do, recomplemented (to hexadecimal 360B4F5C), and came up with the decimal equivalent of that amount.

This recitation of troubles is not meant to suggest any difficulty in using the System/360. Any programmer appreciates the necessity of knowing a good deal about his data and for testing it for validity if he is not sure of it. The purpose in showing these slightly surprising results is simply to clarify how the machine operates, especially since many programmers will not have had previous contact with complement representation of negative numbers.

```
ISSUE           PRICE           TOTAL
0087654+        0087654+        906710876-

REG 4           REG 5
0000001+        906710876-
```

Figure 3-8. Output of the program with values for ISSUE and PRICE that lead to a TOTAL too large to fit in a fullword

MULTIPLICATION AND DIVISION WITH DECIMAL POINTS

The next example involves a little further practice with multiplication, an application of the Divide instruction, and a rather basic question of decimal point handling in binary.

The task is to increase a principal amount named PRINC by an interest rate of 3%. The principal is stored in pennies as in the previous example; for instance, 24.89 would be stored simply as the integer 2489. Later program segments would have to insert any "graphic" decimal point that might be desired for printing; at this point we make a mental note of the true situation, while pretending for programming purposes at the moment that the unit of currency is the penny.

One possible program is shown in Figure 3-9. (There are other ways, as we shall see.) After the usual preliminaries we load the principal into an odd-numbered register preparatory to multiplying. The interest rate is shown as 103, which should be read as 1.03. This is a shortcut: instead of multiplying the principal by 0.03 and adding the product to the principal, we multiply the principal by 1.03. The result is the same either way; our way saves an addition.

The absence of the decimal point is another matter. We are saying here that instead of multiplying by 1.03, we will multiply by 103; the product will be 100 times too large as a result. It will be necessary in a subsequent step to divide by 100 to correct for this. The reason for this is that there is a question of how to represent a decimal fraction in binary form. The question can be answered, as we shall see, leading to a different program. For now, let us take what seems at first to be the easy way out and stay with integers.

Using the sample principal mentioned above, 24.89, the product after multiplication is 256367. We shall assume that the product in all cases is short enough to be held in register 5 alone.

We now wish to round off. We think of the product as \$25.6367; the desired rounded value is \$25.64. Remembering that the computer knows nothing of our behind-the-scenes understanding about decimal points, all we have to do to round off is to add 50 to the integer product. *We* will think of the 50 as \$0.0050, but to the computer it is 50.

Having done this, we need finally to divide by 100 to correct for using 103 in place of 1.03. This requires the Divide instruction, which as we might expect is a close relative to the Multiply instruction. The dividend must be in an even-odd pair of registers as a 64-bit number. This requirement is already met by the way the Multiply leaves the product in an even-odd pair (the machine was designed to make it simple to follow a Multiply with a Divide). The remainder is placed in the even register and the quotient in the odd. Our quotient will be 2564 (we read: \$25.64) and the remainder will be 17 (we don't care about this). The quotient can now be stored back in the location for PRINC, as required in the problem statement.

The question will occur to many: why was it necessary to divide? Why not simply shift two places right to drop the right two digits? The answer is, of course, that we could do precisely that in decimal, but this is binary. Shifting one place to the right in decimal divides the number by 10; shifting one place to the right in binary divides the number by 2. There is no number of binary shifts that divides a number by a factor of decimal 100. Six places divides it by 64, and seven places by 128. With this way of approaching the problem, we have no choice but to divide.

It should be kept clearly in mind that in all examples so far we have explicitly stated that all quantities were to be viewed for programming purposes as integers, whatever we on the outside might understand by the digits. This was by agreement, not necessity. We can work with binary numbers that are taken to have "binary points" elsewhere than at the extreme right. Let us, for instance, attempt to express the factor 1.03 as a binary number.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT    SOURCE  STATEMENT

                                      1            PRINT  NOGEN
000100                                2  INTA      START  256
000100 05B0                           3  BEGIN     BALR   11,0
000102                                4            USING  *,11
000102 5850 B016             00118    5            L      5,PRINC
000106 5C40 B01A             0011C    6            M      4,INT
00010A 5A50 B01E             00120    7            A      5,C50
00010E 5D40 B022             00124    8            D      4,C100
000112 5050 B016             00118    9            ST     5,PRINC
                                     10            EOJ
000118 000009B9                      13  PRINC     DC     F'2489'
00011C 00000067                      14  INT       DC     F'103'
000120 00000032                      15  C50       DC     F'50'
000124 00000064                      16  C100      DC     F'100'
000100                               17            END    BEGIN
```

Figure 3-9. Assembly listing of a program involving binary multiplication and division with the result rounded off

It may be recalled from a study of the conversion rules that there will be in general no *exact* binary equivalent for a decimal fraction. If we try 1.03 we get an infinitely repeating binary fraction. The first twelve bits are

1.00000111101

The binary point is, of course, *understood* (by us).

If we enter such a number as the constant (which we shall see how to do in a moment), we can multiply by it. The machine cares nothing for our understood binary points, and carries out the multiplication. We must then take into account the understood binary point in the product, according to a literal translation of familiar rules: the binary point in the product will have as many places to the right as the sum of the number of places to the right of the binary points in the multiplier and in the multiplicand. If the constant has eleven places to the right, as written above, and the principal is still understood to be an integer (zero places), then the product will have eleven places to the right.

Let us turn to Figure 3-10 to see how this much of the revised program looks.

The Load is the same as before, as is the Multiply. The constant used for multiplication is different, however. Down at INT we see that the DC is

FS11′1.03′

The F stands for fullword, as before. The S stands for Scale factor and is the number of binary places that are to be reserved for the fractional part of the constant. We have indicated eleven places as the number of bits to the right of the binary point in the factor as we write it before.

The Add to round off is the same as before, but once again the constant is different. What we have after the multiplication this time is not an integer but a binary fraction. To the left of the assumed binary point we have a whole number of pennies; to the right a fractional part of a penny. This time, to round off we need a constant that is 0.5 cent expressed in the same form as the fractional part of our product. The Scale factor method shown gives this. (In fact, the constant consists of a 1 followed by ten zeros.)

After rounding off we are left with eleven superfluous bits at the right end of the product. These can be shifted off the end of the register with a suitable shift instruction. "Suitable" in this case means that the shift should be to the right, it should involve a single register, and it should be an algebraic shift so that if the number were negative, proper sign bits would be shifted into the register. The instruction is called Shift Right Single (SRA), in which we specify the register first and then the number of positions of shift desired. Bits shifted off the right end of the register are lost. After the shift we are ready to store the result.

The point of doing all this is that we have replaced a Divide with a Shift, and the latter is considerably faster than the former. In some applications the difference in time could be significant.

If we print the result, we get a surprise: the answer is 2563 ($25.63); rounding seems not to have taken place. The trouble is that the binary "equivalent" of the decimal number 1.03 was not *exactly* equivalent. To prove the point, let us ask for 15 binary places in the fractional part of the constant created for 1.03. We change the rounding constant likewise, and make the shift 15 places. This time, the printout shows 2564 ($25.64) as before.

The moral of this story is that decimal fractions do not usually have exact binary equivalents. Computations that are required to be exact to the penny should be done in integer form, as in the first version of the program. (Even though a larger number of bits led to a correct answer this time, it would not always do so, particularly for larger principal amounts.)

This means, in most situations, that it would be most unwise to go the further possible step of representing penny amounts as binary fractions. Unless approximate results are acceptable, which they sometimes are, of course, the use of anything but integer arithmetic leads to problems more severe than they are worth.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      1         PRINT NOGEN
000100                                2 INTB    START 256
000100 05B0                           3 BEGIN   BALR  11,0
000102                                4         USING *,11
000102 5850 B016            00118     5         L     5,PRINC
000106 5C40 B01A            0011C     6         M     4,INT
00010A 5A50 B01E            00120     7         A     5,HALF
00010E 8A50 000B            0000B     8         SRA   5,11
000112 5050 B016            00118     9         ST    5,PRINC
                                     10         EOJ
000118 000009B9                      13 PRINC   DC    F'2489'
00011C 0000083D                      14 INT     DC    FS11'1.03'
000120 00000400                      15 HALF    DC    FS11'0.5'
000100                               16         END   BEGIN
```

Figure 3-10. A different version of the program of Figure 3-9, using a scale modifier for constants

Some readers may be wondering whether binary arithmetic is worth the trouble. The answer is yes, of course. Many applications of binary arithmetic raise none of the questions suggested here and do not involve the possible complications with complement form either. For the straightforward cases, it is barely necessary to know anything about the binary and complement matters. We present examples like these to warn the unwary and to lay a foundation of understanding for those with problems where the advantages of binary arithmetic are worth the care that must be exercised in using it. It is true that many applications will suggest staying with decimal arithmetic, for users having the decimal instruction set, but even then there will be more than a few occasions where binary operations are the only ones that make sense from a standpoint of time.

SHIFTING AND DATA MANIPULATION

Having introduced the shifting operation briefly in the previous example, let us now turn to an application that will involve considerably more shifting.

We begin with a fullword, supplied by some other program, in which three data items are stored in binary form:

Bits	*Item name*
0 – 11	A
12 – 23	B
24 – 31	C

We are required to separate the three data items and store each in a separate halfword storage location, with names for the latter as shown. All three numbers are positive.

The program shown in Figure 3-11 is a more or less straightforward matter of shifting and storing, but a few notes are necessary to make clear what is happening at certain points.

The numbers in the Comments field are sample contents of registers 6 and 7 as they would appear during execution of the program if the original word were hexadecimal 78ABCDEF. These sample values, of course, were entered when the source program was punched; it is quite impossible for the object program to print anything on the assembly listing.

We begin by loading the fullword into an even-numbered general register. This permits us to continue with a double-length shift that moves bits from the named even-numbered register into the adjacent odd-numbered register, which we think of as being to the right. This is what "double" means in Shift Right Double Logical (SRDL). The "logical" refers to the handling of sign bits and means that zeros are entered at the left of register 6. This is in contrast to the "algebraic" shifts, in which the bits entered at the left are made to be the same as the original "sign bit", that is, the original leftmost bit. Here, we were guaranteed in the problem statement that all three numbers are positive, so we can ignore any question of what the leftmost bit in each item might be. Whether it is zero or one, the number represented is positive.

The SRDL moves the rightmost eight bits into register 7; from there we move them to the right-hand end of the same register, using a single-length logical shift that does not affect register 6. What were originally the rightmost eight bits of the fullword are now properly positioned in register 7 to be stored in a halfword location with the Store Halfword (STH) instruction. The action here is to store the rightmost 16 bits of the register named, in the two bytes identified by the effective address. The register is not disturbed by the operation of the instruction. This is an RX format instruction; it could be indexed if we had occasion to do so.

Now we again shift the two registers together to get the twelve-bit B item into register 7. From there we move it on over to the right-hand end of 7 and store it. A further shift of what was originally the leftmost twelve bits is not needed, since they are now in the right-hand end of 6, from whence they may be stored.

Actually, the restriction to positive numbers is not too difficult to remove. It would have to be agreed that the leftmost bit of each item was its sign bit, that is, that in generating the fullword each negative item was in complement form and of such length as to fit in the item size allotted. With this assumption, the program of Figure 3-12 properly expands the sign bits of the items and stores any negative items in halfwords in complement form. The "expansion" of the sign bit is one of the functions of an algebraic shift, as noted above. The program must also be

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1           PRINT NOGEN
000100                               2 SHIFTA    START 256
000100 05B0                          3 BEGIN     BALR  11,0
000102                               4           USING *,11
000102 5860 B022            00124    5           L     6,FWORD        78ABCDEF   00000000
000106 8C60 0008            00008    6           SRDL  6,8            0078ABCD   EF000000
00010A 8870 0018            00018    7           SRL   7,24           0078ABCD   000000EF
00010E 4070 B02A            0012C    8           STH   7,C            0078ABCD   000000EF
000112 8C60 000C            0000C    9           SRDL  6,12           0000078A   BCD00000
000116 8870 0014            00014   10           SRL   7,20           0000078A   00000BCD
00011A 4070 B028            0012A   11           STH   7,B            0000078A   00000BCD
00011E 4060 B026            00128   12           STH   6,A            0000078A   00000BCD
                                    13           EOJ
000124                              16 FWORD     DS    F
000128                              17 A         DS    H
00012A                              18 B         DS    H
00012C                              19 C         DS    H
000100                              20           END   BEGIN
```

Figure 3-11. Assembly listing of a program to separate three quantities stored in one fullword

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT    SOURCE STATEMENT

                                     1          PRINT NOGEN
000100                               2 SHIFTB   START 256
000100 05B0                          3 BEGIN    BALR  11,0
000102                               4          USING *,11
000102 5860 B02A            0012C    5          L     6,FWORD          78ABCDEF    00000000
000106 8C60 0008            00008    6          SRDL  6,8              0078ABCD    EF000000
00010A 8A70 0018            00018    7          SRA   7,24             0078ABCD    FFFFFFEF
00010E 4070 B032            00134    8          STH   7,C              0078ABCD    FFFFFFEF
000112 8C60 000C            0000C    9          SRDL  6,12             0000078A    BCDFFFFF
000116 8A70 0014            00014   10          SRA   7,20             0000078A    FFFFFBCD
00011A 4070 B030            00132   11          STH   7,B              0000078A    FFFFFBCD
00011E 8C60 000C            0000C   12          SRDL  6,12             00000000    78AFFFFF
000122 8A70 0014            00014   13          SRA   7,20             00000000    0000078A
000126 4070 B02E            00130   14          STH   7,A              00000000    0000078A
                                    15          EOJ
00012C                              18 FWORD    DS    F
000130                              19 A        DS    H
000132                              20 B        DS    H
000134                              21 C        DS    H
000100                              22          END   BEGIN
```

Figure 3-12. Modified version of the program of Figure 3-11, making it operate correctly with negative quantities

changed to expand the sign of item A. The final two shifts are added to do this. Actually, it could be done more efficiently and these extra steps avoided simply by changing the SRDL in statements 6 and 9 to the algebraic SRDA.

Figure 3-13 shows the output of the two programs for the sample input word of 78ABCDEF. The three parts of the combined word, in hexadecimal, were therefore 78A, BCD, and EF. In the first line of Figure 3-13 we see that these have been put into halfwords by the first program as 078A, 0BCD and 00EF, that is, as three positive numbers. In the second line we see that the second program, on the other hand, interpreted the second and third numbers as negative because their leftmost bits were 1's. The three output halfwords are 078A, FBCD, and FFEF, showing that the sign bits of the numbers were properly expanded.

```
PROG SHIFTA          078A        0BCD        00EF

PROG SHIFTB          078A        FBCD        FFEF
```

Figure 3-13. Output of the two programs executed with hexadecimal 78ABCDEF for the fullword

BRANCHES AND DECISION CODES

The Condition Code

Decisions and branching are important parts of data processing, and the programming methods by which these operations are carried out are important aspects of the programming task. The facilities offered by the System/360 are particularly powerful and flexible. The basic action is the setting of the *condition code* by any of a large number of instructions and the subsequent testing of the condition code by a Branch on Condition instruction.

Many arithmetic, shift, and logical instructions have as a part of their action the setting of the condition code to indicate something about the result of the instruction's execution. For instance, after an Add instruction, the condition code indicates whether the sum was zero, positive, negative, or too large for the register. After a Compare instruction the condition code indicates whether the first operand was greater than, equal to, or less than the second operand. The meaning of each of the different states or values of the condition code is specified in the description of each instruction that affects the code. These descriptions may be found in the *System/360 Principles of Operation*, which also contains a complete tabulation of the instructions involved and the meaning of the condition codes.

The condition code occupies two bits (in the control program area of storage). Two bits can, of course, be set in just four ways: 00, 01, 10, and 11; and these four binary settings are equal to decimal values 0, 1, 2, and 3, respectively.

At any time after the condition code has been set by the action of an instruction, it may be tested by using a Branch on Condition (BC) instruction. In this instruction, which is in the RX format, the four bits that in other instructions designate a general register are here used for a *mask* that designates in which states of the condition code we wish a certain branch to occur.

The leftmost bit of the mask checks for a condition code of zero, the next bit for code 1, the next for code 2, and the rightmost for code 3. If the condition code is equal to any of the values selected by the mask bit(s), the Branch is taken. The correspondences between condition codes and mask are summarized in Table 3-1.

Note that the mask bits correspond from left to right with the four condition codes. Another way, perhaps easier to remember, of summarizing this correspondence is as follows:

Condition code	0	1	2	3
Mask used to test code	8	4	2	1

A BC instruction with a decimal mask of 12 (8+4) specifies that a branch is to be made if the condition code is 0 or 1, and is not to be made if the condition code is 2 or 3. A mask of 7 (4+2+1) will cause a branch only if the condition code is 1, 2, or 3.

A decimal mask value of zero makes the instruction test for no condition codes; it thus becomes a no-operation instruction. A mask of 15 tests for *any* condition code; it is thus an unconditional branch.

Table 3-1. Masks for testing various states of the condition code

Mask bits	*Decimal value*	*Condition codes tested*
0000	0	None
0001	1	3
0010	2	2
0011	3	2 or 3
0100	4	1
0101	5	1 or 3
0110	6	1 or 2
0111	7	1, 2, or 3
1000	8	0
1001	9	0 or 3
1010	10	0 or 2
1011	11	0, 2, or 3
1100	12	0 or 1
1101	13	0, 1, or 3
1110	14	0, 1, or 2
1111	15	0, 1, 2, or 3

A Sorting Procedure

To see how some of these ideas are applied, consider a simple example. We are given three fullword data items named A, B, and C. They may be positive or negative. We are required to change any negative values to positive, and then to rearrange the three values in storage to make the number in A the largest, the number in B the next largest, and the number in C the smallest of the three. Figure 3-14 expresses the logic of the method that will be used here to perform the sort; other ways are possible.

We first make all three numbers positive. A comparison is then made between A and B; if A is the smaller, we interchange the two values. Now we know that the value in A is the larger of the two, whether it originally was or not. A similar process compares A and C and interchanges if A is smaller. Having done this, we know that what is in A is the largest of the three. A final comparison of the numbers now

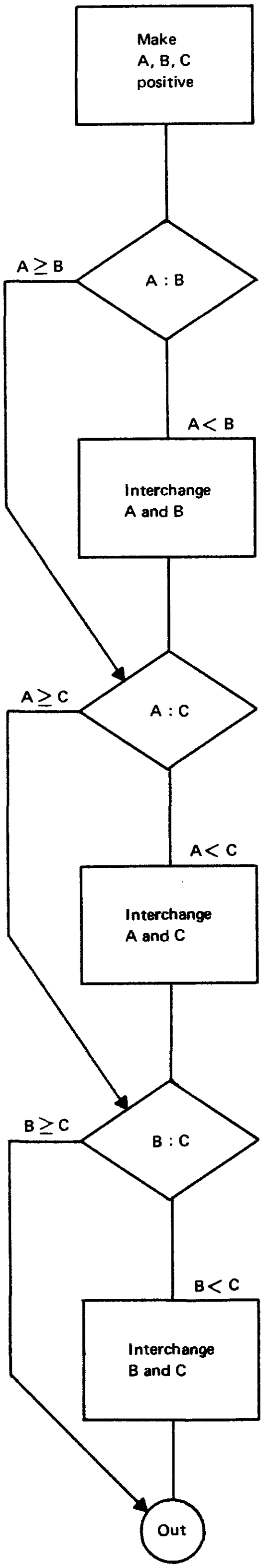


Figure 3-14. Program flowchart of a method of sorting three numbers into descending sequence. Any negative numbers are changed to positive before sorting.

in B and C, and an interchange if necessary, gets the "middle" number in B and the smallest in C.

The program of Figure 3-15 involves some instructions that we have not used before. The Load Multiple (LM) instruction begins loading fullwords from the specified storage location. The first word goes into the first-named register. Successive fullwords go into higher-numbered registers until the second-named register has been loaded. In the program, the result of the LM instruction will be to place A in 2, B in 3, and C in 4.

Now three Load Positive Register (LPR) instructions change any negative numbers to positive, leaving any positive numbers unchanged. This is an RR format instruction, meaning that both of its operands are registers. Here both operands are the same register, as will frequently be the case. The action is to take the value from a register, complement it if it is negative, and place the result back in the same register. If it were necessary, two different registers could of course be used.

Next comes a Compare Register (CR) instruction, which is also in the RR format. This instruction does not change the contents of either register, but simply sets the condition code to zero if the two operands are the same, to 1 if the first operand is low, and to 2 if the first operand is high. (The comparison is algebraic, meaning that signs are taken into account according to the rules of algebra, by which any positive number is greater than any negative number. We know that our numbers are by now all positive, so this feature does not concern us.)

Next comes the Branch on Condition instruction, with a mask of 10 (decimal) and a branch address of COMP2. The mask of 10, checking with the table above, tests for condition code zero or 2. Following a Compare-type instruction, these mean, respectively, that the first operand is equal to or greater than the second operand. If the condition code is either of these, we branch; otherwise the next instruction in sequence is taken. The effect is: if the number in register 2 is already equal to or greater than the number in register 3, we skip down to the second comparison, because A and B are already in correct sequence.

The interchange, if it is necessary, is performed by moving the contents of register 2 to register 6, moving 3 to 2, and finally moving 6 to 3. These transfers are made with the Load Register (LR) instruction.

The remaining instructions repeat these operations twice for the other comparisons. Finally, there is a Store Multiple (STM) instruction to place the rearranged items back in the original three locations, as required by the problem statement.

Figure 3-16 shows before-and-after values of A, B, and C for six possible original orderings of the three values. Each pair of lines is one set. These are hexadecimal numbers; the original value of A in the last set is -3.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      1          PRINT NOGEN
000100                                2 SORT     START 256
000100 05B0                           3 BEGIN    BALR  11,0
000102                                4          USING *,11
000102 9824 B036            00138     5          LM    2,4,A          LOAD REGISTERS WITH 3 NUMBERS
000106 1022                           6          LPR   2,2            MAKE NUMBERS POSITIVE
000108 1033                           7          LPR   3,3
00010A 1044                           8          LPR   4,4
00010C 1923                           9          CR    2,3            COMPARE A AND B
00010E 47A0 B016            00118    10          BC    10,COMP2
000112 1862                          11          LR    6,2            INTERCHANGE IF NECESSARY
000114 1823                          12          LR    2,3
000116 1836                          13          LR    3,6
000118 1924                          14 COMP2    CR    2,4            COMPARE A AND C
00011A 47A0 B022            00124    15          BC    10,COMP3
00011E 1862                          16          LR    6,2            INTERCHANGE IF NECESSARY
000120 1824                          17          LR    2,4
000122 1846                          18          LR    4,6
000124 1934                          19 COMP3    CR    3,4            COMPARE B AND C
000126 47A0 B02E            00130    20          BC    10,OUT
00012A 1863                          21          LR    6,3            INTERCHANGE IF NECESSARY
00012C 1834                          22          LR    3,4
00012E 1846                          23          LR    4,6
000130 9024 B036            00138    24 OUT      STM   2,4,A          STORE SORTED VALUES
                                     25          EOJ
000136 0000
000138 00000001                      28 A        DC    F'1'
00013C 00000002                      29 B        DC    F'2'
000140 00000003                      30 C        DC    F'3'
000100                               31          END   BEGIN
```

Figure 3-15. Assembly listing of a program to carry out the sorting procedure charted in Figure 3-14

```
INPUT1    00000001  00000002  00000003
OUTPUT1   00000003  00000002  00000001

INPUT2    00000001  00000003  00000002
OUTPUT2   00000003  00000002  00000001

INPUT3    00000002  00000001  00000003
OUTPUT3   00000003  00000002  00000001

INPUT4    00000003  00000002  00000001
OUTPUT4   00000003  00000002  00000001

INPUT5    00000003  00000001  00000002
OUTPUT5   00000003  00000002  00000001

INPUT6    FFFFFFFD  00000002  00000001
OUTPUT6   00000003  00000002  00000001
```

Figure 3-16. Six sets of sample input and output for the program of Figure 3-15

FURTHER DECISIONS: THE SOCIAL SECURITY PROBLEM

In this application, which is presumably familiar to many readers, we combine two decisions with some arithmetic processing.

We are given a man's earnings for a week (EARN), his previous ("old") year-to-date earnings (OLDYTD), and his previous year-to-date Social Security tax (OLDFICA). We are to compute his Social Security tax for this week (TAX), his new year-to-date earnings (NEWYTD) and new Social Security tax (NEWFICA). Assume the Social Security tax is computed as 4.4% of earnings (with certain exclusions such as sick pay, which we shall ignore) up to an annual limit on taxable income of $7800. The program must decide whether the employee has yet earned $7800 this year; if so, he is exempt from further Social Security tax. Actually, the situation is slightly more complex than that: if the man has not yet earned $7800 before this week's pay but, counting this week's pay, goes over $7800, only the portion of this week's pay that takes him up to the $7800 limit is taxable.

The flowchart of Figure 3-17 expresses the logic we have just described. Figure 3-18 translates this logic into a program illustrating in the process that there are many ways to implement a flowchart.

We begin by loading the previous year-to-date into a register, and from there immediately load it into another register, in order to have it both places. This method saves a little time over loading twice from storage. We add this week's earnings, giving the new year-to-date, which is stored. Once this is done, we no longer need the same information in register 6, so this register is free for any other processing we will need to do. Now we compare the *old* year-to-date with $7800. The Branch on Condition that follows asks whether the condition code is 1, that is, whether the first operand is low. This can be read: branch if the old year-to-date was less than $7800. If the branch is not taken, the old year-to-date was already over $7800, so there is no tax to pay. We clear register 7, where the tax is developed if there is any, by subtracting it from itself – the fastest and simplest way to clear a register to zero. The Branch on Condition with a mask of 15 is an unconditional branch down to the final instructions where the tax is stored and the Social Security updated.

If the branch is taken, there is a tax to be paid, but we still need to know whether this week took the man over the top. Accordingly, at the instruction labeled YES, we compare the new year-to-date with $7800. The Branch on Condition with a mask of 2 asks whether the first operand – the new year-to-date – was greater than $7800. If so, it is necessary to compute the tax on just that part of this week's pay that takes the total up to $7800. At OVER78, accordingly, we load register 7 with $7800 and subtract the previous year-to-date; the difference is just the amount that is taxable. If the branch was not taken, the full week's earnings are taxable, and they are therefore loaded into register 7 and we branch unconditionally to MULT.

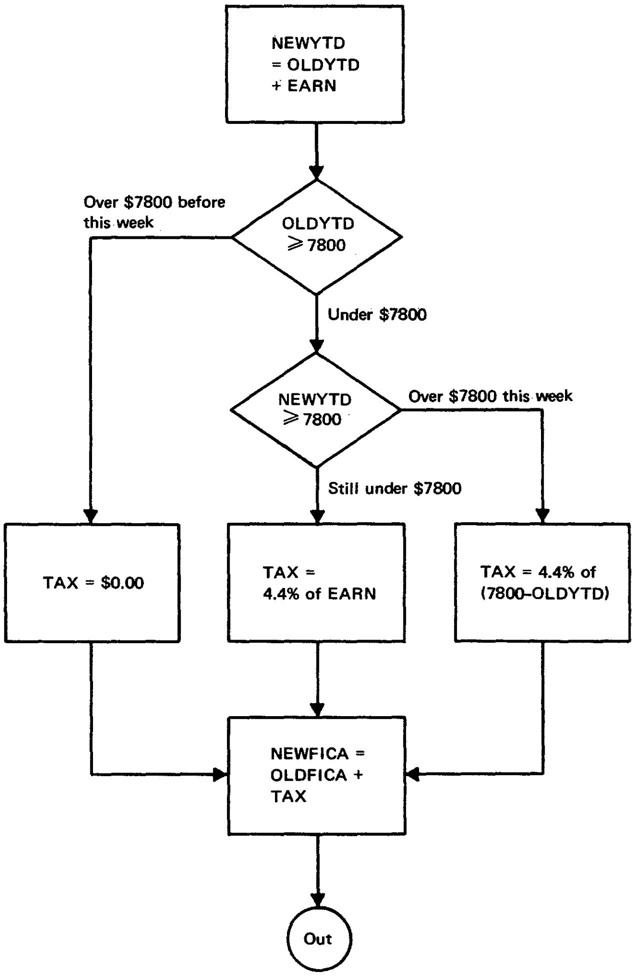


Figure 3-17. Program flowchart of a procedure for computing Social Security tax

At that location is an instruction to multiply whatever is in register 7 – either the full week's pay or some part of it – by 4.4%. This constant is entered as the integer 44. We must think of this number as 0.044, however, remembering that it is a fraction. The constant for rounding, HALF, is therefore 500, and we remove all the excess decimals by dividing by 1000. At this point the tax is in register 7 ready to be stored by the instruction at STORE. This same Store instruction is the one to which we branched if there was no tax to pay, having cleared register 7. A final Add and Store update the year-to-date Social Security.

This program fulfills the requirements of the problem statement and does the processing described by the flowchart – but it is quite unacceptable. The problem is something not mentioned in the problem statement. Let us see what the trouble is by looking at an example.

Suppose we have a man who earns $164.00 per week. Multiplying by 0.044 and rounding to the nearest cent, we get a Social Security tax of $7.22. In 47 weeks of working at this rate, the man will accumulate year-to-date earnings of $7708.00 and a year-to-date Social Security tax of $339.34. Now in the next week his full earnings are not taxable, but only the part that takes him up to $7800, or $92.00; the tax on this amount is $4.05. Adding $4.05 to his previous year-to-date Social Security, we get $343.39, which is more than 4.4% of the $7800 maximum.

The difficulty is in the computation of the tax on one week's earnings. Before rounding, the product of $164.00 and 0.044 is $7.21600. When we round this to $7.22 we add nearly half a cent. For each of the 47 weeks we are adding nearly half a cent.

This would be inaccurate. Social Security tax is seldom computed the way we have shown.

Fortunately, correcting the trouble is not only fairly easy, but leads to a shorter program. The general approach is to compute 4.4% of the new year-to-date earnings, then compute the tax by subtracting from this the previous year-to-date Social Security. The effects of the rounding error are thus balanced from week to week, and we are never more than half a cent off in the accumulated total.

Consider the example given above. The first week of the year, we get $7.22 as the tax. The second week, we begin by computing 0.044 times $328.00, the new year-to-date gross; this gives us $14.43 as the new year-to-date Social Security, which we store. This week's tax is $14.43 minus the previous year-to-date Social Security of $7.22, or $7.21. In other words, where last week we were a fraction of a cent high, now we are a fraction of a cent low; the two tend to cancel each other. The offset may not always be equal; however, we can never be more than half a cent off.

The test for reaching the maximum taxable amount is now made in terms of the tax instead of the earnings. We compute the Social Security on the new year-to-date earnings, then ask whether the result is greater than $343.20. If so, the result is replaced by $343.20 and the tax is computed as before, by subtracting the previous year-to-date Social Security. If that was already $343.20, that is, if the maximum had already been reached, then the tax computed by this method is zero, as it should be. If this week's pay goes over the taxable limit, the tax is the difference between the maximum tax and the amount already paid, which is correct.

The program shown in Figure 3-19 should not be too difficult to follow after the description of the process that has just been given. The program is eight instructions

```
  LOC  OBJECT CODE    ADDR1 ADDR2 STMT   SOURCE STATEMENT

                                1          PRINT NOGEN
000100                          2 FICA1    START 256
000100 05B0                     3 BEGIN    BALR  11,0
000102                          4          USING *,11
000102 5860 B052       00154    5          L     6,OLDYTD
000106 1856                     6          LR    5,6
000108 5A60 B04E       00150    7          A     6,EARN
00010C 5060 B056       00158    8          ST    6,NEWYTD
000110 5950 B066       00168    9          C     5,C7800
000114 4740 B01C       0011E   10          BC    4,YES
000118 1B77                    11          SR    7,7
00011A 47F0 B040       00142   12          BC    15,STORE
00011E 5960 B066       00168   13 YES      C     6,C7800
000122 4720 B02C       0012E   14          BC    2,OVER78
000126 5870 B04E       00150   15          L     7,EARN
00012A 47F0 B034       00136   16          BC    15,MULT
00012E 5870 B066       00168   17 OVER78   L     7,C7800
000132 5B70 B052       00154   18          S     7,OLDYTD
000136 5C60 B06A       0016C   19 MULT     M     6,C44
00013A 5A70 B06E       00170   20          A     7,HALF
00013E 5D60 B072       00174   21          D     6,CHUN
000142 5070 B062       00164   22 STORE    ST    7,TAX
000146 5A70 B05A       0015C   23          A     7,OLDFICA
00014A 5070 B05E       00160   24          ST    7,NEWFICA
                               25          EOJ

000150 00004010                28 EARN     DC    F'16400'
000154 000BBFD0                29 OLDYTD   DC    F'770000'
000158                         30 NEWYTD   DS    F
00015C 00008408                31 OLDFICA  DC    F'33800'
000160                         32 NEWFICA  DS    F
000164                         33 TAX      DS    F
000168 000BE6E0                34 C7800    DC    F'780000'
00016C 0000002C                35 C44      DC    F'44'
000170 000001F4                36 HALF     DC    F'500'
000174 000003E8                37 CHUN     DC    F'1000'
000100                         38          END   BEGIN
```

Figure 3-18. Assembly listing of a program based on the flowchart in Figure 3-17 shorter and considerably less complex. Both versions have been tested with a variety of data; both give "correct" results in that they do what we expect, although of course the results are not identical.

The only new instruction used in this program is BL UNDER, which means Branch on Low to the address UNDER. BL is an *extended mnemonic code*; it is translated by the assembler to the Branch on Condition operation code (47) with a decimal mask of 4. Other extended mnemonics used after Compare instructions are BH (Branch on High) for BC 2, BE (Branch on Equal) for BC 8, BNH (Branch on Not High) for BC 13, and so on. Additional extended mnemonics can be used after arithmetic operations and Test under Mask instructions. They are supplied with most System/360 assemblers and are a great convenience in writing and checking conditional branching instructions, since they specify the conditions. A full list is given in the Appendix.

```
  LOC   OBJECT CODE     ADDR1 ADDR2  STMT    SOURCE STATEMENT

                                       1            PRINT NOGEN
000100                                 2 FICA2      START 256
000100 05B0                            3 BEGIN      BALR  11,0
000102                                 4            USING *,11
000102 5850 B036            00138      5            L     5,OLDYTD
000106 5A50 B032            00134      6            A     5,EARN
00010A 5050 B03A            0013C      7            ST    5,NEWYTD
00010E 5C40 B04A            0014C      8            M     4,C44
000112 5A50 B04E            00150      9            A     5,HALF
000116 5D40 B052            00154     10            D     4,CHUN
00011A 5950 B056            00158     11            C     5,MAX
00011E 4740 B024            00126     12            BL    UNDER
000122 5850 B056            00158     13            L     5,MAX
000126 5050 B042            00144     14 UNDER      ST    5,NEWFICA
00012A 5B50 B03E            00140     15            S     5,OLDFICA
00012E 5050 B046            00148     16 STORE      ST    5,TAX
                                      17            EOJ
000134 00004010                       20 EARN       DC    F'16400'
000138 000BBFD0                       21 OLDYTD     DC    F'770000'
00013C                                22 NEWYTD     DS    F
000140 00008408                       23 OLDFICA    DC    F'33800'
000144                                24 NEWFICA    DS    F
000148                                25 TAX        DS    F
00014C 0000002C                       26 C44        DC    F'44'
000150 000001F4                       27 HALF       DC    F'500'
000154 000003E8                       28 CHUN       DC    F'1000'
000158 00008610                       29 MAX        DC    F'34320'
000100                                30            END   BEGIN
```

Figure 3-19. Assembly listing of a much better version of the program to calculate Social Security tax

SIMPLE LOOPS: FINDING A SUM

A frequent programming requirement is to perform some operation on a set of values arranged in some systematic way in storage. We shall examine some of the coding methods available for such operations in the System/360, in terms of a very simple example.

For our illustrative problem, suppose that there are 20 fullwords in consecutive fullword locations starting with the one identified by the symbol TABLE. We are required to form the sum of the 20 numbers and place it in SUM.

We shall consider the three different ways of doing this. All three involve the use of an index register to modify the effective address in an instruction. The contents of the index register are changed between repetitions of the loop.

The first version of the program is shown in Figure 3-20. We shall use register 8 to accumulate the sum and register 11 as the index register. We want register 8 cleared to zero so that the sum will be correct; as it happens, we want the index register cleared to zero also. Both operations are done with Subtract Register instructions.

Now comes the instruction that does the actual computing. We add to register 8 the contents of some fullword in storage. The first time through the loop we want to add the word at TABLE. The instruction specifies that the contents of index register 11 should be used in computing the effective address – but we just made those contents zero, so the effective address is that of the word at TABLE. The first time through the loop, this instruction therefore adds the word at TABLE to register 8, which was cleared to zero.

The next time through the loop, we want the fullword at TABLE+4 added to register 8. This can be accomplished by adding 4 to the index register. In this version of the program, we do so with an Add instruction.

Now we are at the point in the program where a test for completion must be made. The last of the 20 words is located at TABLE+76. We are modifying before testing, however. At the point where the loop has just been executed with TABLE+76 for an effective address, we will now have 80 in the index register. That is, therefore, the correct constant to use in testing for completion. We do so with a Compare, then Branch on Condition with a mask that asks for a branch if the index was less than 80. We could use the extended mnemonic code BL and write the branch instruction as BL LOOP; the object program would be the same.

The branch will be executed 19 times, giving 20 executions of the Add at LOOP. After that, the branch is not executed, we store the total at SUM, and the program is completed.

The reader will no doubt have recalled the customary names for the parts of a loop. The part at the beginning that gets the loop started is the *initialization* section; here, it consists of the first two instructions. The part that does the

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1           PRINT NOGEN
000100                               2 SUMA      START 256
000100 0530                          3 BEGIN     BALR  3,0
000102                               4           USING *,3
000102 1B88                          5           SR    8,8
000104 1BBB                          6           SR    11,11
000106 5A8B 301A            0011C    7 LOOP      A     8,TABLE(11)
00010A 5AB0 306E            00170    8           A     11,C4
00010E 59B0 3072            00174    9           C     11,C80
000112 4740 3004            00106   10           BC    4,LOCP
000116 5080 306A            0016C   11           ST    8,SUM
                                    12           EOJ
00011C 00000001                     15 TABLE     DC    F'1'
000120 00000002                     16           DC    F'2'
000124 00000003                     17           DC    F'3'
000128 00000004                     18           DC    F'4'
00012C 00000005                     19           DC    F'5'
000130 00000006                     20           DC    F'6'
000134 00000007                     21           DC    F'7'
000138 00000008                     22           DC    F'8'
00013C 00000009                     23           DC    F'9'
000140 0000000A                     24           DC    F'10'
000144 0000000B                     25           DC    F'11'
000148 0000000C                     26           DC    F'12'
00014C 0000000D                     27           DC    F'13'
000150 0000000E                     28           DC    F'14'
000154 0000000F                     29           DC    F'15'
000158 00000010                     30           DC    F'16'
00015C 00000011                     31           DC    F'17'
000160 00000012                     32           DC    F'18'
000164 00000013                     33           DC    F'19'
000168 00000014                     34           DC    F'20'
00016C                              35 SUM       DS    F
000170 00000004                     36 C4        DC    F'4'
000174 00000050                     37 C80       DC    F'80'
000100                              38           END   BEGIN
```

Figure 3-20. First version of a program to form the sum of 20 numbers actual work of the loop is called the *compute* part, and here consists of the Add at LOOP. The *modification* section changes something between repetitions; here, it is the modification of the index contents by the Add. The *testing* section determines whether the action of the loop has been completed, and consists here of the Compare and the Branch on Condition. The sequence of the last three sections is not always as in this example. And as we shall see in the third version, the modification and testing can often be combined into one instruction.

The second version shortens the repeated section of the loop by one instruction. Normally, we do not worry too much about trying to get the last microsecond out of programs, but in heavily repeated parts it is worth some effort.

The method will require us to go "backward" through the table, which in this particular example is permissible; sometimes, of course, it would not be. As shown in Figure 3-21 we again clear register 8. This time, however, instead of loading the index register (11) with zero, we use a new instruction, Load Address, to put 76 in it. The Load Address (LA) simply puts the address part of the instruction itself in the designated register; there is no reference to storage whatsoever. In this case, 76 is actually the displacement and there is no base or index register. If we wanted to state this specifically, the statement could be written LA 11,76(0,0).

Now when we execute the indexed Add instruction at LOOP, the effective address is TABLE+76. Following this, we subtract 4 from the index register. As it happens, the execution of a Subtract sets the condition code. A condition code of zero indicates that the result was zero, 1 indicates a negative result, and 2 a positive result. (A code of 3 indicates an overflow – a result too large to hold in the register. If the program is correct an overflow cannot occur here, so the possibility does not concern us.) We want to branch back to LOOP as long as the result of the subtraction is either positive or zero, so the mask on the Branch on Condition is 10: 8 picks condition code zero and 2 picks up code 2.

The Store is as before.

Where in the first version there were four instructions in the repeated portion of the loop, here there are three. The final version reduces this number to the minimum, two. The technique is to use the Branch on Index Low or Equal instruction (BXLE), which is a combination of an Add, a comparison, and a conditional branch.

Let us assume we have three registers set up as follows: Register 9 will be the index; it initially contains zero. Register 10 will contain the amount by which the index is to be incremented each time around the loop, 4. Register 11 will contain the limit value, the value of the index which is not to be exceeded, 76. If we have the instruction:

BXLE 9,10,LOOP

the action will be as follows: The contents of register 10 (4) are added to register 9, which is the index and initially contains zero. If the sum is less than or equal to the

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1           PRINT NOGEN
000100                               2 SUMB      START 256
000100 0530                          3 BEGIN     BALR  3,0
000102                               4           USING *,3
000102 1B88                          5           SR    8,8
000104 41B0 004C          0004C      6           LA    11,76
000108 5A8B 301A          0011C      7 LOOP      A     8,TABLE(11)
00010C 5BB0 306E          00170      8           S     11,C4
000110 47A0 3006          00108      9           BC    10,LOOP
000114 5080 306A          0016C     10           ST    8,SUM
                                    11           EOJ
00011A 0000
00011C 00000001                     14 TABLE     DC    F'1'
000120 00000002                     15           DC    F'2'
000124 00000003                     16           DC    F'3'
000128 00000004                     17           DC    F'4'
00012C 00000005                     18           DC    F'5'
000130 00000006                     19           DC    F'6'
000134 00000007                     20           DC    F'7'
000138 00000008                     21           DC    F'8'
00013C 00000009                     22           DC    F'9'
000140 0000000A                     23           DC    F'10'
000144 0000000B                     24           DC    F'11'
000148 0000000C                     25           DC    F'12'
00014C 0000000D                     26           DC    F'13'
000150 0000000E                     27           DC    F'14'
000154 0000000F                     28           DC    F'15'
000158 00000010                     29           DC    F'16'
00015C 00000011                     30           DC    F'17'
000160 00000012                     31           DC    F'18'
000164 00000013                     32           DC    F'19'
000168 00000014                     33           DC    F'20'
00016C                              34 SUM       DS    F
000170 00000004                     35 C4        DC    F'4'
000100                              36           END   BEGIN
```

Figure 3-21. Second version of program to form the sum of 20 numbers contents of register 11, the limit, the branch to LOOP is taken; otherwise the next instruction in sequence is taken.

The instruction is written in assembler language in the general form:

BXLE R1,R3,D2(B2)

Three factors, each of which must be located in a register, are required by the BXLE instruction. An index must be in the register specified by R1. An increment must be in the register specified by R3. A limit value must also be in a register but the register is not explicitly specified in the instruction. The BXLE instruction will first add the increment to the index. It will then compare the resultant index against the limit. If the index is less than or equal to the limit, a branch is taken to the location specified by D2(B2); otherwise the next instruction in sequence is taken. The register containing the limit value is always odd-numbered and is chosen in the following way:

1. If the register specified by R3 is an even-numbered register, the limit value is assumed to be in the next higher-numbered register. If we have the instruction:

BXLE 9,10,LOOP

the limit value is in register 11, the next higher-numbered register.

2. If the register specified by R3 is an odd-numbered register, a third register is not used. In this case the BXLE instruction assumes that R3 specifies the register to be used for both the increment and the limit. If we have the instruction:

BXLE 6,7,LOOP

register 7 will be used by BXLE as the source of the increment and the limit.

At first glance this instruction seems more complicated than it is. Let us turn to an example to see how it works. Figure 3-22 is the final version of our summing loop.

We begin the program by loading the three registers that will be used by the BXLE instruction (registers 9, 10, and 11), with the desired initial contents. We then proceed to the Add instruction at LOOP, which is the same as in the previous two versions. Next comes the BXLE, which operates as described.

The operation of the BXLE instruction is most easily remembered if we think in terms of three registers representing the index, the increment, and the limit, in that order.

For a situation where it is desired to work backwards, in which case the increment would be negative, the Branch on Index High (BXH) instruction is available.

The BXLE and BXH instructions are very powerful and very flexible. They will find heavy use in many practical applications, and are well worth the investment of effort necessary to understand them fully.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1           PRINT NOGEN
000100                               2 SUMC      START 256
000100 0530                          3 BEGIN     BALR  3,0
000102                               4           USING *,3
000102 1B88                          5           SR    8,8
000104 1B99                          6           SR    9,9
000106 41A0 0004            00004    7           LA    10,4
00010A 41B0 004C            0004C    8           LA    11,76
00010E 5A89 301A            0011C    9 LOOP      A     8,TABLE(9)
000112 879A 300C            0010E   10           BXLE  9,10,LOOP
000116 5080 306A            0016C   11           ST    8,SUM
                                    12           EOJ
00011C 00000001                     15 TABLE     DC    F'1'
000120 00000002                     16           DC    F'2'
000124 00000003                     17           DC    F'3'
000128 00000004                     18           DC    F'4'
00012C 00000005                     19           DC    F'5'
000130 00000006                     20           DC    F'6'
000134 00000007                     21           DC    F'7'
000138 00000008                     22           DC    F'8'
00013C 00000009                     23           DC    F'9'
000140 0000000A                     24           DC    F'10'
000144 0000000B                     25           DC    F'11'
000148 0000000C                     26           DC    F'12'
00014C 0000000D                     27           DC    F'13'
000150 0000000E                     28           DC    F'14'
000154 0000000F                     29           DC    F'15'
000158 00000010                     30           DC    F'16'
00015C 00000011                     31           DC    F'17'
000160 00000012                     32           DC    F'18'
000164 00000013                     33           DC    F'19'
000168 00000014                     34           DC    F'20'
00016C                              35 SUM       DS    F
000100                              36           END   BEGIN
```

Rc, B = 102

Figure 3-22. Third and shortest version of program to form the sum of 20 numbers, using the BXLE instruction

CASE STUDY: AVERAGING A LIST OF TEMPERATURES

In an attempt to draw together some of the things that have been discussed in this chapter, we shall now consider a final problem that involves several different concepts.

Suppose we have in storage a group of halfwords giving the temperature, to the nearest degree, on each of the days of a month. There may be 28, 29, 30, or 31 of them; the number is given by a halfword named DAYS. The table of temperatures begins at TEMP and continues for a total of 31 halfwords; if there are fewer than 31 days in the month at hand, the last entries of the table are to be ignored. It is possible that the temperature reading may be missing for some days; a missed reading is indicated in storage by a halfword of all 1's. We are to form the average of the temperatures for the month, using only as many good readings as are found. If the entire table should happen to contain bad readings, a halfword of all 1's should be stored to indicate that the average was not computed. In any case, we are to store in NGOOD the number of good readings found. The average should be rounded off to the nearest degree.

The program shown in Figure 3-23 uses the halfword variations of a number of instructions that should be quite familiar in their fullword forms.

Before analyzing the operation of the program, it may be helpful to summarize the functions of the registers used, which will often be a valuable thing for the programmer to do.

Register	*Usage*
3	Base register
4	Index register
5	Word of 1's
6	Left half of dividend
7	Sum of temperatures—right half of dividend
8	Count of nonzero temperatures
10	Increment for BXLE
11	Limit for BXLE

The initialization consists of setting up the contents of the seven registers used by the program. The first one to be set to zero (6) is cleared by a Subtract Register, the others by Load Registers from 6. The Load Halfword to get the

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT    SOURCE STATEMENT

                                      1            PRINT NOGEN
000100                                2 AVGTEMP    START 256
000100 0530                           3 BEGIN      BALR  3,0
000102                                4            USING *,3
000102 4850 3094            00196     5            LH    5,ONES
000106 1866                           6            SR    6,6
000108 1876                           7            LR    7,6
00010A 1886                           8            LR    8,6
00010C 41A0 0002            00002     9            LA    10,2
000110 48B0 3096            00198    10            LH    11,DAYS
000114 4BB0 3092            00194    11            SH    11,ONE
000118 8BB0 0001            00001    12            SLA   11,1
00011C 1846                          13            LR    4,6
00011E 4954 3054            00156    14 LOOP       CH    5,TEMP(4)
000122 4780 302C            0012E    15            BE    ZERO                EXTENDED MNEMONIC FOR BC 8
000126 4A74 3054            00156    16            AH    7,TEMP(4)
00012A 4A80 3092            00194    17            AH    8,ONE
00012E 874A 301C            0011E    18 ZERO       BXLE  4,10,LOOP
000132 4080 309A            0019C    19            STH   8,NGOOD
000136 1288                          20            LTR   8,8
000138 4770 3040            00142    21            BNZ   NOT                 EXTENDED MNEMONIC FOR BC 7
00013C 4050 3098            0019A    22            STH   5,AVER              STORE ONES IF NO GOOD DATA
                                     23            EOJ                       STOP
000142 8B70 0001            00001    26 NOT        SLA   7,1                 TO GET EXTRA BINARY PLACE IN QUOTIENT
000146 1D68                          27            DR    6,8                 DIVIDE REGISTER
000148 4A70 3092            00194    28            AH    7,ONE               ROUND OFF
00014C 8A70 0001            00001    29            SRA   7,1                 DROP THE EXTRA BIT
000150 4070 3098            0019A    30            STH   7,AVER              FINAL RESULT
                                     31            EOJ                       END OF JOB
000156 0001                          34 TEMP       DC    H'1'
000158 0002                          35            DC    H'2'
00015A 0003                          36            DC    H'3'
00015C 0004                          37            DC    H'4'

00018C 001C                          60            DC    H'28'
00018E 001D                          61            DC    H'29'
000190 001E                          62            DC    H'30'
000192 001F                          63            DC    H'31'
000194 0001                          64 ONE        DC    H'1'
000196 FFFF                          65 ONES       DC    X'FFFF'
000198                               66 DAYS       DS    H
00019A                               67 AVER       DS    H
00019C                               68 NGOOD      DS    H
000100                               69            END   BEGIN
```

Figure 3-23. A program to compute average monthly temperature, which takes into account the possibility of omitted readings number of days into register 11 automatically expands the halfword into a fullword, which would mean that the sign bit of a negative number would be filled out. With correct data, the word here cannot be negative, of course. The number of days is to be used to terminate the summing loop that adds up the temperatures. The loop should be *executed* as many times as the number of days; it should be *repeated* (after the first time) one less time than the number of days. We accordingly subtract 1 from register 11 after loading it.

Since the table of data consists of halfwords, the index register will have to be incremented by 2 between loop repetitions, and the proper limit value is two less than double the number of days. We can double a number quite simply by shifting left one place in a binary machine. (If the table had consisted of fullwords, requiring an increment of 4, a left shift of two places would multiply the number of days by 4.)

In the working part of the loop we first check to see whether the particular temperature is valid, by comparing with the word of all 1's that had been set up in register 5. The Compare Halfword expands the halfword from storage to a fullword by propagating the sign bit. This is necessary to us, since the load halfword that put the word of all 1's in register 5 did the same thing. We next branch on equal to the instruction at ZERO, which would happen if the reading was bad. If it was good, the branch is not taken; we add in the temperature, add one to the count of good readings, and then reach the BXLE.

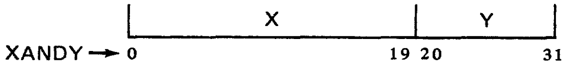
The BXLE increments the index register (4) by 2 (which is in 10) and checks whether the index is now the same as what we put in 11. If the index is low or equal, meaning that the list has not been exhausted, we branch back to LOOP to go around again.

When the loop is finished, we reach the Store Halfword after the BXLE. Here we store the count of good readings at NGOOD; this conceivably could be zero. Next we check whether it was zero, using the Load and Test Register instruction (LTR). With the two register designations being the same, as they are here, the effect of this instruction is to set the condition code according to the sign and magnitude of the count in register 8. The Branch on Condition instruction then asks whether the count was either positive or negative and branches if so. If it was neither of these it must have been zero, in which case we store the word of all 1's for the average in AVER, and stop.

If there was at least one good reading, we are ready now to compute the average. In order to be able to round off to the nearest degree, it is necessary to arrange the division so that the quotient has one binary place in it; this can be done by shifting the dividend to the left one place before dividing. The division is done this time with the Divide Register instruction, since the desired divisor (the count) is already in a register. Following the Divide Register we add 1 to the rightmost bit position of the quotient register to round off. Having done so, we shift the quotient back to the right to get rid of the extra bit and store the result.

QUESTIONS AND EXERCISES

1. The L, A, S, and ST instructions all operate on a (fullword, halfword).
2. The first operand of an instruction *usually* specifies the operand that (sends, receives) information.
3. In a ST instruction the first operand specifies the operand that (sends, receives). Does the ST instruction, in this respect, follow the general rule, or is it an exception to the general rule?
4. Is the instruction M 7,QTY a legitimate instruction? If not, why not?
5. The D instruction specifies ____________________ as the first operand, and the ____________________ as the second operand. After completion of the divide operation, where is the quotient located? Where is the remainder located?
6. Assume that a fullword area of storage (reserved by a DS), to be addressed as XANDY, contains two positive items as below:



Write the program to store X in a fullword area in storage called X, and Y in a halfword area in storage called Y.

7. The instruction BC 5,ROUT3 would branch to ROUT3 if the:
   a. Condition code is 5.
   b. Condition code is 1, 2, or 3.
   c. Condition code is 1 or 3.
8. Write an instruction to branch unconditionally to an instruction called NEWONE.
9. There are four fullwords named X1, X2, X3, and X4 sequentially located in storage. Write one instruction that loads these four fullwords into registers 2, 3, 4, and 5 respectively.
10. Write an instruction that clears register 5 to zero.
11. Consider the instruction named LOOP in Figure 3-20. How will the effective address of TABLE(11) be formed?
12. Write a single instruction that adds the contents of register 6 to register 5, tests to see if the sum now in register 5 is equal to or less than the contents of register 7, and then branches to an instruction called NEWONE if the answer is yes.

Chapter 4: Programming with Base Registers and the USING Instruction

A major programming feature of System/360 is the use of base registers for addressing main storage. One advantage is that compatibility is maintained between the small system with its short addresses and the large system with its longer addresses. The same instruction size and format accommodates both. Also, through appropriate use of base registers it is possible to relocate assembled programs almost at will. Great flexibility in program organization is thus achieved, since storage locations can be reassigned as dictated by the needs of the particular "mixture" of programs or program segments.

Base registers are thus deeply involved in programming and in program execution. However, as we shall see, it is possible to delegate to the assembler almost all the clerical work of keeping track of base registers and computing displacements. With a full understanding of these techniques, the programmer is able to leave the housekeeping to the assembler where appropriate, and to employ more sophisticated methods where needed.

In this chapter we shall see how the automatic techniques are called into operation and how the assembler implements them; and we will explore a few slightly more advanced techniques. As in so many other aspects of programming, particular emphasis must be placed on the question of when various actions occur: during assembly, linkage editing, or program execution.

THE USING INSTRUCTION

Automatic computation of the addresses of all operands in main storage requires the programmer to supply two items of information to the assembler and one to the object program.

With the USING instruction, the programmer tells the assembler:

1. Which general registers may be used as base registers
2. What each one will contain at the time the object program is executed

With this information the assembler can do its work of designating base registers and computing displacements.

It still remains to place in the base registers the values we have promised the assembler will be there. This can in principle be done in many ways, but the most common is to use the Branch and Link Register instruction (BALR). The general format of this instruction is:

BALR R1,R2

R1 receives the address of the next byte after the BALR; R2 supplies a branch address unless it is zero, in which case the next instruction in sequence is taken as usual. For our purposes here, the second operand (R2) is always zero. For instance, in the illustrative program we shall be considering shortly, we have an instruction:

BALR 11,0

This places in register 11 the address of the next byte after the BALR, and there is no branch. Register 11 was arbitrarily chosen as the base register for this program. It is used as a base register for most of the programs in this text. In actual practice, the choice of a base register cannot be a completely arbitrary one. As mentioned earlier, most installations find it necessary to establish rather rigid conventions for register usage. In addition, the various operating systems for System/360 make use of certain general registers for supervisor routines, linkages between separate programs, and other purposes. Under most operating systems, registers 2 through 11 are freely available to the programmer and should be used to avoid any complication.

AN EXAMPLE

These ideas may be made more concrete by considering an example. Figure 4-1 is an assembly listing of a program the processing details of which do not concern us.

The START instruction specifies that the assembled first byte of the program is location $256_{10} = 100_{16}$. We see that the BALR instruction has in fact been placed at 100. (All numbers in the object program area of the assembly listing – on the left-hand side – are hexadecimal.) The BALR instruction specifies that general purpose register 11 is to be loaded with the address of the next machine instruction. This, of course, is done at execution time by the machine. The USING instruction, which is an assembler instruction and takes no space in the object program, informs the assembler that general purpose register 11 is available for use as a base register and will contain the address of the next machine instruction, as signified by the asterisk. The BALR is a two-byte instruction so the next instruction, the Load, is placed at 102. This number, shown in the location counter column in the USING statement, indicates what the assembler assumed would be the contents of base register 11.

Let us look at the Load instruction to see how the assembler handled it. Reading from left to right the operation code is 58, the register loaded with a word from storage is number 2, no index register is specified, the base register is $B_{16} = 11_{10}$, and the displacement is 022_{16}. With base register 11 containing 102 and with a displacement of 22, we get an actual address of 124_{16}, as listed under ADDR2. Looking down the listing we see that 124 is in fact the address corresponding to the symbol DATA, as it should be.

The Add instruction is similar. With base register 11 again automatically designated, we have a base address of 102 and a displacement of 2A for an effective address of 12C, which is the address of the symbol TEN.

The Shift Left Algebraic instruction is a little different. All shift instructions have the RS format, with the index portion unused, but they still must specify a base register. Even though the effective "address" is never used for a storage reference, it is possible to make effective use of a variable number or positions of shift by varying the contents of the base register. In this program, however, such is not the case and we need a base register designation of zero. We see that this was done. The effective address is therefore just the displacement of 1. The remainder of the program presents no new base register concepts.

As always, it is most important to distinguish between what is done at assembly time and what is done at execution time. The assembler, in the example at hand, has filled in base register numbers where needed and has computed displacements. These base register numbers and displacements become part of the actual instructions, as listed down the left side of the assembly listing. In carrying out the assembly operations, the assembler had to know what base register we wished to use and what we planned to put in it; this information we provided with the USING.

The assembler cannot load the base register for the execution of our program, since that can be done only when the program is executed. We therefore provided the BALR instruction, which, at execution time, places the address of the next instruction into the specified register. The remainder of the program can now be carried out, with effective addresses being developed as intended.

The assembler program is actually processed in several separate phases. One of its functions is to determine the length and location of each instruction, area, and constant.

```
  LOC   OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                        1          PRINT NOGEN
000100                                  2 PROGE    START 256
000100 05B0                             3 BEGIN    BALR  11,0
000102                                  4          USING *,11
000102 5820 B022              00124     5          L     2,DATA
000106 5A20 B02A              0012C     6          A     2,TEN
00010A 8B20 0001              00001     7          SLA   2,1
00010E 5B20 B026              00128     8          S     2,DATA+4
000112 5020 B02E              00130     9          ST    2,RESULT
000116 5860 B032              00134    10          L     6,BIN1
00011A 5A60 B036              00138    11          A     6,BIN2
00011E 4E60 B03E              00140    12          CVD   6,DEC
                                       13          EOJ
000124 00000019                        16 DATA     DC    F'25'
000128 0000000F                        17          DC    F'15'
00012C 0000000A                        18 TEN      DC    F'10'
000130                                 19 RESULT   DS    F
000134 0000000C                        20 BIN1     DC    F'12'
000138 0000004E                        21 BIN2     DC    F'78'
000140                                 22 DEC      DS    D
000100                                 23          END   BEGIN
```

Figure 4-1. Listing of a program to show how the assembler calculates and supplies addresses of all storage operands. The processing performed is not intended to be realistic.

While doing this, the assembler constructs a symbol table. As shown in Figure 4-2, this lists for each symbol used in the program: its length in bytes, either its value or location (VALUE), the number of the statement in which it is defined (DEFN), and all statements in which it is referenced. With the length and location of each instruction and the base register information provided by the programmer in the USING instruction, the assembler is able, in a later phase, to calculate the base register and displacement and to list these and the actual assembled addresses of all operands as they appear in Figure 4-1.

In our program, we said with the START instruction that the first byte of the program should be assembled at location $256_{10} = 100_{16}$. Everything said so far has assumed that the program will actually be loaded at 100_{16}. This is not so. In the first place, this location is within the low area of main storage that is occupied by the supervisor and other parts of the control program, and could not be used for program execution. Parenthetically, it should be explained that START 256 is not a standard programming practice. We have chosen it for the examples in the first few chapters of this book to cause our assemblies to begin at some positive value, simply for illustrative purposes. The usual practice is to specify a zero START, which greatly simplifies the programmer's chore of calculating addresses, a necessity when debugging a program.

The second reason that our program will not be loaded at location 100_{16} is that, regardless of the location we give in the START statement, our assembled object program is in relocatable form and it is not executable until processed by the *linkage editor*. The linkage editor is an IBM service program that is part of the operating or programming support system.

The linkage editor assigns the actual starting address in main storage for each object program in a job input stream, and edits these into executable programs. It uses information supplied by the assembler regarding the length of the program, its name (given in the START statement, PROGE in this case), the assembled locations of any relocatable address constants, and other details necessary to perform the relocation.

When the program is in executable form, all statements, constants, reserved storage spaces, etc., remain in the same relative positions as in the assembly listing. Nothing needs to be changed to make the object program operate correctly from the new location or, at a later time, from still another location. All that is involved is the relocation factor.

Suppose the linkage editor assigns location 3200_{16} as the starting address. When the program has been loaded, it begins with execution of the BALR instruction. Now, what is the address of the next instruction after the BALR? The answer is 3202. This value goes into register 11 and becomes the base address. The displacements in the assembled instructions have not changed, of course. The effective address in the Load instruction is now 3202 + 22 = 3224. With the new starting location, 3224 is exactly where DATA appears. All other addresses are correctly computed as well, including the "address" in the Shift, which is completely unchanged since no base register is used.

It is also possible for the programmer by use of a control card to tell the linkage editor which starting location to assign. A complete relocation of the program after assembly is thus a simple matter. In more complex program structures, the linkage editor has more work to do than this example might suggest, but it is nevertheless feasible to execute programs from whatever storage locations may be convenient and available under any particular set of circumstances.

As we have noted, this simplicity of program relocation was one of the reasons for providing base registers in System/360.

The techniques of program relocation and the functions of the linkage editor will be discussed in more detail in the chapter on subroutines and program relocation.

```
                                              CROSS-REFERENCE

SYMBOL     LEN   VALUE   DEFN

BEGIN     00002 000100 00003    0023
BIN1      00004 000134 00020    0010
BIN2      00004 000138 00021    0011
DATA      00004 000124 00016    0005   0008
DEC       00008 000140 00022    0012
PROGE     00001 000100 00002
RESULT    00004 000130 00019    0009
TEN       00004 00012C 00018    0006


NO STATEMENTS FLAGGED IN THIS ASSEMBLY
```

Figure 4-2. Symbol cross-reference table constructed and listed by the assembler for the assembly in Figure 4-1

MORE THAN ONE BASE REGISTER

The displacement in an instruction is limited to a positive number in the range 0–4095_{10} = 0–FFF_{16}, since this is the limit that can be expressed in an unsigned 12-bit number. This means that an effective address cannot be less than the base address or more than 4095 greater, when an index register is not being used. If a program must reference a range of addresses greater than 4095, the easiest and most common approach in routine programming is to use more than one base register.

It should be noted, however, that it takes a rather large program segment to exhaust the range of displacements using one base register. With average length instructions, it takes a full pad of coding paper to use up 4096 bytes. It will usually be desirable to break a program this large into smaller segments anyway, so it will probably be extremely rare in practice to need more than one base register because of program length. Long sections of storage for data or results are another matter. Frequently it may be advantageous to assign one base register to the program and another to data. This is done in the last example in this chapter.

For now, to establish some basic ideas, let us make up a program that does use more than 4096 bytes for combined data and program. We shall not actually write an illustrative program that large, but we can simulate the effect of such a size by using the ORG assembler instruction to advance the location counter.

The partial program shown in Figure 4-3 was designed with the sole purpose of illustrating base register ideas; the "processing" is not intended to be meaningful. After the usual START, we have a BALR to load base register 11 with the address of the next instruction. The USING instruction is slightly different this time. Instead of using an asterisk to denote the address of the first byte of the following instruction, we give that instruction a symbolic name (HERE) and use the symbol. This gives exactly the same effect with respect to register 11, and permits us to refer to the contents of 11 in terms of a symbol, which we shall need for loading register 9. (The choice of register 9 was arbitrary.)

In loading the second base register, we cannot use a BALR; we want register 9 to contain not the address of the next instruction, but 4096 more than whatever went into 11. To accomplish this we use an address constant, named BASE in this case, which is written with the address HERE+4096. We see that the constant BASE has been assembled as we instructed: hexadecimal 1102 is 1000 greater than the value of the symbol HERE, and 1000_{16} = 4096_{10}.

Base register 9 will thus be loaded with 1102_{16} at execution time. This information is given to the assembler with a USING that has the address HERE+4096.

It is worthwhile noting which base register was used in the Load instruction that loaded base register 9: we see that the base register is 11 (which contains 102) and there is a displacement of A (+10 decimal). The effective address is thus 10C, which we see is indeed the address of the constant BASE. It is important to realize that at the time register 9 is being loaded, the only base register available is 11; the effective address of the instruction that loads 9 therefore cannot be more than 4096 greater than the contents of 11. Thus the address constant BASE cannot be at the end of the entire program, which would be more than 4096 bytes away. We have chosen to place it at almost the beginning and branch around it. Other placements are possible, so long as they do not cause the assembler to try to use a displacement in the Load instruction at HERE that is negative or greater than 4095.

```
  LOC  OBJECT CODE   ADDR1 ADDR2  STMT  SOURCE STATEMENT

000100                               1 PROGF   START 256
000100 05B0                          2 BEGIN   BALR  11,0
000102                               3         USING HERE,11
000102 5890 B00A           0010C     4 HERE    L     9,BASE
001102                               5         USING HERE+4096,9
000106 47F0 B00E           00110     6         BC    15,FIRST
00010A 0000
00010C 00001102                      7 BASE    DC    A(HERE+4096)
000110 5820 BFFE           01100     8 FIRST   L     2,DATA
000114 5A20 900E           01110     9         A     2,TEN
000118 47F0 9002           01104    10         BC    15,SECOND
001100                              11         ORG   *+4068
001100 0000007B                     12 DATA    DC    F'123'
001104 5830 BFFE           01100    13 SECOND  L     3,DATA
001108 5A30 900E           01110    14         A     3,TEN
00110C 47F0 B00E           00110    15         BC    15,FIRST
001110 0000000A                     16 TEN     DC    F'10'
000100                              17         END   BEGIN
```

Figure 4-3. Listing of an incomplete program with an Origin (ORG) assembler instruction to simulate a length of over 4096 bytes, thus requiring two base registers

As an example of an attempt to use a negative displacement, suppose we were to put the address constant BASE at the very beginning of the program, between the START and the BALR: then the displacement in the Load would need to be -6, which is impossible.

Following the constant BASE, we have two instructions that are meant to suggest the processing steps of the program, and then a branch to an instruction near the end. For the sake of illustration, we want the program to look as though it is more than 4096 bytes long. This we can simulate by an ORG that, in this case, advances the location counter by 4068. This arbitrary-appearing number was chosen to put DATA at the end of a 4096-byte segment controlled by base register 11, which means that the following instructions and data are referenced by base register 9.

Let us now investigate how the assembler assigned base registers and computed displacements.

The next instruction is a Branch on Condition with a mask of 15, which indicates a branch on any condition, or an unconditional branch. This branch to FIRST involves a location under the control of base register 11; if base register 9 were specified, the displacement would have to be negative. The Load at FIRST refers to DATA. The base is 11, with a large displacement of $FFE_{16} = 4094_{10}$. The Add refers to a location that is more than 4096 bytes away from the beginning of the program, so base register 11 cannot be used. We see that 9 has been indicated, with a displacement of $E_{16} = 14_{10}$. The following Branch on Condition references a storage location 2 greater than what was placed in register 9, so register 9 is the base and the displacement is 002.

Down at SECOND, the base registers and displacements for getting DATA and TEN are exactly as they were before; these matters are unaffected by the location of the instructions. The assembled Branch on Condition to FIRST is precisely the same as the assembled Branch on Condition that appeared earlier just before BASE.

The essential concept is that the assembler assigns whatever base register is necessary to get a displacement less than 4096. If the program has been written so that two or more base registers have contents that satisfy this rule, the assembler chooses the one that leads to the smallest displacement. Later we shall see an instance in which this rule for choosing base registers is important.

SEPARATE BASE REGISTERS FOR INSTRUCTIONS AND DATA

We have suggested that it will be rare for a program segment to be so long as to require more than one base register. On the other hand, it may be fairly common to want separate base registers for instructions and data, even though the instructions take far fewer than 4096 bytes. How this can happen is illustrated in the following problem.

Suppose we have six records in storage, each record consisting of 80 characters. The six records are in consecutive storage locations; the first of the 480 bytes has the symbolic address DATA. Within each record there are eight fields of ten characters each, named FIELD1, FIELD2, etc. Each field is in packed decimal format. We are required to add FIELD1 and FIELD2 and place the result in FIELD3. The other five fields are not used in this program. This processing is to be done for each of the six records, using a loop.

Now the question is, how do we attack the loop? The arithmetic will use decimal instructions, which have the SS format and do not provide for use of an index register. We *could* write instructions to modify the displacement of every instruction that refers to the records, but this is very poor form if there is a better way available.

The solution proposed here is to modify the base register contents to pick up the records in succession, which means that between loop repetitions we will add 80 to the base register. But now we have a new problem. If only one base register is used, how do we modify its contents and still get a correct base for Branch instructions and for references to program constants? The simplest answer is probably obvious: use two base registers, the second of which refers *only* to the data processed by the loop.

A program is shown in Figure 4-4. The loading of base registers is much as it was in Figure 4-3, except that this time register 8 is loaded with the address corresponding to DATA, rather than with 4096 more than what register 11 contained. As a matter of fact, it turns out that register 11 contains 102_{16}, and register 8 contains $12C_{16}$. This will mean that the first byte of the area named DATA could be obtained by adding a displacement of 2A to register 11, or by adding a displacement of zero to register 8. As we noted, the assembler picks the way that gives the smaller displacement. It is essential for us to be able to depend on this fact.

We see also that in this program the address constant for loading register 8 has been placed at the end of the instructions rather than in the instruction stream. This is permissible as long as we are sure that it is not more than 4096 bytes away from the beginning of the program, which it obviously is not.

It is assumed, for the purposes of this illustration of base register ideas, that the data is provided by another program segment and will be used later by still another program. We therefore provide space for the data with DS instructions that allot space for the required number of characters but do not assemble constants to be entered. The DS for DATA, in fact, does even less than that: it provides a reference point for the symbol, but does not even reserve space since a zero is written for the duplication factor. Thus

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      1          PRINT NOGEN
000100                                2 LOOPA    START 256
000100 05B0                           3 BEGIN    BALR  11,0
000102                                4          USING *,11
000102 5880 B01E            00120     5 LOOP1    L     8,BASE
00012C                                6          USING DATA,8
000106 D209 8014 8000 00140 0012C     7 LOOP2    MVC   FIELD3,FIELD1
00010C FA99 8014 800A 00140 00136     8          AP    FIELD3,FIELD2
000112 5A80 B022            00124     9          A     8,EIGHTY
000116 5980 B026            00128    10          C     8,TEST
00011A 4770 B004            00106    11          BNE   LOOP2
                                     12          EOJ
000120 0000012C                      15 BASE     DC    A(DATA)
000124 00000050                      16 EIGHTY   DC    F'80'
000128 0000030C                      17 TEST     DC    A(DATA+480)
00012C                               18 DATA     DS    0F
00012C                               19 FIELD1   DS    CL10
000136                               20 FIELD2   DS    CL10
000140                               21 FIELD3   DS    CL10
00014A                               22 FIELD4   DS    CL10
000154                               23 FIELD5   DS    CL10
00015E                               24 FIELD6   DS    CL10
000168                               25 FIELD7   DS    CL10
000172                               26 FIELD8   DS    CL10
00017C                               27          DS    5CL80
000100                               28          END   BEGIN
```

Figure 4-4. Program with separate base registers for processing and data, showing how a base register can be used to provide indexing for loop control

DATA and FIELD1 both refer to the same byte. The point of this approach is to have DATA for a name for the entire 480-character storage area, and still use names for the fields within the first record. An alternative approach would be to use DATA as the name of the first field, DATA + 10 for the second, DATA + 20 for the third, etc., but the loss of meaningful names would be a disadvantage. Another alternative would be to omit the entry for DATA and use FIELD1 wherever DATA appears earlier. This would also be a little less meaningful, perhaps. The final DS reserves 400 bytes of storage for the remaining five records.

The Move Characters instruction at LOOP2 moves the first field to the third field location. Reading across the assembled instruction, which we note is in the SS format, we see: the actual operation code is D2, the length code is 09, the base register for the first operand is 8, the displacement for the first operand is 014, the base register for the second operand is also 8, and the displacement for the second operand is zero. The length code of 9 is correct for a field of length 10; the assembler picked up the implied length from the DS entry for FIELD3, and subtracted 1 from the length to get the length code. Checking the address calculations, we see that a base address of 12C plus a displacement of 014 give an effective address of 140, which is correct for FIELD3. A base address of 12C and a displacement of zero give the address of FIELD1.

The Add Decimal instruction that follows does the required addition. This instruction has two length codes, both 9 in this case, for two fields of length 10. The displacement of 00A, together with the base address of 12C, correctly lead to 136, the address of FIELD2. The addressing of FIELD3 is as before.

Now we are ready to add 80 to the base register associated with DATA and go back to process more records if more remain. We add 80 to base register 8 and then compare with an address constant to test for completion of the loop. What should the test constant be? Since we modify before testing, and since there are 480 characters in the six records, we should stop repeating if at this point the base register contains a number 480 greater than what it was to start. It was originally the equivalent of the symbol DATA, so the test value ought to be DATA + 480, as shown. The Branch on Not Equal here is the extended mnemonic for BC 7. If the branch is not executed, we are finished and the next instruction ends the job.

If the program were written to use only one base register, we would be in trouble with the address of the Branch instruction. The assembler would assume a certain value for the base register and compute a displacement accordingly. After modifying the base register contents, we would no longer have the desired branch address.

It is of course true that we are modifying the contents of base register 8 also, but we have carefully arranged that it is not used as a base for anything besides DATA. No confusion is caused, therefore, because we have "cheated" by changing the contents of a base register from what we promised the assember would be there. What we told the assembler will lead correctly to the first record processed; by the time the contents are actually changed during execution, the assembler will no longer be on the scene to know that anything happened.

In practice it would normally be necessary to process many blocks of six records, not just one. In that case we would have to get register 8 back to its starting value. This is done simply by re-executing the Load instruction at LOOP1.

If a program like this were to be executed, it is perhaps obvious that something would have to be done during loading to take care of the address constants at BASE and TEST. It would clearly not be enough for the linkage editor just to assign the initial program loading location. This matter is properly handled by an automatic flagging of all address constants in the relocation dictionary produced by the assembler, and by suitable modifications performed by the linkage editor.

In order to illustrate one last facet, suppose that there were some compelling reason to place additional instructions *after* DATA. This could be done by a Branch to them. Suppose that within these additional instructions there were Branches to locations within the new group. What would the base register situation be? With the size of program and data shown, either base register 11 or 8 could supply a displacement of acceptable size; the assembler could pick the one leading to the smaller displacement, register 8. But the contents of 8 change as the loop is executed; how can we tell the assembler that 11 is wanted, not 8?

The answer is the DROP instruction, in which we would say DROP 8 at the beginning of the new group of instructions. This says to the assembler that general purpose register 8 may no longer be used as a base register. The only one left is then 11, so it is the one used, as desired.

QUESTIONS AND EXERCISES

Consider the following programs. Note that some of the program statements have been omitted from the listings. The locations assigned to each instruction, constant, and area are listed in hexadecimal, as in all program listings. The locations are such that you should have no difficulty with hexadecimal arithmetic. Questions 1 to 4 refer to Figure 4-5.

1. In the program in Figure 4-5,

a. What instruction informs the assembler that register 11 is to be used as a base register, and tells the assembler what value it must assume to be in that base register?

b. What instruction causes register 11 to be loaded with the base address at execution time?

2. In the spaces provided in the diagram,

a. Write the value the assembler assumes to be in base register 11.

b. Using the symbol table and answer 2a, write the base register and displacement appearing in the object instruction for each encircled operand.

3. Using the specified base register and displacement, write (in the spaces provided) the effective address developed during assembly for each encircled operand.

4. Assume that the program, when loaded for execution, is located starting at 3200_{16} instead of 200_{16}. In the spaces provided, list:

a. The locations into which each instruction, area, and constant (that is, each statement) is loaded.

b. The value placed in register 11 at execution time.

c. The effective address computed at execution time for each encircled operand.

5. Consider the program in Figure 4-6. In the spaces provided, list:

a. The symbol table prepared by the assembler (symbol and location only).

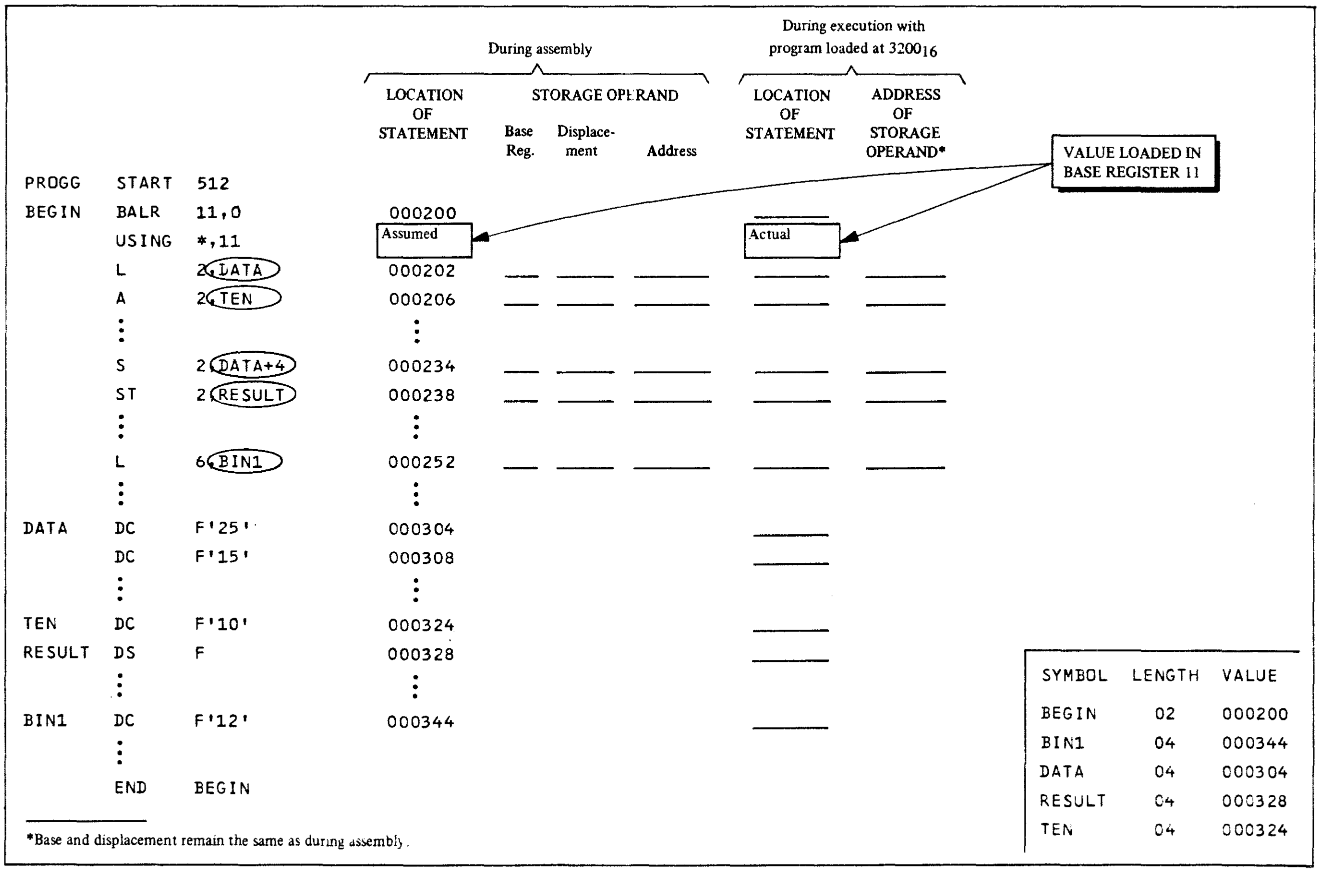
b. The contents of base registers 9, 10, and 11 assumed by the assembler.

c. The base register and displacement for each encircled operand.

d. The values actually placed in registers 9, 10, and 11 at execution time, assuming the program is loaded at 1000_{16}.

e. The location of each statement at execution time.

f. The effective address computed at execution time for each encircled operand.



```
PROGG    START   512
BEGIN    BALR    11,0        000200
         USING   *,11
         L       2,DATA      000202
         A       2,TEN       000206
           :                   :
         S       2,DATA+4    000234
         ST      2,RESULT    000238
           :                   :
         L       6,BIN1      000252
           :                   :
DATA     DC      F'25'       000304
         DC      F'15'       000308
           :                   :
TEN      DC      F'10'       000324
RESULT   DS      F           000328
           :                   :
BIN1     DC      F'12'       000344
           :
         END     BEGIN
```

SYMBOL	LENGTH	VALUE
BEGIN	02	000200
BIN1	04	000344
DATA	04	000304
RESULT	04	000328
TEN	04	000324

---

*Base and displacement remain the same as during assembly.

Figure 4-5. Program for questions 1 to 4

			During assembly				During execution with program loaded at 1000_{16}	
			LOCATION OF STATEMENT	STORAGE OPERAND			LOCATION OF STATEMENT	ADDRESS OF STORAGE OPERAND*
				Base Reg.	Displacement	Address		
PROGH	START	0						
BEGIN	BALR	11,0	000000				____	
	USING	FIRST,11						
FIRST	BC	15,SKIP	000002				____	
DATA	DC	F'3472'	000008				____	
	⋮		⋮					
BASE1	DC	A(FIRST+4096)	000024				____	
BASE2	DC	A(FIRST+8192)	000028				____	
	⋮		⋮					
SKIP	L	10,BASE1	000104	____	____	____	____	____
	USING	FIRST+4096,10						
	L	9,BASE2	000108	____	____	____	____	____
	USING	FIRST+8192,9						
	⋮		⋮					
	BC	15,CK8	001504	____	____	____	____	____
	⋮		⋮					
LOOP	A	4,DATA	001898	____	____	____	____	____
	⋮		⋮					
LOOPB	S	5,DATA	002204				____	
	⋮		⋮					
	BC	8,LOOP	002508	____	____	____	____	____
	⋮		⋮					
CK8	BC	8,LOOPB	002904	____	____	____	____	____
	END	BEGIN						

REGISTER	VALUE LOADED INTO BASE REGISTERS	
	During assembly (assumed)	During execution (actual)
11		
10		
9		

SYMBOL	VALUE
____	____
____	____
____	____
____	____
____	____
____	____
____	____
____	____
____	____

---

*Base and displacement remain the same as during assembly.

Figure 4-6. Program for question 5

Chapter 5: Decimal Arithmetic

The decimal instruction set is an optional feature of System/360, but one that most users elect. Besides making it possible to do arithmetic in the more familiar decimal system, the decimal instruction set includes instructions for editing data, that is, preparing data for printing by the insertion of characters such as commas, periods, and dollar signs. The decimal instruction set permits operations on variable length data since the operations are performed in storage areas rather than in registers. It includes the following instructions:

Add Decimal
Compare Decimal
Divide Decimal
Edit
Edit and Mark
Multiply Decimal
Subtract Decimal
Zero and Add

The student will find a detailed description of the basic operation of these instructions in the *System/360 Principles of Operation.* This chapter will provide examples of their use in various problem situations and will attempt to show how the programmer can make them a working part of his strategy.

Data operated upon by instructions in the decimal set must be in one of two forms, *packed* or *zoned*, depending on the instruction. As a generalization, we can say that the packed format is required for arithmetic and the zoned for input/output. The two formats are shown in Figure 5-1.

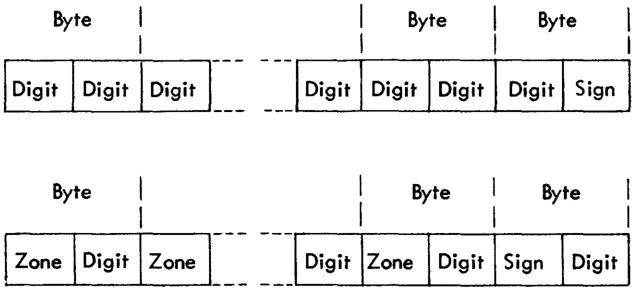


Figure 5-1. Formats of packed and zoned decimal numbers

In the packed format, two decimal digits are placed in each byte except the rightmost of the field, which contains a digit and the sign of the entire number. Digits and sign occupy four bits each. The decimal digits 0–9 have the binary codes 0000–1001. The codes 1010–1111 are not valid as digits. In the sign position, the code combinations 1010, 1100, 1110, and 1111 are taken to mean plus, and 1011 and 1101 are recognized as minus. When a sign is generated as a part of an arithmetic result, a plus is 1100 and a minus is 1101. As mentioned before, all reference to binary codes in this book is to System/360 EBCDIC unless another is specified.

In the zoned format the rightmost four bits of a byte are called the numeric portion of the byte and contain a digit. The leftmost four bits are called the zone and contain either a zone code or, in the case of the rightmost byte, the sign of the number. The codes for signs are treated as described for the packed format. The code for all zones is 1111.

Decimal instructions have precise requirements that operands be in packed or zoned format. The Pack and Unpack instructions, standard instructions of the system, are available for converting from one form to another. The Move with Offset instruction, another of the standard instructions, is often used for shifting factors used or developed in decimal arithmetic operations. Instructions for converting from binary to packed and from packed to binary are also part of the standard instruction set. We shall see examples of all of these operations later.

Decimal instructions use the SS (Storage-to-Storage) format. The machine format is:

Op Code	L_1	L_2	B_1	D_1	B_2	D_2

In assembler format, as written in the source program, the sequence of an SS instruction is:

$$\text{Op code} \quad D_1(L_1,B_1),D_2(L_2,B_2)$$

There are two addresses, both of course referring to core storage. Each address is formed from a base register contents and a displacement. The address *always* refers to the *leftmost* byte of an operand.

For each operand there is a separate length in most cases. *In the machine instruction,* the length code may vary between 0000 and 1111, or zero and 15. These correspond to lengths of one to 16. In other words, the actual length is one greater than what appears in the length code of the object program. In assembler language programming, lengths will quite often be implicit in the data definitions, but when we do write an explicit length, it is the *actual* length. The generation of the proper code in the machine instruction (one less than whatever we write) is the function of the assembler.

With these preliminaries in mind, let us turn to an example.

ADDITION AND SUBTRACTION IN DECIMAL

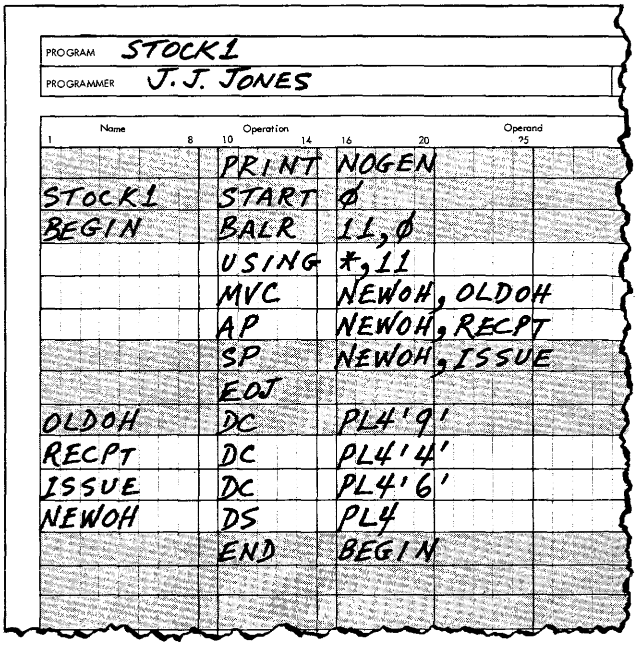
Let us take the first example used in the chapter on fixed-point arithmetic and write it with decimal arithmetic. The application is an inventory updating. We were given an old on-hand (OLDOH), a number received (RECPT), and a number issued (ISSUE); we were to compute the new on-hand (NEWOH). For this program we shall assume that all data entries are already in packed format and are four bytes long. Four bytes can contain, in packed format, seven decimal digits and the sign.

In Figure 5-2 let us look first at the data definitions. The DC instructions for OLDOH, RECPT, and ISSUE and the DS for NEWOH all have operands that start with PL4. The P stands for packed format, and the L4 for a length of 4. Lengths are always in bytes, never digits. This is our first contact with a length modifier in a DC instruction. Here, we are specifying that the constants *must* be four bytes long. If we had omitted the length, the constant generated by the assembler would have been as long as needed to hold the data value we wrote, in this case one byte. (Length modifiers are permitted for other types of data, too.)

Looking at the assembly listing in Figure 5-3, we see that the DC entries have resulted in four-byte constants. In each case, with the data shown, there are six zeros, followed by a digit, followed by a hexadecimal C (binary 1100), which signifies a plus sign in EBCDIC.

Turning back to the instructions of the program, we see the familiar PRINT, START, BALR, and USING instructions. Note that the START instruction specifies zero. This is the usual programming practice at most computer installations, and we will follow it in this book from now on. The START instruction simply tells the assembler where to begin the program during assembly. The linkage editor will assign the actual starting address later, that is, the address in core storage at which the program will be located during execution.

The first processing instruction is a new one, Move Characters (MVC). This is an SS format instruction of a slightly different sort: it moves from storage to storage, but there is only one length, because the "sending" and "receiving" fields must be of the same length. That length may be from one to 256 bytes. Looking at the assembled instruction, we see that a length code of 3 has been supplied by the assembler; this is the correct code for a length of four bytes. The length of the operands was *implied* by the data definitions. It is also possible, and frequently necessary, to write *explicit* lengths to override what the assembler would infer.



PROGRAM STOCK1
PROGRAMMER J.J. JONES

```
         PRINT NOGEN
STOCK1   START 0
BEGIN    BALR  11,0
         USING *,11
         MVC   NEWOH,OLDOH
         AP    NEWOH,RECPT
         SP    NEWOH,ISSUE
         EOJ
OLDOH    DC    PL4'9'
RECPT    DC    PL4'4'
ISSUE    DC    PL4'6'
NEWOH    DS    PL4
         END   BEGIN
```

Figure 5-2. An assembler language program to perform a simple calculation in arithmetic, using the System/360 decimal instruction set

The generation of an address from the base register contents and the displacement is as before: for instance, for OLDOH the base register contains 002, the displacement is 014; the sum of these is 016 which we see is the address for OLDOH.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      1          PRINT NOGEN
000000                                2 STOCK1   START 0
000000 05B0                           3 BEGIN    BALR  11,0
000002                                4          USING *,11
000002 D203 B020 B014 00022 00016     5          MVC   NEWOH,OLDOH
000008 FA33 B020 B018 00022 0001A     6          AP    NEWOH,RECPT
00000E FB33 B020 B01C 00022 0001E     7          SP    NEWOH,ISSUE
                                      8          EOJ
000016 0000009C                      11 OLDOH    DC    PL4'9'
00001A 0000004C                      12 RECPT    DC    PL4'4'
00001E 0000006C                      13 ISSUE    DC    PL4'6'
000022                               14 NEWOH    DS    PL4
000000                               15          END   BEGIN
```

Figure 5-3. Assembly listing of the decimal arithmetic program in Figure 5-2

The purpose of the Move Characters instruction is to get the old on-hand quantity into a location where we can perform arithmetic without disturbing the original quantity. The decimal instructions make no use of the general registers (except, of course, to specify the base), so we must provide storage locations for all data. We do not wish to destroy the old on-hand, so we must arrange for the arithmetic results to go somewhere else. In this case, the obvious place is NEWOH, where we want the eventual result anyway. In other problems, as we shall see, it is often necessary to provide temporary working storage.

The Add Decimal (AP, for Add Packed) instruction adds the quantity received to the old on-hand, which by now is in NEWOH. Note that the result of an arithmetic operation is always stored in the *first* operand location. The two fields in an Add Decimal instruction need not be the same length, since there are two length codes in the instruction. Here, they are the same, as it happens. The Subtract Decimal (SP) instruction deducts the quantity issued.

There is no need for something equivalent to a Store instruction; every instruction already involves two storage addresses, one of which receives the result.

The output in Figure 5-4 shows that the result has been correctly computed.

```
0000009C    0000004C    0000006C    0000007C
```

Figure 5-4. Output of the program of Figure 5-3, showing OLDOH, RECPT, ISSUE, AND NEWOH, in that order

DECIMAL MULTIPLICATION

For a simple example of decimal multiplication, let us write a program for the computation of a new principal amount.

We are given a principal (PRINC), here taken to be four bytes, and an interest factor (INT), two bytes; we are to compute the new principal amount after adding in the year's interest. The interest rate of 3% is expressed as the factor 1.03, so that a single multiplication does the whole job. A program is shown in Figure 5-5.

The Multiply Decimal (MP) instruction takes the second operand to be the multiplier; the first operand initially contains the multiplicand, and at the end of the operation contains the product. However, we cannot begin with a multiply instruction specifying PRINC as the multiplicand, as we might be inclined, because extra space is required. The first operand is required to have at least as many high-order zeros as the size of the multiplier field. We need, therefore, to move the principal to a working storage area having extra positions at the left. These extra positions must be cleared to zero before the multiplication starts.

The Zero and Add (ZAP) does just what we need. The effect of the instruction is to clear the first operand (PROD, in this case) to zero, then add the second operand (PRINC) to it. PROD is two bytes longer than PRINC; these extra four digit positions will be cleared to zeros before PRINC is added in. This provides the zeros needed to satisfy the multiplication rule.

Now we multiply. With the sample data shown, the result in PROD will be 00000256367C, as shown in the comments field. We were regarding 2489 as meaning $24.89, and 103 as meaning 1.03, so there are four places to the right of the understood decimal point in the product, which we therefore regard as 0000025.6367+. We would now like to round this off to $25.64. This can be done in a number of ways. Here we simply add a constant (ROUND) properly set up to add a 5 into the second place from the right. The second operand in an Add Decimal instruction is permitted to be shorter than the first (which holds the result). When this is done, any carries that occur are properly propagated.

We are now ready to discard the two digits at the right end of the product. But this is not quite as simple as just not moving them to PRINC, because if we did that, PRINC would not be a legal operand in any subsequent arithmetic operation, since it would not have a sign. Before moving the result back to PRINC, therefore, we must move the sign from where it is to the byte just to the left. This we can do with a Move Numeric (MVN) instruction, which transmits only the numeric portions of the bytes. The instruction says: Take the numeric portion of the byte at PROD+5 (which is the rightmost byte of the PROD, and contains the sign) and move it to the byte at PROD+4 (which is the byte to the left and will be the rightmost byte of PRINC after the next instruction); the field to be moved is one byte long. The length for this instruction cannot be left to the assembler; the implied length here would be 6 (the length of PROD), which would destroy the result. The Move Numeric instruction has only one length code, so we need give only one explicit length.

Finally, we are ready to move the result to the field where it is required to be at the end of the program, PRINC. Remember that PROD is six bytes long. The leftmost byte contains two zeros, we assume, and the maximum size of the result is taken to be seven digits. The validity of such an assumption as always, is the responsibility of the programmer. The rightmost byte of PROD contains a digit and sign that we now wish to drop. To drop the leftmost byte, we write the address as PROD+1. To drop the rightmost, we need a length of 4, which happens to be the implied length of PRINC, so no explicit length is necessary.

```
  LOC  OBJECT CODE    ADDR1 ADDR2 STMT   SOURCE STATEMENT

                                    1          PRINT NOGEN
000000                              2 INTC     START 0
000000 05B0                         3 BEGIN    BALR  11,0
000002                              4          USING *,11
                                    5 *                                   NUMBERS BELOW SHOW CONTENTS
                                    6 *                                   OF PROD AFTER INSTR IS EXECUTED
                                    7 *                                   C IS PLUS SIGN IN PACKED FORMAT
                                    8 *
000002 F853 B026 B020 00028 00022   9          ZAP   PROD,PRINC           00 00 00 02 48 9C
000008 FC51 B026 B024 00028 00026  10          MP    PROD,INT             00 00 02 56 36 7C
00000E FA51 B026 B02C 00028 0002E  11          AP    PROD,ROUND           00 00 02 56 41 7C
000014 D100 B02A B02B 0002C 0002D  12          MVN   PROD+4(1),PROD+5     00 00 02 56 4C 7C
                                   13 *
00001A D203 B020 B027 00022 00029  14          MVC   PRINC,PROD+1         CONTENTS OF PRINC WILL BE
                                   15 *                                   00 02 56 4C
                                   16          EOJ
000022 0002489C                    19 PRINC    DC    PL4'2489'
000026 103C                        20 INT      DC    PL2'103'
000028                             21 PROD     DS    PL6
00002E 050C                        22 ROUND    DC    PL2'50'
000000                             23          END   BEGIN
```

Figure 5-5. Listing of a program that performs decimal multiplication. Step-by-step results to be expected during execution are shown in the comments field.

DECIMAL DIVISION

Some of the operations in working with the decimal instruction set are different enough from similar operations in other machines that it may be well to pause and consider them in somewhat more detail than we have devoted to other topics. Division is one such operation; Move instructions, used as the equivalent of shifting and considered later in the section on shifting of decimal fields, is another.

The Divide Decimal (DP) instruction is in the SS format. The first operand is the dividend (the number divided into), the second the divisor (the number divided by). After the operation is completed, the first operand field holds the quotient (at the left) and the remainder (at the right). The remainder is the same length as the divisor. Let us see how this description works out in an example.

Suppose we begin with the symbolic locations DIVID and DIVIS as follows:

$\text{DIVID}_{\text{before}}$ 0 0 0 0 0 4 2 4 6 +
DIVIS 0 3 1 +

We have indicated DIVID as a "before" value, because after the division the same field will contain both the quotient and the remainder. All operands are in packed format, as with other decimal arithmetic operations. After executing the instruction:

DP DIVID,DIVIS

the contents of DIVIS would be unchanged; the contents of DIVID would be:

$\text{DIVID}_{\text{after}}$ 0 0 1 3 6 + 0 3 0 +

This means that 4246 divided by 31 in this way gives a quotient of 136 and a remainder of 30. The divisor was two bytes, so the remainder is two bytes. The quotient takes up the remaining space in the first operand field.

The question of the lengths of the various fields can be answered with a useful rule:

Number of bytes in dividend = number of bytes
in divisor + number of bytes in quotient

It is perhaps most common to know the number of bytes in the divisor and the number desired in the quotient, the question being how much space to allow in the dividend in order to get the specified size of the quotient. If two of the three lengths are known, the formula can be used to get the length of the third.

Note that the formula is stated in terms of the number of bytes, not the number of digits. The reason is that the first operand field contains only one sign at the beginning, when it is the dividend, but two afterward, when it contains both quotient and remainder. This change would invalidate a rule stated in terms of digits.

A very similar rule gives the relationship among decimal points. If we agree that by "decimal places" we mean the number of digits to the right of an assumed decimal point, the rule is:

Number of places in dividend = number of places
in divisor + number of places in quotient

In the example given above, we assume that all quantities are integers, that is, they have no decimal places. The rule still holds, although in its most elementary form:

$$0 = 0 + 0$$

Let us see what the result would be if we were to arrange the dividend of the example so that it has one decimal place:

$\text{DIVID}_{\text{before}}$ 0 0 0 0 4 2 4 6 0 +

In other words, we now view the dividend as 4246.0. The result is:

$\text{DIVID}_{\text{after}}$ 0 1 3 6 9 + 0 2 1 +

The rule says that the quotient should have one decimal place: the dividend has one and the divisor has zero. The quotient must therefore be interpreted as meaning 136.9. (And if anything has to be done with the remainder, it should be taken as meaning 2.1.)

Suppose the dividend were shifted one more place to the left:

$\text{DIVID}_{\text{before}}$ 0 0 0 4 2 4 6 0 0 +
$\text{DIVID}_{\text{after}}$ 1 3 6 9 6 + 0 2 4 +

This result should be read as 136.96.

What would happen if we tried to set up the dividend with yet one more shift to the left? There is room in the dividend – but there is no more space in the quotient field. This constitutes a divide exception, which occurs whenever the quotient is too large to fit in the field available to it. An interrupt occurs.

It is possible to check for the possibility of a divide exception, given sample numbers. To do this, the leftmost digit position of the divisor is aligned with the second digit position from the left of the dividend. When the divisor, so aligned, is less than or equal to the dividend, a divide exception will occur. Take the situation suggested:

$\text{DIVID}_{\text{before}}$	0	0	4	2	4	6	0	0	0	+
DIVIS		0	3	1	+					

This is the alignment described by the rule. As aligned, the divisor is smaller. We saw before that there would not be enough room for the quotient.

This question does depend on the particular numbers involved, of course. Suppose the quantities were aligned the same way but that the dividend were 2246 instead of 4246:

DIVID$_{\text{before}}$	0	0	2	2	4	6	0	0	0	+
DIVIS		0	3	1	+					

This is entirely acceptable.

To be completely confident that a divide exception cannot occur, we have to know the maximum possible size of the dividend and the minimum possible size of the divisor, or we must know the maximum size of the quotient.

Further examples of decimal division will be given after we have studied shifting, which is often needed to arrange the dividend so as to give the necessary number of decimal places.

SHIFTING OF DECIMAL FIELDS

Shifting *as such* is not provided in System/360 decimal operations. As in other variable-field-length computers, the equivalent of shifting is performed by appropriate combinations of data movement instructions.

The matter is made somewhat more complex by the factor of packed formats, with two digits per byte and with the special status of the sign position. This is a small price to pay for the increased storage economy of the two-digits-per-byte arrangement.

It is also necessary to exercise caution when overlapping fields are to be manipulated, in order to be sure that no data is destroyed. This is another occasion where it is absolutely essential to remember that *all* operands are addressed by the leftmost byte.

Shifting to the Right

Let us begin with the simplest type of shift: a decimal right shift of an even number of places. Suppose that we have a five-byte, nine-digit number in SOURCE; we are to move it to a five-byte field named DEST with the last two digits dropped and two zeros at the left. We can do this two ways: with or without disturbing the original contents of SOURCE. Let us do it first without disturbing them.

Suppose that the two fields originally contain:

```
         SOURCE                     DEST
   12  34  56  78  9S        55  55  55  55  55
```

The S stands for a plus or minus sign, whichever it might be. The instructions for accomplishing the shift could be as follows, where we have also shown the contents of the two fields after the execution of each instruction:

```
                            SOURCE           DEST
MVC
 DEST+1(4),SOURCE      12 34 56 78 9S   55 12 34 56 78
MVN
 DEST+4(1),SOURCE+4    12 34 56 78 9S   55 12 34 56 7S
MVC
 DEST(1),ZERO          12 34 56 78 9S   00 12 34 56 7S
```

In the first Move Characters instruction, an explicit length of 4 is stated; this length applies to both fields. With the first operand address being DEST+1, the four bytes of the destination are the rightmost four. The second operand is given simply as SOURCE, so the four bytes there are the leftmost. The last two digits (one byte) have been dropped.

But the sign has been dropped too, in the process. We accordingly use a Move Numeric instruction to attach it to the shifted number. This must be done with an explicit length of one, to avoid disturbing any of the digits of DEST. Both addresses must be written with the "+4" to pick out the proper single character. Finally, we move one byte of a constant named ZERO (not shown), which contains zeros, to the first byte of DEST. This clears to zero whatever may have been there before.

If the contents of SOURCE are no longer needed in their original form, the following sequence is a bit shorter.

```
                               SOURCE           DEST
MVN
 SOURCE+3(1),SOURCE+4    12 34 56 7S 9S   55 55 55 55 55
ZAP
 DEST,SOURCE(4)          12 34 56 7S 9S   00 12 34 56 7S
```

The Move Numeric moves the sign to the byte which will contain the sign in the eventual result. The Zero and Add picks up four bytes of SOURCE and adds them to DEST after clearing DEST to zeros. The Zero and Add has two length codes. For DEST we use the implied length of 5; for SOURCE it is necessary to give an explicit length in order to drop the last two digits.

Finally, suppose that for some reason it is necessary to leave the shifted result in SOURCE, without resorting to the expedient of simply moving the sign and appending zeros at the left.

```
                                  SOURCE
MVN  SOURCE+3(1),SOURCE+4     12  34  56  7S  9S
ZAP  SOURCE,SOURCE(4)         00  12  34  56  7S
```

The sign movement is as before. In the Zero and Add, the second operand is given as SOURCE(4), which means a four-byte field the leftmost byte of which has the address SOURCE; this is just 12 34 56 7S. The first operand is simply SOURCE, with its implied length of 5, which means the whole field.

It is important to know that this type of overlap is permitted when the first operand field is at least as long as the second operand, but not when it is too short to contain all significant digits of the second operand. A little study shows that a violation of this rule would result in destroying bytes of the second operand before they have been moved.

Let us now turn to a slightly more complex shift, one that involves an odd number of places. This requires the use of a special instruction designed for the purpose, the Move with Offset. The action of this instruction can be described as follows. The sign of the first operand is not disturbed; the second operand is placed to the left and adjacent to the four low-order bits (the sign bits) of the first operand. Any unused high-order digit positions in the first operand are filled with zeros.

Looking at an example, take the fields described in the previous illustration, but suppose that the shift must be three positions instead of two.

```
                            SOURCE           DEST
MVO
 DEST,SOURCE(3)        12 34 56 78 9S   00 01 23 45 65
MVN
 DEST+4(1),SOURCE+4    12 34 56 78 9S   00 01 23 45 6S
```

In the Move with Offset, the second operand is given as SOURCE(3), which picks up a three-byte field starting at the left, namely, the bytes containing 12 34 56. The first operand is DEST, with its implied length of 5. The digits 12 34 56 are moved to DEST with an offset of four bits, or one digit, leaving 00 01 23 45 65 in DEST; the rightmost 5 is the one that was there to begin with. A final Move Numeric attaches the source sign to the destination field.

If the shift is required to leave the result in SOURCE, only one instruction is needed, since the Move with Offset instruction has no effect on the sign of the first operand, and the left end of the receiving field is filled with zeros.

```
                                SOURCE
MVO   SOURCE,SOURCE(3)      00 01 23 45 6S
```

The overlapping fields here cause no trouble, since again the movement is to the right of the original contents. (Actually, overlap of any type is *permitted*; it is the programmer's responsibility to make sure that the result is meaningful.)

Shifting to the Left

A shift to the left presents slightly different problems. This time suppose that we have a source field of three bytes and a destination of five.

```
Before       SOURCE          DEST
            12 34 5S    99 99 99 99 99
```

Let us take our problem, to move the number at SOURCE to DEST, with four zeros to the right at DEST, and with DEST left ready to do arithmetic. An acceptable sequence of instructions is shown below.

```
                           SOURCE          DEST
MVC  DEST(3),SOURCE       12 34 5S    12 34 5S 99 99
MVC  DEST+3(2),ZEROS      12 34 5S    12 34 5S 00 00
MVN  DEST+4(1),DEST+2     12 34 5S    12 34 5S 00 0S
MVN  DEST+2(1),ZEROS      12 34 5S    12 34 50 00 0S
```

The first Move Characters needs an explicit length on DEST; otherwise, the length would, improperly for our problem, be interpreted from DEST as 5. The last two bytes of DEST are unaffected by the first Move; a second clears them. A Move Numeric transfers the sign, and a second Move Numeric clears the now extraneous sign that went with the source data on the first Move Characters.

Another way to clear the extraneous sign is available, using the And Immediate instruction. "Anding" two quantities gives a result that has a one bit wherever *both* operands had 1's, and a zero elsewhere. For instance, if we "And" 1100 and 1010, the result is 1000; only in the first bit position did both operands have ones. In the And Immediate instruction (NI), both operands are exactly eight bits long. One of them is given by the byte specified by the address; the other is contained in the instruction itself (which is the reason for the term "immediate"). The result replaces the byte specified in storage.

In the example at hand, we wish to leave the first four bits of the byte at DEST+2 just as they were; this can be done by placing ones in the corresponding positions in the part of the instruction that will be "And-ed". (This is usually called the mask.) We wish to make the right four bits of DEST+2 zero, whatever they were before; this can be done by placing zeros in that part of the mask. The mask, in short, should be 11110000, expressed in binary. To write the instruction, we can either convert this to its decimal equivalent 240, or, better, write it in hexadecimal, X'F0'. In other words, we can replace the last instruction with either of the following:

```
NI    DEST+2,240
NI    DEST+2,X'F0'
```

Finally, consider a shift to the left of an odd number of places. For an example, take the data of the preceding illustration, but suppose there are to be three zeros at the right instead of four.

```
                          SOURCE          DEST
     Before              12 34 5S    99 99 99 99 99
MVC  DEST(3),SOURCE      12 34 5S    12 34 5S 99 99
MVC  DEST+3(2),ZEROS     12 34 5S    12 34 5S 00 00
MVN  DEST+4(1),DEST+2    12 34 5S    12 34 5S 00 0S
NI   DEST+2,X'F0'        12 34 5S    12 34 50 00 0S
MVO  DEST(4),DEST(3)     12 34 5S    01 23 45 00 0S
```

The first four instructions are just the same as in the previous example, except that the And Immediate is substituted for the Move Numeric. The final instruction now is a Move with Offset that shifts one digit position to the right.

DECIMAL DIVISION WITH SHIFTING

We are now prepared to approach a realistic problem in decimal division.

Suppose that in a four-byte field named SUM we have the total of the number of hours worked by all the employees in a factory, given to tenths of an hour. In NUMBER we have the number of employees included in the sum; this is a two-byte number. We are to calculate the average workweek, to tenths of an hour, rounded, and place it in a two-byte location named AVERAG.

We begin the analysis of the problem knowing that the dividend (SUM) has one decimal place to start, and the divisor (NUMBER) has none. If we set up the division this way, we would get a quotient having one place; this would not permit rounding. Evidently we shall have to allow extra places to the right. One more would be sufficient, but this would involve a shift of an odd number of places; it would be simpler for us and faster in the machine to make a shift of two places and simply ignore the extra digit. The dividend therefore should be set up like this:

XX XX XX X0 0+

The X's stand for any digits.

Now we turn to the rule stating that the number of bytes in the dividend is equal to the number of bytes in the divisor plus the number of bytes in the quotient. We know that we have two bytes in the divisor as it stands. The quotient need be only three: there can be no more than two digits before the decimal point, there will be three after the decimal point, and there will be a sign. (There will be three decimal places in the quotient because there are three in the dividend and none in the divisor.) The dividend evidently should be five bytes. As it happens – which will by no means always be the case – that is just how long it will be as the result of the shifting we decided upon.

With this much background, let us now look at the program shown in Figure 5-6. We assume that it is permissible to destroy the original contents of SUM; if this were not so, it would be a matter of one extra instruction to move the contents of SUM to a working storage location.

Notice in the list of constants at the end of the program that a one-byte constant named PAD has been established just after, and therefore to the right of, SUM. Now, instead of actually moving the contents of SUM in order to accomplish a shift, we simply extend the field by one byte. This is the function of the first two instructions. We have assumed, reasonably enough, that the sum is always positive, so a plus sign is moved with the first Move Characters, and the original sign is simply erased with the And Immediate.

The Divide Decimal might seem to carry the possibility of a divide exception. We must fall back on a knowledge of the data, which is the eventual foundation of any intelligent programming. We simply observe that the average hours worked would not be as great as 100 hours – and anything less can be contained in the space provided.

Rounding is accomplished by adding 5 in the proper position. We move the sign to where it is needed, and finally transfer the result to the specified location in storage.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1            PRINT NOGEN
000000                               2 AVG        START 0
000000 05B0                          3 BEGIN      BALR  11,0
000002                               4            USING *,11
                                     5 *                                  NUMBERS BELOW SHOW CONTENTS
                                     6 *                                  OF SUM AFTER INSTR IS EXECUTED
                                     7 *
000002 D200 B028 B02F 0002A 00031    8            MVC   SUM+4(1),ZERO     01 93 64 8C 0C
000008 94F0 B027      00029          9            NI    SUM+3,X'F0'       01 93 64 80 0C
00000C FD41 B024 B029 00026 0002B   10            DP    SUM(5),NUMBER     39 76 3C 21 9C
000012 FA21 B024 B02D 00026 0002F   11            AP    SUM(3),ROUND      39 81 3C 21 9C
000018 D100 B025 B026 00027 00028   12            MVN   SUM+1(1),SUM+2    39 8C 3C 21 9C
00001E D201 B02B B024 0002D 00026   13            MVC   AVERAG,SUM        AVERAG  WILL BE 39 8C
                                    14            EOJ
000026 0193648C                     17 SUM        DC    PL4'0193648'
00002A                              18 PAD        DS    PL1
00002B 487C                         19 NUMBER     DC    PL2'487'
00002D                              20 AVERAG     DS    PL2
00002F 050C                         21 ROUND      DC    PL2'50'
000031 0C                           22 ZERO       DC    PL1'0'
000000                              23            END   BEGIN
```

Figure 5-6. Assembled program showing decimal division and "shifting". Step-by-step results to be expected during execution are included in the comments field.

FORMAT AND BASE CONVERSIONS

It is often necessary to convert from zoned to packed format and vice versa, and also to convert from binary to decimal and vice versa. In this section, we shall examine a program that has been constructed as an exercise in manipulating the form of data. For practice purposes, some new instructions are introduced for these maneuvers, which might be accomplished more simply in a realistic situation.

We are given a fullword named REG, in binary format. Actual data for the three-byte field named PREM is read in directly from an input card on which the sign is in the high-order position, instead of the low-order. That is, a positive number was punched with a 12 zone over the leftmost digit, and a minus number was punched with an 11 zone over the leftmost digit. We are required to place the sum of REG and PREM in ANS, as a decimal number in the normal zoned format, that is, with the sign in the zone of the low-order byte. The zone bits that result in a byte in storage from a 12 zone on the card, are the zone bits required for a plus sign in the EBCDIC zoned format in storage. An 11 zone likewise is translated into the correct zone bits for a minus sign. Our problem, then, is simply to move the zone bits of the high-order byte to the zone bits of the low-order byte.

In the program of Figure 5-7 we have shown at the right of the first half-dozen instructions the contents of the last eight bit positions of registers 5 and 6, to aid in understanding how the instructions operate on sample data consisting of the three bytes:

1101 0011   1111 0111   1111 1001

With the card column assignments we have described, this is the EBCDIC representation of -379.

The program begins with a new instruction: Insert Character (IC). This is an RX format instruction that gets one character (byte) from the specified storage location and places it in the rightmost byte position of the register named. The other bit positions of the register are not disturbed. We do not know what might be in them, but it will not matter, as it happens, since the following instruction clears them. This is an And to erase the numeric bits of the high-order character of our sample data.

Next we perform the similar operations on the low-order byte, using register 6, except that this time we erase the zone bits.

Now we have in register 6 the numeric bits of the low-order byte, and in register 5 the zone bits that are to be attached to that byte. They can be combined with an Or Register (OR) instruction. "Or-ing" two operands is a bit-by-bit operation that results in a 1 wherever *either* operand had a 1, and zero where both had zero. The result of this instruction is to combine the two groups of bits, leaving the result in register 5. This now is the byte that we want in the low-order position, so we use a Store Character instruction (STC) to place it there.

Insert Character and Store Character do not require the character to be on any sort of integral boundary. They are the only indexable instructions for which this is true. The various decimal instructions do not require boundary alignment either, of course, but they are not indexable. The two

```
 LOC   OBJECT CODE    ADDR1 ADDR2 STMT   SOURCE STATEMENT

                                     1          PRINT NOGEN
000000                               2 CONVERT  START 0
000000 05B0                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
                                     5 *                                LAST BYTE (BITS 24 TO 31) OF REGS 5
                                     6 *                                AND 6 AFTER EXECUTION OF EACH INSTR
                                     7 *                                IS SHOWN BELOW
                                     8 *
                                     9 *                                  REG 5              REG 6
000002 4350 B03A            0003C   10          IC    5,PREM            1101 0011
000006 5450 B032            00034   11          N     5,MASK1           1101 0000
00000A 4360 B03C            0003E   12          IC    6,PREM+2          1101 0000         1111 1001
00000E 5460 B036            00038   13          N     6,MASK2           1101 0000         0000 1001
000012 1656                         14          OR    5,6               1101 1001         0000 1001
000014 4250 B03C            0003E   15          STC   5,PREM+2          1101 1001         0000 1001
000018 F212 B03D B03A 0003F 0003C   16          PACK  WORK,PREM
00001E 5860 B042            00044   17          L     6,REG
000022 4E60 B046            00048   18          CVD   6,DOUBLE
000026 FA71 B046 B03D 00048 0003F   19          AP    DOUBLE,WORK
00002C F357 B04E B046 00050 00048   20          UNPK  ANS,DOUBLE
                                    21          EOJ
000034                              24          DS    0F
000034 000000F0                     25 MASK1    DC    X'000000F0'
000038 0000000F                     26 MASK2    DC    X'0000000F'
00003C                              27 PREM     DS    ZL3
00003F                              28 WORK     DS    PL2
000044                              29 REG      DS    F
000048                              30 DOUBLE   DS    D
000050                              31 ANS      DS    ZL6
000000                              32          END   BEGIN
```

Figure 5-7. Assembled program showing various instructions for changing the format of data. Contents of registers 5 and 6 to be expected during execution are given in the comments field.

And (N) instructions, however, *do* require their operands to be on fullword boundaries. This is the purpose of the DS 0F before the DC's for the masks.

At this point we have merely got the sign where it is expected to be in the zoned format of a decimal number. Now we must convert from zoned to packed format, which is the function of the PACK instruction. The second operand names a field in zoned format; the first names the field where the packed format should be stored. Both fields carry length codes. Here, we are able to leave the lengths implied: three bytes for PREM and two for WORK (two bytes allow space enough for three digits and sign in packed format). The PACK instruction ignores all zones except the rightmost, which is taken to carry the sign. Therefore we can leave the zone of the high-order byte as it was without disturbing the operation.

With the PREM amount finally in packed format, we are almost ready to do the addition – but not quite, because the REG amount is still in binary. The next instruction, accordingly, is a Load followed by a Convert to Decimal (CVD). Convert to Decimal takes the binary number in the specified register and converts it to packed format decimal in the location given, which must be aligned on a doubleword boundary.

At last it is possible to do the addition, which is done in decimal. A final instruction, Unpack (UNPK), converts back from packed to zoned, as required in the problem statement. This will leave the final answer with the sign in the zone bits of the low-order byte, which was stated to be the desired position for whatever processing might follow. If it were necessary to get the result into the same format as PREM originally was, we could of course do so.

DECIMAL COMPARISON: OVERTIME PAY

Logical tests and decisions are as necessary in decimal operations as elsewhere. System/360 provides a Compare Decimal instruction, and the condition code is set as a result of this and three decimal arithmetic instructions.

For an example we take the familiar calculation of gross pay, with time-and-a-half for hours over 40. We have a RATE, given in dollars and cents, and an HOURS, to tenths of an hour. We are to place the total wages in GROSS.

There are several ways to approach the overtime computation. We choose here to begin by figuring the pay at the straight-time rate, on the full amount in HOURS. We then inspect the hours worked, and if it was not over 40 the job is finished. If there was overtime, we multiply the hours over 40 by the pay rate, and multiply this product by one-half to get the premium, which is then added to the previous figure. Several other ways to arrange the sequence of decisions and multiplications are obviously possible. This one probably minimizes the computation time if most employees do not work overtime; if most did work overtime, a different sequence might be a little better.

The program in Figure 5-8 begins with a three-instruction sequence to set up the multiplicand in a work area, multiply, and round. The Move with Offset instruction drops one digit in the move; this is the extra digit that was rounded off. The Move with Offset instruction does not transmit the sign; we have shown GROSS as a DC to get a plus sign there from the outset. Since the pay can never properly be negative, the plus sign will simply remain there throughout the operation of the program.

The Compare Decimal (CP) instruction is not greatly different in concept from Compare instructions we have seen previously. The two operands are compared algebraically; the condition code is set depending on the relative sizes of the two; neither operand is changed. The mask of 12 on the Branch on Condition will cause a branch if the contents of HOURS are less than or equal to FORTY, in which case there is no overtime to compute, and we simply branch out to whatever follows.

If the man did work more than 40 hours, we compute his pay on the amount over 40, then multiply by 5, which we view as having a decimal point, that is, as being one-half. This is done because we have already computed the straight-time pay on the amount over 40; now we need only to compute the extra premium. After the multiplication by 5 we round off, using a different rounding constant this time because the multiplication by 0.5 has added another decimal place. (It is necessary to check that WORK is long enough to satisfy the rule about at least as many zeros as the size of the multiplier. Assuming that no employee could make $1000 in one week, the rule is satisfied.)

After a Move Numerics to move the sign, we can add the rounded amount to GROSS to get the total pay. In the Add Decimal, note the length of 3 to drop the last byte, which after rounding is extraneous. We now reach the termination of the program, the same point to which we transferred if there was no overtime. In other words, both paths would lead, in a real program, to the same continuation point.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1          PRINT NOGEN
000000                               2 OTPAY    START 0
000000 05B0                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
                                     5 *                                  NUMBERS BELOW SHOW CONTENTS OF
                                     6 *                                  FIRST OPERAND (WORK OR GROSS)
                                     7 *                                  AFTER INSTRUCTION IS EXECUTED
                                     8 *
000002 F831 B056 B050 00058 00052    9          ZAP   WORK,HOURS          00 00 44 6C
000008 FC31 B056 B04E 00058 00050   10          MP    WORK,RATE           00 78 05 0C
00000E FA30 B056 B05A 00058 0005C   11          AP    WORK,FIVE           00 78 05 5C
000014 F132 B052 B056 00054 00058   12          MVO   GROSS,WORK(3)       00 07 80 5C
00001A F911 B050 B05D 00052 0005F   13          CP    HOURS,FORTY
000020 47C0 B04C            0004E   14          BC    12,OUT
000024 F831 B056 B050 00058 00052   15          ZAP   WORK,HOURS          00 00 44 6C
00002A FB31 B056 B05D 00058 0005F   16          SP    WORK,FORTY          00 00 04 6C
000030 FC31 B056 B04E 00058 00050   17          MP    WORK,RATE           00 08 05 0C
000036 FC30 B056 B05A 00058 0005C   18          MP    WORK,FIVE           00 40 25 0C
00003C FA31 B056 B05B 00058 0005D   19          AP    WORK,FIFTY          00 40 30 0C
000042 D100 B058 B059 0005A 0005B   20          MVN   WORK+2(1),WORK+3    00 40 3C 0C
000048 FA32 B052 B056 00054 00058   21          AP    GROSS,WORK(3)       00 08 20 8C
                                    22 OUT      EOJ
000050 175C                         25 RATE     DC    PL2'1.75'
000052 446C                         26 HOURS    DC    PL2'44.6'
000054 0000000C                     27 GROSS    DC    PL4'0'
000058                              28 WORK     DS    PL4
00005C 5C                           29 FIVE     DC    PL1'5'
00005D 050C                         30 FIFTY    DC    PL2'50'
00005F 400C                         31 FORTY    DC    PL2'40.0'
000000                              32          END   BEGIN
```

Figure 5-8. Assembled program that computes a man's gross pay, including any overtime pay, in decimal arithmetic. Results expected during execution are shown in the comments field.

THE SOCIAL SECURITY PROBLEM IN DECIMAL

For a little further practice in applying decimal operations, we may rewrite the Social Security calculation of Figure 3-19 in the chapter on fixed-point operations. The logic of the decimal program shown in Figure 5-9 is the same as that of the earlier one. No new instructions are introduced, so a few notes should be all that is required to explain the program.

We begin by moving the old year-to-date to the new year-to-date location. The purpose is simply to get one of the two operands in the following addition where we want the result to be. Following is a Zero and Add to get the new year-to-date into working location where we can continue the processing without disturbing the NEWYTD location. From here on, the right side of Figure 5-9 shows the contents of the WORK field for sample data as shown in the DC instructions.

In the Multiply Decimal instruction that computes the Social Security tax on the new year-to-date figure, we use a constant for the 4.4% that has been set up with an extra zero at the right. This was done to put the product in a position where a Move with Offset would not be necessary. As it has been done, after rounding and moving the sign, we can carry out all following operations on the Social Security amount on the second, third and fourth bytes of WORK. Since the implied length from the DS is 6, an explicit length must be given. The explicit length specifications in the two Move Characters (statements 17 and 19) are unnecessary, however, because NEWFICA and TAX are defined as 3 bytes, and the assembler already has that information.

Except for the points discussed here, the operations closely parallel the program in the earlier version.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                    1          PRINT NOGEN
000000                              2 FICA3    START 0
000000 05B0                         3 BEGIN    BALR  11,0
000002                              4          USING *,11
000002 D203 B04F B04B 00051 0004D   5          MVC   NEWYTD,OLDYTD
000008 FA32 B04F B048 00051 0004A   6          AP    NEWYTD,EARN
                                    7 *                                 CONTENTS OF WORK AFTER EXECUTION
                                    8 *                                 OF EACH INSTR ARE SHOWN BELOW
                                    9 *
00000E F853 B064 B04F 00066 00051  10          ZAP   WORK,NEWYTD        00 00 07 86 40 0C
000014 FC51 B064 B05F 00066 00061  11          MP    WORK,C44           00 34 60 16 00 0C
00001A FA52 B064 B061 00066 00063  12          AP    WORK,HALF          00 34 60 21 00 0C
000020 D100 B067 B069 00069 0006B  13          MVN   WORK+3(1),WORK+5   00 34 60 2C 00 0C
000026 F932 B064 B05C 00066 0005E  14          CP    WORK(4),MAX        00 34 60 2C 00 0C
00002C 4740 B034            00036  15          BC    4,UNDER            00 34 60 2C 00 0C
000030 D202 B065 B05C 00067 0005E  16          MVC   WORK+1(3),MAX      00 34 32 0C 00 0C
000036 D202 B056 B065 00058 00067  17 UNDER    MVC   NEWFICA(3),WORK+1  00 34 32 0C 00 0C
00003C FB22 B065 B053 00067 00055  18          SP    WORK+1(3),OLDFICA  00 00 52 0C 00 0C
000042 D202 B059 B065 0005B 00067  19          MVC   TAX(3),WORK+1      00 00 52 0C 00 0C
                                   20          EOJ
00004A 16400C                      23 EARN     DC    PL3'16400'
00004D 0770000C                    24 OLDYTD   DC    PL4'770000'
000051                             25 NEWYTD   DS    PL4
000055 33800C                      26 OLDFICA  DC    PL3'33800'
000058                             27 NEWFICA  DS    PL3
00005B                             28 TAX      DS    PL3
00005E 34320C                      29 MAX      DC    PL3'34320'
000061 440C                        30 C44      DC    PL2'440'
000063 05000C                      31 HALF     DC    PL3'5000'
000066                             32 WORK     DS    PL6
000000                             33          END   BEGIN
```

Figure 5-9. Assembled program to calculate Social Security tax in decimal arithmetic. Results expected during execution are shown in the comments field.

THE "INDIAN" PROBLEM

A certain programming exercise has been done by so many generations of IBM students that it is a classic. We present it here, worked out with the calculation in decimal and the counting in binary.

> The Indians sold Manhattan Island in 1627 for $24. If the Indians had banked their $24 in 1627, what would their bank balance be in 1965 at a 3% interest rate compounded annually?

To make the problem a little more interesting, let us assume that the principal, $24, the interest rate factor, 1.03, and the number of years, 338, are all initially in zoned format. The program of Figure 5-10 accordingly begins with three PACK instructions to get from zoned to packed format.

The general scheme of the program will be to multiply the principal by 1.03 as many times as there are years. In other words, we shall go around a loop repeatedly, each time performing a multiplication and subtracting 1 from a count. When the count has been reduced to zero, the computation of the balance is completed. This counting down from 338 to zero could, of course, be done in decimal, testing for zero with a Compare Decimal instruction. It is better programming practice, however, to remove time-consuming operations from the repeated part of the loop wherever possible. Doing the repeated combination of an Add Decimal, a Compare Decimal, and a Branch on Condition is much more time-consuming than another approach that is available to us. This other way is to convert the years to binary once, before entering the loop, then use a Branch on Count (BCT) in the loop, a single instruction that will subtract 1, test, and conditionally branch.

The fourth instruction of the program is therefore a Convert to Binary (CVB) instruction, which in our program takes the doubleword at YEARSP and converts to a binary number in register 4. The Convert to Binary instruction requires an *aligned* doubleword operand, which is why the DS for YEARSP was set up as it was instead of with a CL8.

The repeated part of the loop starts with a Multiply Decimal that should by now be moderately familiar. PRINCP was set up to be long enough to hold the size of number that previous runnings of the program have shown will be necessary. The programmer facing this problem completely fresh would have to make some preliminary calculations as to the possible size.

Now comes a familiar sequence of decimal instructions to round, move the sign, and shift right two digits (one byte). One might be tempted to replace the Move Characters and Zero and Add instructions with a single one of the sort:

MVC PRINCP+1(6),PRINCP

thinking that a right-to-left operation would permit this sort of overlap. A check of the *Principles of Operation* manual, however, discloses that Move Characters works from left to right! The instruction suggested would therefore propagate the leftmost character through the entire field! This can be quite useful on occasion, and is permitted, but it is hardly what we want here. Overlapping fields must be treated with caution.

The Branch on Count subtracts 1 from register 4; if the result is not zero, a branch occurs. If the result is zero, the next instruction in sequence is taken. The loop will be carried out 338 times, as required.

A final Unpack instruction puts the result into a location named BALANCE in zoned format. The answer obtained by execution of our program is $523,998.22. Carrying the calculations out to more decimal places would of course give a more precise result.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1          PRINT NOGEN
000000                               2 INDIAN   START 0
000000 05B0                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
000002 F263 B04A B040 0004C 00042    5          PACK  PRINCP,PRINCZ
000008 F212 B051 B044 00053 00046    6          PACK  INTP,INTZ
00000E F272 B056 B047 00058 00049    7          PACK  YEARSP,YEARZ
000014 4F40 B056            00058    8          CVB   4,YEARSP
000018 FC61 B04A B051 0004C 00053    9 LOOP     MP    PRINCP,INTP
00001E FA61 B04A B05E 0004C 00060   10          AP    PRINCP,ROUND
000024 D100 B04F B050 00051 00052   11          MVN   PRINCP+5(1),PRINCP+6      MOVE SIGN
00002A D205 B060 B04A 00062 0004C   12          MVC   TEMP,PRINCP               DROP LOW-ORDER BYTE
000030 F865 B04A B060 0004C 00062   13          ZAP   PRINCP,TEMP
000036 4640 B016            00018   14          BCT   4,LOOP                    SUBTRACT 1 FROM REG 4
00003A F386 B066 B04A 00068 0004C   15          UNPK  BALANCE,PRINCP
                                    16          EOJ
000042 F2F4F0C0                     19 PRINCZ   DC    ZL4'24.00'
000046 F1F0C3                       20 INTZ     DC    ZL3'1.03'
000049 F3F3C8                       21 YEARZ    DC    ZL3'338'
00004C                              22 PRINCP   DS    PL7
000053                              23 INTP     DS    PL2
000058                              24 YEARSP   DS    D
000060 050C                         25 ROUND    DC    PL2'50'
000062                              26 TEMP     DS    PL6
000068                              27 BALANCE  DS    ZL9
000000                              28          END   BEGIN
```

Figure 5-10. Assembled program to compute compound interest (the "Indian" problem), with counting in binary and calculations in decimal arithmetic

QUESTIONS AND EXERCISES

1a. Write the assembler instruction to define a packed decimal constant of 3 to be named CON3 and to occupy 5 bytes of storage.

b. Show how this constant appears on the assembly listing.

2. A length code in an instruction is called *implied* if it is supplied by the ________________ on the basis of ________________. An explicit length code is supplied by the ________________.

3. An explicit length code is (equal to, one more than, one less than) the actual number of bytes to be dealt with.

4. The length code in the object instruction is (equal to, one more than, one less than) the actual number of bytes to be dealt with.

5a. In an MP instruction, the first operand specifies the location of a storage area containing ________________.

b. Where is the product at the end of the multiplication?

6. If there were two successive DC statements of:

```
PRINC  DC     PL4'2489'
INT    DC     PL2'103'
```

and PRINC were assigned a location of 158:

a. Byte by byte, what would be in the storage locations assigned to these constants?

b. To what storage location would the operand INT−2 refer?

7. A DP instruction specifies in its first operand the location of the ________________, and in its second operand the location of the ________________. Where will the quotient and remainder be after the completion of a DP instruction?

8. Assume two fields:

SOURCE containing 66 55 44 33 22 11
DEST containing 11 22 33 44 55 6S (S = sign)

Show the contents of SOURCE and DEST after the execution of the instructions below. In each case, assume that before execution the contents of SOURCE and DEST are as shown above.

a. MVC DEST+2(3),SOURCE
b. MVN DEST+3(1),DEST+5
c. MVO DEST,SOURCE+2(2)

9. Assume the same fields (SOURCE and DEST) as given in question 8.

Would the instruction ZAP DEST,SOURCE be a legitimate one? If not, why not?

10. Assume a 5-byte field called FACTOR, which contains 12 34 56 78 9S (S = sign)

a. Write the instruction or instructions to store the leftmost 8 digits (12345678) and the sign in a 6-byte field called RESULT.

b. Write the instruction or instructions to store the leftmost 7 digits and the sign in RESULT.

11a. The NI (And Immediate) instruction is a ________________ format instruction.

b. Write the NI instruction(s) that will change the contents of a field named HOLD from 11 22 33 44 6S to 00 22 33 44 6S.

c. 11 22 33 44 6S to 11 22 33 04 6S.

12. What is the difference between the And Immediate and Or Immediate instructions?

13. Decimal arithmetic can be performed only on (zoned decimal, packed decimal) fields.

14. What instruction converts information from zoned decimal to packed decimal form?

15. What instruction converts information from packed decimal to zoned decimal form?

16. Write DC's to store the number 578 as:

a. A fixed-point number.
b. A 3-byte zoned decimal number.
c. A 2-byte packed decimal number.

17. Write a DC to store the hexadecimal equivalent of 75_{10}.

18. Write an instruction that will place a byte named OLD in the rightmost byte position of register 6 without disturbing the remaining positions of register 6.

19. Write an instruction that will store the contents of the rightmost byte position of register 6 in a storage byte named OLD.

20. Consider the following excerpts from an assembly listing. Mask is located at 13E.

```
         N      6,MASK
         .   .
         .   .
         .   .
MASK     DC     X'0000000F'
```

a. Will the N 6,MASK instruction be successfully executed? If not, why not?

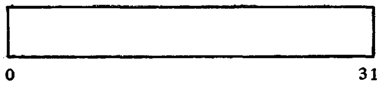
b. If not, what statement or statements could be inserted to correct the condition?

c. How could the DC itself be rewritten to correct the situation?

Chapter 6: Logical Operations on Characters and Bits

So far we have been dealing mainly with the arithmetic operations of System/360. Now we turn to an area of particular fascination to the programmer, one that opens up a nearly unlimited range of flexibility and inventiveness in the performance of his task. The logical operations of System/360 provide means for the testing and manipulation of data in a *logical* sense, rather than arithmetic or algebraic. Among these special assembler language instructions are: the Logical Compares, Test under Mask, some new Move instructions, the Logical Shifts, Insert Character, Store Character, and the highly versatile Ands, Ors, and Exclusive Ors. One or more forms of each of these instructions, which are part of the System/360 standard instruction set, will be demonstrated in examples in this chapter. Other logical instructions–the standard Translate and the decimal feature Edit instructions–have such highly specialized functions that they will be the subject of a separate chapter.

The most important thing for us to realize about the logical instructions is that (except for the Edit instructions) they treat all data as unsigned binary quantities, that is, all bits are treated alike and no distinction is made between sign and numeric bits. Remember the data format for a System/360 fixed-point number, with its sign in the first bit? And for a zoned or packed decimal number, with its sign in the first four or last four bits of the final byte? Well, the logical instructions are non-algebraic, and they treat all data as unstructured logical quantities, not as numbers. Fixed-length data such as a word in a register is regarded this way:



Variable-length data in storage is looked at this way:

In practice, the operands are generally characters or groups of bits.

Since the logical operations do not recognize any signs as such, it is incumbent upon the programmer to know when and where they are in his data. He can use signed numeric data with whatever logical instructions may fill his needs as long as he knows that any data examined will be regarded strictly as a binary quantity. Some of the instructions do not even examine data. The Move Numerics operation, for example, which was designed as a convenient way of moving just the numeric portions of zoned decimal numbers, will move any group or groups of four bits that are in the right location just as cheerfully as it moves valid numerics.

The programmer will find it important to differentiate carefully between the action of the fixed-point Compare and Shift instructions, which are algebraic, and the Logical Compare and Shift instructions, which of course are not. An L in the mnemonics of these logical instructions is a convenience.

In logical operations, processing is performed bit by bit from left to right, whereas arithmetic processing is generally from right to left. Processing may be done either in storage or in general registers. Some of the instructions may be used in a choice of four different formats: RR, RX, SI, or SS. Operands may be four bits, a byte, a word, a doubleword, or as many as 256 bytes for variable-length data in storage. The programmer may select a single bit for attention. The "Immediate" instructions (in the SI format) provide a streamlined method of introducing one byte of immediate data in the instruction statement itself. The action of most of the logical instructions sets the condition code and thus provides a basis for decision-making and branching.

Since the logical operations are covered in detail in the *System/360 Principles of Operation,* these introductory remarks are limited to generalizations, which give only a hint of their range and flexibility. The student is urged to consult the *Principles of Operation* for precise descriptions of their action, for useful programming suggestions, and for examples of their use. He will find it rewarding reading.

The program example in the first section of this chapter demonstrates a method for sorting three items into ascending sequence. The next two sections will show examples of testing combinations of bits with a mask and of setting specified bits on and off. Another program example uses a self-checking number routine to illustrate logical operations on a sequence of characters. A final example demonstrates a series of bit and byte manipulations on input data fields.

ALPHAMERIC COMPARISON: AN ADDRESS SORT

A frequent requirement in commercial data processing is the comparison of two alphameric quantities, such as names or account numbers, for relative "magnitude". Sometimes this is done to establish correspondence between records in two files, both of which are in ascending sequence on the name or account number, which is called the key. Another common application is in arranging a group of records into ascending or descending sequence on keys contained in the records. Let us consider this problem, which is usually called *sorting,* although *sequencing* might in some ways be a preferable term.

The problem will be to arrange three "records" of 13 characters each into ascending sequence on a five-character key contained in the middle five positions of the record. The rearranged records are to be moved to three new record areas named SMALL, MEDIUM, and LARGE.

The basic operation in the program will be an alphameric comparison of two five-character keys to determine relative magnitude. This will be done with a Compare Logical Character instruction (CLC). The word "logical" in the name means that in comparing two characters, all possible bit combinations are valid, and the comparison is made purely on binary values. In a table of EBCDIC character codes, we can see that, according to such a scheme, all letters will be "smaller" than all digits; if punctuation characters occur, they rank smaller than either letters or digits. If we were working with the USASCII code, we would find, on the other hand, that the positions of letters and digits are just the opposite.

For our purposes here, we are not too concerned about the intricacies of where the various characters are ranked by the machine's *collating sequence*; all we really need to know is that names will be correctly alphabetized and that digits are *consistently* ranked somewhere.

The Compare Logical Character instruction is in the SS format and operates on variable-length fields. There is one length code, which applies to both operands. The comparison is from left to right, and continues either until two characters are found that are not the same, or until the end of the fields is reached. As soon as two characters are found to be different, there is no need to continue the comparison. If we are comparing SMITH and SMYTH, we know that SMITH is "smaller" as soon as the I and Y are compared, regardless of what characters follow.

With this much preliminary, let us consider the program in Figure 6-1. Perhaps we should begin by looking at the storage allocation. We see DS entries for A, B, and C, the three original records; these are 13 characters each. Next come three entries that define the addresses of A, B, and C, as ADDRA, ADDRB, and ADDRC, respectively. When we write ADDRA as the operand in a Load, what we get in the register is not A, but its address. Finally, there are DS's for

```
  LOC  OBJECT CODE    ADDR1 ADDR2 STMT   SOURCE STATEMENT

                                     1          PRINT NOGEN
000000                               2 SORTABC  START 0
000000 05B0                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
000002 9824 B072            00074    5          LM    2,4,ADDRA      LOAD REGISTERS WITH ADDRESSES
000006 D504 2004 3004 00004 00004    6          CLC   4(5,2),4(3)    COMPARE A WITH B
00000C 47C0 B014            00016    7          BC    12,X           BRANCH IF A ALREADY LESS OR EQUAL
000010 1862                          8          LR    6,2            IF NOT EXCHANGE ADDRESSES OF A AND B
000012 1823                          9          LR    2,3
000014 1836                         10          LR    3,6
000016 D504 2004 4004 00004 00004   11 X        CLC   4(5,2),4(4)    COMPARE A WITH C
00001C 47C0 B024            00026   12          BC    12,Y           BRANCH IF A ALREADY LESS OR EQUAL
000020 1862                         13          LR    6,2            IF NOT EXCHANGE ADDRESSES OF A AND C
000022 1824                         14          LR    2,4
000024 1846                         15          LR    4,6
000026 D504 3004 4004 00004 00004   16 Y        CLC   4(5,3),4(4)    COMPARE B WITH C
00002C 47C0 B034            00036   17          BC    12,MOVE        BRANCH IF B ALREADY LESS OR EQUAL
000030 1863                         18          LR    6,3            IF NOT EXCHANGE ADDRESSES OF B AND C
000032 1834                         19          LR    3,4
000034 1846                         20          LR    4,6
000036 D20C B07E 2000 00080 00000   21 MOVE     MVC   SMALL,0(2)     ADDRESS OF SMALLEST IS NOW IN REG 2
00003C D20C B08B 3000 0008D 00000   22          MVC   MEDIUM,0(3)    ADDRESS OF MEDIUM IS NOW IN REG 3
000042 D20C B098 4000 0009A 00000   23          MVC   LARGE,0(4)     ADDRESS OF LARGEST IS NOW IN REG 4
                                    24          EOJ                  PROGRAM TERMINATION
00004A                              27 A        DS    CL13
000057                              28 B        DS    CL13
000064                              29 C        DS    CL13
000071 000000
000074 0000004A                     30 ADDRA    DC    A(A)
000078 00000057                     31 ADDRB    DC    A(B)
00007C 00000064                     32 ADDRC    DC    A(C)
000080                              33 SMALL    DS    CL13
00008D                              34 MEDIUM   DS    CL13
00009A                              35 LARGE    DS    CL13
000000                              36          END   BEGIN
```

Figure 6-1. A program to sort three 13-character items into ascending sequence on keys in the middle of each item. The three items are in A, B, and C, and when sorted will be placed in SMALL, MEDIUM, and LARGE.

SMALL, MEDIUM, and LARGE, where the results are to go.

The processing begins by loading the addresses of A, B, and C into registers 2, 3, and 4, respectively, with a Load Multiple. Now we begin a sequence of comparisons and (if necessary) interchanges that will put the three quantities into ascending sequence. We first compare A and B. If A is already equal to or smaller than B, we do nothing; but, if A is larger, we interchange the addresses of A and B. Let us see how this works.

The Compare Logical Character (CLC) instruction following the Load Multiple is written with explicit base registers and explicit lengths. The general format of the instruction is

CLC D1(L1,B1),D2(B2)

As we have written the instruction here, the displacement for operand 1 is 4, the length of both operands is 5, the base register for the first operand is 2, the displacement for the second operand is 4, and the base register for the second operand is 3. Exactly what character positions do these addresses refer to? Remember that base register 2 contains the address of A. This base, plus a displacement of 4, gives the address of the fifth character. Since we said that the key was to be the middle five characters of each record, what we have here is the address of the leftmost character of the key of record A. The length of the key is given explicitly as 5. Operand 2, likewise, gives the address of the key of record B.

The Branch on Condition asks whether the first operand (the key of A) was less than or equal to the second operand (the key of B). If so, there is a branch down to the next comparison, at X, since A and B are already in correct sequence.

If the Branch is not taken, we reach the interchange of A and B. Now, an actual interchange of two 13-character records is a somewhat time-consuming operation; and, of course, this example is only symbolic of real applications, where the records to be sorted might be hundreds of characters long. It is much faster to interchange the *addresses* of A and B than to interchange the records themselves; the addresses are only four characters instead of 13, and, as written here, they are in registers rather than in storage. Three Load Register instructions, which are executed very rapidly, carry out the interchange.

Now, when we continue to the comparison at X, what is the address situation? We know that we want to compare whichever of A and B was the smaller with C. Accordingly, we write addresses using base registers 2 and 4. We cannot say whether 2 contains the address of A or B; but, whichever it is, it is the address of the smaller of the two. That is all we need to know. After this comparison and (possible) interchange, we are guaranteeed that base register 2 contains the address of the smallest of the three numbers.

A final comparison using whatever addresses are by now in registers 3 and 4 gives us the address of the "middle" number in 3 and the address of the largest of the three in 4.

Now, at MOVE, we are able to write three instructions that perform the rearrangement. In the first Move Characters, we pick up the smallest, using whatever is in base register 2. The displacement this time is zero; we want the entire 13 characters. The length can be left implicit this time; it will be implied from SMALL, which is 13 characters long.

With the program loaded at 2000_{16}, Figure 6-2 shows the contents of registers 2, 3, and 4 at four points during execution of the program: at the beginning, at X, at Y, and at MOVE. The three actual data items used for A, B, and C, in order, were 1111CCCCC1111, 2222BBBBB2222, and 3333AAAAA3333. In other words, the items were in reverse order according to their keys.

In practical applications there are usually far too many records to be sorted internally for the keys of all of them to be held in base registers. On the other hand, the records are ordinarily so long that it is a saving in time to work with addresses held in storage rather than with the records themselves. The basic concept suggested here can readily be generalized.

AFTER EXECUTION OF	REG 2	REG 3	REG 4
STATEMENT 5	0000204A	00002057	00002064
STATEMENT 10	00002057	0000204A	00002064
STATEMENT 15	00002064	0000204A	00002057
STATEMENT 20	00002064	00002057	0000204A

Figure 6-2. The contents of registers 2, 3, and 4 at four points during execution of the program in Figure 6-1, loaded at 2000

LOGICAL TESTS

The Wallpaper Problem

Problems sometimes arise in which it is necessary to work with combinations of logical tests, where each test is of the yes-or-no variety. Such situations are often most conveniently attacked as logical operations on sets of binary variables. If the data can be suitably arranged, the tests can sometimes be made very simply with the Test under Mask (TM) instruction.

Consider the following problem. A wallpaper manufacturer classifies his products according to the colors each style contains. There are only four colors: red, blue, green, and orange. For each style there is a group of four bits at the right-hand side of a character named PATTRN. These bits represent, from left to right, the four colors, in the order named. For each bit position, a 1 means that the style contains the color, and a zero means that it does not. For instance, 0001 means a style with orange only; 1010 describes a pattern with red and green, but no blue or orange.

We wish to see how to set up instructions to answer questions of the following sort:

Does this pattern have *either* red or green, or both?

Does this pattern have red, or green, or orange, or any two of these, but not all three?

Does this pattern have both red and orange, whether or not it has blue and/or green?

Does this pattern have neither green nor orange?

Does this pattern have red but *not* orange?

Let us consider these questions in order.

Red, or green, or both. Looking at the four color-bits, we are interested in the first and third. If we let X stand for a bit that we want to be a 1, and D for a bit about which we don't care, the required pattern is XDXD.

The Test under Mask instruction can handle this situation with just two instructions:

```
TM   PATTRN,X'0A'
BC   5,YES
```

In the Test under Mask instruction, the 0A is the mask, written here in hexadecimal. Writing it out as a binary number, we have 00001010. The two 1's here pick out the two bits in the character at PATTRN that are to be tested. The resulting condition codes have meanings as follows: a code of zero means that all the selected bits were zero; a code of 1 means that the selected bits were mixed zeros and 1's; a condition code of 3 means that the selected bits were all 1's. (A condition code of 2 is not possible with this instruction.) The question to be answered was: Does this pattern contain either red, or green, or both? We have selected the two bits that describe the presence or absence of red and green. If the two bits selected were a mixture of zeros and 1's we have just one of the two colors in the pattern. If the two bits selected were both 1's, the pattern contains both colors. Either situation answers the question affirmatively. We accordingly write a Branch on Condition instruction that tests for the presence of condition codes 1 or 3. (Remember that 8, 4, 2, and 1 in the R1 field of a BC correspond to condition codes of 0, 1, 2, and 3, respectively. Branch on Condition with an R1 field of 5, therefore, tests for a condition code of either 1 or 3.) At YES, we assume there would be instructions to do whatever action depended on an affirmative answer to the question.

Red, green or orange, but not all three. Here we need a mask that tests bits according to this scheme: XDXX. The necessary mask is 00001011, which is 0B in hexadecimal. The condition code that describes the wallpaper design specified is 1: mixed zeros and 1's. We want at least one 1, and two would do, but we must have at least one zero among the bits tested because the pattern must not have all three colors. The required instructions are:

```
TM   PATTRN,X'0B'
BC   4,YES
```

The conditional branch could equally well have been written as the extended mnemonic used after Test under Mask instructions, BM YES (BM means Branch if Mixed).

Both red and orange. This one is fairly simple. We pick out bits according to XDDX, and then ask whether they are all (both) 1's. The instructions are:

```
TM   PATTRN,X'09'
BC   1,YES (or BO YES)
```

Neither green nor orange. This is not very difficult, either. The bits are shown by DDXX, and we want to know whether they are all (both) zero. The instructions are:

```
TM   PATTRN,X'03'
BC   8,YES (or BZ YES)
```

Red but not orange. This is a different problem that cannot be done with a single Test under Mask. We turn to the logical instructions And, and Exclusive Or. The bits in question are shown as X's in XDDX. We want the leftmost X to be a 1, and the rightmost to be a zero.

We begin by moving PATTRN to WORK, where we may destroy its original value. An And Immediate instruction with an immediate portion of 09 (in binary: 00001001) erases all bits except the ones we want. In the two positions of interest, if there was a 1 before, there still is, and if there was a zero, there still is. All other bit positions are guaranteed to be zero. If the pattern is to pass the test, there must now be exactly one 1 in WORK, and it must be in this position: 0000X000. Whether this is so could be determined with a comparison or two Test under Mask instructions. But let us continue with the logical operations.

Exclusive Or is a logical operation; like And and Or, it is a bit-by-bit operation. In each bit position, the result is 1 if the two operands had exactly one 1 in that position; the result bit is zero if both operand bits were zero or if both were 1. Suppose we write an Exclusive Or Immediate in which the immediate portion is 00001000; the 1 here is in the position for red. The result after the Exclusive Or Immediate will be zero in this position if there had been a 1, and vice versa.

In other words, if the result really were 00001000 after the And Immediate, there would be *all* zeros after the Exclusive Or Immediate. If, on the other hand, there were a zero in the position for red, there would now be a 1. And if there were a 1 in the position for orange, there would still be a 1 there. In short, a zero result corresponds to an answer of "yes, there is red but no orange". As it happens, the various logical operations used here all set the condition code; and, in the case of the Exclusive Or, a condition code of zero means that the result was zero. The program can thus be:

```
MVC    WORK,PATTRN
NI     WORK,X'09'
XI     WORK,X'08'
BC     8,YES (or BZ YES)
```

Test under Mask is a most useful instruction where it applies, and its usefulness is by no means limited to color-blind wallpaper manufacturers. It is useful partly because it is selective, testing only the bits specified by the mask, and partly because it gives a three-way description of the selected bits: all zero, mixed, or all 1's. It does have the drawback, however, that only one character can be tested at a time.

If it were necessary to extend the application to cover, say, 20 different yes-no descriptions, the Test under Mask instruction could not be used, except in combinations that would get rather involved. In such a situation, we would turn instead to the RX forms of the logical instructions. After moving the pattern to a register, which can hold a 32-bit pattern, we would use an And to "select" the bits of interest. The operand of the And instruction would be a fullword in storage that has 1's where there are bits of interest in the pattern.

What we do next depends on our answers to certain questions.

Question: Were *any* of the selected bits 1's?

Action: We need only test the condition code, which tells whether the result was all zeros or had at least one 1.

Question: Were certain of the selected bits 1, with the others being zero?

Action: We execute an Exclusive Or to change to zero the bits that should be 1's, then ask whether the result is all zero.

Working with larger groups of bits is thus seen not to be a great deal more difficult than working with a single character.

Setting Bits On and Off

A problem related to the one we have been considering is to set a specified bit of a character or a word to be zero or 1, or perhaps to reverse them from whatever they are. This might be necessary, for instance, if we were writing a program to develop the wallpaper codes that we tested in the preceding section.

Bearing in mind that fullword operands represent only a minor amount of additional programming effort, let us see how to carry out these operand operations on one-character operands.

To set a specified bit to 1, an Or Immediate is sufficient. Suppose that we are still working with a character named PATTRN, which now uses all eight bits, and that we want 1, 3, 6, and 7 to be "on" (1). We are not interested in the status of bits 0, 2, 4, and 5. In other words, we want the pattern to be D1D1DD11, where the D's stand for "don't care" or "leave them whatever they were". This action is precisely what will result from an Or Immediate in which the immediate part is 01010011 (53 hexadecimal). The Or results in a 1 in any bit position in which either operand, or both, had a 1. (The case of both having 1 is not excluded, as in the Exclusive Or. The ordinary Or is sometimes called the "inclusive" Or to distinguish between the two.)

The instruction could be

```
OI   PATTRN,X'53'
```

If the required action is to set the same four bit-positions to zero, regardless of their previous values, and leave the others as they were, we would use an And Immediate with zeros where we want zeros and 1's where we want the previous contents undisturbed. The necessary immediate portion is 10101100 (AC in hexadecimal). The instruction is therefore

```
NI   PATTRN,X'AC'
```

The And places a 1 in bit positions in which *both* operand bits were 1, and zero elsewhere. Wherever we put zeros in the immediate portion, therefore, there will be zeros in the result, as required. Wherever we placed 1's there will be a 1 if there was before, or a zero if there was a zero before. This is exactly what we need.

Sometimes it is necessary to change a bit to 1 if it was zero, and to zero if it was 1. This is called *complementing* a bit. If we place 1's in the immediate portion wherever we want this complementing action, the Exclusive Or Immediate does precisely what is needed. Other bit positions will be unchanged. Assuming we are still working with bits 1, 3, 6, and 7, the instruction is

```
XI   PATTRN,X'53'
```

A SELF-CHECKING NUMBER ROUTINE

It is fairly common practice in business to devise account numbers for things like credit cards so that the number is self-checking. This means that one of the digits is assigned to provide a certain amount of protection against fraud and clerical errors. This digit is assigned by some fixed sequence of operations on the other digits.

We shall work in this section with a ten-digit account number, the last (rightmost) of which is a check digit. This digit is computed when the number is assigned. It consists of the last digit of the sum found by adding together the second, fourth, sixth, and eighth digits, together with three times the sum of the first, third, fifth, seventh, and ninth digits. For instance, if a nine-digit account number is 123456789, the check digit is the last digit of the sum

$$(2+4+6+8)+3\,(1+3+5+7+9)=95$$

The last digit is five, so the complete account number would be 1234567895.

There is a certain protection against fraud here; unless the person attempting the fraud knows the system, there is only one chance in ten that an invented account number will be a valid one.

More important, perhaps, there is considerable protection against clerical error. If any one digit is miscopied, the erroneous account number will not pass the check. Furthermore, most transpositions of two adjacent digits will cause the check to fail. For instance, the check digit for 132456789 would be

$$(3+4+6+8)+3\,(1+2+5+7+9)=93$$

The computed check digit of 3 is obviously not the same as the one in the number, so the account number is rejected as invalid.

We wish now to study a program that will determine whether an account number that has been entered into the computer is valid. We begin the program with a nine-digit account number in ACCT, in zoned format. Immediately following ACCT is a one-digit check digit named CHECK, also in zoned format.

In the program in Figure 6-3 we begin by loading register 3 with a 1. This will be used to determine whether a digit should be multiplied by 3 or not, as we shall see below. Register 4 is loaded with a 9; this is an index register, used to get the digits in order from right to left. A Move Character puts a signed zero in SUM where the sum of the digits will be developed. A Subtract Register clears register 5 to zero.

At LOOP we begin the processing of digits. With index register 4 containing 9, the effective address the first time through the loop will be ACCT+8, which is the address of the rightmost digit. The index is reduced by one each time around the loop by the Branch on Count Instruction, so we pick up the digits one at a time, from right to left, as stated.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1          PRINT NOGEN
000000                               2 ACCTNO   START 0
000000 05B0                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
000002 4130 0001            00001    5          LA    3,1             REG 3 WILL HAVE SIGN REVERSED IN LOOP
000006 4140 0009            00009    6          LA    4,9             COUNTER FOR 9 DIGITS IN NUMBER
00000A D201 B064 B066 00066 00068    7          MVC   SUM,ZERO        SUM OF DIGITS KEPT IN SUM
000010 1B55                          8          SR    5,5             CLEAR REG 5
000012 4354 B059            0005B    9 LOOP     IC    5,ACCT-1(4)     PICK UP 1 DIGIT OF INDEXED NUMBER
000016 8950 0004            00004   10          SLL   5,4             SHIFT LEFT 4 BITS
00001A 5650 B06A            0006C   11          O     5,PLUS          ATTACH A PACKED PLUS SIGN
00001E 4250 B068            0006A   12          STC   5,DIGIT         STORE IN TEMPORARY LOCATION
000022 FA10 B064 B068 00066 0006A   13          AP    SUM,DIGIT       ADD TO SUM OF DIGITS
000028 1333                         14          LCR   3,3             REVERSE SIGN OF REG 3
00002A 4720 B038            0003A   15          BC    2,EVEN          SKIP NEXT 2 INSTR ON PLUS ODD TIMES THRU
00002E FA10 B064 B068 00066 0006A   16          AP    SUM,DIGIT       IF NOT SKIPPED ADD DIGIT TO SUM
000034 FA10 B064 B068 00066 0006A   17          AP    SUM,DIGIT       SAME. HAS EFFECT OF MULTIPLYING BY 3
00003A 4640 B010            00012   18 EVEN     BCT   4,LOOP          BRANCH BACK IF NOT ALL DIGITS PROCESSED
00003E 4350 B063            00065   19          IC    5,ACCT+9        PUT CHECK DIGIT IN REG 5
000042 8950 0004            00004   20          SLL   5,4             SHIFT LEFT 4 BITS
000046 5650 B06A            0006C   21          O     5,PLUS          ATTACH SIGN TO PUT IN SAME FORMAT AS SUM
00004A 4250 B064            00066   22          STC   5,SUM           PUT ONE BYTE IN LEFT BYTE OF SUM
00004E D500 B064 B065 00066 00067   23          CLC   SUM(1),SUM+1    IS THIS BYTE SAME AS CHECK DIGIT
000054 4770 B058            0005A   24          BNE   ERROR           BRANCH TO ERROR ROUTINE IF NOT EQUAL
                                    25 OUT      EOJ                   PROGRAM WOULD NORMALLY CONTINUE HERE
                                    28 ERROR    EOJ
00005C                              31 ACCT     DS    CL9
000065                              32 CHECK    DS    CL1
000066                              33 SUM      DS    CL2
000068 000C                         34 ZERO     DC    PL2'0'
00006A                              35 DIGIT    DS    CL1
00006C                              36          DS    0F
00006C 0000000C                     37 PLUS     DC    XL4'0C'
000000                              38          END   BEGIN
```

Figure 6-3. A self-checking account number routine that recalculates a check-digit and verifies it

The digit inserted in register 5 is shifted left four bits. This puts the numeric part of the digit, which was in zoned format, into the leftmost four bits of an eight-bit byte at the right end of the register, and brings in four zeros at the right. Or-ing with PLUS puts a plus sign into the rightmost four bits (note that the machine code generated by this DC is 0000000C), and we have a one-digit byte in correct packed format for use with an Add Decimal. We therefore put the assembled digit into a working storage location at DIGIT and add it to SUM.

Now comes the question of whether or not this is a digit that is to be multiplied by 3. The rule requiring digits to be so multiplied can be stated thus: the first digit is multiplied by 3; after that, every other digit is so multiplied. In other words, we need some technique for getting a branch *every other* time through the loop. The method shown here is to reverse the sign of the contents of register 3 every time, then to ask whether the result is positive. The first time through we change a +1 to -1; the answer is "no, the result is not positive". The second time through we change a -1 to +1, and the answer is "yes, the result is positive". The third time through the +1 gets changed back to -1, and the answer is no. In short, every other time we ask whether the result of reversing the sign of register 3 is positive, the answer will be yes. We accordingly Branch on Condition to EVEN if register 3 is positive. This means that for digits in even positions 2, 4, 6, and 8, the two additional Add Decimal instructions will be skipped. These, if they are executed, have the effect of adding in a digit three times instead of once, which is equivalent to multiplying and somewhat faster.

At EVEN we Branch on Count back to LOOP if, after reducing the contents of 4 by one, the result is not zero. The loop will therefore be executed the last time around with 1 in register 4, so the last digit picked up is at ACCT, as it should be.

Once all nine digits have been added to sum, we are ready to see whether the last digit of SUM is the same as CHECK. But it isn't quite that simple; the digit at CHECK is still in zoned format. We accordingly go through the steps necessary to convert it to packed format, storing it for comparison in the left byte of SUM, which we no longer need. A Compare Logical Character with an explicit length of one now determines whether the check digit that came with the account number, which is now in SUM, is the same as the computed check digit, which is now in SUM+1. We have ended the error path as well as the normal path with an End of Job macro instruction. In a real situation additional steps would be included to enable investigation of an invalid account number, and both paths would branch back to LOOP to continue with the next account number in the input stream.

There are, of course, many other techniques for computing check digits which give greater protection or make the check digit operations simpler.

A FINAL EXAMPLE

We are given two numbers, NUMBER and COMB. NUMBER is a seven-digit quantity in zoned format. We are to test each of the seven numeric portions separately in order to be certain that each represents a digit, that is, that the value of the numeric portion is less than ten. If each character contains a valid digit, we go on to the next test; if any one contains numeric bits not valid for a digit, we shall simply go to an End of Job. After completing this test, we are to check the zone bits of the rightmost byte of NUMBER to be sure that it contains a sign. The other zone positions are of no interest. As before, if there is an error condition, we go to an EOJ.

Next, we start with an eight-byte composite field named COMB. We shall assume for the purposes here that the numeric portions of each byte all represent valid digits; if this were questionable, they could be checked. The zones of the eight bytes contain either plus or minus signs. A plus sign is to be taken as meaning 1 and a minus sign as meaning zero; we are to assemble a one-byte quantity that contains a binary number formed from the signs. For instance, Figure 6-4 shows a card field that could have produced the data in COMB. If this field were viewed as an alphabetic quantity in normal IBM card code, it would be ABLMEOGQ. We want to view it, instead, as being a positive number 12345678 together with a binary number (contained in the zone punching area) of 11001010. The 1's and zeros here correspond to the zones: + + - - + - + -. We are to separate the two items contained in COMB, placing the number in NUMERC as a packed decimal number and the zones in CODES as a one-byte binary number.

A flowchart showing the logic of this problem is shown in Figure 6-5 and the program in Figure 6-6 does the processing required. We start by placing a 7 in register 10, for use as an index. Register 9 is cleared. The instructions from LOOP to OK pick up the digits in turn, strip off the zone bits with a suitable And, and compare the numeric portions with 10.

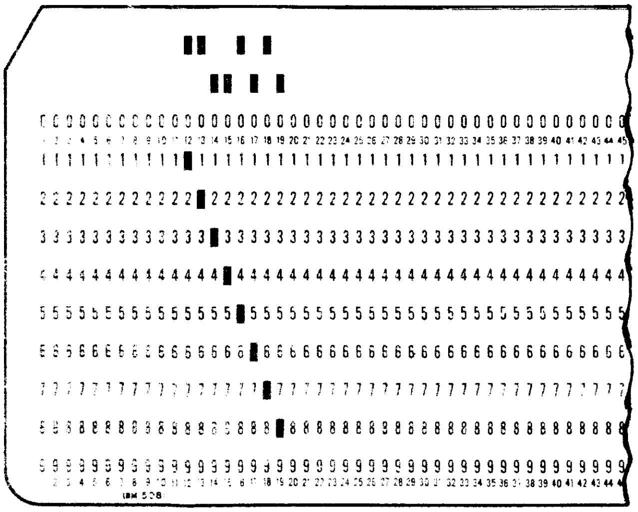


Figure 6-4. Alphabetic input for COMB that can be viewed as two numbers: 12345678 and binary 11001010

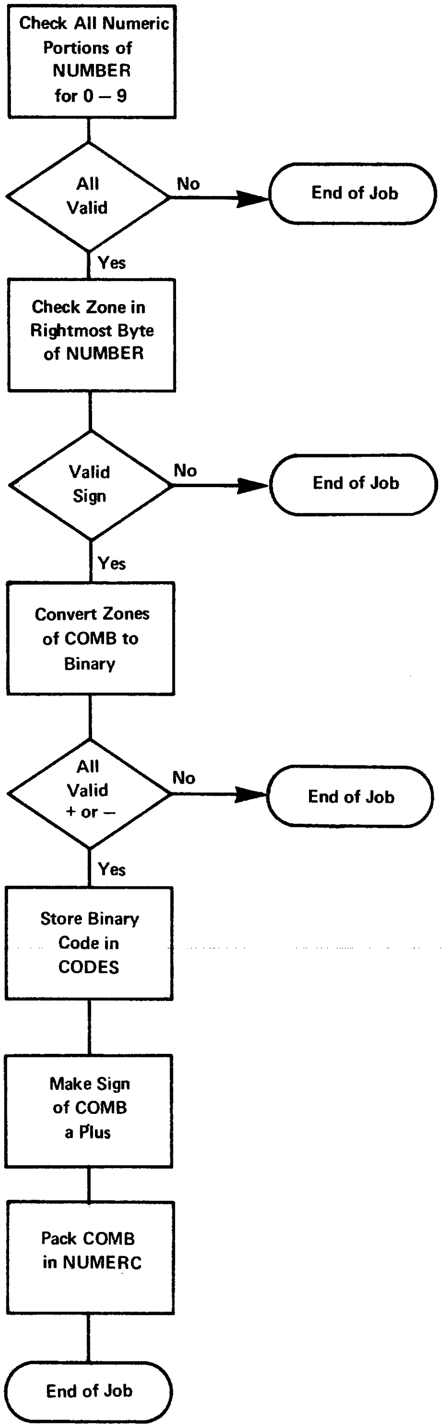


Figure 6-5. A flowchart of the steps required to solve the problem

The instruction after OK picks up the rightmost byte of NUMBER; this should have either a plus sign or a minus sign. Another And, but with a different mask, strips off the numeric portion and the rightmost bit of the sign; we do not care whether the sign is plus or minus, a distinction which is made in the rightmost bit of the sign. A comparison then establishes whether the left three bits of the sign are 110, which they should be for an EBCDIC sign.

At OK2 we are ready to go to work on the combined digits and zones at COMB. In preparation for what follows, we clear registers 8, 9, and 10. At LOOP2 there is a shift – before anything has been placed in the register shifted. The idea is that we want to shift the contents of this register seven times for eight bits. One way to accomplish this is to place the shift instruction so that it has no net effect the first time around.

The Insert Character is indexed with register 10, which initially contains zero. We will therefore pick up the digits from left to right this time. For each digit we use an And to drop the numeric bits, then test against constants so as to determine whether the sign is plus or minus. If it is neither, we get out; there should be one or the other. If the sign is plus, we branch to YES, where a 1 is added into register 9 – the one that we shifted at the beginning of the loop. Whether the sign is plus or minus, we now reach NO, where we add 1 to the index register and branch back to LOOP2 if the contents are less than eight.

Now, when we branch back, we again shift the contents of register 9 one position to the left. This means that each time we again reach the beginning of this loop, whatever has been assembled in register 9 so far is shifted left one place, thereby making room for another bit at the rightmost position of the register. Thus, when we finally get out of the loop and arrive at the Store Character, the last byte of register 9 will contain a 1 in positions corresponding to plus signs in COMB, and zeros in positions corresponding to minus signs. The byte stored at CODES is just what the problem statement required.

An And Immediate now erases the zone positions of the rightmost byte of COMB, and an Or Immediate places a plus sign there. The Pack instruction does not check zones, except in the rightmost byte, so we can proceed to it immediately, with no concern for the other zone positions.

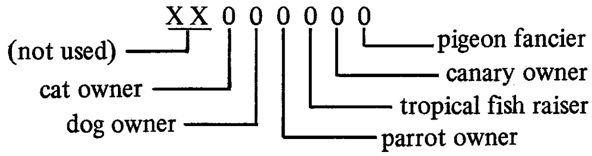
```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1          PRINT NOGEN
000000                               2 FORMAT   START 0
000000 05B0                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
000002 41A0 0007            00007    5          LA    10,7            REG 10 IS USED AS AN INDEX
000006 1B99                          6          SR    9,9             CLEAR REG 9
000008 439A B075            00077    7 LOOP     IC    9,NUMBER-1(10)  INSERT 1 DIGIT IN REG 9--INDEXED
00000C 5490 B08E            00090    8          N     9,MASK1         STRIP OFF SIGN
000010 5990 B0A2            000A4    9          C     9,TEN           IS NUMBER LESS THAN 10
000014 4740 B018            0001A   10          BL    OK              BRANCH AROUND EOJ IF OK
                                    11          EOJ                   NOT A DIGIT
00001A 46A0 B006            00008   14 OK       BCT   10,LOOP         REDUCE CONTENTS OF REG 10 BY 1 & BRANCH
00001E 4380 B07C            0007E   15          IC    8,NUMBER+6      IF HERE, ALL DIGITS CHECKED OK
000022 5480 B092            00094   16          N     8,MASK2         STRIP OFF LAST DIGIT & FINAL SIGN BIT
000026 5980 B09A            0009C   17          C     8,PLUS          COMPARE 3 REMAINING BITS WITH SIGN
00002A 4780 B02E            00030   18          BE    OK2             BRANCH IF OK
                                    19          EOJ                   NOT AN EBCDIC SIGN
000030 1B88                         22 OK2      SR    8,8             CLEAR REG 8
000032 1898                         23          LR    9,8             CLEAR REG 9 BY LOADING FROM REG 8
000034 18A8                         24          LR    10,8            CLEAR REG 10 BY LOADING FROM REG 8
000036 8B90 0001            00001   25 LOOP2    SLA   9,1             SHIFT REG 9 LEFT 1 BIT
00003A 438A B07D            0007F   26          IC    8,COMB(10)      INSERT 1 BYTE IN REG 8--INDEXED
00003E 5480 B096            00098   27          N     8,MASK3         STRIP OFF DIGIT PART
000042 5980 B09A            0009C   28          C     8,PLUS          COMPARE WITH CODING FOR PLUS
000046 4780 B052            00054   29          BE    YES             BRANCH IF PLUS
00004A 5980 B09E            000A0   30          C     8,MINUS         COMPARE WITH CODING FOR MINUS
00004E 4780 B056            00058   31          BE    NO              BRANCH IF MINUS
                                    32          EOJ                   NEITHER PLUS NOR MINUS
000054 5A90 B0A6            000A8   35 YES      A     9,ONE           IF PLUS ADD 1 TO CONTENTS OF REG 9
000058 5AA0 B0A6            000A8   36 NO       A     10,ONE          ADD 1 TO REG 10 FOR LOOP TEST
00005C 59A0 B0AA            000AC   37          C     10,TEST         COMPARE
000060 4770 B034            00036   38          BNE   LOOP2           BRANCH BACK IF NOT FINISHED
000064 4290 B085            00087   39          STC   9,CODES         STORE LAST BYTE OF REG 9
000068 940F B084      00086         40          NI    COMB+7,X'0F'    STRIP OFF OLD ZONE
00006C 96C0 B084      00086         41          OI    COMB+7,X'C0'    ATTACH ZONED PLUS SIGN
000070 F247 B086 B07D 00088 0007F   42          PACK  NUMERC,COMB     CONVERT TO PACKED FORMAT
                                    43          EOJ                   PROGRAM TERMINATION
000078                              46 NUMBER   DS    CL7
00007F                              47 COMB     DS    CL8
000087                              48 CODES    DS    CL1
000088                              49 NUMERC   DS    CL5
000090                              50          DS    0F
000090 0000000F                     51 MASK1    DC    X'0000000F'
000094 000000E0                     52 MASK2    DC    X'000000E0'
000098 000000F0                     53 MASK3    DC    X'000000F0'
00009C 000000C0                     54 PLUS     DC    X'000000C0'
0000A0 000000D0                     55 MINUS    DC    X'000000D0'
0000A4 0000000A                     56 TEN      DC    F'10'
0000A8 00000001                     57 ONE      DC    F'1'
0000AC 00000008                     58 TEST     DC    F'8'
000000                              59          END   BEGIN
```

Figure 6-6. A program that checks a decimal field at NUMBER for validity and converts a composite field at COMB into separate binary and packed decimal quantities. The flowchart in Figure 6-5 was used as a guide for the programming

QUESTIONS AND EXERCISES

1. The byte at location KEY in main storage contains four program switches in bit positions 4–7. Each of these bit positions may be 1 (on) or 0 (off). Write an instruction that will reverse the setting of the program switches and leave bits 0-3 unchanged.

2. In the following byte, located at ADDR in main storage, a 1 in a particular position shows the presence of a characteristic and a zero its absence. Write instructions that will branch to ANIMAL for owners of dogs or cats or both, and proceed sequentially for all others.



3. Using the preceding, write instructions to branch to LIST2 for owners of fish but not canaries, or canaries but not fish.

4. Suppose location SUM contains 05432+ in packed decimal format, and suppose that general register 2 initially contains zero. Show what register 2 will contain (in hexadecimal or binary) after:

   a. IC 2, SUM
   b. IC 2,SUM+2
   c. IC 2,SUM+1

5. At most, the TM (Test Under Mask) instruction can test ______________ bit(s) or ______________ byte(s) with one instruction.

6. At most, the CLC (Compare Logical Character) instruction can compare ______________ bit(s) or ______________ byte(s) with one instruction.

7. The CLC instruction will successfully compare two operands in only one of the following forms. Which is it?

   a. Packed decimal numbers
   b. Alphameric characters
   c. Zoned decimal numbers

8. In the CLC instruction, comparison proceeds from left to right, byte by byte, but ceases before the end of the operand is reached, as soon as one of the following is encountered (select one):

   a. The EBCDIC sign code
   b. A special character
   c. An inequality
   d. An improper zone code

9. Neglecting leading zeros, give in decimal the contents of general register 5 after execution of each of the following:

```
a.         LA     5,5
b.         LA     5,2
c.         LA     5,3(0,1)
d.         LA     5,FIELD
           .......
   FIELD   DS     F
```

10. Write instructions to determine whether or not the byte at main storage location FIELD contains a 5 (0000 0101 in binary).

11. In the following hypothetical program, the rows of dots represent straightforward instruction sequences of any reasonable length, whose nature need not concern us.

```
        LA     2,10
LOOP    . . . . .
        . . . . .
INST    BC     0,ADDR
        OI     INST+1,X'F0'
        . . . .
        . . . .
ADDR    . . . .
        . . . .
        BCT    2,LOOP
```

Which part of the BC instruction is addressed by the relative address INST+1?

12. Bearing in mind that in question 11 the hexadecimal immediate data X'F0' is simply a convenient way of specifying binary 11110000 (or decimal 240), can you say that the OI (Or Immediate) instruction:

   a. Will be executed once and only once?
   b. Causes certain instructions within the BCT loop to be skipped on all but the first execution of the loop?
   c. Alters the bit structure of a mask field?
   d. Does all of the above?

13. Assume that the overall loop of the following sequence will be executed a number of times. What will be the effect of the XI (Exclusive Or) instruction?

```
LOOP    . . . .
        XI     INST+1,X'F0'
INST    BC     0,ADDR
        . . . .
ADDR    . . . .
        BCT    5,LOOP
```

14. Suppose that general register 5 contains a number of which only the high-order (leftmost) byte is of interest. Write a logical instruction to zero the three low-order bytes, together with any instructions necessary to define masks, load other registers, etc., as required.

Chapter 7: Edit, Translate, and Execute Instructions

This chapter will be devoted to several highly specialized and useful instructions that are part of the assembler language. They call into play some new concepts, and their functions and machine actions are different in many ways from any of the instructions we have encountered so far. Since they may be regarded as irregular verbs, so to speak, of System/360 Assembler Language, we will subject each of them to careful scrutiny.

The Execute (EX) instruction is a special type of branching instruction that causes one other instruction in main storage to be executed out of sequence without actually branching to its location. Since Execute can also modify the remote instruction before it is executed, it offers considerable economy in the number of instructions needed to achieve certain results.

The other instructions covered in this chapter are Edit, Edit and Mark, Translate, and Translate and Test. These are part of the System/360 logical operations discussed in the preceding chapter. We begin with a detailed demonstration of how the Edit, and the almost identical Edit and Mark, instructions work. These two instructions are invaluable aids to any programmer concerned with decimal arithmetic. Translate can be used for code conversion or to provide a control function. The description of the Translate instruction is necessary for an understanding of the Translate and Test (TRT), which follows it. Detailed program examples are included, with special emphasis on the use of the powerful combination of TRT and EX in various applications. The programmer will find many additional applications for the techniques demonstrated in this chapter.

THE EDIT INSTRUCTION

The Edit instruction is one of the most powerful in the repertoire of the System/360. It is used in the preparation of printed reports to give them a high degree of legibility and therefore greater usefulness. It makes it possible, as we shall see, to suppress nonsignificant zeros, insert commas and decimal points, insert minus signs or credit symbols, and specify where suppression of leading zeros should stop for small numbers. All of these actions are done by the machine in *one* left-to-right pass. The condition code can be used to blank all-zero fields with two simple instructions. A variation of the instruction, Edit and Mark, makes possible the easy insertion of floating currency symbols.

We shall study the application and results of this highly flexible instruction by applying it to successively more complex situations.

We begin with a simple requirement to suppress leading zeros; no punctuation is to be inserted. We have a field to be edited, called DATA. It is four bytes long, and the decimal data is in packed format. The packed format for data to be edited is a requirement of the Edit (ED) instruction, which is a decimal instruction. As we saw in an earlier chapter, data used in decimal arithmetic operations is always in packed format. If we happened to have source data in some other form, we would have to pack it before editing.

The data to be edited is named as the second operand of the Edit. The first operand must name a field containing a "pattern" of characters that controls the editing; after execution of the instruction, the location specified by the first operand contains the edited result. (The original pattern is destroyed by the editing process.) The pattern is in zoned format, as is the result; the Edit instruction causes the conversion of the data to be edited from packed to zoned format, since zoned format is what is needed for most output operations.

We said that in our example the data field to be edited was four bytes long, that is, seven decimal digits and sign, which we shall assume to be plus. The pattern must accordingly be at least eight bytes long: seven for the digits and one at the left to designate the "fill character". The fill character is of our choosing, but is usually a blank. This is the character that is substituted for nonsignificant zeros.

The leftmost character of the pattern in our case will be the character blank (hexadecimal 40 in System/360 EBCDIC coding). The other seven characters will contain hexadecimal 20, a control character called a digit selector, which is used to indicate to the Edit instruction that a digit from the source data may go into the corresponding position.

Let us see how all this works out in our example. Suppose we set up an eight-byte working storage field named WORK into which we move the pattern (located in an area called PATTRN). Then we will perform our edit using WORK and DATA as the two operands. The two instructions necessary to do the job are:

```
MVC     WORK,PATTRN
ED      WORK,DATA
```

After execution of the two instructions, WORK contains our edited result. PATTRN still contains the original pattern and can transmit that original pattern to WORK for the editing of any new value in DATA. At PATTRN there should be the following characters, written here in hexadecimal:

40 20 20 20 20 20 20 20

or as they would appear in an actual program, defined as a hexadecimal constant:

```
PATTRN   DC   X'4020202020202020'
```

In EBCDIC, 40 is the hexadecimal code for a blank and 20 for the digit selector control character. Hex is used to specify control characters, since there are no written or printed symbols to represent them. In this section, all patterns are shown exactly as they would appear in constants, except of course that the spaces would be closed up.

In our example, suppose that at DATA there is

00 01 00 0+

The edited result would be

b b b b 1 0 0 0

where the b's stand for blanks. All zeros to the left of the first nonzero digit have been replaced by blanks; but zeros to the ***right*** of the first nonzero digit have been moved to WORK without change. This is the desired action. Figure 7-1 shows a series of values for DATA and the resultant edited results in WORK, using the pattern stated. Note that the high-order position of WORK contains the fill character, a blank. The values of DATA are packed decimal; the edited results are changed during execution of the Edit instruction to zoned decimal format.

```
 BDDDDDDD
 40 20 20 20 20 20 20 20

1234567   1234567
0120406    120406
0012345     12345
0001000      1000
0000123       123
0000012        12
0000001         1
0000000
```

Figure 7-1. Results of Editing source data in left-hand column. Two lines at top give editing pattern in symbolic form (B represents a blank, D a digit selector) and in hexadecimal coding.

The fill character that we supply as the leftmost character of the pattern may be any character that we wish. It is fairly common practice to print dollar amounts with asterisks to the left of the first significant digit in order to protect against fraudulent alteration. This is usually called asterisk protection.

To do this, we need only change the leftmost character of the pattern of the previous example. The hexadecimal code for an asterisk is 5C; hence the new pattern is

5C 20 20 20 20 20 20 20

Figure 7-2 shows the edited results for the same DATA values that we used in Figure 7-1.

```
*DDDDDDD
5C 20 20 20 20 20 20 20

1234567  *1234567
0120406  **120406
0012345  ***12345
0001000  ****1000
0000123  *****123
0000012  ******12
0000001  *******1
0000000  ********
```

Figure 7-2. Editing results with an asterisk as the fill character

Any characters in the pattern other than the digit selector and two other control characters that we shall study later are called message characters. They are *not* replaced by digits from the data. Instead, they are either replaced by the fill character (if a significant digit has not been encountered yet), or left as they are (if a significant digit has been found). Suppose, for instance, that we set up a PATTRN as follows:

40 20 6B 20 20 20 6B 20 20 20

The 6B is hexadecimal coding for a comma, and it is a message character. The edited result will contain commas in the two positions shown, unless they are to the left of the first nonzero digit, in which case they are suppressed. Figure 7-3 shows the results for the same data values.

```
BD,DDD,DDD
40 20 6B 20 20 20 6B 20 20 20

1234567   1,234,567
0120406     120,406
0012345      12,345
0001000       1,000
0000123         123
0000012          12
0000001           1
0000000
```

Figure 7-3. Editing results with blank fill and the insertion of commas

The message characters inserted are, naturally, not limited to commas. A frequent application is to insert a decimal point as well as commas. Let us assume that the data values we have been using are now to be interpreted as dollars-and-cents amounts. We need to arrange for a comma to set off the thousands of dollars, and a decimal point to designate cents. The characters in PATTRN, where 6B is a comma and 4B is a decimal point, should be as follows:

40 20 20 6B 20 20 20 4B 20 20

The edited results this time are in Figure 7-4.

We see here something that would normally not be desired: amounts under one dollar have been edited with the decimal point suppressed. We would ordinarily prefer to have the decimal point. This can be done by placing a significance starter in the pattern. This control character, which has the hexadecimal code 21, is either replaced by a digit from the data or replaced by the fill character, just as a digit selector is. The difference is that the operation proceeds *as though* a significant digit had been found in the position occupied by the significance starter. In other words, succeeding characters to the right will not be suppressed. (An exception to this generalization may occur when we want to print sign indicators, a subject that will be explored later.)

```
BDD,DDD.DD
40 20 20 6B 20 20 20 4B 20 20

1234567   12,345.67
0120406    1,204.06
0012345      123.45
0001000       10.00
0000123        1.23
0000012          12
0000001           1
0000000
```

Figure 7-4. Editing results with blank fill and the insertion of comma and decimal point

The pattern for this action, assuming we still want the comma and decimal point as before, should be

40 20 20 6B 20 20 21 4B 20 20

The effect is this: if nothing but zeros has been found by the time we reach the significance starter (hex 21) in a left-to-right scan, the significance starter will turn on the significance indicator. This indicator will cause succeeding characters to be treated as though a nonzero digit had been found. The result is that the decimal point will always be left in the result, as will zeros to the right of the decimal point. The edited results this time are shown in Figure 7-5.

One useful point to remember is that the total number of digit selectors plus significance starters in the pattern must equal the number of digits in the field to be edited. Note that this is the case in all our examples.

```
BDD,DDS.DD
40 20 20 6B 20 20 21 4B 20 20

1234567    12,345.67
0120406     1,204.06
0012345       123.45
0001000        10.00
0000123         1.23
0000012          .12
0000001          .01
0000000          .00
```

Figure 7-5. Editing results with blank fill, comma and decimal point insertion, and significance starter. In the symbolic pattern, S stands for significance starter.

We can begin to get a little idea of how the machine does its work on this instruction by noting that the significance indicator is initially in the off state before the scan begins. Scanning proceeds source digit by source digit. The significance indicator stays off until a nonzero data digit is found, or until the significance starter is encountered; either event causes the indicator to be turned on.

Source digits 1–9 always replace a digit selector or significance starter, but whether a zero source digit will do so depends upon the state of the significance indicator. If the significance indicator is on, then we know that either a significant digit was found at some previous character position, or a significance starter has been encountered; in either case, a zero from the source data is inserted. If the significance indicator is off, we know that no significant digit has been found so far during the scan; therefore, the fill character appears in the result, rather than a zero from the data.

It may be useful to refer to Table 7-1, which includes a summary of how the state of the significance indicator affects the editing operation under all conditions of consequence that you may encounter. The table also shows how the significance indicator itself is affected.

In the table, the four columns at the left list all the significant combinations of the four conditions that can be encountered in the execution of the editing operation. The two columns at the right under Results show the action taken for each case – that is, the type of character placed in the result field and the new setting of the significance indicator. Use of the field separator will be discussed in a later paragraph.

We have so far ignored the sign portion of the source data, which (in the packed decimal format required for the Edit instruction) is in the four low-order bits of the rightmost byte. These bits are examined each time the Edit instruction is executed. If the sign is plus, the significance indicator will then be turned off, as shown in the table; if the sign is minus, the significance indicator will be left on. The information will not appear in the result, however, if there are no further pattern characters to be scanned. As a matter of fact, if any of the source fields in the examples above had been negative, the results shown would have been exactly the same.

Suppose, however, that pattern characters remain after the sign position has been examined. The action of the significance indicator in controlling the instruction continues just as before, although the setting of the significance indicator was accomplished by a different condition. There are, of course, no more digits to move. Hence we will not want to place digit selectors in the pattern in this position,

Table 7-1. Summary of Editing Functions

CONDITIONS				RESULTS	
Pattern Character	Previous State of Significance Indicator	Source Digit	Low-Order Source Digit is a Plus Sign	Result Character	State of Significance Indicator at End of Digit Examination
Digit selector	off	0	*	fill character	off
	off	1-9	no	source digit	on
	off	1-9	yes	source digit	off
	on	0-9	no	source digit	on
	on	0-9	yes	source digit	off
Significance starter	off	0	no	fill character	on
	off	0	yes	fill character	off
	off	1-9	no	source digit	on
	off	1-9	yes	source digit	off
	on	0-9	no	source digit	on
	on	0-9	yes	source digit	off
Field separator	*	**	**	fill character	off
Message character	off	**	**	fill character	off
	on	**	**	message character	on

*No effect on result character and new state of significance indicator.
**Not applicable because source digit is not examined.

but, rather, sign indicators, such as a minus sign or CR for credit. The action taken with the characters in the pattern is the same now as it was before: they remain unchanged if the significance indicator is on, but are replaced by the fill character if the significance indicator is off.

Let us set up a suitable pattern for the example data. Let us print the letters CR for negative numbers, with one blank between the rightmost digit and the C. In hexadecimal, CR is C3 D9, so the pattern becomes

40 20 20 6B 20 20 21 4B 20 20 40 C3 D9

Figure 7-6 shows the results for sample data values as before, together with two negative values.

```
  BDD,DDS.DDBCR
  40 20 20 6B 20 20 21 4B 20 20 40 C3 D9

 1234567   12,345.67
 0120406    1,204.06
 0012345      123.45
 0001000       10.00
 0000123        1.23
 0000012         .12
 0000001         .01
 0000000         .00
-0098765      987.65 CR
-0000000         .00 CR
```

Figure 7-6. Editing results with blank fill, comma and decimal point insertion, significance starter, and CR symbol for negative numbers

If we use an asterisk now as the fill character, positive quantities will have three asterisks following the cents, as shown in Figure 7-7. This may or may not be desired. There are other ways to handle the signs, as we shall see next.

We have seen above that an amount of zero prints in the general form .00 when a significance starter is used. It may in some cases be desirable to make such an amount print as all blanks or all asterisks. This is very easily done by making use of the way the condition code is set by execution of the Edit instruction:

Code	*Instruction*
0	Result field is zero
1	Result field is less than zero
2	Result field is greater than zero

This means that after completion of the Edit we can make a simple Branch on Condition test of the condition code and move blanks or asterisks to the result field if it is zero. The movement is particularly simple because the fill character is still there in the field and an overlapped Move Characters instruction can be used as follows:

```
          BC     6,SKIP
          MVC    WORK+1(12),WORK
SKIP
```

```
  *DD,DDS.DDBCR
  5C 20 20 6B 20 20 21 4B 20 20 40 C3 D9

 1234567   *12,345.67***
 0120406   **1,204.06***
 0012345   ****123.45***
 0001000   *****10.00***
 0000123   ******1.23***
 0000012   *******.12***
 0000001   *******.01***
 0000000   *******.00***
-0098765   ****987.65 CR
-0000000   *******.00 CR
```

Figure 7-7. Same with asterisk fill

The explicit length of 12 is based on the most recent pattern, which has a total of 13 characters. The MVC, as written, picks up the leftmost character and moves it to the leftmost-plus-one position. It then picks up the leftmost-plus-one character and moves it to the leftmost-plus-two position, etc., effect propagating the leftmost character through the field. This is precisely what we want if the fill character is the one to be substituted.

Figure 7-8 shows our familiar data values with zero fields blanked, and Figure 7-9 shows them with zero fields filled with asterisks. Only the fill character differs in the two programs that would produce the results shown in Figures 7-8 and 7-9; the Edit, the Branch on Condition, and the Move Characters are the same in both cases.

```
  BDD,DDS.DDBCR
  40 20 20 6B 20 20 21 4B 20 20 40 C3 D9

 1234567   12,345.67
 0120406    1,204.06
 0012345      123.45
 0001000       10.00
 0000123        1.23
 0000012         .12
 0000001         .01
 0000000
-0098765      987.65 CR
-0000000
```

Figure 7-8. Editing results showing the blanking of zero fields by the use of two additional instructions

```
  *DD,DDS.DDBCR
  5C 20 20 6B 20 20 21 4B 20 20 40 C3 D9

 1234567   *12,345.67***
 0120406   **1,204.06***
 0012345   ****123.45***
 0001000   *****10.00***
 0000123   ******1.23***
 0000012   *******.12***
 0000001   *******.01***
 0000000   *************
-0098765   ****987.65 CR
-0000000   *************
```

Figure 7-9. Same with zero fields filled with asterisks

The condition code can also be used to distinguish between positive and negative numbers when it is necessary to present the sign in some manner that is not possible by using the automatic features of the Edit. We might, for instance, wish to test the condition code and use the results of the test to place a plus sign *or* minus sign to the left of the edited result.

The Edit instruction can be used to edit several fields with one instruction. Doing so uses a final control character, the field separator (hexadecimal 22). This character is replaced in the pattern by the fill character, and causes the significance indicator to be set to the off state. The characters following, both in the pattern and in the source data, are handled as described for a single field. In other words, it is possible to set up a pattern to edit a whole series of quantities, even an entire line, with one instruction. The packed source fields must, of course, be contiguous in storage, but this is often no inconvenience. One limitation is that the condition code, upon completion of such an instruction, gives information only about the last field encountered after a field separator.

Let us consider the example shown in Figure 7-10. Suppose that at DATA we have a sequence of three fields. The leftmost of the fields has four bytes, the next has three, and the rightmost has five bytes. The first is to be printed with commas separating groups of three digits. The values are always positive and, therefore, no sign control is desired. Zero values will be blank since we shall not use a significance starter.

The second field is to be printed with three digits to the right of the decimal point, with a significance starter to force amounts less than 1 to be printed with a zero before the decimal point. Positive quantities are to be printed without a sign, and negative quantities are to be printed with a minus sign immediately to the right of the number.

The third number is a dollar amount that could be as great as $9,999,999.99. Commas and decimal point are needed just as shown. Amounts less than $1 are to be printed with the decimal point as the leftmost character. Zero amounts are to be blanked. Signs are not to be printed.

There is to be at least one blank between the first and second edited result, and at least three between the second and third.

Let us write out the necessary pattern in shorthand form, with b standing for a blank, d for digit selector, f for field separator, s for significance starter, and other characters for themselves:

```
bd,ddd,dddfsd.ddd-fbbd,ddd,dds.dd
```

The required blank between the first and second edited result will be placed there by the replacement of the field separator with the fill character. The significance starter in the part of the pattern corresponding to the second field will give the required handling of quantities less than 1. The extra two blanks between the second and third results are provided by the blanks in the part of the pattern corresponding to the third data item. (These are not treated as new fill characters; only the leftmost character in the entire pattern is so regarded.) Notice that the total of digit selectors plus significance starters is equal to the number of digits in each field to be edited.

Instructions to do the required actions are as follows:

```
         MVC    WORK,PATTRN
         ED     WORK,DATA
         BC     6,SKIP
         MVC    WORK+30(3),WORK+18
SKIP
```

The choice of addresses in the final MVC that blanks a zero field is somewhat arbitrary. We reason that if the entire field is zero, the first three positions of it are surely blank by now; hence a three-character MVC from there to the last three positions of the field will be correct.

Figure 7-10 shows initial source data values and edited results. The packed source fields must be adjacent as shown; we address the leftmost character.

```
1234567C12345C123456789C    1,234,567 12.345    1,234,567.89

0123456C01234C012345678C      123,456  1.234      123,456.78

0010009C00123C001000000C       10,009  0.123       10,000.00

0004502C98007D000001210C        4,502 98.007-          12.10

0000800C00012C000000006C          800  0.012             .06

0000001C00001D000000001C            1  0.001-            .01

0000000C00000C000000000C               0.000
```

Figure 7-10. Examples of multiple edits. On each line the first field is a combination of three items; all three were edited with one Edit, giving the three results shown to the right. The editing pattern is shown in the text.

THE EDIT AND MARK INSTRUCTION

The Edit and Mark instruction (EDMK) makes possible the insertion of floating currency symbols. By this we mean the placement in the edited result of a dollar sign (or pound sterling symbol) in the character position immediately to the left of the first significant digit. This serves as protection against alteration, since it leaves no blank spaces. It is a somewhat more attractive way to provide protection than the asterisk fill.

The operation of the instruction is precisely the same as the Edit instruction, with one additional action. The execution of the Edit and Mark places in register 1 the address of the first significant digit. The currency symbol is needed one position to the left of the first significant digit. Consequently, we subtract one from the contents of register 1 after the execution of the Edit and Mark and place a dollar sign in that position.

There is one complication: if significance is forced by a significance starter in the pattern, nothing is done with register 1. Before going into the Edit and Mark, therefore, we place in register 1 the address of the significance starter plus one. Then, if nothing happens to register 1, we still get the dollar sign in the desired position by using the procedure described above.

Let us suppose that we are again working with a four-byte source data field, which we are to edit with a comma, a decimal point, and CR for negative numbers. Accordingly, the pattern (in shorthand form) should be

bdd,dds.ddbCR

The significance starter here is six positions to the right of the leftmost character of the pattern. The complete program to give the required editing and the floating dollar sign is as follows:

```
MVC    WORK,PATTRN
LA     1,WORK+7
EDMK   WORK,DATA
BCTR   1,0
MVI    0(1),C'$'
```

The Load Address instruction as written places in register 1 the address of the position one beyond the significance starter. If significance is forced, this address remains in register 1, but otherwise the address of the first significant digit is placed in register 1 as part of the execution of the Edit and Mark. The Branch on Count Register instruction with a second operand of zero reduces the first operand register contents by 1 and does not branch. There are, of course, other ways to subtract 1 from the contents of register 1, but this is the easiest and fastest. In the Move Immediate instruction we write an explicit displacement of zero and an explicit base register number of 1. The net effect is to move one byte of immediate data, a dollar sign, to the address specified by the base in register 1. This is the desired action.

Figure 7-11 shows the effect on sample data values. Zero fields could be blanked by methods we have already discussed.

```
  BDD,DDS.DDBCR
  40 20 20 6B 20 20 21 4B 20 20 40 C3 D9

 1234567   $12,345.67
 0120406    $1,204.06
 0012345      $123.45
 0001000       $10.00
 0000123        $1.23
 0000012         $.12
 0000001         $.01
 0000000         $.00
-0098765      $987.65 CR
-0000000         $.00 CR
```

Figure 7-11. Examples of the application of the Edit and Mark instruction to get a floating currency symbol

THE TRANSLATE INSTRUCTION

How It Works

Another powerful programming feature of System/360 is the ability, through the Translate instruction, to convert very rapidly from one coding system of eight or fewer bits to another coding system. Using a conversion table, we can convert a string of characters from one form to another at speeds that compare favorably with that of decimal addition.

Suppose that we have an input stream in which the data is in proper arrangement for processing, but is in Baudot teletypewriter code. Before System/360 can process the input, it must be converted to EBCDIC. The Baudot code is a five-bit code with shifting, which makes it the equivalent of a six-bit code. For simplicity, we will omit control characters, punctuation marks, and fractions; for our purposes, they are "invalid". As shown in Figure 7-12, our transmission receiving equipment adds two zero bits at the beginning of each character, which do not change its binary value, and converts the code signals into the equivalent binary bit patterns shown in the illustration, so that our input stream is in the necessary eight-bit bytes. It remains for us to translate this stream into the corresponding EBCDIC characters by programming.

The Translate instruction (TR) is in the SS format with two storage operands. The first operand names the leftmost byte of a field to be translated; this field may be from one to 256 bytes in length. In programming parlance, it is called the *argument*. The second operand names the start of a list, or table, that contains the characters of the code into which we wish to make the translation. The table may be 256 bytes in length or it may be shorter. It is called the *function*.

Our first step before using the Translate instruction is to construct a table like the one in Figure 7-13. Note that it is 64 bytes in length, which is the maximum length of a six-bit code ($2^6 = 64$), and provides us with one byte for every binary value that we might receive. We give our table a name, TABLE, so that we will be able to refer to its symbolic storage address regardless of where it is. To create the table, a DC statement like the following might be used. In this case, we are arbitrarily filling the unused bytes with FF's.

```
TABLE DC X'FFE3FFD640C8D5D4FFD3D9C7C9D7C3E5C5E9
            ............F2FFFFF7F1FFFF'
```

Then, assuming each record is a maximum of 80 bytes in length, we set up a storage area for the Baudot data that is to be translated:

```
RECORD  DS  CL80
```

After moving the first input item to RECORD, the only processing instruction that is necessary to convert it to EBCDIC characters is:

```
TR  RECORD,TABLE
```

The operation of this instruction is byte by byte, from left to right, until the end of the first operand field. Like the other logical instructions we have studied, TR treats all data as unstructured logical data, that is, as unsigned binary quantities. Say the first byte of RECORD is 0000 1010 (hex 0A), which is R in Baudot code. The machine action will be to go to the address TABLE+0A (that is, the byte at 0A within the table) and to replace the 0A in RECORD by the bit pattern it finds in that byte of the table. This is hex D9 or 1101 1001, which is R in EBCDIC. If the next Baudot byte is 0010 1100 (hex 2C) for the numeral 8, it will be replaced by the contents of the byte at TABLE+2C: hex F8 or 1111 1000, which is 8 in EBCDIC. If the next Baudot byte is either hex 04 or 24, a space, it will be replaced by hex 40, the EBCDIC blank, which we placed in

BIT POSITIONS 0, 1, 2, 3	HEX VALUE → ↓	BIT POSITIONS 4, 5, 6, 7															
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0		T		O	sp	H	N	M		L	R	G	I	P	C	V
0001	1	E	Z	D	B	S	Y	F	X	A	W	J		U	Q	K	
0010	2		5		9	sp						4		8	0		
0011	3	3					6				2			7	1		

Figure 7-12. Baudot teletypewriter code. This is a five-bit code that, with shifting, has the capacity of six bits, or 64 characters. Control characters, punctuation marks, and fractions have been omitted.

Symbolic address of this byte is TABLE *This is TABLE+07*

Address*	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Contents†		E3 T		D6 O	40 sp	C8 H	D5 N	D4 M		D3 L	D9 R	C7 G	C9 I	D7 P	C3 C	E5 V
Address	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
Contents	C5 E	E9 Z	C4 D	C2 B	E2 S	E8 Y	C6 F	E7 X	C1 A	E6 W	D1 J		E4 U	D8 Q	D2 K	
Address	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
Contents		F5 5		F9 9	40 sp						F4 4		F8 8	F0 0		
Address	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
Contents	F3 3					F6 6				F2 2			F7 7	F1 1		

*Location of function byte within table, given in hex.

†Contents of function byte. The actual bit configuration is shown in hex at upper left; character at lower right is EBCDIC character represented by that bit pattern.

Figure 7-13. Table for translation of Baudot code to EBCDIC. Unused bytes may be filled with FF's to test for invalid characters.

the bytes at both TABLE+04 and TABLE+24.

There are some important things to note about our translate table:

1. It contains the characters of the code into which we are translating, the function bytes.
2. It is in order, not by the binary sequence of the characters it contains, but by the binary sequence of the characters of the code from which we are translating, the argument bytes.
3. It is 64 bytes long, the length we determined was equal to the maximum number of bit combinations we might have to deal with in the Baudot code.

If we were confronted by the reverse situation and needed to translate from EBCDIC to Baudot code, using the same letters, numbers, and blanks as before, we would have to construct a 256-byte table in order to have the required indexing or referencing capacity to 256 different addresses. (EBCDIC is an eight-bit code, and $2^8 = 256$.) The table would contain the 38 characters of interest to us, but the contents of these function bytes would now have to be in the Baudot code bit configuration, and they would be in order by the sequence of the EBCDIC characters, which are now the argument bytes.

We have not so far mentioned the unused function bytes of our table in Figure 7-13. We could store blanks or zeros as constants, but a better procedure is available to us, especially when a code without validity checks, like the Baudot, is transmitted. If we fill the spaces with a single unused character, such as hex FF (1111 1111), all invalid codes received would be translated to FF. After translation of each record, it would then be very simple to scan it for invalid characters by using a Compare Logical Immediate instruction.

The Translate instruction may be used to convert any characters of no more than eight bits to any other characters, not necessarily from one standard code to another. It may be used to perform a control function, as in the program example in which we shall see the instruction at work. At first glance, this program may seem to be rather complicated, but it is simply a variation on an example of a sorting technique that we discussed earlier.

An Example

In the example we shall use the Translate instruction to accomplish a reversal of letters and digits in the collating sequence. When we compare a letter and a digit in normal EBCDIC coding the letter will always show as "smaller" than the digit. We shall assume that, for some special reason, it is necessary to arrange things so that letters sort as "larger".

It should be realized that we need to reverse the ordering of letters and digits *as complete groups*. It is therefore not possible simply to reverse the paths taken on the comparisons in the program. Consider an example. With EBCDIC coding and the Compare Logical Character instruction, this is the binary sequence, and the machine's normal collating sequence, of the following five items:

ADAMS
JONES
SMITH
12345
56789

We want to modify the sorted order to:

12345
56789
ADAMS
JONES
SMITH

If we were simply to reverse the paths taken after the comparison, the sorted order would be:

56789
12345
SMITH
JONES
ADAMS

We shall see how, using the Translate instruction, we can rearrange the letters and digits so that digits sort ahead of letters, while retaining normal numerical and alphabetical order. The translated characters will be used *only* for the sorting operation; we are not required to translate the characters into anything that would be otherwise meaningful.

The only thing we need to do in setting up the table, therefore, is to replace digits with something smaller than what we replace letters with. There are, of course, a great many ways to do this. In the program of Figure 7-14 we have chosen a scheme for its simplicity. The digits 0–9 are replaced by hexadecimal 01–10, A–I are replaced by 11–19, J–R by 21–29, and S–Z by 32–39. These replacements satisfy the one basic requirement, that digits sort earlier than letters. The scheme also preserves the ordering of the letters within the alphabet. The *particular* choices for the letters are not critical, but they will seem reasonable to someone familiar with punched cards.

The program begins with an unfamiliar operand in the PRINT instruction, the assembler instruction that controls the content of the assembly listing. The operand DATA has the effect of causing constants to be printed out in full in the listing. If DATA is omitted or NODATA is specified, only the leftmost eight bytes of any constant will be printed, no matter how large it may be. In actual programming, it is generally considered good practice to include PRINT DATA routinely–and also to let generated macro statements appear in full (by the use of GEN or the omission of NOGEN in the PRINT statement). Then the programmer will always have a complete listing for program checking and debugging. A full explanation of the PRINT instruction may be found in the assembler language reference manual.

We are assuming, for the purposes of this program, that the input stream contains nothing but letters and digits. There are only 36 of these. The other 220 positions of the table have been filled with blanks (represented by 40 in EBCDIC), which is not quite representative of what we might do in practice. In an actual application, if our data had already been run and verified and we really knew that nothing else could appear, we would use relative addressing with a minus factor to reference the table, and would not store the blanks at the beginning of the table. If, as is more likely, we were concerned about the possibility of erroneous data, we might use the full 256-byte table to check validity.

The task is to sort into the stated sequence three records of 13 characters each, using as the sort key the middle five characters of each record. In other words, the sorted records, which are named A, B, and C, are to be in sequence on their middle five characters after the execution of the program.

In the program of Figure 7-14 we begin by moving the keys to locations in which they can be translated; we do not want to destroy the actual records. The working storage areas have been named KEYA, etc. We shall see shortly why these need to be 13 characters. The three Translate instructions make the conversions of coding on the keys that we have described in detail above. The original records are not disturbed.

Now we load three general registers with the addresses of A, B, and C, that is with A(A), A(B), and A(C). It is these addresses that will be moved during the bulk of the sorting, not the records themselves. The Compare Logical that comes next must be studied carefully. The instruction says that the first operand begins 43 bytes after the address contained in register 2 and that the first operand is five bytes long. Register 2 at this point contains the *address* of A because of the Load Multiple just before this instruction. Looking at the data layout, we see that 43 bytes past the beginning of A is the beginning of the translated *key* of A. Similarly, the second operand refers to the key of B. (Only one length is required on this instruction.) We are thus asking for a comparison between the translated key of A and the translated key of B. If the key of A is already equal to or smaller than the key of B, we Branch on Not High down to X where the next comparison is made. If the key of A is larger than the key of B, we proceed in sequence to the three instructions that interchange the contents of registers 2 and 3. This means that when we arrive at X, register 2 contains the address of the smaller of the keys of A and B, whether or not there was an interchange.

In the addressing scheme described in the preceding paragraph, it is essential that there be a fixed relationship between the address of an item and the address of its translated key. In other words, the translated key of A in KEYA has to be the same distance beyond A as the translated key of B in KEYB is beyond B, and similarly with KEYC and C, so that the same displacement of 43 can be used for all three items. (This, in turn, is why KEYA, KEYB, and KEYC were made 13 characters long even though the keys are only five.) This addressing scheme is necessary because on the second and third comparisons, we will not know which keys are being compared–A, B, or C.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT  SOURCE STATEMENT

                                     1          PRINT DATA,NOGEN
000000                               2 SORTABC2 START 0
000000 05B0                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
000002 D204 B097 B070 00099 00072    5          MVC   KEYA+4(5),A+4       MOVE KEYS TO POSITION FOR TRANSLATE
000008 D204 B0A4 B07D 000A6 0007F    6          MVC   KEYB+4(5),B+4
00000E D204 B0B1 B08A 000B3 0008C    7          MVC   KEYC+4(5),C+4
000014 DC04 B097 B0ED 00099 000EF    8          TR    KEYA+4(5),TABLE     TRANSLATE KEYS TO CHANGE COLLATE SEQ
00001A DC04 B0A4 B0ED 000A6 000EF    9          TR    KEYB+4(5),TABLE
000020 DC04 B0B1 B0ED 000B3 000EF   10          TR    KEYC+4(5),TABLE
000026 9824 B0BA            000BC   11          LM    2,4,ADDRA           PUT ADDRESSES IN REGS 2, 3, 4
00002A D504 202B 302B 0002B 0002B   12          CLC   43(5,2),43(3)       COMPARE KEYA WITH KEYB
000030 47D0 B038            0003A   13          BNH   X                   BRANCH IF ALREADY IN SEQUENCE
000034 1862                         14          LR    6,2                 INTERCHANGE
000036 1823                         15          LR    2,3
000038 1836                         16          LR    3,6
00003A D504 202B 402B 0002B 0002B   17 X        CLC   43(5,2),43(4)       COMPARE SMALLER OF A AND B WITH KEYC
000040 47D0 B048            0004A   18          BNH   Y                   BRANCH IF ALREADY IN SEQUENCE
000044 1862                         19          LR    6,2                 INTERCHANGE
000046 1824                         20          LR    2,4
000048 1846                         21          LR    4,6
00004A D504 302B 402B 0002B 0002B   22 Y        CLC   43(5,3),43(4)       COMPARE TWO LARGER KEYS
000050 47D0 B058            0005A   23          BNH   MOVE                BRANCH IF ALREADY IN SEQUENCE
000054 1863                         24          LR    6,3                 INTERCHANGE
000056 1834                         25          LR    3,4
000058 1846                         26          LR    4,6
00005A D20C B0C6 2000 000C8 00000   27 MOVE     MVC   SMALL,0(2)          MOVE USING ADDRESSES IN REGISTERS
000060 D20C B0D3 3000 000D5 00000   28          MVC   MEDIUM,0(3)
000066 D20C B0E0 4000 000E2 00000   29          MVC   LARGE,0(4)
                                    30          EOJ
00006E                              33 A        DS    CL13
00007B                              34 B        DS    CL13
000088                              35 C        DS    CL13
000095                              36 KEYA     DS    CL13
0000A2                              37 KEYB     DS    CL13
0000AF                              38 KEYC     DS    CL13
0000BC 0000006E                     39 ADDRA    DC    A(A)
0000C0 0000007B                     40 ADDRB    DC    A(B)
0000C4 00000088                     41 ADDRC    DC    A(C)
0000C8                              42 SMALL    DS    CL13
0000D5                              43 MEDIUM   DS    CL13
0000E2                              44 LARGE    DS    CL13
0000EF 4040404040404040             45 TABLE    DC    CL193' '
0000F7 4040404040404040
0000FF 4040404040404040
000107 4040404040404040
00010F 4040404040404040
000117 4040404040404040
00011F 4040404040404040
000127 4040404040404040
00012F 4040404040404040
000137 4040404040404040
00013F 4040404040404040
000147 4040404040404040
00014F 4040404040404040
000157 4040404040404040
00015F 4040404040404040
000167 4040404040404040
00016F 4040404040404040
000177 4040404040404040
00017F 4040404040404040
000187 4040404040404040
00018F 4040404040404040
000197 4040404040404040
00019F 4040404040404040
0001A7 4040404040404040
0001AF 40
0001B0 1112131415161718             46          DC    X'111213141516171819'
0001B8 19
0001B9 40404040404040               47          DC    CL7' '
0001C0 2122232425262728             48          DC    X'212223242526272829'
0001C8 29
0001C9 4040404040404040             49          DC    CL8' '
0001D1 3233343536373839             50          DC    X'3233343536373839'
0001D9 404040404040                 51          DC    CL6' '
0001DF 0102030405060708             52          DC    X'01020304050607080910'
0001E7 0910
0001E9 404040404040                 53          DC    CL6' '
000000                              54          END   BEGIN
```

Figure 7-14. A program to sort three fields named A, B, and C into ascending sequence on file-character keys in each field. The Translate instruction is used to make digits sort ahead of letters.

We now carry out the same actions using the addresses in registers 2 and 4, thus comparing the smaller of KEYA and KEYB with KEYC. The two addresses are interchanged if necessary, to make the address in register 2 that of the smaller. After this sequence of instructions, therefore, we can be positive that register 2 contains the address of the smallest of the translated keys. The same set of actions on registers 3 and 4 gets them in proper sequence.

Now we know that whatever rearrangements may or may not have been carried out, register 2 contains the address of the smallest of the keys, register 3 the address of the middle-sized, and register 4 the address of the largest. We can therefore proceed to the three instructions that place the proper three records in SMALL, MEDIUM, and LARGE. For instance, the first of these instructions, the one at MOVE, says to move 13 characters from the address given in register 2, whatever it may be, to SMALL. The other two instructions do the same with registers 3 and 4.

Figure 7-15 shows the contents of registers 2, 3, and 4 at four points during the execution of the program: at the beginning, at X, at Y, and at MOVE. The three actual data items, in order, were:

1111SMITH1111
2222ADAMS2222
3333567893333

In other words, the original items were in reverse order to the sequencing pattern we wanted.

AFTER EXECUTION OF	REG 2	REG 3	REG 4
STATEMENT 11	0000206E	0000207B	00002088
STATEMENT 16	0000207B	0000206E	00002088
STATEMENT 21	00002088	0000206E	0000207B
STATEMENT 26	00002088	0000207B	0000206E

Figure 7-15. The contents of registers 2, 3, and 4 during execution of the program in Figure 7-14, loaded at 2000

THE TRANSLATE AND TEST INSTRUCTION AND THE EXECUTE INSTRUCTION

The Translate and Test instruction (TRT) adds great power to the processing capability of System/360. It is related to the Translate instruction and has the same format, but is very different in operation. It is used to scan a data field for characters with a special meaning. Since it merits our close attention, we shall study it in the three remaining programs in this chapter.

As with Translate, we work with a table as the second operand that is accessed exactly the same way. That is, a first operand argument byte addresses a particular entry in the table by an address computation. Once again the table must be in order by the binary sequence of the code of the source material, which in this and the following sections will be standard EBCDIC input. This time, however, we must put zeros in the table to indicate characters without any special meaning and some nonzero value for each character with a special meaning.

In further contrast to the Translate instruction, there is no change in the argument bytes as a result of the TRT operation, despite the "translate" in its name. Instead, the argument bytes are merely inspected, byte by byte, from left to right. If the first argument byte references a function byte that is zero, the next argument byte is inspected, and so forth. If all the function bytes that are referenced are zero, the condition code is set to zero and the operation is complete. However, if a nonzero function byte is referenced, the contents of that byte are placed by the machine in register 2 and the address of the *argument* byte is placed in register 1. The condition code is set to 1 or 2, and the operation is terminated. A condition code of 1 indicates that there are more argument bytes to inspect, a condition code of 2 that the nonzero function byte is at the end of the field. The programmer may then make use of the information in the registers and in the condition code.

This means that we can inspect a complete stream of argument bytes, looking for whatever interests us: error characters, end-of-message codes, blanks and commas that separate parts of a line, or whatever. The following problem shows one way to use the instruction.

We are given the starting address of a string of characters of unknown length. The string contains an unknown number of names and addresses. Each name is of unknown length; each address component is of unknown length; there may be from one to four lines of address; we do not know how many names and addresses there are. All we do know is that after each "line" of information there is a dollar sign ($), after the last line of an address there are two dollar signs ($$), and at the end of the entire string there is a dollar sign followed by an asterisk ($*). We are required to set up each name and address in four lines named LINE1, LINE2, LINE3, and LINE4. Any unused lines must be blanked. When an address has been assembled in this manner, it is to be printed, after which we return to set up and print the next address.

The table required for this application must be 256 bytes in length in order to reference the complete range of EBCDIC binary values. It will consist of 254 zeros, with entries only in positions 5B (91) and 5C (92), corresponding to dollar sign and asterisk respectively. For the dollar sign we have chosen to enter 01 and for asterisk 02. These choices are highly arbitrary; as we shall see, any other two numbers would be just as good. All we need to know about the input stream is where the dollar signs and asterisks appear; we care nothing about any other characters.

The program in Figure 7-16 begins by placing in register 3 the address of the first character of the input stream that we shall break into names and addresses. On the assumption that there is only one such stream to process, this instruction is never repeated in this program. The next instruction is returned to each time another name and address is to be processed. It places a 4 in register 9 to be used as a guard against incorrect input streams; if ever a name and address would seem to require more than the four lines we have allotted, the program will stop. The next Load Address places in register 10 the address of the first line of the output. The next two instructions are overlapping Move Characters that clear to blanks the output areas. With assumed line lengths of 120 characters, this makes 480 bytes to clear. Since the maximum length in a Move Characters is 256 bytes, two instructions are needed, each clearing two lines. The first MVC instruction clears to blanks the first two lines and the first position of the third line. The second MVC instruction uses the blank now in the first position of the third line to blank the remaining positions of the third line and all of the fourth line.

Now we come to the Translate and Test. The first operand starts at the address in register 3, which we set up with the starting address of the input stream; it is stated to be a maximum of 120 characters in length. The second operand address names the table. If the input stream is correct, a dollar sign will be found within 120 characters. If, because of an error, there is no end-of-line dollar sign, we will have a condition code of zero at the completion of the execution of the instruction. A Branch on Zero, accordingly, takes us to an error exit (this could also be written as BC 8,ERROR).

In the normal case of finding a dollar sign to indicate the end of the first line, what do we have in the registers? Register 1 contains the address of the dollar sign that stopped the Translate and Test. We wish to do a little arithmetic on this address without destroying it, so we move it to register 4. Now we subtract from the address of the dollar sign the address of the first character of the line. The difference is the length of the line, in bytes. We are about

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1          PRINT DATA,NOGEN
000000                               2 MAILLIST START 0
000000 05B0                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
000002 1B22                          5          SR    2,2                     CLEAR REG FOR LATER COMPARE INSTR
000004 4130 B23D             0023F   6          LA    3,NAME                  PUT STARTING ADDR OF RECORD IN REG
000008 4190 0004             00004   7 AGAIN    LA    9,4                     FOR ERROR CHECKING
00000C 41A0 B05D             0005F   8          LA    10,LINE1                INITIALIZE TO START OF FIRST LINE
000010 D2F0 B05D B05C 0005F 0005E    9          MVC   LINE1(241),BLANK        BLANK LINES 1 & 2, 1ST POS LINE 3
000016 D2EE B14E B14D 00150 0014F   10          MVC   LINE3+1(239),LINE3      BLANK BAL LINE 3 & LINE 4
00001C DD77 3000 B30A 00000 0030C   11 LOOP     TRT   0(120,3),TABLE          SCAN RECORD FOR DELIMITER
000022 4780 B054             00056  12          BZ    ERROR                   BRANCH IF NO DELIMITER IN 120 CHARS
000026 1841                         13          LR    4,1                     GET LENGTH CODE OF LINE
000028 1B13                         14          SR    1,3
00002A 5B10 B302             00304  15          S     1,ONE
00002E 4740 B044             00046  16          BM    OUT                     BRANCH IF 2 DELIMITERS IN SEQUENCE
000032 4410 B056             00058  17          EX    1,MVCINS                MOVE LINE TO PRINTING POSITION
000036 4130 4001             00001  18          LA    3,1(0,4)                SET UP NEXT TRT
00003A 41AA 0078             00078  19          LA    10,120(10)              TO GET NEXT LINE
00003E 4690 B01A             0001C  20          BCT   9,LOOP                  BRANCH UNLESS FIFTH LINE
000042 47F0 B054             00056  21          B     ERROR                   MORE THAN 4 LINES
000046 0700                         22 OUT      NOPR  0                       THE PRINT ROUTINE WOULD START HERE
                                    23 *        *
                                    24 *        *
                                    25 *        *
000048 4130 4001             00001  26          LA    3,1(0,4)                SET UP FOR NEXT NAME & ADDRESS
00004C 5920 B306             00308  27          C     2,ENDCON                SEE IF DELIMITER WAS AN ASTERISK
000050 4770 B006             00008  28          BNE   AGAIN                   BRANCH IF NOT
                                    29          EOJ                           ALL FINISHED IF HERE
                                    32 ERROR    EOJ                           ERROR STOP
000058 D200 A000 3000 00000 00000   35 MVCINS   MVC   0(0,10),0(3)            EX INSTR ADDS LENGTH FROM REG 4
00005E 40                           36 BLANK    DC    CL1' '
00005F                              37 LINE1    DS    CL120
0000D7                              38 LINE2    DS    CL120
00014F                              39 LINE3    DS    CL120
0001C7                              40 LINE4    DS    CL120
                                    41 NAME     DC    C'SMITH$DETROIT$$J. C. JACKSON$1234 MAIN STREET$CHICAGO,X
00023F E2D4C9E3C85BC4C5                                ILLINOIS$$F. C. R. ANDERSON$553 MAPLE PLACE APARTMENT '
000247 E3D9D6C9E35B5BD1
00024F 4B40C34B40D1C1C3
000257 D2E2D6D55BF1F2F3
00025F F440D4C1C9D540E2
000267 E3D9C5C5E35BC3C8
00026F C9C3C1C7D66B40C9
000277 D3D3C9D5D6C9E25B
00027F 5BC64B40C34B40D9
000287 4B40C1D5C4C5D9E2
00028F D6D55BF5F5F340D4
000297 C1D7D3C540D7D3C1
00029F C3C540C1D7C1D9E3
0002A7 D4C5D5E340
                                    42          DC    C'5C$WHITE PLAINS, NEW YORK$$D. D. ADAMS AND FAMILY$505 X
0002AC F5C35BE6C8C9E3C5                               GRATHSON$APT. 31$READING, PENN.$*'
0002B4 40D7D3C1C9D5E26B
0002BC 40D5C5E640E8D6D9
0002C4 D25B5BC44B40C44B
0002CC 40C1C4C1D4E240C1
0002D4 D5C440C6C1D4C9D3
0002DC E85BF5F0F540C7D9
0002E4 C1E3C8E2D6D55BC1
0002EC D7E34B40F3F15BD9
0002F4 C5C1C4C9D5C76B40
0002FC D7C5D5D54B5B5C
000303 00
000304 00000001                     43 ONE      DC    F'1'
000308 00000002                     44 ENDCON   DC    F'2'
00030C 0000000000000000             45 TABLE    DC    91X'00'
000314 0000000000000000
00031C 0000000000000000
000324 0000000000000000
00032C 0000000000000000
000334 0000000000000000
00033C 0000000000000000
000344 0000000000000000
00034C 0000000000000000
000354 0000000000000000
00035C 0000000000000000
000364 000000
000367 0102                         46          DC    X'0102'
000369 0000000000000000             47          DC    163X'00'
000371 0000000000000000
000379 0000000000000000
000381 0000000000000000

0003F9 0000000000000000
000401 0000000000000000
000409 000000
000000                              48          END   BEGIN
```

Figure 7-16. A program to print names and addresses. The input stream contains an unknown number of names and addresses, each name and address contains a variable number of lines, and each line is of variable length.

ready to execute a Move Characters instruction in which we will use this computed address; but in the instruction itself the length *code* is always one less than the actual length. So we now subtract 1 from the difference residing in register 1.

What would it mean if this difference were now negative? We shall see, in further analysis of the program, that it would indicate the double dollar sign that denotes the end of a name and address. We therefore Branch on Minus (or BC 4) to OUT, where we would normally process the completed name and address.

Let us review the status of things. We have in register 3 the starting address of a group of characters that should be moved; in register 10 we have the address to which they should be moved; in register 1 we have the correct length code for a Move Characters instruction. We need either to place that length code in an instruction – or do something equivalent. "Something equivalent" is precisely what the Execute (EX) instruction provides. We say

```
EX   1,MVCINS
```

This means to execute the instruction at the second operand address named (MVCINS), after Or-ing together the last eight bits of register 1 and the length code portion of MVCINS. Looking down at MVCINS we see that a Move Characters instruction has been set up to do all the things just outlined as necessary, with the exception of the length. The instruction set up at MVCINS says to move a group of bytes starting at the address given in register 3 to another location given by the address in register 10. Both displacements are zero, because the base addresses are exactly what are wanted. The length code is zero in the instruction; the actual length is supplied by the last eight bits of register 1. One line of the complete name and address is thus moved to a printing position.

The Execute instruction is a very serviceable tool in the hands of a resourceful programmer, especially when it is used in a loop that deals with varying conditions. It is an unusual branching instruction that causes one instruction anywhere in a program to be executed out of sequence. Then, unless the remote instruction itself happens to be a successful branch, the program continues with the next instruction after the Execute. As we have seen, Execute can actually modify the remote instruction before execution. It can specify length codes, immediate data, register operands, or whatever information goes into the second byte in the format of the remote instruction. It does this by Or-ing with the last eight bits of a register, which the programmer may use to store information, do arithmetic, or whatever. We will see further examples of the Execute instruction in the next two programs.

We are now about ready to go back for another look at the input stream. To do that, register 3 must contain the address of the next valid data character in the stream. Register 4 contains almost what we need; it has the address of the dollar sign just prior to the next valid character. We accordingly use a Load Address instruction to get the desired address into register 3. The instruction operates as follows. The displacement of one is added to the contents of the base register to get an effective address. (If an index register had been specified, its contents would also have been added in.) This address is then placed in register 3, with no actual reference to storage. It would have been legitimate to place the sum back in register 4, if that had been desired. Load Address provides a fast and simple way to add a small positive amount to a register.

In the next Load Address instruction we see register 10 being incremented by 120 by use of the method just described. The purpose is to set up the next line as the destination the next time around the loop. Finally we Branch on Count back to inspect the input stream again. If this would mean trying for a fifth line, the branch is not taken and we reach the error exit.

At OUT, which we reach on discovering either two dollar signs in sequence or a dollar sign followed by an asterisk, we would normally include a series of instructions to print the output. Since input/output operations are outside the scope of this book, we simply indicate by a No-Operation instruction that this action would occur here in the program. NOPR is an extended mnemonic for Branch on Condition with a mask of zero, which never causes a branch to occur.

Following the output operations we are ready to go back for another name and address, unless this was the last one in the stream. Whether that was the case can be determined by looking at the function byte in register 2 to see whether it is that produced by a dollar sign or by an asterisk, that is, a 1 or a 2 respectively. A comparison with ENDCON, which contains a 2 in proper form for a comparison with a fullword register, makes the determination. If the function byte is not that produced from an asterisk, we Branch on Not Equal back to AGAIN to repeat the whole process. Otherwise we reach the normal exit from the program.

Figure 7-17 shows successive groups of output, based on the input stream assembled with the program.

```
SMITH
DETROIT

J. C. JACKSON
1234 MAIN STREET
CHICAGO, ILLINOIS

F. C. R. ANDERSON
553 MAPLE PLACE APARTMENT 5C
WHITE PLAINS, NEW YORK

D. D. ADAMS AND FAMILY
505 GRATHSON
APT. 31
READING, PENN.
```

Figure 7-17. Four names and addresses produced by the program in Figure 7-16

AN ASSEMBLER APPLICATION OF TRANSLATE AND TEST AND EXECUTE

Another example of the powerful combination provided by the Translate and Test instruction with the Execute instruction is provided by a simplified version of part of the work an assembler must do.

We are given an input stream consisting of one type of operand field in an assembler language program. The field that we shall process will always consist of two operands: the first will be a general register, the second a symbolic address of not more than six letters. Relative addressing with either an increment or a decrement may or may not be included in the second operand. Accordingly, our field will start with one or two decimal digits, a comma, and from one to six letters. After the final letter there will be either: (1) a blank, or (2) a plus or a minus sign followed by from one to four decimal digits and a blank.

We are required to place the register number in REG as a binary number, to place the symbol in SYMBOL, and to place in INCDEC the increment or decrement as a properly signed binary number.

We are, of course, defining away a great deal of the actual work of an assembler program, which must sort out many different kinds of instructions and operands, and errors too.

The task of the Translate and Test Instruction this time will be to detect the "delimiters" that separate one part of the operand field from another. The delimiters in the job as we have defined it are the comma, the plus sign or the minus sign, and the blank. These set off register from symbol, symbol from increment or decrement, and mark the end of the address. We will need a translate table with entries in the positions corresponding to these four delimiters.

The input stream begins at symbolic location COL16, a name chosen to suggest where the operand field might begin on a card, although we realize that, in the System/360 assembler language, it is not *required* to begin there.

The program of Figure 7-18 begins by clearing to blanks the location set up for the symbol. This must be done because we do not know whether the symbols we shall find will always have six characters; therefore, any previous contents of SYMBOL must be erased. A similar consideration applies to INCDEC. There may or may not be an increment or decrement, hence we are required to place zero there. It seems to be a little easier to clear INCDEC at the beginning and then to leave it zero, if nothing is placed there, rather than to clear it later if necessary. REG need not be cleared; we will always place something there.

This time we construct the function table by entering a constant of 256 bytes of zeros in storage, and use the Move Immediate instruction to insert arbitrary values in the EBCDIC positions corresponding to the four delimiters. To find the correct positions, we need only read off the hexadecimal values from an EBCDIC chart. For the delimiters, a value of 1 is used for a blank, 2 for a comma, 3 for a plus sign, and 4 for a minus sign. When the program is executed, these values will be moved into position in place of zeros in TABLE, which will then be in storage, and of course the values will be in the specified bytes before execution of the TRT instructions.

Following a procedure somewhat similar to that used in the name and address program of the preceding section, we now place in register 3 the address of the leftmost character of the stream. A Translate and Test will stop after two or three characters, depending on whether the register number has one or two digits. We now compute in register 4 the proper length code, either zero or 1, and use an Execute to carry out a Pack instruction that is stored at PCKINS. This remote PACK takes its second operand from the address given in register 3, its length from register 4, and places the result in WORK. The latter was set up as a doubleword, so we may now do a Convert to Binary, placing the result in register 5 from whence we store it in REG. The first required action is complete.

We are now ready to get the symbol, after some preliminaries. When we have found the delimiter after the symbol (a blank, a plus, or a minus), it will be necessary to compute the length of the symbol. In order to be able to do this later, we need now to put in register 3 the address of the first character of the symbol. This can be done with a Load Address instruction using register 1 as a base and a displacement of 1. The same scheme (base register 1 and displacement of 1) gives the correct starting address for the Translate and Test instruction also.

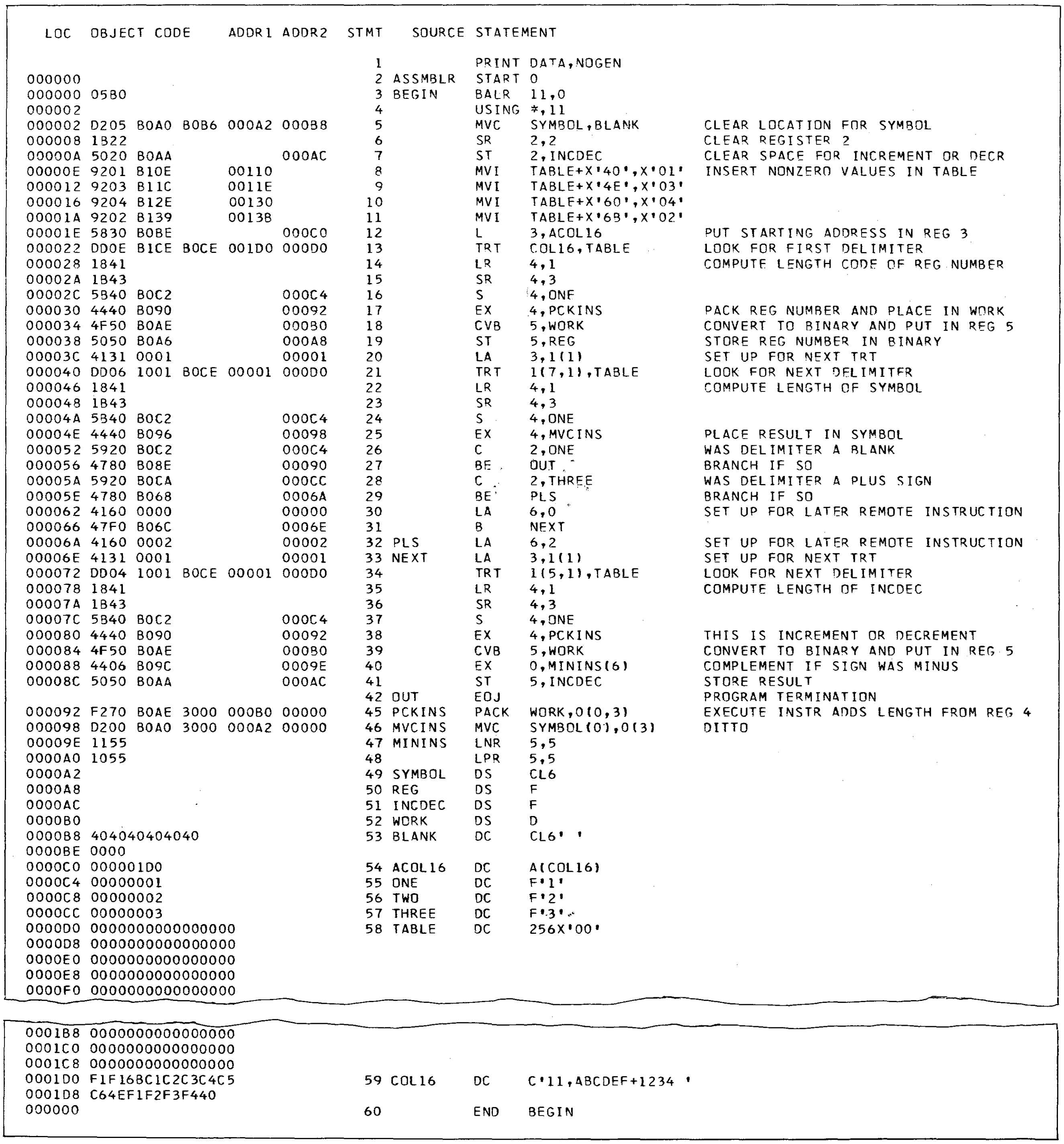
Once again, after completing the Translate and Test, we compute the length of the symbol and use an Execute, this time to move the symbol from its position in the input stream to SYMBOL. When this has been done, we inspect the delimiter. If it is a blank, signified by a function byte of 1 in the TABLE, we are finished because there is no increment or decrement.

If it is not a blank, then it must be either a plus or a minus, always assuming for this example that there are no errors. If it is a plus, we place a 2 in register 6; otherwise a zero. The purpose of this will become clear in a moment.

At NEXT we once again place the address of the next character in the stream in 3, this time to be able to compute the length of the increment or decrement. The next six instructions are much as they were before, resulting in the value of the increment or decrement being placed in register 5 in binary. It will be positive; the sign was not included.

Now we come to an Execute instruction used in a rather different way for a rather different purpose. We have specified register zero for the Or-ing, which means that the executed instruction is not modified. Then we have indexed the address of the instruction to be modified. We will therefore execute either the instruction at MININS, if register 6 contains a zero, or the instruction two bytes later, if register 6 contains 2. The net effect is to do nothing to register 5 if the sign is plus, and to make register 5 negative if the sign is minus.

Having done this, we store the contents of register 5 at INCDEC and our assigned task is completed; we have placed various parts of the operand in separate locations where they can be separately addressed. In the real world of an assembler, many more operations would have to be performed on this operand. Our small task of separating the various parts of the operand would facilitate these further operations.



```
  LOC  OBJECT CODE    ADDR1 ADDR2 STMT   SOURCE STATEMENT

                                     1          PRINT DATA,NOGEN
000000                               2 ASSMBLR  START 0
000000 0580                          3 BEGIN    BALR  11,0
000002                               4          USING *,11
000002 D205 B0A0 B0B6 000A2 000B8    5          MVC   SYMBOL,BLANK        CLEAR LOCATION FOR SYMBOL
000008 1B22                          6          SR    2,2                 CLEAR REGISTER 2
00000A 5020 B0AA            000AC    7          ST    2,INCDEC            CLEAR SPACE FOR INCREMENT OR DECR
00000E 9201 B10E      00110          8          MVI   TABLE+X'40',X'01'   INSERT NONZERO VALUES IN TABLE
000012 9203 B11C      0011E          9          MVI   TABLE+X'4E',X'03'
000016 9204 B12E      00130         10          MVI   TABLE+X'60',X'04'
00001A 9202 B139      0013B         11          MVI   TABLE+X'6B',X'02'
00001E 5830 B0BE            000C0   12          L     3,ACOL16            PUT STARTING ADDRESS IN REG 3
000022 DD0E B1CE B0CE 001D0 000D0   13          TRT   COL16,TABLE         LOOK FOR FIRST DELIMITER
000028 1841                         14          LR    4,1                 COMPUTE LENGTH CODE OF REG NUMBER
00002A 1B43                         15          SR    4,3
00002C 5B40 B0C2            000C4   16          S     4,ONE
000030 4440 B090            00092   17          EX    4,PCKINS            PACK REG NUMBER AND PLACE IN WORK
000034 4F50 B0AE            000B0   18          CVB   5,WORK              CONVERT TO BINARY AND PUT IN REG 5
000038 5050 B0A6            000A8   19          ST    5,REG               STORE REG NUMBER IN BINARY
00003C 4131 0001            00001   20          LA    3,1(1)              SET UP FOR NEXT TRT
000040 DD06 1001 B0CE 00001 000D0   21          TRT   1(7,1),TABLE        LOOK FOR NEXT DELIMITER
000046 1841                         22          LR    4,1                 COMPUTE LENGTH OF SYMBOL
000048 1B43                         23          SR    4,3
00004A 5B40 B0C2            000C4   24          S     4,ONE
00004E 4440 B096            00098   25          EX    4,MVCINS            PLACE RESULT IN SYMBOL
000052 5920 B0C2            000C4   26          C     2,ONE               WAS DELIMITER A BLANK
000056 4780 B08E            00090   27          BE    OUT                 BRANCH IF SO
00005A 5920 B0CA            000CC   28          C     2,THREE             WAS DELIMITER A PLUS SIGN
00005E 4780 B068            0006A   29          BE    PLS                 BRANCH IF SO
000062 4160 0000            00000   30          LA    6,0                 SET UP FOR LATER REMOTE INSTRUCTION
000066 47F0 B06C            0006E   31          B     NEXT
00006A 4160 0002            00002   32 PLS      LA    6,2                 SET UP FOR LATER REMOTE INSTRUCTION
00006E 4131 0001            00001   33 NEXT     LA    3,1(1)              SET UP FOR NEXT TRT
000072 DD04 1001 B0CE 00001 000D0   34          TRT   1(5,1),TABLE        LOOK FOR NEXT DELIMITER
000078 1841                         35          LR    4,1                 COMPUTE LENGTH OF INCDEC
00007A 1B43                         36          SR    4,3
00007C 5B40 B0C2            000C4   37          S     4,ONE
000080 4440 B090            00092   38          EX    4,PCKINS            THIS IS INCREMENT OR DECREMENT
000084 4F50 B0AE            000B0   39          CVB   5,WORK              CONVERT TO BINARY AND PUT IN REG 5
000088 4406 B09C            0009E   40          EX    0,MININS(6)         COMPLEMENT IF SIGN WAS MINUS
00008C 5050 B0AA            000AC   41          ST    5,INCDEC            STORE RESULT
                                    42 OUT      EOJ                       PROGRAM TERMINATION
000092 F270 B0AE 3000 000B0 00000   45 PCKINS   PACK  WORK,0(0,3)         EXECUTE INSTR ADDS LENGTH FROM REG 4
000098 D200 B0A0 3000 000A2 00000   46 MVCINS   MVC   SYMBOL(0),0(3)      DITTO
00009E 1155                         47 MININS   LNR   5,5
0000A0 1055                         48          LPR   5,5
0000A2                              49 SYMBOL   DS    CL6
0000A8                              50 REG      DS    F
0000AC                              51 INCDEC   DS    F
0000B0                              52 WORK     DS    D
0000B8 404040404040                 53 BLANK    DC    CL6' '
0000BE 0000
0000C0 000001D0                     54 ACOL16   DC    A(COL16)
0000C4 00000001                     55 ONE      DC    F'1'
0000C8 00000002                     56 TWO      DC    F'2'
0000CC 00000003                     57 THREE    DC    F'3'
0000D0 0000000000000000             58 TABLE    DC    256X'00'
0000D8 0000000000000000
0000E0 0000000000000000
0000E8 0000000000000000
0000F0 0000000000000000

0001B8 0000000000000000
0001C0 0000000000000000
0001C8 0000000000000000
0001D0 F1F16BC1C2C3C4C5             59 COL16    DC    C'11,ABCDEF+1234 '
0001D8 C64EF1F2F3F440
000000                              60          END   BEGIN
```

Figure 7-18. A program to break down the operands of an assembler language instruction into its constituent parts, using TRT and EX

PROCESSING VARIABLE-LENGTH BLOCKED RECORDS

The following illustrative program applies techniques that are highly useful in certain commerical applications, and that the features of System/360 make particularly easy to accomplish. The task is the processing of blocked tape records (that is, many logical records in one physical block) with a variable number of records per block and with variable-length records. We shall take a record layout, furthermore, that places certain fixed-length items after the variable-length portion of the record.

Each record in a block to be processed by the program of this example will contain four fields, with characteristics as follows:

Field	*Length*	*Type*
DESC	variable, at most 60 characters	alphameric
ACCT	7 characters	alphameric
QOH	4 bytes	binary
DOLL	4 bytes	binary

The first field is a variable-length description of a stock item; it is alphameric and at most 60 characters. The next field is an account number, of exactly seven alphameric characters. The third field is four bytes long. It is a binary number giving the quantity on hand. The fourth and last field is also a four-byte binary number giving the year-to-date sales of the stock item to the nearest dollar. However long the description may be, its final character is always an equal sign to serve as a sentinel marking the end of the variable-length portion of the record. There is an unknown number of records. Immediately following the last record is another equal sign, which is the last character in the block.

We are required to process such a block, which we assume has already been read into core storage. We are to set up a line for printing that contains the account number, the quantity on hand, the sales, and the description, in that order. The numeric quantities are to be in zoned format. After printing a line for each record in the block, we are to print the total dollar sales from all records on a separate line.

The program is shown in Figure 7-19. After the usual preliminaries we clear register 4 and store the resulting zero in TOTAL in order to be sure that the accumulator for total sales is zeroed. Register 7 is next loaded with the address of the first character of the block; register 7 will always contain the address of the first character of the *next* record as the loop is repeated. The MVI instruction inserts a one in the equal sign position of our translate table. This will occur during execution, of course.

In the body of the loop we first blank out the space assigned to the description because, in general, it will be possible for a long description to be followed by a short one; without a prior blanking, the end of the previous line would still be there. The MVC instruction used here will blank out the DESC area for its entire 60-byte implied length *provided* that the first operand DESC in storage is one byte to the right of the second operand BLANK. Checking statements 50 and 51 of the assembly listing, we see that this is so.

The Translate and Test instruction references a table in which the only nonzero entry corresponds to an equal sign. The effective address of the first operand in the Translate and Test is just the contents of register 7 because the explicit displacement is zero. The length of 60 sets a limit on the search for an equal sign. If no equal sign is found within 60 bytes, the condition code will be zero; a Branch on Condition transfers to an error routine if this happens.

We now are ready to move the description from its place in the block to the space from which it will be printed. This can be done readily enough once we have available the length code of the description. Register 1 after the Translate and Test contains the address of the equal sign. Subtracting from this address the address of the first byte of the description gives the length of the description in bytes; one less than this number is the length code of the description. With this number in register 3, we can Execute a remote Move Characters instruction that moves the description from the block storage area to a location from which it can be printed.

Just before doing so, however, we have a Branch on Minus instruction to detect a negative number after the computation of the length code of the description; this would happen only if the first character of the "description" were an equal sign, which would signal the end of the block.

Getting the account number from the block area to the printing location is an easy matter. We know that the account number begins one byte beyond the address of the equal sign, which is contained in register 1. The effective address of the account number is therefore just register 1 as a base with a 1 for displacement. The address of the quantity on hand is just eight bytes beyond the address in register 1. Here we must be careful of word boundaries. The quantity on hand was said to be a four-byte binary number, but, because of the variable length of the description, it may not be aligned on a word boundary in the block storage area. We therefore use a Move Characters instruction to move it to a temporary storage area that is definitely aligned on a word boundary. TEMP1 is on a word boundary because the DS says so.

Now this binary quantity can be loaded into a register and converted to decimal in a doubleword. From here it is unpacked to the location from which it will be printed, named QOH.

The same sequence of operations gets the year-to-date sales into DOLL. Because the sales are still in register 4 in binary, they can be added to the total for the block.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      1          PRINT DATA,NOGEN
000000                                2 VARBLK   START 0
000000 0580                           3 BEGIN    BALR  11,0
000002                                4          USING *,11
000002 1B44                           5          SR    4,4                 CLEAR REGISTER 4 TO ZEROS
000004 5040 B0EE            000F0     6          ST    4,TOTAL             PUT ZEROS IN TOTAL
000008 5870 B07A            0007C     7          L     7,AFIRST            PUT ADDRESS OF 1ST RECORD IN REG 7
00000C 1B11                           8          SR    1,1                 CLEAR REGISTER 1
00000E 9201 B1E6      001E8           9          MVI   TABLE+X'7E',X'01'   PUT 1 IN EQ SIGN POSITION OF TABLE
000012 D23B B0A3 B0A2 000A5 000A4    10 AGAIN    MVC   DESC,BLANK          START OF RECORD LOOP
000018 DD3B 7000 B168 00000 0016A    11          TRT   0(60,7),TABLE       LOOK FOR SENTINEL
00001E 4780 B070            00072    12          BC    8,ERROR             NO DELIMITER FOUND IN 60 CHARACTERS
000022 1831                          13          LR    3,1                 COMPUTE LENGTH CODE OF DESCRIPTION
000024 1B37                          14          SR    3,7
000026 5B30 B07E            00080    15          S     3,ONE
00002A 4740 B06E            00070    16          BM    OUT                 BRANCH IF EQ SIGN IS 1ST CHARACTER
00002E 4430 B072            00074    17          EX    3,MVCINS            MOVE DESCRIPTION FOR PRINTING
000032 D206 B082 1001 00084 00001    18          MVC   ACCT,1(1)           MOVE ACCOUNT NUMBER
000038 D203 B0E2 1008 000E4 00008    19          MVC   TEMP1,8(1)          MOVE QOH TO TEMPORARY STORAGE AREA
00003E 5840 B0E2            000E4    20          L     4,TEMP1             QOH TO REGISTER FOR PROCESSING
000042 4E40 B0E6            000E8    21          CVD   4,TEMP2             CONVERT TO DECIMAL
000046 F377 B08C B0E6 0008E 000E8    22          UNPK  QOH,TEMP2           UNPACK AND MOVE FOR PRINTING
00004C D203 B0E2 100C 000E4 0000C    23          MVC   TEMP1,12(1)         SAME PROCESSING FOR DOLLARS
000052 5840 B0E2            000E4    24          L     4,TEMP1
000056 4E40 B0E6            000E8    25          CVD   4,TEMP2
00005A F377 B097 B0E6 00099 000E8    26          UNPK  DOLL,TEMP2
000060 5A40 B0EE            000F0    27          A     4,TOTAL             ADD DOLLARS TO TOTAL
000064 5040 B0EE            000F0    28          ST    4,TOTAL
000068 4171 0010            00010    29          LA    7,16(1)             PUT ADDRESS OF NEXT RECORD IN REG 7
                                     30 *        *                         PRINT ROUTINE WLD BE INCLUDED HERE
                                     31 *        *
                                     32 *        *
                                     33 *        *
00006C 47F0 B010            00012    34          B     AGAIN               GO BACK FOR NEXT RECORD
                                     35 OUT      EOJ                       NORMAL END OF JOB
                                     38 ERROR    EOJ                       ERROR TERMINATION
000074 D200 B0A3 7000 000A5 00000    41 MVCINS   MVC   DESC(0),0(7)        EXECUTE INSTR ADDS LENGTH FROM REG 3
00007A 0000
00007C 000000F4                      42 AFIRST   DC    A(RECORD)
000080 00000001                      43 ONE      DC    F'1'
000084                               44 ACCT     DS    CL7
00008B 404040                        45          DC    CL3' '
00008E                               46 QOH      DS    CL8
000096 404040                        47          DC    CL3' '
000099                               48 DOLL     DS    CL8
0000A1 404040                        49          DC    CL3' '
0000A4 40                            50 BLANK    DC    C' '
0000A5                               51 DESC     DS    CL60
0000E4                               52 TEMP1    DS    F
0000E8                               53 TEMP2    DS    D
0000F0                               54 TOTAL    DS    F
0000F4 C2C5E5C5D36B40C2              55 RECORD   DC    C'BEVEL, BLUE, 6 INCH='
0000FC D3E4C56B40F640C9
000104 D5C3C87E
000108 F1F2F3C1C2C3F4                56          DC    C'123ABC4'
00010F 000001CA                      57          DC    FL4'458'
000113 000015CA                      58          DC    FL4'5578'
000117 C1D5C7D3C56B40D9              59          DC    C'ANGLE, RED, 8 INCH FORGED='
00011F C5C46B40F840C9D5
000127 C3C840C6D6D9C7C5
00012F C47E
000131 F2F3F4E7E8E9F7                60          DC    C'234XYZ7'
000138 00001F40                      61          DC    FL4'8000'
00013C 000125C0                      62          DC    FL4'75200'
000140 C6D3C1D5C7C56B40              63          DC    C'FLANGE, 2 INCH, MAGNESIUM='
000148 F240C9D5C3C86B40
000150 D4C1C7D5C5E2C9E4
000158 D47E
00015A F7F5F3C7C8D1F8                64          DC    C'753GHJ8'
000161 0000000C                      65          DC    FL4'12'
000165 00001EB0                      66          DC    FL4'7856'
000169 7E                            67          DC    C'='
00016A 0000000000000000              68 TABLE    DC    256X'00'
000172 0000000000000000
00017A 0000000000000000
000182 0000000000000000
00018A 0000000000000000

000242 0000000000000000
00024A 0000000000000000
000252 0000000000000000
00025A 0000000000000000
000262 0000000000000000
000000                               69          END   BEGIN
```

Figure 7-19. A program to prepare for printing a series of variable-length blocked records, each consisting of four fields. Total dollar sales are computed at the same time.

This completes the actions needed to make our first line of information ready for printing, and we would normally include a printer output routine at this point. There may still be another record in the block, so we branch back to AGAIN to see whether there is. During execution, the program will continue to go through the loop each time there is another record. After the final record, the equal sign delimiter that follows it will produce a result of −1 for the length-code computation, and this will cause the program to branch (on the Branch on Minus instruction) to our EOJ macro at OUT.

The sample block that appears at RECORD involves a little bit of trickery. One of the essential aspects of the assignment is that the binary fields appear in the block not aligned on word boundaries. In real life such a block would have been set up by a previous program. Here, in attempting to set it up with DC entries, we run into the automatic boundary alignment that is normally performed on fullwords. This action can be overridden, however, by specifying a length modifier. A length of 4 is, of course, the same as the implied length of a fullword; the whole purpose is to prevent boundary alignment.

QUESTIONS AND EXERCISES

For questions 1–6, show the contents of WORK after the execution of ED WORK,SOURCE. The characters in WORK have the following meanings:

Character	*Meaning*	*Hexadecimal Equivalent*
B	Blank	40
S	Significance starter	21
D	Digit selector	20
,	Comma	6B
.	Decimal	4B
C	C C3	C3
R	R D9	D9
*	*	5C
F	Field separator	22

1. WORK BDDDDDDD
   SOURCE 0001540+
2. WORK BDDDDDDDCR
   SOURCE 0005721+
3. WORK BDD,DDS.DDBCR
   SOURCE 0000001-
4. WORK BDDD,DDCR
   SOURCE 00000+
5. WORK BSD,DDD.DDCR
   SOURCE 0000010+
6. WORK BDD,DDS.DDCRFDD,DDS.DDBCR
   SOURCE 0010143–0000107–

7a. Write a DC named PATRN to set up the editing pattern for a 9-digit amount to be printed as follows:

BX,XXX,XXX.XXBBB (for a positive amount)
BX,XXX,XXX.XXBCR (for a negative amount)

Insignificant zeros should print as blanks. However, amounts less than one dollar must be punctuated with a decimal point.

b. If SOURCE contains 009250001–and we execute ED PATRN,SOURCE, what would PATRN then contain?

c. What would PATRN contain if EDMK instead of ED were the operation?

8. PATRN DC X′4020206B2020214B202040C3D9′
   EDMK PATRN,SOURCE

Assume SOURCE contains 0123456–. Choose the address that would be in bits 8–31 of general register 1 after execution of the EDMK instruction:

a. PATRN
b. PATRN+1
c. PATRN+2
d. PATRN+3

9. Does the ED instruction affect general register 1?

10. What would be in location AREA as a result of the following operations?

```
AREA     DC   X'00020103'
TABLE    DC   C'ABCD'
         TR   AREA,TABLE
```

a. ABCD
b. DBCA
c. DCBA
d. ADBC
e. ACBD

11. What would be in general registers 1 and 2 as a result of the following operations:

```
AREA     DC   X'00010203'
TABLE    DC   X'00000100'
         TRT  AREA,TABLE
```

a. Address of AREA+3 and X′03′ respectively
b. Address of TABLE+2 and X′01′ respectively
c. Address of TABLE+3 and X′04′ respectively
d. Address of AREA+2 and X′01′ respectively

12. Assume the following sequence:

```
CON1  DC   F'10'
WORK  DC   CL16'1234567899123456'
AREA  DS   CL20
      L    2,CON1
      MVI  AREA,C'0'
      MVC  AREA+1(19),AREA
      EX   2,MOVE
      B    ROU2
MOVE  MVC  AREA(0),WORK
```

What will AREA contain after the instruction B ROU2 is executed?

13. What would AREA contain if the EX instruction were EX 0,MOVE?

Chapter 8: Subroutine Linkages and Program Relocation

Subroutines are an important element in programming. Storage space is conserved when a subroutine at one storage location is branched to from many points in a main section instead of being inserted each time it is needed. Programming, compilation, and debugging time are conserved when an existing subroutine can be incorporated into a new program.

A subroutine is a set of instructions that performs a particular function. It may be used in more than one program or more than once within a single program. Subroutines have been used in scientific programming for many years. Common subroutines used are the sine, cosine, and square root functions. Subroutines have now become equally important in commercial programming. In many cases, a main program may be little more than a sequence of branches to subroutines, some of which may be used many times, some only once. When a long and involved program is to be written, it is frequently divided into a number of separate subroutines to be written by different programmers. After the general plan is determined, each part may be relatively simple to program, and a considerable saving of time can be achieved. Each section can be assembled and debugged independently.

Subroutines may be classified as either "open" or "closed". An open subroutine is included each time it is required in the main program. The open subroutine is not normally branched to but is inserted into the main program and as such has little or no difficulty communicating with the main program. The closed subroutine, which is the kind we shall investigate in this chapter, is included once in a program and in storage no matter how many times it is branched to. Since the subroutine may be entered from many points in the main program, communication of data to the subroutine and of results back to the main program can be a problem unless standards are set.

In this chapter we shall be concerned primarily with the standards that have already been established for subroutine communication. By demonstrating the techniques in actual program examples, we shall answer questions like:

How does the subroutine know where to return in the main program?

How does the main program pass data to the subroutine?

How does the subroutine pass results back to the main program?

How can one program reference areas in another program that the assembler does not know about?

Much of the above is accomplished through register addressing. For the rest we will look to functions of the assembler and the linkage editor.

SUBROUTINE LINKAGES

The basic idea of a subroutine is to put it in storage at one place, then branch to it whenever its function is needed. If we are using a square root subroutine, for instance, we put it in one section of storage available for use as needed. Then, at any point in the main program that we need to take a square root, we branch to the square root subroutine, compute the square root, and branch back to the point in the main program where we left off.

This raises two questions: How does the subroutine know where to return when its work is finished? How does the main program provide the subroutine with information on the location of the number to be processed and where the result is to be left?

The question of where to return is answered by a *linkage* that places in a register the address of the next instruction after the one that branches to the subroutine. In System/360 we do this with the Branch and Link Register (BALR) instruction that we have seen so frequently for loading a base register. But now we specify a second operand other than zero, so that it really is a branch. The technique is to place in a register, usually 15, the address of the first instruction of the subroutine. Then, if we have chosen register 14 to hold the link, we write the instruction BALR 14,15. When executed, this instruction places in register 14 the address of the next byte after the BALR, and causes a branch to the address in register 15. At the end of the subroutine it is merely necessary to specify an unconditional branch to the address in register 14. This is done with a Branch Register Unconditional (extended mnemonic BR).

We can make these ideas much more clear by considering an example. It is not our purpose now to explore new ideas in information processing, so we chose an unrealistically simple job for the subroutine to do: to double a number by shifting it left one place. Communicating data and the location of results between the main routine and the subroutine is handled easily by placing the number to be doubled in a register, in this case register 3, before the branch to the subroutine, and leaving the doubled result in register 3 on the return to the main program. Figure 8-1 is a listing of a single program consisting of a main, or calling, routine and a subroutine.

The START, BALR, and USING instructions in Figure 8-1 are still necessary; they are unchanged by the fact that a subroutine will be used. Next comes the first processing instruction of the main routine, to load register 3 with a number that is to be doubled by the subroutine. Register 15 is then loaded with the address of the subroutine, using an address constant, in preparation for branching to the subroutine with the BALR.

Address constants, a subject we have not so far encountered, provide a means of communicating between separate parts of a program or between separately assembled programs. We could have used other means in this single assembly, but since address constants will appear throughout the rest of this chapter, they are well worth some study. An address constant (adcon for short) is a storage address that is translated into a constant. Unlike other types of DC's, it is enclosed in parentheses. We are particularly interested here in two types of address constants: A and V.

An A-type address constant may be absolute (its value does not change upon program relocation) or it may be

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1           PRINT NOGEN
000000                               2 LINK1     START 0
000000 05B0                          3 BEGIN     BALR  11,0
000002                               4           USING *,11
000002 5830 B022            00024    5           L     3,FIRST           FIRST NUMBER TO BE DOUBLED
000006 58F0 B01E            00020    6           L     15,ADSR1          SUBROUTINE ADDRESS
00000A 05EF                          7           BALR  14,15             LINKAGE RETURN ADDRESS GOES INTO 14
00000C 5030 B02A            0002C    8           ST    3,ANS1            RETURN POINT FROM SUBROUTINE
000010 5830 B026            00028    9           L     3,SECOND          SECOND NUMBER TO BE DOUBLED
000014 58F0 B01E            00020   10           L     15,ADSR1          SUBROUTINE ADDRESS AGAIN
000018 05EF                         11           BALR  14,15             LINKAGE
00001A 5030 B02E            00030   12           ST    3,ANS2            STORE SECOND RESULT
                                    13           EOJ                     END OF JOB
000020 00000034                     16 ADSR1     DC    A(SR1)            SUBROUTINE ADDRESS
000024 00000001                     17 FIRST     DC    F'1'
000028 00000004                     18 SECOND    DC    F'4'
00002C                              19 ANS1      DS    F
000030                              20 ANS2      DS    F
                                    21 *
                                    22 *   THIS IS THE END OF THE MAIN PROGRAM
                                    23 *   THE SUBROUTINE MAY USE ITS OWN BASE REGISTER
                                    24 *   WHICH MUST BE LOADED AND IDENTIFIED
                                    25 *
000034 05A0                         26 SR1       BALR  10,0
000036                              27           USING *,10
000036 8B30 0001            00001   28           SLA   3,1               THIS IS THE ONLY PROCESSING INSTRUCTION
00003A 07FE                         29           BR    14                UNCONDITIONAL BRANCH TO MAIN ROUTINE
000000                              30           END   BEGIN
```

Figure 8-1. Listing of a single program that consists of a main, or calling, routine and a subroutine. Standard linkage registers are used.

relocatable. The storage address is calculated by the assembler and is stored in binary integer form. If no length is specified, it is stored as a fullword, aligned to a fullword boundary. We note that in statement 16 the object code for the DC named ADSR1 is 00000034. The operand is A(SR1); the A stands for address and the SR1 in parentheses is the same as the label of the first machine instruction in the subroutine, which is at location 000034.

A V-type address constant, which we shall see later, is similar to the A-type, but it *must* be relocatable. It is used to reserve storage for the address of a symbol that is defined in a program or program segment *external* to the program it appears in. During assembly the V-type constant is given a zero value, and it is placed in the assembler's external symbol dictionary, to be resolved later by the linkage editor.

The BALR as written in statement 11 takes its branch address from register 15 and places in register 14 the address of the next instruction. The Branch and Link (BAL) instruction can sometimes be used instead of BALR, thereby avoiding the loading of a register before branching. The restriction is that the address of the subroutine must be within the range of addresses of the current program base register. This will not always be true, and will never be true for separately assembled routines, as we shall discuss later. BALR is probably a good habit even when not strictly needed.

We have now branched to the subroutine, which in this highly simplified example consists of just one processing instruction. The contents of register 3 are shifted left one place, which doubles the number, and the processing is finished. We are now ready to return to the instruction following the BALR in the main routine. This address is precisely what is in register 14 now, so an unconditional branch to the address specified in register 14 is the correct return. The BR instruction is unconditional.

On returning to the main routine, we store the doubled number at ANS1 and proceed to load another number into register 3 for doubling by the subroutine. We again go through the operations of loading register 15 with the address of the subroutine and linking to it. Although it is true that register 15 still has the address of the subroutine in it from the last time, we prefer, even in this example, to load it again as a matter of good programming habit.

Figure 8-2 shows the values of FIRST, SECOND, ANS1, and ANS2, in that order, after the execution of the program in Figure 8-1.

```
00000001  00000004  00000002  00000008
```

Figure 8-2. Values of FIRST, SECOND, ANS1, and ANS2, respectively, after execution of program in Figure 8-1

In Figure 8-3 we add a feature to the program. Shifting a number left can, of course, result in loss of a bit from large numbers. Let us arrange things so that such a loss would be signaled back to the main routine as an error. Our method of signaling may be as follows. If such a loss of information occurs, the subroutine would return to the instruction after

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      1          PRINT NOGEN
000000                                2 LINK2    START 0
000000 05B0                           3 BEGIN    BALR  11,0
000002                                4          USING *,11
000002 5830 B02E            00030     5          L     3,FIRST          FIRST NUMBER TO BE DOUBLED
000006 58F0 B02A            0002C     6          L     15,ADSR1         SUBROUTINE ADDRESS
00000A 05EF                           7          BALR  14,15            LINKAGE-RETURN ADDRESS
00000C 47F0 B026            00028     8          B     ERROR            ERROR RETURN
000010 5030 B036            00038     9          ST    3,ANS1           RETURN POINT FROM SUBROUTINE
000014 5830 B032            00034    10          L     3,SECOND         SECOND NUMBER TO BE DOUBLED
000018 58F0 B02A            0002C    11          L     15,ADSR1         SUBROUTINE ADDRESS AGAIN
00001C 05EF                          12          BALR  14,15            LINKAGE
00001E 47F0 B026            00028    13          B     ERROR            ERROR RETURN
000022 5030 B03A            0003C    14          ST    3,ANS2           STORE SECOND RESULT
                                     15          EOJ                    PROGRAM TERMINATION
                                     18 ERROR    EOJ                    ERROR PROGRAM TERMINATION
00002A 0000
00002C 00000040                      21 ADSR1    DC    A(SR1)
000030 00000010                      22 FIRST    DC    F'16'
000034 7FFFFFFF                      23 SECOND   DC    X'7FFFFFFF'
000038                               24 ANS1     DS    F
00003C                               25 ANS2     DS    F
                                     26 *
                                     27 *   THIS IS THE END OF THE MAIN PROGRAM
                                     28 *   THE SUBROUTINE MAY USE ITS OWN BASE REGISTER
                                     29 *   WHICH MUST BE LOADED AND IDENTIFIED
                                     30 *
000040 05A0                          31 SR1      BALR  10,0
000042                               32          USING *,10
000042 8830 0C01            00001    33          SLA   3,1              THIS IS THE ONLY PROCESSING INSTRUCTION
000046 4710 E000            00000    34          BC    0(0,14)          GO TO ERROR RETURN
00004A 47F0 E004            00004    35          B     4(0,14)          UNCONDITIONAL BRANCH TO MAIN PROGRAM
000000                               36          END   BEGIN
```

Figure 8-3. The program of Figure 8-1 modified to give the subroutine a choice between two return points the BALR; if there is no loss of information, the subroutine would return to an instruction that is four bytes after the BALR. In other words, the instruction after the BALR would be executed only in the error condition, and it would be called the *error return*. The *normal return* would branch back beyond this point.

We shall insert in the program, following each BALR to the subroutine, a branch to a routine labeled ERROR to discontinue the program if the error arises. (In practical applications, of course, we would take some corrective action rather than give up completely.) The choice of whether to go back to the error return or the normal return will be made by the subroutine. Figure 8-3 shows the modifications required. After shifting left, we execute a Branch on Overflow (BO) instruction that tests the condition code set by the Shift Left Single (SLA) instruction. If we have overflowed, the branch is taken and the error return is reached. If we have not overflowed, we go back to the normal return, which is four bytes beyond the address in register 14. This is done with a Branch Unconditional instruction (extended mnemonic B) that uses register 14 for a base register and has a displacement of 4.

Figure 8-4 shows the information at the end of execution of the program, with the values for FIRST, SECOND, and ANS1. A doubled value for SECOND has not been stored, since the error return was taken and the instruction (ST 3, ANS2) was never reached.

```
00000010  7FFFFFFF  00000020
```

Figure 8-4. Values of FIRST, SECOND, and ANS1, respectively, after execution of the program in Figure 8-3. ANS2 was not stored because the error return was taken during the doubling of SECOND.

STANDARD LINKAGE REGISTERS

So far we have seen a typical subroutine linkage in action, with a variation that allows a choice between two return points. Communication of data and results between the main program and the subroutine was made easily because the programmer knew which registers were used for what purpose in both. Supposing they had been written by different programmers?

To ease the problem of assuring proper communications between program segments, which often are written by different programmers, standard register assignments and techniques have been defined in each of the IBM operating or programming support systems. (They are similar, but not identical, in all System/360 operating systems.) Standard register assignments in the Disk Operating System (DOS) are shown in Table 8-1. The items in this chart will be explained in the following pages of text, as their use is discussed. These registers are used for the purposes shown, both by programmers and by the DOS macros. The DOS macros for subroutine linking are CALL, SAVE, and RETURN.

Table 8-1. DOS Linkage Registers

Register Number	Register Function	Contents
0	Parameter register	Parameters to be passed to the subroutine.
1	Parameter register or Parameter list register	Parameters to be passed to the subroutine. Address of a parameter list to be passed to either the control program or a user's subroutine.
13	Save area register	Address of the register save area to be used by the subroutine.
14	Return register	Address of the location in the main program to which control should be returned after execution of the subroutine.
15	Entry point register	Address of the entry point in the subroutine.

Let us examine the standard linkage registers. We have been using register 14 for the return register and register 15 for the entry point register. Data may be passed to a subroutine using registers 0 and 1. However, a more common practice is to use register 1 to hold the address of a *parameter list*, because we usually have more data than will fit in two registers. (The expression "parameter list" merely signifies a list of numbers of any desired value.) The parameter list may consist of either data or the addresses of data. Addresses are used more often so that data of varying lengths can be handled easily. One common technique is to write the data and/or data addresses in the instruction stream immediately following the BALR. This is a point from which the subroutine can readily obtain them.

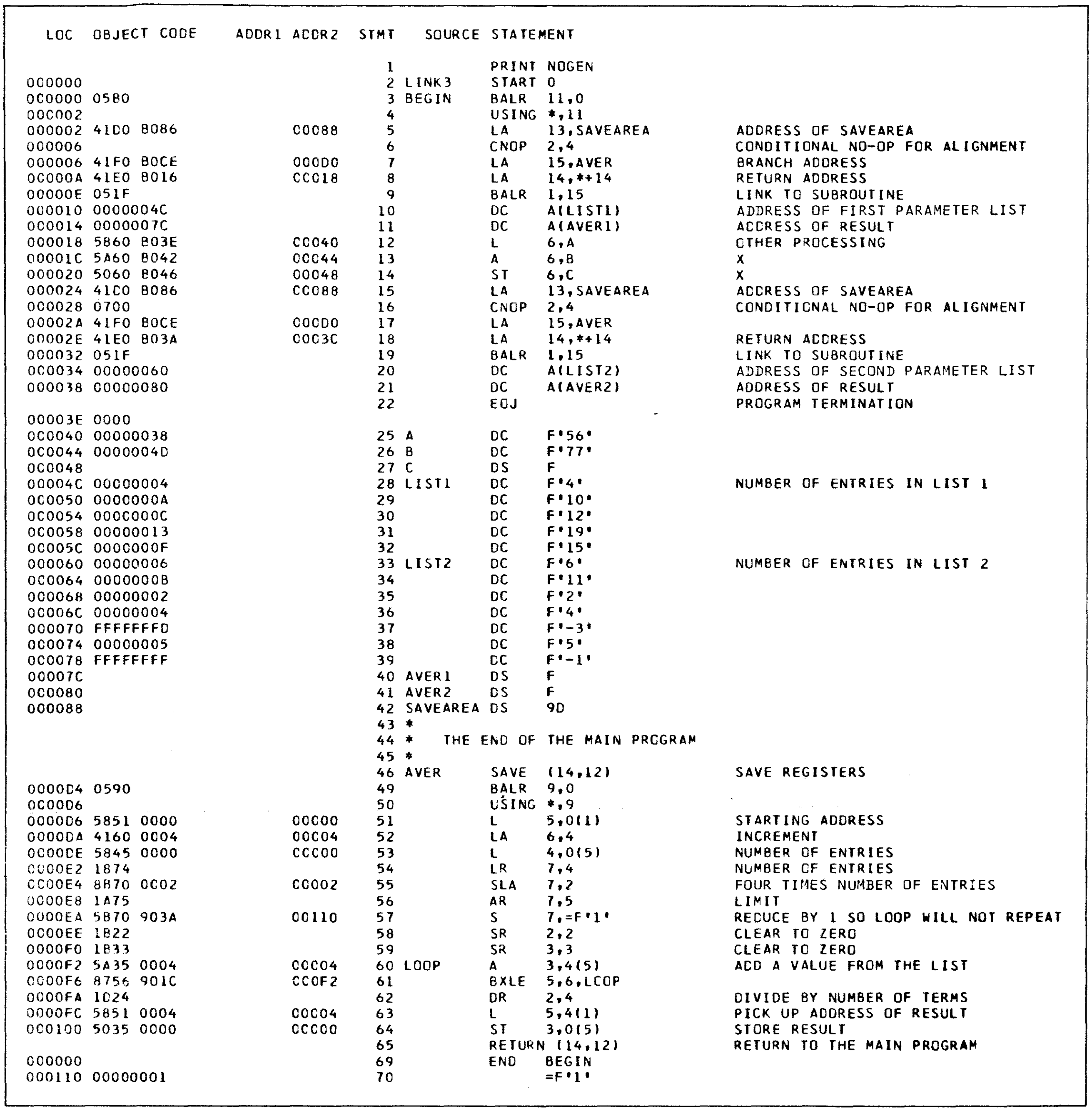
Figure 8-5 illustrates the use of most of these techniques and includes a subroutine that averages a series of numbers. The main program stores two series of numbers as lists in consecutive fullword locations. The average of each list is to be calculated and stored, so the subroutine will be used twice and each time will return to different points in the main program. To simplify the averaging routine, each list begins with the total number of entries in the list. Each list is identified by its starting address. This address is to be communicated to the subroutine, along with the address at which the subroutine is to store the average.

There are several possible ways to give the necessary information to the subroutine. We chose one that is representative, using BALR and A-type address constants:

```
BALR  1,15       Link to subroutine
DC    A(LIST1)   Address of first parameter list
DC    A(AVER1)   Address of second parameter list
```

The address of the first word of the list and the address at which the average should be stored will immediately follow the BALR that branches to the subroutine. The subroutine will be required to pick up the information it needs from this parameter list. It can find it because it will have the address of the first word after the BALR in register 1, loaded there by the BALR. Of course, prior to this we had to load the address of our subroutine entry point into register 15. This was done by use of the Load Address (LA) instruction. In addition the return address was loaded into register 14. The return address must be carefully calculated to assure that the proper return point is stored. This address is the current value of the assembler's location counter at the *start* of this instruction plus the length of the LA (4 bytes) plus the length of the BALR (2 bytes) plus the length of the two address constants (8 bytes). This could also have been accomplished by labeling the return point and using that in a Load Address instruction. In any case, after we branch to the subroutine, register 1 contains the address of the first DC in the parameter list.

What about register 13, and what is a "save area"? Usually the subroutine will need to use the same registers that are used in the main program, but for different purposes. The main program may use the registers for base addresses, index addresses, intermediate results, or other data vital to the main program. To keep this data from being destroyed by the subroutine it has become conventional to store the contents of these registers in an area called a save area and defined by the main program. This area is 18 words in length in most System/360 systems and its address is stored in register 13 prior to branching to the subroutine. It is aligned on a doubleword boundary. In a standard save area, word 1 is used only by PL/I. Words 2 and 3 are used to trace subroutines that are branched to by



```
  LOC  OBJECT CODE        ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                         1          PRINT NOGEN
000000                                   2 LINK3    START 0
000000 05B0                              3 BEGIN    BALR  11,0
000002                                   4          USING *,11
000002 41D0 B086                00088    5          LA    13,SAVEAREA         ADDRESS OF SAVEAREA
000006                                   6          CNOP  2,4                 CONDITIONAL NO-OP FOR ALIGNMENT
000006 41F0 B0CE                000D0    7          LA    15,AVER             BRANCH ADDRESS
00000A 41E0 B016                00018    8          LA    14,*+14             RETURN ADDRESS
00000E 051F                              9          BALR  1,15                LINK TO SUBROUTINE
000010 0000004C                         10          DC    A(LIST1)            ADDRESS OF FIRST PARAMETER LIST
000014 0000007C                         11          DC    A(AVER1)            ADDRESS OF RESULT
000018 5860 B03E                00040   12          L     6,A                 OTHER PROCESSING
00001C 5A60 B042                00044   13          A     6,B                 X
000020 5060 B046                00048   14          ST    6,C                 X
000024 41D0 B086                00088   15          LA    13,SAVEAREA         ADDRESS OF SAVEAREA
000028 0700                             16          CNOP  2,4                 CONDITIONAL NO-OP FOR ALIGNMENT
00002A 41F0 B0CE                000D0   17          LA    15,AVER
00002E 41E0 B03A                0003C   18          LA    14,*+14             RETURN ADDRESS
000032 051F                             19          BALR  1,15                LINK TO SUBROUTINE
000034 00000060                         20          DC    A(LIST2)            ADDRESS OF SECOND PARAMETER LIST
000038 00000080                         21          DC    A(AVER2)            ADDRESS OF RESULT
                                        22          EOJ                       PROGRAM TERMINATION
00003E 0000
000040 00000038                         25 A        DC    F'56'
000044 0000004D                         26 B        DC    F'77'
000048                                  27 C        DS    F
00004C 00000004                         28 LIST1    DC    F'4'                NUMBER OF ENTRIES IN LIST 1
000050 0000000A                         29          DC    F'10'
000054 0000000C                         30          DC    F'12'
000058 00000013                         31          DC    F'19'
00005C 0000000F                         32          DC    F'15'
000060 00000006                         33 LIST2    DC    F'6'                NUMBER OF ENTRIES IN LIST 2
000064 0000000B                         34          DC    F'11'
000068 00000002                         35          DC    F'2'
00006C 00000004                         36          DC    F'4'
000070 FFFFFFFD                         37          DC    F'-3'
000074 00000005                         38          DC    F'5'
000078 FFFFFFFF                         39          DC    F'-1'
00007C                                  40 AVER1    DS    F
000080                                  41 AVER2    DS    F
000088                                  42 SAVEAREA DS    9D
                                        43 *
                                        44 *   THE END OF THE MAIN PROGRAM
                                        45 *
                                        46 AVER     SAVE  (14,12)             SAVE REGISTERS
0000D4 0590                             49          BALR  9,0
0000D6                                  50          USING *,9
0000D6 5851 0000                00000   51          L     5,0(1)              STARTING ADDRESS
0000DA 4160 0004                00004   52          LA    6,4                 INCREMENT
0000DE 5845 0000                00000   53          L     4,0(5)              NUMBER OF ENTRIES
0000E2 1874                             54          LR    7,4                 NUMBER OF ENTRIES
0000E4 8B70 0002                00002   55          SLA   7,2                 FOUR TIMES NUMBER OF ENTRIES
0000E8 1A75                             56          AR    7,5                 LIMIT
0000EA 5B70 903A                00110   57          S     7,=F'1'             REDUCE BY 1 SO LOOP WILL NOT REPEAT
0000EE 1B22                             58          SR    2,2                 CLEAR TO ZERO
0000F0 1B33                             59          SR    3,3                 CLEAR TO ZERO
0000F2 5A35 0004                00004   60 LOOP     A     3,4(5)              ADD A VALUE FROM THE LIST
0000F6 8756 901C                000F2   61          BXLE  5,6,LOOP
0000FA 1D24                             62          DR    2,4                 DIVIDE BY NUMBER OF TERMS
0000FC 5851 0004                00004   63          L     5,4(1)              PICK UP ADDRESS OF RESULT
000100 5035 0000                00000   64          ST    3,0(5)              STORE RESULT
                                        65          RETURN (14,12)            RETURN TO THE MAIN PROGRAM
000000                                  69          END   BEGIN
000110 00000001                         70                =F'1'
```

Figure 8-5. A program with a subroutine that averages a series of numbers. The subroutine will be used twice and will store the results at AVER1 and AVER2.

subroutines. Word 2 is the save area address of a preceding routine. Word 3 is the save area address of a succeeding routine. Words 4–18 are the contents of registers 14–12, respectively–that is, all the general registers except register 13.

Note that early in the main program after the usual preliminaries, we load register 13 with the address of the save area. Then follows the CNOP (which we will discuss later on), and the loading of the entry point and return addresses. In the subroutine we begin by saving the contents of the registers in the save area. This is accomplished through the use of the SAVE macro in DOS, or by regular assembler language Store instructions in systems where this macro is not available. Register 14 is specified as the first register to be stored, and all additional registers are stored simply by specifying the last register to be stored. Thus the registers are stored in the order 14, 15, then 0 through 12. The instruction generated by the SAVE macro is STM 14, 12, 12(13). Therefore the registers are stored in the save area (its address is designated by the contents of register 13) starting at a point that is twelve bytes past the beginning of the area (we recall that the first three words of the save area are reserved for other data). Now the subroutine can use the registers for its own purposes and, when its processing is finished, can restore the registers to their status when the subroutine was entered. This entire procedure is normal practice; it can almost never be assumed that any registers are available to the subroutine unless their contents are first saved.

Let's proceed with the work performed by the subroutine. Statement 51 gets the address of LIST1 by a Load instruction in which the effective address is simply the contents of register 1. After execution of this LOAD, the address in register 1 (00004C) is placed in register 5 for subsequent use. Stepping through the list will be done with a Branch on Index Low or Equal instruction (BXLE), so we proceed to set up the other parameters required. Register 6 is accordingly loaded with a 4, the increment between locations that must be added on each repetition of the loop. With register 6 containing the increment, register 7 must contain the final address, that is, the starting address plus four times the number of entries.

We load register 4 with the number of entries, which is the first fullword in each list. It is to be left in register 4 for computing the average later. For purposes of controlling the loop, we move it to register 7, shift left two places (in effect multiplying by four), and add the starting address. Since the BXLE will repeat the loop on an equal, we must now reduce the value in register 7 by 1, using the literal term =F′1′ to introduce the value one. After clearing registers 2 and 3, we are ready to go into the loop.

A literal term in assembler language is a way of introducing data into a program in a machine instruction, and is (in that one sense only) like immediate data in an SI instruction. It is simply a constant preceded by an equal sign. It represents data rather than a reference to data. It can be used to enter a number for calculation, an address constant, or words or phrases for printing out a message. Unlike immediate data, a literal term is not assembled into the instruction in which it appears. The assembler generates the values of all literals in a program, collects them, and stores them in a "pool," usually at the end of the program. Their addresses, rather than their values, are assembled into the instructions in which they are used, and so literals are considered relocatable.

The Add instruction at LOOP uses as its address the contents of register 5, which is the index register for the loop. The address in register 5 is adjusted by 4 so that we will address the first number to be averaged rather than the number of entries. Between loop repetitions, register 5 is incremented by the contents of register 6, which we set at 4. Each time register 5 is lower than, or equal to, register 7 on comparison, we will branch back to LOOP. The looping stops when all entries in the list have been added to register 3.

To compute the average, which is simply a matter of dividing the contents of registers 2 and 3 (the sum of all the numbers in the list) by the contents of register 4 (the number of entries in the list), a Divide Register instruction (DR) is used. The quotient is the average, which is in register 3. We are now ready to store the average; where does it go? The answer is to be found by looking at the fullword address that is four bytes beyond the address specified by the contents of register 1; the address of the average is placed in register 5 by the Load. A Store instruction using this address now completes the work of the subroutine. By use of the RETURN macro, we restore the registers that had been saved and branch back to the main routine.

The RETURN macro is coded identically to the SAVE macro, that is, the operands are the starting and ending registers to be restored, RETURN 14,12. It restores the saved data to the specified registers and returns to the main program via the return address in register 14. The coding generated by the RETURN macro is a Load Multiple (LM) instruction and an unconditional branch.

Back in the main routine, we do some simple processing using register 6. We can use the same registers as we used in the subroutine because the SAVE and RETURN macros allow each section of programming to view the registers as their own.

We then wish to average another list, LIST2. At statement 15, note that we must again place the address of the save area in register 13 because we must enter the subroutine again. The execution of the subroutine follows the same lines as before, although this time it operates on different data and places the result in a different place.

The CNOP assembler instruction, which we have not discussed so far, appears twice in the main program during the preparations for each branch to the subroutine. The function of the CNOP (Conditional No-Operation) is to make sure that the two address constants appear in storage immediately after the BALR.

If the instruction before the BALR ended on a fullword boundary, the BALR (a two-byte instruction) would then occupy the first two bytes of the next word. The assembler, automatically aligning on a fullword boundary an A-type constant for which no length is specified, would skip two bytes before locating the constant. Then, when the BALR is executed, register 1 would contain the address of the byte following the BALR instruction, but this address would not be correct for the parameter list.

Before reaching the BALR we write the instruction CNOP 2,4. If the assembler's location counter is already set to a value that is two bytes greater than a fullword boundary, the CNOP is ignored, as is the case in statement 6. If not, as is the case in statement 16, the assembler inserts a Branch on Condition (BC) instruction with a mask of zero, which never causes a branch and therefore is equivalent to a "No-Operation". It occupies two bytes and thus causes the parameter list to be located immediately following the BALR.

The CNOP in statement 6 has no effect; we see that the location counter is already located as described by the CNOP (that is, it is in the second half of a fullword–000006). Therefore, no instruction is generated for the CNOP. Note that the DC will be on a fullword boundary without skipping any space between the BALR and the address constant.

The CNOP in statement 16 resulted in the creation of a No-Operation instruction because the assembler's location counter is at 000028, which is not two bytes beyond a fullword. If this had not been done, the assembler would have placed the BALR at 000030 and the A-type constant at 000034, thereby leaving a two-byte gap. The next byte after the BALR (000032) would go into register 1. The subroutine's ensuing attempt to call for a fullword from 000032 would have caused a specification exception and a program interruption.

The CNOP could be used before or after the two LA instructions that load the return and entry point addresses into registers 14 and 15. We have used the CNOP ahead of the two LA instructions in this program.

Figure 8-6 shows the two lists of numbers followed by the average of each, as computed by the program in Figure 8-5.

```
10  12  19  15                 14
11   2   4  -3   5  -1          3
```

Figure 8-6. The data and results of the program in Figure 8-6. The last number in each line is the average.

PROGRAM RELOCATION

In an operating system environment, a program must be processed by the linkage editor program before it can be executed. During the process it is usually relocated, that is, assigned a starting location in main storage other than the locations assigned during assembly. Most programs are run more than once. A standard subroutine may be stored in a part of the system library in relocatable form and be used very frequently. A different core location may be assigned each time. We know that the storage location indicated by the START assembler instruction is tentative. The locations calculated by the assembler merely establish the relative storage locations of data and instructions within a program. Most programs are assembled relative to zero, but are never executed there because of restrictions on the use of lower core. System/360 was designed to run under a control program, and, under operating conditions, part of the control program is always resident in the low region of main storage for handling interruptions, error recovery routines, etc. In many systems this occupies several thousand bytes. Problem programs must be executed beyond this area.

Relocation is necessary for a number of other reasons not of direct interest to the programmer. These are (1) the overall storage requirements of other programs that are to go into core at the same time and (2) various operating considerations dependent upon the type of installation, operating system environment, etc.

The Linkage Editor

When the capabilities of the linkage editor are added to program relocatability, great programming efficiency can be achieved by dividing a large program into separate sections for coding. Each section can be written by a different programmer and compiled and checked out separately. It is even possible to code some of the subroutines in a different programming language. Each part of the programming operation is greatly simplified. Time is saved by having several people work independently and simultaneously on the program. When all the routines have been compiled, they are in relocatable form and can be link edited in any sequence. The linkage editor will assign storage locations and match up all address references between the routines, so that the entire program can be executed correctly as one program. If it should be necessary to correct a routine, only that one routine would have to be reassembled or recompiled and then link edited again with the other routines. This facility also makes it relatively simple to maintain a large program that may have to be updated from time to time.

The output of the assembler (or any language translator) is called an object module. It may consist of a single program or many. We should perhaps use the more exact term *control section*. A control section is the smallest separately relocatable unit of a program. It is an entity declared as such by the programmer by use of the START statement or another assembler instruction called CSECT. A program may consist of one control section or many control sections.

In an operating system environment, an object module has two major characteristics:

(1) It is relocatable. This means that all address constants are in a form that can be modified to compensate for a change in the starting location.

(2) It is not executable.

The object module may call for other object modules assembled at other times and stored in the system library in relocatable form. Programmers frequently indicate standard I/O or other object modules to be included as subroutines in a new program. This is perfectly feasible. The linkage editor, which is a service program, will find all required modules, process one after the other, and combine them into a single, executable *load module* (or *program phase*, as it is called in some systems).

In Chapter 1 we described the output of the assembler program. The major items of input to the linkage editor program are the object code (or *text*), the external symbol dictionary (ESD), and the relocation dictionary (RLD). (Linkage editor control cards are also included, but will not be covered in this book.)

The text consists of the actual instructions and data fields of one or more control sections in the module. The dictionaries contain the information necessary for the linkage editor to resolve references between different modules.

For each control section in an assembly, the ESD contains its length in bytes, assembled address, and any name given in the START or CSECT statement. Also included is information about any V-type address constants, external references (a linkage symbol used in this control section but defined in another), and entry points (a linkage symbol defined in this control section but used in another).

The relocation dictionary contains all address constants that appear in an assembly. RLD information includes (1) the control section in which the adcon is used as an operand, (2) the control section in which it is defined, (3) whether it is a V-type or other type, (4) how long it is, and (5) the assembled address where it is stored.

The linkage editor, working under control of the job control program, builds up composite dictionaries of the ESD and RLD data found in the object modules. It resolves all linkages between different control sections as if they had been assembled as one object module. It relocates each control section as necessary and assigns the entire load module (or program phase) to a contiguous area of main storage. It adds the relocation factor to the location given by the assembler's location counter at the start of each assembly. It modifies all relocatable address constants to contain the relocated value of their symbols. (Except for adcons, no address values within the instructions and data fields are changed during link editing. These remain in base and displacement form, as we shall see shortly in the dumps of the next program.) The load module is constructed by building the text in the form in which it will actually be loaded into core; it is then executable and nonrelocatable.

The CALL and PDUMP Macros

Perhaps it would help to clarify just what happens during program relocation if we could see our program in main storage while it is being executed. We can do nearly that by getting a "dump" of storage during execution. In this section we shall see how a program appears, first relocated to one area of storage and then to another. We shall use a program almost identical to the last one, a main program assembled with a subroutine.

In the last program, Figure 8-5, statements 6 through 11 and 16 through 21 were necessary to link to the subroutine, communicate data both ways, and get back to the right point in the main program. We had to be very careful about boundary alignment and using the correct standard linkage registers for the right functions. In DOS all of this can be taken care of very simply by use of the CALL macro, which generates instructions similar to the six statements in the last program. In the program in Figure 8-7 CALL appears in statements 7 and 18.

The CALL macro instruction was designed primarily for use with separately assembled programs to pass control from one program to a specified entry point in another. It works equally well within a single assembly, however, because the assembler and linkage editor carry out their functions just

```
  LOC  OBJECT CODE    ACDR1 ACCR2  STMT   SOURCE STATEMENT

                                      1          PRINT NOGEN
                                      2          ENTRY AVER
0C0000                                3 LINK4    START 0
0C0000 05B0                           4 BEGIN    BALR  11,0
0C0002                                5          USING *,11
0C0002 41C0 B08E            CC090     6          LA    13,SAVEAREA           ACCRESS OF SAVEAREA
                                      7          CALL  AVER,(LIST1,AVER1)    LINK TO SUBROUTINE
000018 5860 B046            CC048    14          L     6,A                   OTHER PROCESSING
00001C 5A60 B04A            CC04C    15          A     6,B                   X
0C0020 5060 B04E            CCC50    16          ST    6,C                   X
000024 41D0 B08E            CC090    17          LA    13,SAVEAREA           ADDRESS OF SAVEAREA
                                     18          CALL  AVER,(LIST2,AVER2)    LINK TO SUBROUTINE
                                     25          PDUMP BEGIN,BEGIN+X'200'    CUMP ROUTINE
                                     30          EOJ                         PROGRAM TERMINATION
000048 00000038                      33 A        DC    F'56'
0C004C 0000004D                      34 B        DC    F'77'
0C0050                               35 C        DS    F
000054 00000C04                      36 LIST1    DC    F'4'                  NUMBER OF ENTRIES IN LIST 1
000058 0000000A                      37          DC    F'10'
0C005C 0000000C                      38          DC    F'12'
000060 00000013                      39          DC    F'19'
0C0064 0000000F                      40          DC    F'15'
0C0068 C0000006                      41 LIST2    DC    F'6'                  NUMBER OF ENTRIES IN LIST 2
0C006C 0000000B                      42          DC    F'11'
0C0070 00000002                      43          DC    F'2'
0C0074 00000004                      44          DC    F'4'
000078 FFFFFFFD                      45          DC    F'-3'
0C007C 00000005                      46          DC    F'5'
0C0080 FFFFFFFF                      47          DC    F'-1'
0C0084                               48 AVER1    CS    F
0C0088                               49 AVER2    DS    F
000Q90                               50 SAVEAREA DS    9D
                                     51 *
                                     52 *   THE END OF THE MAIN PRCGRAM
                                     53 *
                                     54 AVER     SAVE  (14,12)               SAVE REGISTERS
0000CC 0590                          57          BALR  9,0
0C00DE                               58          USING *,9
0000DE 5851 0000            0C000    59          L     5,0(1)                STARTING ADDRESS
0000E2 4160 0004            C0004    60          LA    6,4                   INCREMENT
0000E6 5845 0000            CCC00    61          L     4,0(5)                NUMBER OF ENTRIES
0C00EA 1874                          62          LR    7,4                   NUMBER CF ENTRIES
0000EC 8B70 0002            0CC02    63          SLA   7,2                   4(NUMBER OF ENTRIES)
0000F0 1A75                          64          AR    7,5                   LIMIT
0C00F2 5B70 904E            0012C    65          S     7,=F'1'               REDUCE BY 1 SO LOOP WILL NOT REPEAT
0000F6 1B22                          66          SR    2,2                   CLEAR TC ZERO
C000F8 1B33                          67          SR    3,3                   CLEAR TO ZERO
0000FA 5A35 0004            00C04    68 LOOP     A     3,4(5)                ACD A VALUE FROM THE LIST
0000FE 8756 901C            CCCFA    69          BXLE  5,6,LCOP
CC0102 1C24                          7C          DR    2,4                   CIVIDE BY NUMBER OF TERMS
000104 5851 0004            00C04    71          L     5,4(1)                PICK UP ADDRESS OF RESULT
000108 5035 0000            CCC0C    72          ST    3,0(5)                STORE RESULT
                                     73          RETURN (14,12)              RETURN TO THE MAIN PROGRAM
0C0000                               77          END   BEGIN
000118 5B5BC2C7C4E4D4D7              78                =CL8'$$BPDUMP'
0C0120 000000C000000200              79                =A(BEGIN,BEGIN+X'200')
0C0128 00000000                      80                =V(AVER)
00012C 00000001                      81                =F'1'
```

Figure 8-7. A slightly different version of the program in Figure 8-5, modified by use of two macro instructions, CALL and PDUMP the same. As our program example is organized, the macro requires (1) the symbolic address of the entry point (AVER) as operand, which generates a V-type address constant, and (2) the separate ENTRY AVER statement. Since this is not a typical example of the ENTRY assembler instruction, it would be preferable to wait for the next program example before discussing it. The addresses of the parameter lists also appear in the operand field of the CALL macro, in parentheses, and will be in register 1 (the parameter list register) when the called subroutine is entered.

The first three literals that appear at the end of the assembly appeared in the instructions generated by the PDUMP and CALL macros. As it was written, the CALL generates a V-type address constant rather than the A-type. We note that zeros and not the address of AVER are assembled in statement 80. This address will be supplied later by the linkage editor.

Our plan for the program shown assembled in Figure 8-7 is to have it link edited, and, using linkage editor control cards, have it loaded and executed first at storage location 3000_{16}, and then at 4000_{16}. Re-assembly is not necessary. The linkage editor can override these cards, but will accept the instructions if they do not create a problem. Normally the programmer is not involved in the question of where a program is to be loaded. The storage area from which a program executes is properly an operational, not a programming, decision.

The DOS PDUMP macro causes the system, during program execution, to print out a hexadecimal dump of all the registers and any particular area of storage we are interested in. Such a dump is often used for program checkout, and learning to read one is a worthwhile exercise. We decide we want to see the state of affairs in storage at just the point when all calculations have been executed and the results stored, so we insert the PDUMP just before the EOJ. It will be reached just once, after the second execution of the subroutine.

The programmer must supply two address expressions in the operand field of the PDUMP to show the beginning and end of the storage area wanted. One or both of the addresses may be given in registers, but, since we want a dump from two different core locations, we use symbolic addresses. BEGIN will give us the beginning of the program, and BEGIN+200 is more than enough space to take us through to the end, as we know from the previous assembly of a similar program. Execution of the PDUMP macro will make no difference in execution of the program; processing will continue with the next sequential instruction. (Some other types of dumps result in termination of the program.)

Reading a Dump

Figure 8-8 shows the entire dump that was printed out (this is done by a line printer) during the first execution of the program in Figure 8-7. After assembly the program was link edited, loaded at 3000, and executed almost to the end. The point at which we see the dump is immediately after execution of the PDUMP macro. The EOJ macro instruction has not yet been reached.

A hexadecimal format is used because it represents the binary contents exactly. The contents of all the registers are shown at the top of the printout. General registers 0–7 are in the first line, and 8–15 in the second, a fullword each. The doubleword floating-point registers 0, 2, 4, and 6 are in the third line.

Next comes the hex dump of main storage. The six-digit column at the left shows the storage location of the first byte in each line. There are 20_{16} (or 32_{10}) bytes to a line, divided into fullwords. Locating an address is simplified by the wide space in the center of each line; the next byte after this space is simply hex 10 beyond the location in the lefthand column. For example, in the line that begins at 003020, the B0 in the fifth fullword is at location 003030. The last two bytes in this line are at 00303E and 00303F. The entire storage area shown is from 003000 to 0031FF, since the boundaries given in the PDUMP macro were BEGIN and BEGIN+X′200′. Printing of repeated lines of zeros at the end is suppressed.

At the extreme right, any alphameric characters in a line are also represented in characters. In many cases this is very helpful in interpreting the hexadecimal material and speeds up analysis of a dump. It is not particularly useful in our program, however, so the characters will be omitted to let us get a closer look at the hex. The reader may wish to arm

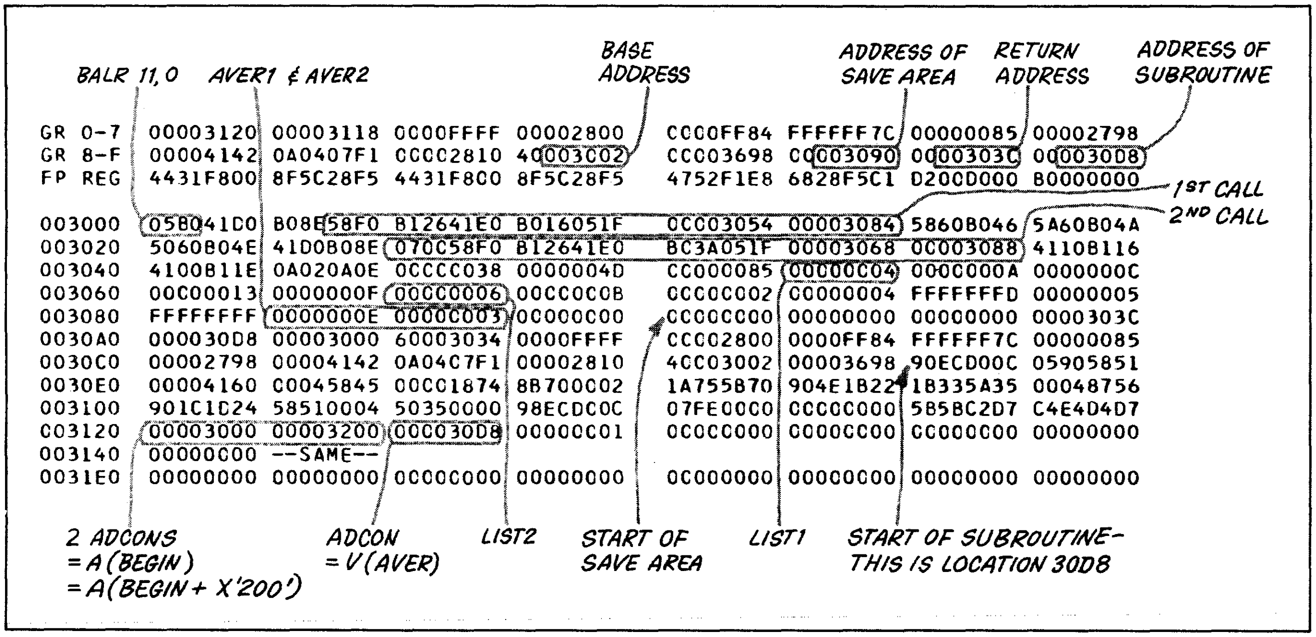
```
GR 0-7   00003120 00003118 0C00FFFF 00002800   CCC0FF84 FFFFFF7C 00000085 00002798
GR 8-F   00004142 0A0407F1 0CCC2810 40003C02   CC003698 CC003090 0000303C 000030D8
FP REG   4431F800 8F5C28F5 4431F800 8F5C28F5   4752F1E8 6828F5C1 D200D000 B0000000

003000   05B041D0 B08E58F0 B12641E0 B016051F   0C003054 00003084 5860B046 5A60B04A   .......0........ .........-...-..
003020   5060B04E 41D0B08E 070C58F0 B12641E0   BC3A051F 00003068 0C003088 4110B116   &-.........0.... ................
003040   4100B11E 0A020A0E 0CCCC038 0000004D   CC000085 00000C04 0000000A 0000000C   ................ ................
003060   00C00013 0000000F 000C0006 00CC0C0B   0CC0C002 C0000004 FFFFFFFD 00000005   ................ ................
003080   FFFFFFFF 0000000E 00000003 00000C00   CC0CCC00 00000000 00000000 0000303C   ................ ................
0030A0   000030D8 00003000 60003034 0000FFFF   CC002800 0000FF84 FFFFFF7C 00000085   ...Q....-....... ...........@....
0030C0   00002798 00004142 0A0407F1 00002810   40C03002 00003698 90ECD00C 05905851   ...........1....  ...............
0030E0   00004160 00045845 00C01874 8B700002   1A755B70 904E1B22 1B335A35 00048756   ...-............ ..$.............
003100   901C1D24 58510004 50350000 98ECDC0C   07FE0000 C0C0C000 5B5BC2D7 C4E4D4D7   ........&....... ........$$BPDUMP
003120   00003000 00003200 000030D8 00000C01   0C0C0000 00000000 CC0000C00 00000000  ...........Q.... ................
003140   00000000 --SAME--                                                             ....
0031E0   00000000 00000000 00000000 00000000   0C000000 00000000 00000000 00000000   ................ ................
```

Figure 8-8. Hex dump of registers and storage produced by execution of the PDUMP macro in the program in Figure 8-7. At right, EBCDIC characters are represented by characters.

himself with the card *IBM System/360 Reference Data* (see Preface), which is helpful for reading a dump.

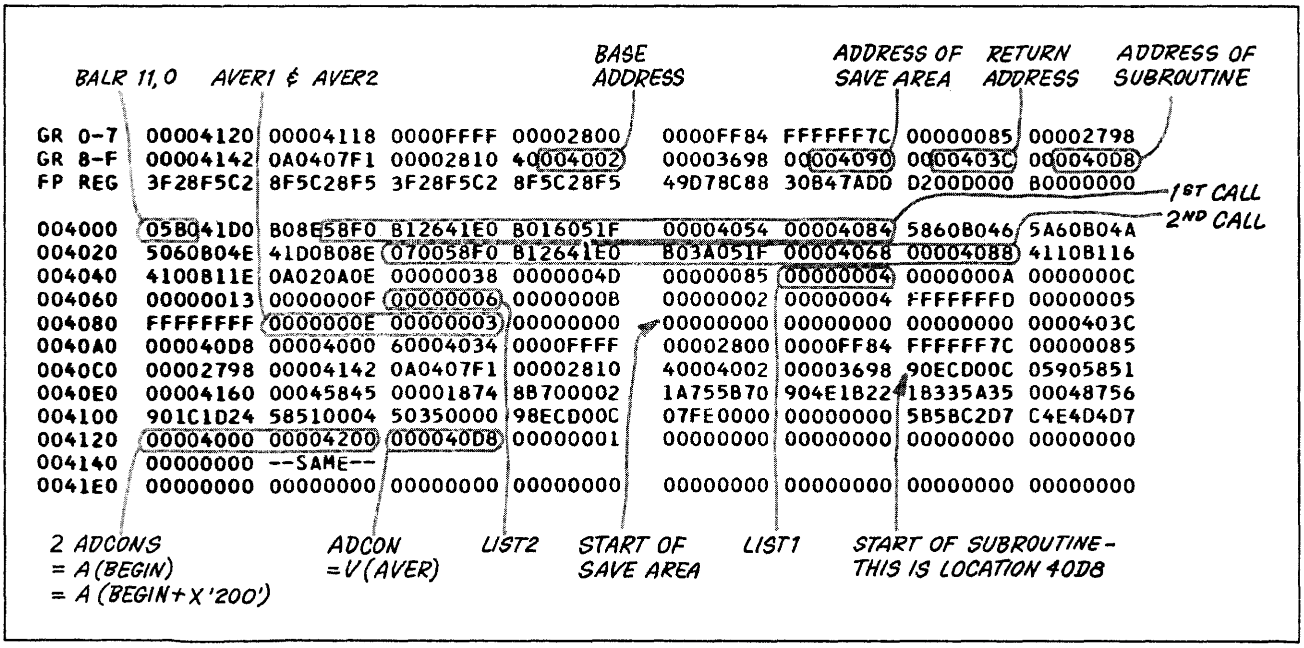
Figures 8-9 and 8-10 show the dumps with the program loaded first at 3000 and then at 4000. Key areas have been labelled to help the reader tie the dump listing to the assembly listing. In Figure 8-9 at location 3000 is 05B0, which is the object code for BALR 11,0. Next is 41D0 B08E, which is for the instruction LA 13,SAVEAREA (see assembly listing). Next is the first CALL. Next is 5860 B046 for the instruction L 6,A. In this way, the entire program was entered into storage and the registers, instruction by instruction and DC by DC, just as it appears in the object code. This dump was printed after execution, however, and therefore the storage areas and registers affected by the instructions have been altered in accordance with the operations they specified.

Let's see what happened when execution of this program began. The linkage editor supplied the starting address 3000, and the program was loaded starting there. BALR was executed; it put the current address from the PSW (by now



```
GR 0-7    00003120 00003118 0000FFFF 00002800   0000FF84 FFFFFF7C 00000085 00002798
GR 8-F    00004142 0A0407F1 00002810 40003002   00003698 00003090 0000303C 000030D8
FP REG    4431F800 8F5C28F5 4431F800 8F5C28F5   4752F1E8 6828F5C1 D200D000 B0000000

003000    05B041D0 B08E58F0 B12641E0 B016051F   00003054 00003084 5860B046 5A60B04A
003020    5060B04E 41D0B08E 070058F0 B12641E0   B03A051F 00003068 00003088 4110B116
003040    4100B11E 0A020A0E 00000038 0000004D   00000085 00000004 0000000A 0000000C
003060    00000013 0000000F 00000006 0000000B   00000002 00000004 FFFFFFFD 00000005
003080    FFFFFFFF 0000000E 00000003 00000000   00000000 00000000 00000000 0000303C
0030A0    000030D8 00003000 60003034 0000FFFF   00002800 0000FF84 FFFFFF7C 00000085
0030C0    00002798 00004142 0A0407F1 00002810   40003002 00003698 90ECD00C 05905851
0030E0    00004160 00045845 00001874 8B700002   1A755B70 904E1B22 1B335A35 00048756
003100    901C1D24 58510004 50350000 98ECD00C   07FE0000 00000000 5B5BC2D7 C4E4D4D7
003120    00003000 00003200 000030D8 00000001   00000000 00000000 00000000 00000000
003140    00000000 --SAME--
0031E0    00000000 00000000 00000000 00000000   00000000 00000000 00000000 00000000
```

Figure 8-9. Hex dump of the program (Figure 8-7) loaded at 3000



```
GR 0-7    00004120 00004118 0000FFFF 00002800   0000FF84 FFFFFF7C 00000085 00002798
GR 8-F    00004142 0A0407F1 00002810 40004002   00003698 00004090 0000403C 000040D8
FP REG    3F28F5C2 8F5C28F5 3F28F5C2 8F5C28F5   49D78C88 30B47ADD D200D000 B0000000

004000    05B041D0 B08E58F0 B12641E0 B016051F   00004054 00004084 5860B046 5A60B04A
004020    5060B04E 41D0B08E 070058F0 B12641E0   B03A051F 00004068 00004088 4110B116
004040    4100B11E 0A020A0E 00000038 0000004D   00000085 00000004 0000000A 0000000C
004060    00000013 0000000F 00000006 0000000B   00000002 00000004 FFFFFFFD 00000005
004080    FFFFFFFF 0000000E 00000003 00000000   00000000 00000000 00000000 0000403C
0040A0    000040D8 00004000 60004034 0000FFFF   00002800 0000FF84 FFFFFF7C 00000085
0040C0    00002798 00004142 0A0407F1 00002810   40004002 00003698 90ECD00C 05905851
0040E0    00004160 00045845 00001874 8B700002   1A755B70 904E1B22 1B335A35 00048756
004100    901C1D24 58510004 50350000 98ECD00C   07FE0000 00000000 5B5BC2D7 C4E4D4D7
004120    00004000 00004200 000040D8 00000001   00000000 00000000 00000000 00000000
004140    00000000 --SAME--
0041E0    00000000 00000000 00000000 00000000   00000000 00000000 00000000 00000000
```

Figure 8-10. Hex dump of the program (Figure 8-7) loaded at 4000 updated to the next available byte, 3002) into register 11 for use as a base address. We see that 3002 is still in register 11. (System/360 uses the rightmost 24 bits of a register for its addressing scheme and ignores the leftmost byte. The 40 here has no effect.)

Next the LA (41D0 B08E) puts the address of SAVEAREA into register 13. This address in the instruction is in the form of a base register and displacement. Will it be correct now that the program has been relocated? The address is arrived at by adding 003002 (the contents of base register 11) and 08E (the displacement) = 003090. The assembly shows 000090 for SAVEAREA, and we shall see that this is where the SAVE macro stores the contents of the registers.

Skipping the CALL, the next instruction (5860B046) loads the value of A into register 6. It looks for A in location 003002 + 046 = 003048. In the assembly, at 000048, we have a DC named A for a fullword of decimal value 56 (this is hex 38), and that is the value at location 003048 in the dump. Continuing in the same fashion with the next instruction (5A60B04A), we see that B (77_{10} or $4D_{16}$) is added to A in register 6. The total in register 6, which is not affected by any later instruction, is 85. This is the hex equivalent of decimal 133, and it is correct. The next instruction (5060B04E) does indeed store the total of 85 in C, location 003002 + 04E = 003050. where a fullword was reserved by a DS.

In this way it is entirely possible to follow the workings of a program. Registers that are not used in a program may have values left over from previous runs—the floating-point registers here, for example. The contents of any register or storage area that is used during a program will be the result of the *last* processing in it before execution of the PDUMP.

To check on the matter of maintaining linkages when a program is relocated, let's look at registers 11 (the base register), 14 (the standard return register), and 15 (the standard entry point register) in both dumps. These are different by the amount 1000, the difference between a relocation factor of 3000 and one of 4000. We also note that, beginning at word 4 in the save area (this program does not use words 1, 2, and 3), we have the contents of registers 14, 15, and 0 through 12 as they were in each case when the SAVE was executed. The code and data produced by the CALL macro can be checked fairly closely by going back to the assembly of the program in Figure 8-5. In this program the CALL generated a V-type address constant for the address of the subroutine, and it was assembled as 000000. The linkage editor supplied the values 30D8 and 40D8 from calculations on ESD and RLD data (although, as we know, a V-type adcon is not really necessary in a single assembly). The PDUMP generated two A-type adcons, and the linkage editor supplied their new values simply by adding the relocation factor (3000 and 4000 in these examples) to the assembled values.

We may also see, from a comparison of the instructions and data constants in both dumps, that the linkage editor does not change any assembled object code except for relocatable address constants. To find a storage address, the CPU simply uses a base-plus-displacement calculation. No matter where in main storage a program is loaded, the relative locations of elements within the program always remain the same.

COMMUNICATION BETWEEN SEPARATE PROGRAMS

The preceding examples have shown how it is possible for a program to keep track of addresses within itself during program relocation. We now turn to the important related question: how do two programs that are assembled separately keep track of addresses in each other, even if they are both relocated by different amounts?

Let us investigate this question in terms of the program in Figures 8-5 and 8-7. This time we shall assemble the main program and the subroutine separately. This method allows subroutines, written and tested separately, to be used in any program. Out of the two assemblies we shall get two object programs which we wish to be able to load at the same time, relocating them by different amounts, and have everything work just as it did before. Once again, we shall use AVER as the entry point into the subroutine, but this time the assembler will have no way of knowing its assembled location. In the single assembly in Figure 8-5, when we wanted to load the address of AVER into register 15, the assembler simply calculated a base-plus-displacement address. If we were to take the main program part of Figure 8-5 and assemble it, AVER would be an undefined symbol, and the assembly could not be completed.

We seem to need some way to say to the assembler: "AVER is a symbol that is *used* in this program but *defined* elsewhere. Whenever you find the symbol AVER, which will be only in address constants, assemble zeros and mark the location as one that will be supplied during the link editing of the object program".

This is precisely what the assembler instruction EXTRN does. We place the EXTRN at the beginning of the program, identify AVER in the operand field, and leave the name field blank. This will cause the action outlined above. The symbol AVER will then be treated, not as an undefined symbol, but as an external symbol defined outside this program.

Figure 8-11 is the assembly listing of the main program. It includes the CALL macro instruction the same as before. An assembler TITLE instruction has been used in order to get identification (in this case, MAIN) into the object deck in columns 73–76, thus distinguishing this object deck from others. Just before the START is the EXTRN. Nothing is printed on the program listing to describe the action of the EXTRN. What it does is to cause an external reference to be listed in the assembler's external symbol

```
       MAIN CALLING PROGRAM FOR SEPARATE ASSEMBLY AND RELOCATION

  LOC  OBJECT COCE    ACDR1 ACCR2  STMT   SOURCE STATEMENT

                                      2          PRINT NOGEN
                                      3          EXTRN AVER
000000                                4 MAIN1    START 0
0C0000 05B0                           5 BEGIN    BALR  11,0
0C0002                                6          USING *,11
000002 41D0 B096            00098     7          LA    13,SAVEAREA         ADDRESS OF SAVEAREA
                                      8          LOAD  SUBR
                                     14          CALL  AVER,(LIST1,AVER1)  LINK TO SUBROUTINE
000020 5860 B04E            00050    21          L     6,A                 OTHER PROCESSING
000024 5A60 B052            00054    22          A     6,B                 X
000028 5060 B056            CC058    23          ST    6,C                 X
00002C 41D0 B096            00098    24          LA    13,SAVEAREA         ADDRESS OF SAVEAREA
                                     25          CALL  AVER,(LIST2,AVER2)  LINK TO SUBROUTINE
                                     32          PDUMP BEGIN,BEGIN+X'200'  CUMP ROUTINE
                                     37          EOJ                       PROGRAM TERMINATION
000050 00000038                      40 A        DC    F'56'
000054 0000004D                      41 B        DC    F'77'
0C0058                               42 C        DS    F
00005C 00000004                      43 LIST1    DC    F'4'                NUMBER OF ENTRIES IN LIST 1
000060 0000000A                      44          CC    F'10'
0C0064 0000000C                      45          DC    F'12'
0C0068 00000013                      46          DC    F'19'
00006C 0000000F                      47          DC    F'15'
0C0070 00000006                      48 LIST2    DC    F'6'                NUMBER OF ENTRIES IN LIST 2
0C0074 0000000B                      49          DC    F'11'
0C0078 00000002                      50          DC    F'2'
0C007C 00000004                      51          DC    F'4'
0C0080 FFFFFFFD                      52          DC    F'-3'
0C0084 00000005                      53          DC    F'5'
0C0088 FFFFFFFF                      54          DC    F'-1'
0C008C                               55 AVER1    DS    F
000090                               56 AVER2    DS    F
000098                               57 SAVEAREA DS    9D
                                     58 *
                                     59 *   THE END OF THE MAIN PROGRAM
                                     60 *
0C0000                               61          END   BEGIN
0000E0 E2E4C2D940404040              62                =CL8'SUBR'
0C00E8 5B5BC2D7C4E4D4D7              63                =CL8'$$BPDUMP'
0C00F0 000C000000000200              64                =A(BEGIN,BEGIN+X'200')
0C00F8 00000000                      65                =V(AVER)
```

Figure 8-11. The same main program assembled separately. The EXTRN assembler instruction and the LOAD macro have been added.

dictionary. When the linkage editor encounters the named symbol in another control section, it will resolve the ESD item. It happens that the V-type address constant =V(AVER) generated by the CALL macro is also entered in the ESD (and the relocation dictionary), so we have some duplication of effort here. Normally, with an EXTRN statement, we would set up the linkage through use of an A-type address constant.

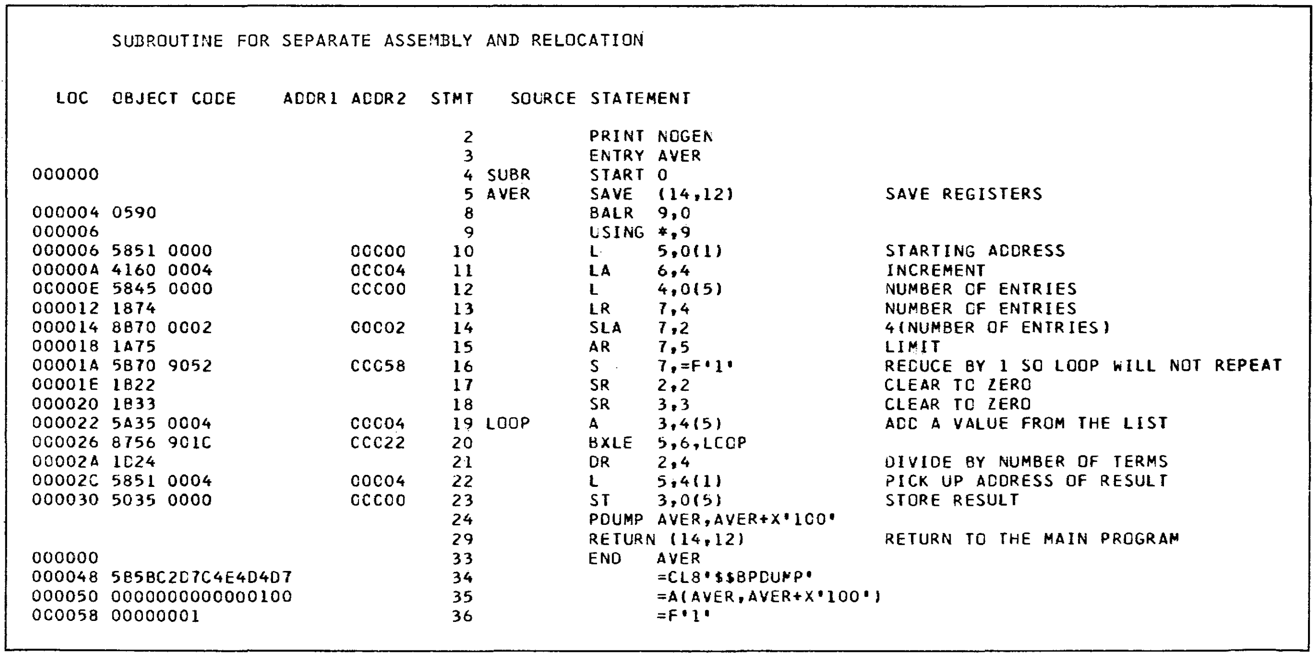
The subroutine (Figure 8-12) has been assembled separately with its own START statement. What about AVER, which is defined here by being used as the name of a statement? Does the symbol have to be identified in any way? The answer is yes, it does. If the subroutine had been assembled just as it was in Figure 8-7, there would be nothing to indicate to the assembler (and later to the linkage editor) that there was anything special about AVER. But there is something special: this symbol is used in the link editing process to supply information missing in the main program. The assembler cannot know this without explicit notification, because we are not assembling the two programs at the same time. What is used is the ENTRY assembler instruction, which says that the symbol given in the operand field is used by some other program, but is defined in this one. AVER also appears in the program in Figure 8-12 as the name of a statement, as required.

If AVER were the name of the program (that is, if it were given in either a START or CSECT statement), it would be listed in the ESD without further ado, and the ENTRY statement would not be necessary. However, we have chosen to name the subroutine SUBR. It is important, for linkage purposes, for the subroutine to have a name. The assembler can process an ENTRY statement that contains a symbol defined in an unnamed control section, but the (DOS) linkage editor cannot process the resulting deck.

Except for the LOAD macro in the main program and another PDUMP in the subroutine, the balance of the two programs in Figures 8-11 and 8-12 is the same as before. The subroutine, we recall, is to be entered twice. The LOAD macro was used to bring in the separate subroutine load module (or program phase), although this might have been done by other means.

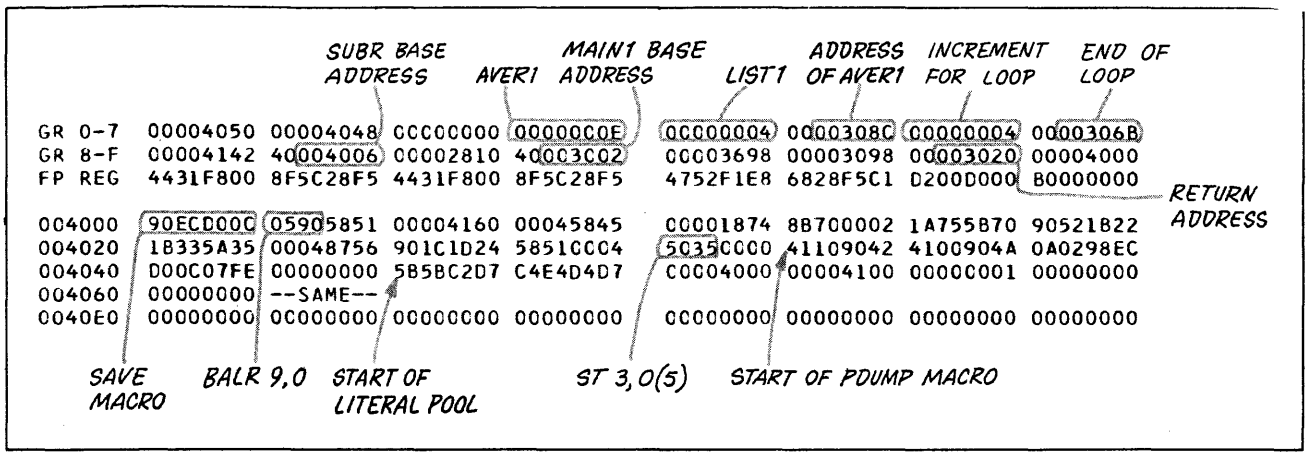
After completion of the assemblies, the two programs were link edited, and the main program was loaded at 3000 and the subroutine at 4000. Execution produced three dump printouts (Figures 8-13, 8-14, and 8-15). These are shown in the order in which they were executed. Figure 8-13 shows a printout of the contents of the registers and the storage area of interest produced by the PDUMP during the first execution of the subroutine, Figure 8-14 during the second execution. Figure 8-15 was produced by execution of the PDUMP in the main program. Various locations in the dumps are identified to help the reader follow the sequence of operations in the registers and main storage, as described below. A careful study of the dumps will help to make clear exactly how communication between programs is maintained and how control is returned to the correct points, even with separate assembly and relocation by different factors. This capacity is not limited to just two programs or control sections. A subroutine may link to another subroutine, which may link to another, etc. Also, one control section can refer to many external symbols and have many entry points from other programs.

The sequence of events, in brief, during execution of the programs in Figures 8-11 and 8-12 was as follows. We can



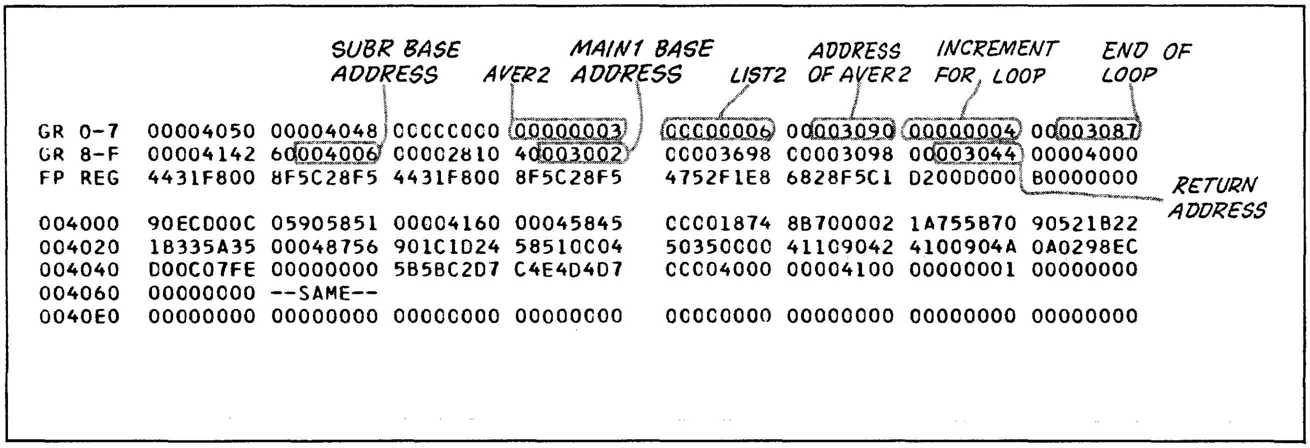
```
         SUBROUTINE FOR SEPARATE ASSEMBLY AND RELOCATION

  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      2          PRINT NOGEN
                                      3          ENTRY AVER
000000                                4 SUBR     START 0
                                      5 AVER     SAVE  (14,12)            SAVE REGISTERS
000004 0590                           8          BALR  9,0
000006                                9          USING *,9
000006 5851 0000            0CC00    10          L     5,0(1)             STARTING ADDRESS
00000A 4160 0004            0CC04    11          LA    6,4                INCREMENT
0C000E 5845 0000            CCC00    12          L     4,0(5)             NUMBER CF ENTRIES
000012 1874                          13          LR    7,4                NUMBER CF ENTRIES
000014 8B70 0C02            00C02    14          SLA   7,2                4(NUMBER OF ENTRIES)
000018 1A75                          15          AR    7,5                LIMIT
00001A 5B70 9052            CCC58    16          S     7,=F'1'            RECUCE BY 1 SO LOOP WILL NOT REPEAT
00001E 1B22                          17          SR    2,2                CLEAR TC ZERO
000020 1B33                          18          SR    3,3                CLEAR TC ZERO
000022 5A35 0004            CCC04    19 LOOP     A     3,4(5)             ADC A VALUE FROM THE LIST
0C0026 8756 901C            CCC22    20          BXLE  5,6,LCCP
00002A 1D24                          21          DR    2,4                DIVIDE BY NUMBER OF TERMS
00002C 5851 0004            00C04    22          L     5,4(1)             PICK UP ADDRESS OF RESULT
000030 5035 0000            CCC00    23          ST    3,0(5)             STORE RESULT
                                     24          PDUMP AVER,AVER+X'1C0'
                                     29          RETURN (14,12)           RETURN TO THE MAIN PROGRAM
000000                               33          END   AVER
000048 5B5BC2D7C4E4D4D7              34                =CL8'$$BPDUMP'
000050 0000000000000100              35                =A(AVER,AVER+X'100')
0C0058 00000001                      36                =F'1'
```

Figure 8-12. The same subroutine assembled separately. The START and ENTRY assembler instructions and the PDUMP macro have been added.



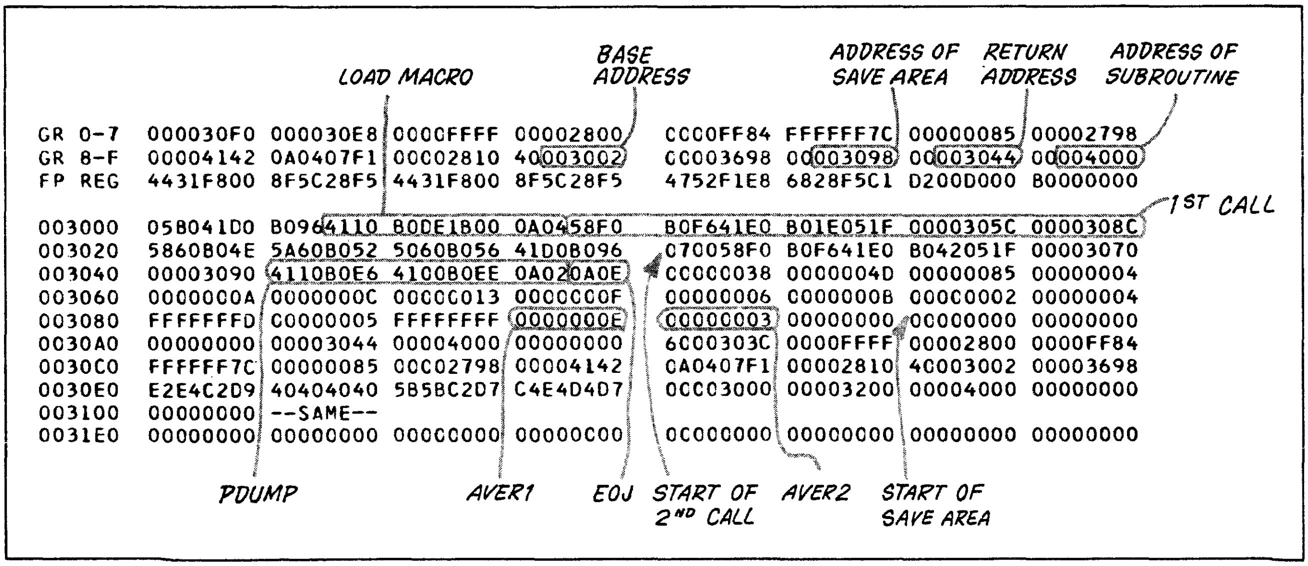
```
GR 0-7   00004050 00004048 00000000 00000000E   00000004 0000308C 00000004 0000306B
GR 8-F   00004142 40004006 00002810 40003002    00003698 00003098 00003020 00004000
FP REG   4431F800 8F5C28F5 4431F800 8F5C28F5    4752F1E8 6828F5C1 D200D000 B0000000

004000   90ECD00C 05905851 00004160 00045845    00001874 8B700002 1A755B70 90521B22
004020   1B335A35 00048756 901C1D24 58510004    50350000 41109042 4100904A 0A0298EC
004040   D00C07FE 00000000 5B5BC2D7 C4E4D4D7    00004000 00004100 00000001 00000000
004060   00000000 --SAME--
0040E0   00000000 00000000 00000000 00000000    00000000 00000000 00000000 00000000
```

Figure 8-13. First dump produced by the subroutine in Figure 8-12, SUBR



```
GR 0-7   00004050 00004048 00000000 00000003    00000006 00003090 00000004 00003087
GR 8-F   00004142 60004006 00002810 40003002    00003698 00003098 00003044 00004000
FP REG   4431F800 8F5C28F5 4431F800 8F5C28F5    4752F1E8 6828F5C1 D200D000 B0000000

004000   90ECD00C 05905851 00004160 00045845    00001874 8B700002 1A755B70 90521B22
004020   1B335A35 00048756 901C1D24 58510004    50350000 41109042 4100904A 0A0298EC
004040   D00C07FE 00000000 5B5BC2D7 C4E4D4D7    00004000 00004100 00000001 00000000
004060   00000000 --SAME--
0040E0   00000000 00000000 00000000 00000000    00000000 00000000 00000000 00000000
```

Figure 8-14. Second dump produced by the subroutine in Figure 8-12, SUBR



```
GR 0-7   000030F0 000030E8 0000FFFF 00002800    0000FF84 FFFFFF7C 00000085 00002798
GR 8-F   00004142 0A0407F1 00002810 40003002    00003698 00003098 00003044 00004000
FP REG   4431F800 8F5C28F5 4431F800 8F5C28F5    4752F1E8 6828F5C1 D200D000 B0000000

003000   05B041D0 B0964110 B0DE1B00 0A0458F0    B0F641E0 B01E051F 0000305C 0000308C
003020   5860B04E 5A60B052 5060B056 41D0B096    070058F0 B0F641E0 B042051F 00003070
003040   00003090 4110B0E6 4100B0EE 0A020A0E    00000038 0000004D 00000085 00000004
003060   0000000A 0000000C 00000013 0000000F    00000006 0000000B 00000002 00000004
003080   FFFFFFFD 00000005 FFFFFFFF 0000000E    00000003 00000000 00000000 00000000
0030A0   00000000 00003044 00004000 00000000    6000303C 0000FFFF 00002800 0000FF84
0030C0   FFFFFF7C 00000085 00002798 00004142    0A0407F1 00002810 40003002 00003698
0030E0   E2E4C2D9 40404040 5B5BC2D7 C4E4D4D7    00003000 00003200 00004000 00000000
003100   00000000 --SAME--
0031E0   00000000 00000000 00000000 00000000    00000000 00000000 00000000 00000000
```

Figure 8-15. Dump produced by the main program in Figure 8-11, MAIN1 follow these events fairly clearly in the dumps, remembering that each dump is produced at just one particular point during processing.

1. The calling program MAIN1 began with execution of the BALR 11,0 that is at 3000, then LA, then the LOAD macro.

2. The LOAD caused the subroutine load module to be entered into core beginning at 4000, and control to be returned to MAIN1.

3. Next, in MAIN1, execution of the CALL macro (see Figure 8-5 for the actions included) branched to and turned control over to the called program SUBR. It also informed SUBR where to find the parameter list and where to place the final result. We note that the last two fullwords in the CALL macro, as shown in the dump in Figure 8-15, are the addresses of LIST1 and AVER1.

4. SUBR was executed once, beginning with the SAVE macro at 4000, which stored the existing contents of all the general registers in the save area beginning at location 3098. Every instruction in SUBR was executed in turn, including repetitions of the loop, through the PDUMP macro. The reader may wish here to go back to the discussion about the subroutine in Figure 8-5 for a detailed review of the processing included.

5. The dump in Figure 8-13 was produced at this point. We note that the averaging calculations in SUBR used registers 2, 3, 4, 5, 6, and 7, and that its base address was in register 9. All this is reflected in the contents of these registers in Figure 8-13.

6. The final instruction in SUBR, the RETURN macro, restored the original contents of the registers from the save area and returned control to MAIN1 at location 3020 (L 6,A in statement 14).

7. MAIN1 then did its processing in register 6, and stored the result (85) at C (the fullword at 3058). It again loaded the save area address in register 13 and executed the second CALL.

8. Beginning at the same location as before (4000), SUBR was executed again in its entirety. The contents of the registers were stored, the registers used, and the contents restored in the same way; and then control was returned to MAIN1. The dump in Figure 8-14 was produced before the registers were restored.

9. This time control was returned to MAIN1 at location 3044, where the PDUMP macro was immediately executed, producing Figure 8-15. Next came the EOJ at location 304E, and, with the Supervisor Call instruction (0A0E), control went back to the control program.

The remaining coding in the dump in Figure 8-15 is not executable, but consists of the constants and storage areas we set up in the original program and also those generated by the various macros. We note that the last item (at 30F8) is a value of 4000 for =V(AVER), the address constant for AVER. This was assembled as 00000000. The value was supplied by the linkage editor.

Two observations can be made from this review of the programs' execution. The first is that program linkage is closely related to the specification of base registers for each program. Throughout execution, the base-plus-displacement addressing system continues to work efficiently on the basis of the values originally assigned by the assembler. Second, communication between programs is easily maintained as long as the data and addresses needed by each is in a known location. When routines are written by different programmers and assembled separately, communication is simplified by use of standard linkage registers for specific functions. Although details differ in certain respects, the necessary linkages can be established similarly in all the operating systems by use of either regular assembler language instructions or macro instructions.

QUESTIONS AND EXERCISES

1a. What functions does BALR 14,15 perform?

b. What functions does BAL 14,SUB perform?

c. What instruction is used to return to the main program after either a. or b. above?

2. Match register numbers with their conventional usage.

REGISTER	
1	a. return address
13	b. address of subroutine entry
14	c. save area address
15	d. address of parameter list

3. List 5 operations that are performed by the CALL macro.

4. The CNOP updates the value in the instruction counter during the first phase of the assembly process. If the counter is now at a value of 000402, what will it be after each of the following:

a. CNOP 0,8
b. CNOP 0,4
c. CNOP 4,8
d. CNOP 6,8
e. CNOP 2,8
f. CNOP 2,4

5a. What is generated by a SAVE (14,12)?

b. What is generated by a RETURN (14,12)?

6a. When a program is branching to an instruction not defined within the confines of that program, what instruction is needed?

b. When a program is to be branched to from another program, what may be used to identify the label of the instruction to be executed first?

Chapter 9: Floating-Point Arithmetic

With the growing use of mathematical and statistical methods to solve business and industrial problems, floating-point arithmetic, long the province of scientists and engineers, is being used more and more by commercial programmers. Although FORTRAN and PL/I are far more efficient for the programmer who wants to solve a complex mathematical problem, floating-point arithmetic can simplify programming in assembler language when the values used in a computation cover a very wide range or are unpredictable. This is so because, in floating-point operations, the machine automatically keeps track of the decimal or binary point and the alignment of intermediate arithmetic results. The programmer need not expend the time and effort required to do this in involved decimal or binary calculations.

Floating-point arithmetic may also save considerable storage space when the values used are either very small or very large. A value up to approximately 7×10^{75} can be expressed in just four bytes. That number is equivalent to 7 followed by 75 zeros. Represented in packed decimal, it would use up over 30 bytes of storage. Since all floating-point numbers are exactly either four or eight bytes in length (at the option of the programmer), he reaps some additional benefits. He does not need to estimate the maximum possible sizes of his data, intermediate results, and final results for purposes of reserving sufficient space. Also, he does not run the risk of losing high-order digits from a register. He can, in fact, perform most calculations almost as directly as he would by hand.

The System/360 floating-point feature performs the same arithmetic calculations as decimal and binary instructions: addition, subtraction, multiplication, and division. There are also similar instructions for comparing, loading, and storing. Just one different kind of instruction is included: the Halve instruction, which has the effect of dividing by two. The entire floating-point instruction set, although it may appear long and complicated (the list is presented later in this chapter), consists only of variations of these basic operations. These variations permit the programmer to choose between (1) long-precision and short-precision numbers, (2) normalized and unnormalized addition or subtraction, and (3) register-to-register and storage-to-register operations.

This brief chapter describes how floating-point numbers are represented in System/360, shows a few examples of floating-point instructions, and explains the new terms used in the preceding paragraph. It is a simplified introduction to the subject for the non-mathematician who may have some curiosity about floating-point operations or who may anticipate using the floating-point feature.

FLOATING-POINT NUMBERS

Floating-point numbers are expressed in a form similar to that commonly used for scientific notation, which is a concise means of expressing very large or very small numbers. For example, the mean distance from the earth to the sun is roughly 93,000,000 miles. In scientific notation, we would give this number as $9.3 \cdot 10^7$. This expression consists of two factors, the significant digits multiplied by a power of 10. The exponent 7 indicates that the base 10 is to be multiplied by itself seven times, and this will give the entire number the proper magnitude. The number base need not be 10, but it is the most common and the easiest for us to understand. The base might be 2 or 8 or 9 or 12 or whatever. In fact, in System/360 it is 16. Let's look at a couple of other examples of scientific notation in base 10. A light year, which is a common term for expressing large distances, represents a distance of 5,880,000,000,000 (or $5.88 \cdot 10^{12}$) miles. A unit that may be used for measuring the wavelength of light is 0.00000001 (or $0.1 \cdot 10^{-7}$) centimeters.

A number in this form of notation is generally, but not always, expressed with one integer to the left of the decimal point. Sometimes it is more convenient to place it elsewhere, either to make some numerical relationship clearer or to simplify computation. In such a case, the exponent is simply increased or decreased by the same number as the number of places the decimal point is moved. The following shows equivalent values for our three examples.

$$93{,}000{,}000 = 9.3 \cdot 10^7 = 93.0 \cdot 10^6 = \underline{.93 \cdot 10^8} = .093 \cdot 10^9$$

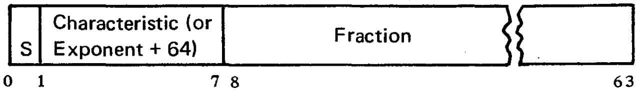
$$5{,}880{,}000{,}000{,}000 = 5.88 \cdot 10^{12} = 5880.0 \cdot 10^9$$
$$= \underline{.588 \cdot 10^{13}} = .00588 \cdot 10^{15}$$

$$0.00000001 = \underline{.1 \cdot 10^{-7}} = 1.0 \cdot 10^{-8} = .01 \cdot 10^{-6}$$
$$= .000001 \cdot 10^{-2}$$

In System/360 a floating-point number, written as a decimal number by the programmer, is converted internally by the machine to a form very much like the underscored examples. In these, the part of each value to the left of the multiplication sign is a fractional quantity, without any whole numbers before the decimal point. Note that the first digit after the decimal point is a nonzero digit. A number in this form is known as a *normalized* number. The final example in each group of examples shows the form of an *unnormalized* number, in which the fraction has one or more high-order zeros.

Floating-point numbers are fixed in length, either a fullword (32 bits) for short precision or a doubleword (64 bits) for long precision. The format of a short floating-point number is as follows:

This format allows 24 bits for the fraction. A long floating-point number has the same arrangement, except that the fraction is 56 bits in length:



A value may be expressed in either short or long form; the short form will give greater speed and use less space, the long will give greater precision.

In either format, the first bit is the sign of the fraction, 0 for plus, 1 for minus, and indicates whether the entire number is positive or negative. The next seven bits are used for the exponent, which in System/360 is called the characteristic by analogy with logarithms. The characteristic also includes a sign (but in an indirect way that will be explained shortly), giving us a plus exponent for large values (over 1) and a minus for small values (below 1). For example, $16^1 = 16$ and $16^{-1} = 1/16$. Similarly, $-16^{+1} = -16$ and $-16^{-1} = -1/16$. The characteristic is a power of 16, of course, not 10, and is a 7-bit binary number with a range of values from 0 to 127_{10}. The fraction is expressed in *hexadecimal digits*, 6 digits for short precision and 14 for long, and in a normalized number its value is between 1/16 and 1. The fraction 1/16 is 0.1 in hexadecimal, with a bit pattern of 0001. Note that normalization applies to hexadecimal digits, not bits, and that the three high-order bits may be zero. The decimal point does not appear in storage, but is understood.

The method devised for indicating the sign of the characteristic in System/360 floating-point numbers is to use what is called excess-64 notation. This avoids the complications of a second sign. As we mentioned, seven bits can represent a range of values from 0 to 127. If 64_{10} is always added to the actual exponent, a range from -64 through +63 can be represented without further indication of a sign. In this scheme, a characteristic of 65 is equivalent to an exponent of +1, 66 to +2, and so on up to 127, which is equivalent to +63. In the low range, a characteristic of 63 is equivalent to an exponent of -1, 62 to -2, and zero to -64. Table 9-1 is given to help in understanding the actual value of some frequently used characteristics.

Table 9-1. Equivalent Values of the Characteristics of Some Floating-Point Numbers

Characteristic		Actual power of 16	Decimal value of characteristic
Dec.	Hex		
68	44	+4	65,536.0
67	43	+3	4,096.0
66	42	+2	256.0
65	41	+1	16.0
64	40	0	1.0
63	3F	-1	0.0625 (or 1/16)
62	3E	-2	0.00390625 (or 1/256)
61	3D	-3	0.000244140625
60	3C	-4	0.0000152587890625

Although the programmer does need to understand the internal form of floating-point numbers, he will never have to do the calculations to break down a value into its hexadecimal exponent and fraction. The machine does that with the greatest of ease. To enter a value into storage in an instruction, the programmer need only define a constant, giving the value in decimal (with or without a decimal point) and specifying its type as E for a short floating-point number or D for a long number. Here are some examples:

```
DC  E'138.25'
DC  E'138'
DC  E'.00138'
DC  E'9.3E+7'
DC  D'9.3E+7'
```

The last two show how the expression $9.3 \cdot 10^7$ is specified as a constant. The E inside the quotation marks simply indicates an exponent.

Figures 9-1 through 9-6, which follow, show various assembly listings of DC entries of floating-point numbers.

Each constant is specified by a decimal number, which we see may be an integer, a fraction, or a mixed number. A decimal point may be placed before, within, or after the number, or it may be omitted. A number without a decimal point is assumed by the machine to be an integer. The number may be signed or unsigned, and a number without a sign is assumed to be positive.

The assembled object code for the floating-point numbers appears at the left in the listings (see Figure 9-1). This is a hexadecimal representation of the actual storage contents. The first two digits represent the sign plus the characteristic. The remaining digits represent the fraction. The decimal point is understood and does not appear in storage. The same numbers are shown in the comments column in Figure 9-2 in a form that is easy to read. A plus or minus sign is printed, depending upon whether the first bit is a zero or a one. The two digits following the sign give the characteristic, the first digit representing the value of the first three bits of the seven-bit characteristic. A decimal point (actually a hexadecimal point) is printed preceding the fraction.

In these figures some of the decimal values specified are integers between 1 and 15. We see that they are represented in floating-point numbers by the corresponding hexadecimal digit in the fraction, with a characteristic of 41. To take the 9 as an example, +41.900000 should be considered as

$$16^1 \cdot \frac{9}{16}.$$

Decimal 16 becomes +42.100000, which we consider as

$$16^2 \cdot \frac{1}{16}.$$

Decimal 32 becomes +42.200000, or

$$16^2 \cdot \frac{2}{16}.$$

```
000148 41100000                                  DC    E'1'
00014C 41200000                                  DC    E'2'
000150 41300000                                  DC    E'3'
000154 41900000                                  DC    E'9'
000158 41A00000                                  DC    E'10'
00015C 41B00000                                  DC    E'11'
000160 41F00000                                  DC    E'15'
000164 42100000                                  DC    E'16'
000168 42110000                                  DC    E'17'
00016C 421F0000                                  DC    E'31'
000170 42200000                                  DC    E'32'
000174 42210000                                  DC    E'33'
000178 42FF0000                                  DC    E'255'
00017C 43100000                                  DC    E'256'
000180 43101000                                  DC    E'257'
000184 43FFF000                                  DC    E'4095'
000188 44100000                                  DC    E'4096'
00018C 44100100                                  DC    E'4097'
```

Figure 9-1. Assembly listing of decimal integers specified as short floating-point constants

The same constants are specified in Figure 9-3 as negative values. Looking at the actual storage values in the object code column, we see that the first digit in these cases is C. This represents the total of the first four bits of our floating-point numbers. In other words, the value of the sign bit (decimal 8) is added to the value of the first three bits of the characteristic. This is of no consequence when the sign is plus and is a zero bit. When it is negative, however, we get binary 1100 0001 (or hexadecimal C1, equal to decimal 12 and 1) for a sign and characteristic of -41. There is still another interesting fact to observe in these representations of negative floating-point numbers. Note that the values are all in true notation, and not in two's complement form as in other types of System/360 arithmetic.

In Figure 9-4 we have some decimal numbers that are fractional and mixed numbers, not integers. Decimal 0.5, for instance, becomes hexadecimal +40.800000, which we consider as

$$16^0 \cdot \frac{8}{16}.$$

The decimal number 1.5 becomes +41.180000, or

$$16^1 \cdot \frac{24}{16^2}.$$

It is interesting to note that the simple decimal number 0.1 is transformed into a nonterminating hexadecimal fraction; there is no exact hexadecimal representation for decimal 0.1. On the other hand, complex-looking decimal fractions that happen to be negative powers of 16 are transformed into particularly simple hexadecimal numbers, as 0.00390625 = +3F.100000.

Figure 9-5 shows a few long floating-point numbers. The scheme is the same, the only difference being the presence of eight additional hexadecimal digits, which make the fraction a total of 14 digits. This permits more accurate representation of numbers that do not have an exact hexadecimal representation and naturally permits much greater precision when arithmetic is performed.

Figure 9-6 shows some examples of short and long floating-point numbers specified by decimal numbers with exponents. The decimal numbers are all in the form of our examples of scientific notation at the beginning of this

```
000190 41100000                          DC     E'1'                +41.100000
000194 41200000                          DC     E'2'                +41.200000
000198 41300000                          DC     E'3'                +41.300000
00019C 41900000                          DC     E'9'                +41.900000
0001A0 41A00000                          DC     E'10'               +41.A00000
0001A4 41B00000                          DC     E'11'               +41.B00000
0001A8 41F00000                          DC     E'15'               +41.F00000
0001AC 42100000                          DC     E'16'               +42.100000
0001B0 42110000                          DC     E'17'               +42.110000
0001B4 421F0000                          DC     E'31'               +42.1F0000
0001B8 42200000                          DC     E'32'               +42.200000
0001BC 42210000                          DC     E'33'               +42.210000
0001C0 42FF0000                          DC     E'255'              +42.FF0000
0001C4 43100000                          DC     E'256'              +43.100000
0001C8 43101000                          DC     E'257'              +43.101000
0001CC 43FFF000                          DC     E'4095'             +43.FFF000
0001D0 44100000                          DC     E'4096'             +44.100000
0001D4 44100100                          DC     E'4097'             +44.100100
```

Figure 9-2. A listing of the same examples as in Figure 9-1, showing them in the comments field in a form that is easy to read

```
000220 C1100000                          DC     E'-1'               -41.100000
000224 C1200000                          DC     E'-2'               -41.200000
000228 C1300000                          DC     E'-3'               -41.300000
00022C C1900000                          DC     E'-9'               -41.900000
000230 C1A00000                          DC     E'-10'              -41.A00000
000234 C1B00000                          DC     E'-11'              -41.B00000
000238 C1F00000                          DC     E'-15'              -41.F00000
00023C C2100000                          DC     E'-16'              -42.100000
000240 C2110000                          DC     E'-17'              -42.110000
000244 C21F0000                          DC     E'-31'              -42.1F0000
000248 C2200000                          DC     E'-32'              -42.200000
00024C C2210000                          DC     E'-33'              -42.210000
000250 C2FF0000                          DC     E'-255'             -42.FF0000
000254 C3100000                          DC     E'-256'             -43.100000
000258 C3101000                          DC     E'-257'             -43.101000
00025C C3FFF000                          DC     E'-4095'            -43.FFF000
000260 C4100000                          DC     E'-4096'            -44.100000
000264 C4100100                          DC     E'-4097'            -44.100100
```

Figure 9-3. The same values shown as negative numbers section. In E'12.78E+8' the decimal value is $12.78 \cdot 10^8$, which we see becomes +48.4C2CBC in hexadecimal. In D'-0.00057E-5' the decimal value is $-0.00057 \cdot 10^{-5}$. This like all the examples we have seen, is converted to a floating-point number in normalized form, that is, with no high-order zeros in the fraction. The fraction is always normalized unless the programmer specifies a decimal number with a scale factor. (Since scaling has not been discussed in this book and is not needed for our elementary computations, it is suggested that a student interested in the subject refer to his assembler specification manual.)

In reviewing the hexadecimal values given by the assembler, we notice that in all the illustrations there are some fractions in which the first digit is 1. Hexadecimal 1, of course, is equivalent to binary 0001. It is important to realize that normalization refers to hexadecimal digits rather than to bits, and a normalized fraction may have as many as three leading zero bits.

```
000268 40800000                DC    E'0.5'              +40.800000
00026C 41180000                DC    E'1.5'              +41.180000
000270 41140000                DC    E'1.25'             +41.140000
000274 41120000                DC    E'1.125'            +41.120000
000278 41110000                DC    E'1.0625'           +41.110000
00027C 411C0000                DC    E'1.75'             +41.1C0000
000280 411E0000                DC    E'1.875'            +41.1E0000
000284 411F0000                DC    E'1.9375'           +41.1F0000
000288 4019999A                DC    E'0.1'              +40.19999A
00028C 3F28F5C3                DC    E'0.01'             +3F.28F5C3
000290 3E418937                DC    E'0.001'            +3E.418937
000294 3D68DB8C                DC    E'0.0001'           +3D.68DB8C
000298 3CA7C5AC                DC    E'0.00001'          +3C.A7C5AC
00029C 4111999A                DC    E'1.1'              +41.11999A
0002A0 40400000                DC    E'0.25'             +40.400000
0002A4 40100000                DC    E'0.0625'           +40.100000
0002A8 3F100000                DC    E'0.00390625'       +3F.100000
```

Figure 9-4. Some fractional and mixed decimal numbers expressed as short floating-point constants

```
0002B8 4110000000000000        DC    D'1'                +41.10000000000000
0002C0 4120000000000000        DC    D'2'                +41.20000000000000
0002C8 4210000000000000        DC    D'16'               +42.10000000000000
0002D0 4980000000000000        DC    D'34359738368'      +49.80000000000000
0002D8 4BB3A73CE5B59000        DC    D'12345678912345'   +4B.B3A73CE5B59000
0002E0 4080000000000000        DC    D'0.5'              +40.80000000000000
0002E8 401999999999999A        DC    D'0.1'              +40.1999999999999A
0002F0 411199999999999A        DC    D'1.1'              +41.1199999999999A
0002F8 C11A86BD134658D5        DC    D'-1.65789516'      -41.1A86BD134658D5
000300 3E10000000000000        DC    D'0.000244140625'   +3E.10000000000000
```

Figure 9-5. Some long floating-point constants

```
000308 484C2CBC                DC    E'12.78E+8'         +48.4C2CBC
00030C 5156BC76                DC    E'1E+20'            +51.56BC76
000310 B819256E                DC    E'-22.87035E-12'    -38.19256E
000314 EABF9572                DC    E'-2.8E+50'         -6A.BF9572
000318 7A25179157C93EC7        DC    D'0.1E+70'          +7A.25179157C93EC7
000320 173BDCF495A9703E        DC    D'0.1E-49'          +17.3BDCF495A9703E
000328 D0891087B9F3A6EC        DC    D'-9.87654321555E+18' -50.891087B9F3A6EC
000330 BA187B375E0424FA        DC    D'-0.00057E-5'      -3A.187B375E0424FA
000338 401F9ADD3739635F        DC    D'12345.6789E-5'    +40.1F9ADD3739635F
```

Figure 9-6. Some decimal values with exponents expressed as floating-point constants

FLOATING-POINT INSTRUCTIONS

Four special registers, used only by the floating-point instructions, are part of the System/360 floating-point feature. They are 64 bits in length and are numbered 0, 2, 4, and 6. All 64 bits are used for long-precision operands and results, and only 32 bits for short-precision (except for the product in Multiply). Use of registers for floating-point arithmetic avoids the many operations that would otherwise be necessary for storing and loading results and operands. All floating-point operations are register-to-register (RR) or storage-to-register (RX), and most of the instructions are available with a choice of either format.

All floating-point instructions are also available with a choice between the use of long or short numbers. In addition, the programmer may select an Add or Subtract instruction in the execution of which the intermediate and final results are normalized or are not normalized. All these choices mean a long list of instructions in the floating-point instruction set (as we see in Table 9-2, there are eight separate Add instructions), but the basic functions are simply to Add, Subtract, Multiply, Divide, Halve, Compare, Store, and Load. The Load instructions also provide the programmer with the ability to control the signs of operands. Note that the mnemonics of instructions for long precision are distinguished by the letter D, and for short precision by the letter E. In Add Unnormalized and Subtract Unnormalized, these change to W and U.

Perhaps the best way to get an idea of how the instructions actually operate is to study an example. Figure 9-7 is an assembly listing of a program to evaluate the following formula, using short precision throughout.

$$Y = \left(\frac{A + \frac{B - C}{2}}{3.17 - 2D} \right)^2$$

The first processing instruction is Load Short (LE), which places the value of D in floating-point register 2. The fact that the 2 in this instruction refers to a floating-point register, rather than to a general purpose register, is implied in the operation code; floating-point is understood by the assembler when it encounters the code LE. This short operation will load the left half of the double-length register, leaving the low-order half unchanged. Any previous value in the low-order 32 bits, will ordinarily have no significant effect on later operations.

The second instruction multiplies the contents of floating-point register 2, which we just loaded, by the constant 2 in floating-point form. The result is left in the same register, destroying the previous contents. No other register is involved, in contrast to fixed-point multiplication. The lower half of the floating-point register is involved, however, because the entire register is used for the result of a Multiply operation. In short precision, the fraction of the product has 14 hexadecimal digits, of which at least two are always zero.

Table 9-2. Instruction Set for the System/360 Floating-Point Feature

Name	Mnemonic	Format
Load (Long)	LDR	RR
Load (Long)	LD	RX
Load (Short)	LER	RR
Load (Short)	LE	RX
*Load and Test (Long)	LTDR	RR
*Load and Test (Short)	LTER	RR
*Load Complement (Long)	LCDR	RR
*Load Complement (Short)	LCER	RR
*Load Positive (Long)	LPDR	RR
*Load Positive (Short)	LPER	RR
*Load Negative (Long)	LNDR	RR
*Load Negative (Short)	LNER	RR
*Add Normalized (Long)	ADR	RR
*Add Normalized (Long)	AD	RX
*Add Normalized (Short)	AER	RR
*Add Normalized (Short)	AE	RX
*Add Unnormalized (Long)	AWR	RR
*Add Unnormalized (Long)	AW	RX
*Add Unnormalized (Short)	AUR	RR
*Add Unnormalized (Short)	AU	RX
*Subtract Normalized (Long)	SDR	RR
*Subtract Normalized (Long)	SD	RX
*Subtract Normalized (Short)	SER	RR
*Subtract Normalized (Short)	SE	RX
*Subtract Unnormalized (Long)	SWR	RR
*Subtract Unnormalized (Long)	SW	RX
*Subtract Unnormalized (Short)	SUR	RR
*Subtract Unnormalized (Short)	SU	RX
*Compare (Long)	CDR	RR
*Compare (Long)	CD	RX
*Compare (Short)	CER	RR
*Compare (Short)	CE	RX
Halve (Long)	HDR	RR
Halve (Short)	HER	RR
Multiply (Long)	MDR	RR
Multiply (Long)	MD	RX
Multiply (Short)	MER	RR
Multiply (Short)	ME	RX
Divide (Long)	DDR	RR
Divide (Long)	DD	RX
Divide (Short)	DER	RR
Divide (Short)	DE	RX
Store (Long)	STD	RX
Store (Short)	STE	RX

---

*Operation sets condition code.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                      1          PRINT NOGEN
000000                                2 SHORTFP  START 0
000000 05B0                           3 BEGIN    BALR 11,0
000002                                4          USING *,11
000002 7820 B032            00034     5          LE    2,D            LOAD FLOATING POINT REGISTER 2 WITH D
000006 7C20 B036            00038     6          ME    2,FTWO         MULTIPLY D IN REGISTER 2 BY 2
00000A 3322                           7          LCER  2,2            REVERSE SIGN OF PRODUCT
00000C 7A20 B03A            0003C     8          AE    2,CON1         ADD CONSTANT 3.17
000010 7840 B02A            0002C     9          LE    4,B            LOAD FLOATING POINT REGISTER 4 WITH B
000014 7B40 B02E            00030    10          SE    4,C            SUBTRACT C
000018 3444                          11          HER   4,4            USE HALVE INSTRUCTION TO DIVIDE BY 2
00001A 7A40 B026            00028    12          AE    4,A            ADD A
00001E 3D42                          13          DER   4,2            DIVIDE NUMERATOR BY DENOMINATOR
000020 3C44                          14          MER   4,4            SQUARE THE QUOTIENT
000022 7040 B03E            00040    15          STE   4,Y            STORE THE FINAL RESULT
                                     16          EOJ
000028                               19          DS    0F
000028 41123456                      20 A        DC    E'1.1377772805'
00002C 43356800                      21 B        DC    E'854.50'
000030 43252600                      22 C        DC    E'594.3750'
000034 3E2D3EFD                      23 D        DC    E'6.904E-4'
000038 41200000                      24 FTWO     DC    E'2'
00003C 4132B852                      25 CON1     DC    E'3.17'
000040                               26 Y        DS    F
000000                               27          END   BEGIN
```

Figure 9-7. Assembly listing of a program to perform simple computations in short floating-point arithmetic

In the execution of a Multiply instruction, the machine normalizes both operands, if the fractions have leading zeros, before any arithmetic is performed. This is done by shifting the fraction left until the leftmost hexadecimal digit is a nonzero digit and reducing the characteristic by the number of shifts required. When this is done before the arithmetic process (as it is in both Multiply and Divide), the action is called *prenormalization*. With both operands prenormalized, the product will either be normalized already or have at most one leading zero. In the latter case, the product fraction is shifted left one hexadecimal position to *postnormalize* it, and the product characteristic is reduced by one.

In floating-point multiplication, the arithmetic process is very simple and follows the familiar rules for exponents. It consists of adding the characteristics and multiplying the fractions. To illustrate the procedure, let's consider a simple problem in base 10: to multiply 12,300 by 60. Expressed with decimal exponents, this is $(.123 \cdot 10^5) \cdot (.6 \cdot 10^2)$. Multiplying the fractions, we get $.123 \cdot .6 = .0738$. Adding the exponents, we get $10^{5+2} = 10^7$. Together, they give $.0738 \cdot 10^7 = 738{,}000$. In System/360, of course, the machine also has to subtract 64 from the characteristic of an intermediate product because, with both operands in the excess-64 notation, adding the characteristics gets the extra 64 into the product twice instead of once.

For those who wish to follow the arithmetic in the program example, the details are given in Figure 9-8. Each line shows the contents of the two registers used for computation after the execution of each of the floating-point instructions. The operation codes are given in the left-hand column. The program used for this output specified the addition of a point in printing the hexadecimal register contents. The decimal equivalents are in the usual form for floating-point numbers and show the true value of the exponents. The decimal numbers are not all exact equivalents, because exact equivalents of fractional quantities often do not exist in base 10 and base 16. Inspection will show that these discrepancies are small for most practical purposes; they can be made much smaller by the use of long precision, as will be seen later.

We noted before that the product fraction of a short-precision Multiply is 14 hexadecimal digits in length, including some trailing zeros. Normally, after the ME operation in Figure 9-8, we would expect to find at least some nonzero digits in the low-order half of register 2. In this case, however, the two fractions that are multiplied yield only six significant digits (.2D3EFD x .200000 = .5A7DFA 000000), so the low-order half of the register contains eight zeros. The more usual situation can be seen in register 4 after execution of the MER instruction.

The next instruction in our program, the Load Complement (RR), reverses the sign of the product as written here. (The instruction can also be used with two different register numbers.) It would of course be acceptable programming practice to have stored the constant 2 as a negative number.

Now we add the constant 3.17, using an Add Normalized instruction. Floating-point addition starts with a comparison of the two operand characteristics; if they are the same, addition of the fractions takes place immediately. Otherwise, the fractional part of the number with the smaller characteristic is shifted right, as many places as the difference in characteristics, until they agree. When this is done, the decimal points (hexadecimal points, really) are "lined up", as addition requires. The fractions are then added. The larger of the two characteristics becomes the "provisional" characteristic of the sum; we say provisional because it may have to be adjusted for a possible overflow carry in the fraction or for postnormalization.

If the addition caused overflow of the fraction, the result fraction is now shifted right one place and the characteristic accordingly increased by 1. On the other hand, the addition might have resulted in a sum with leading zeros, which would happen if the operands were of about the same size but of opposite sign, and the characteristic would be decreased in the process of normalization.

If these actions cause the characteristic to go below or above the range of zero to 127, *exponent underflow* or *overflow* is signaled, and normally a program interruption occurs. If addition or subtraction results in an all-zero fraction, the loss of significance is complete, which may in some cases destroy the validity of all results of the computation. If this happens without the problem originator's knowledge, he may place confidence in results that are in fact meaningless. For this reason, System/360 provides a warning in the form of a *significance exception*, and a program interruption occurs, enabling the programmer to cope with the situation in a subroutine. For certain types of data, the programmer may wish to prevent an interruption, and he can do so in case of an exponent underflow or a significance exception. In case of exponent overflow, however, the interrupt action cannot be overridden.

With the values that have been entered in our sample program, there will be no loss of significance or other exceptions. To review what has been covered in the program so far, we have evaluated the denominator within the parentheses. We leave the result in floating-point register 2 and turn now to an evaluation of the numerator.

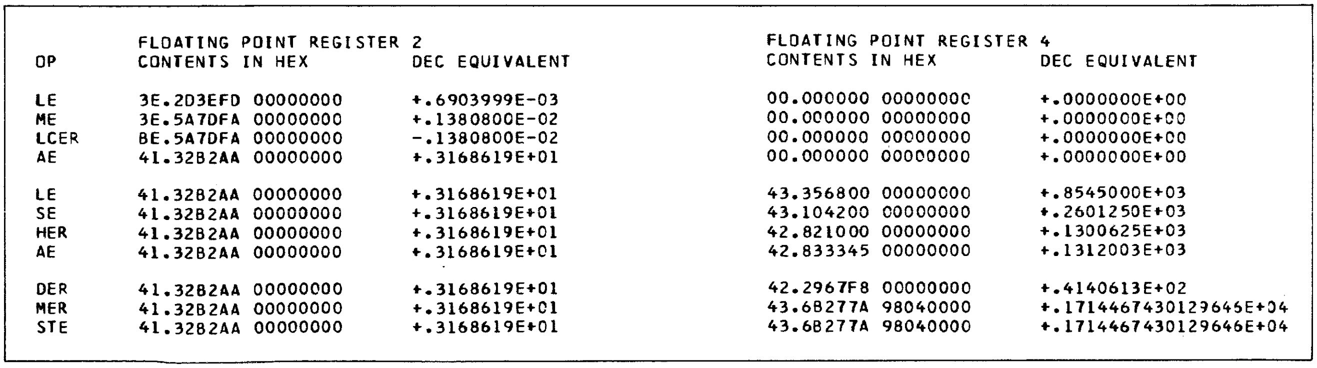
In loading B and subtracting C, instructions are used that are now familiar. Floating-point subtraction is just like addition, with the sign of the second operand reversed before adding the fractions. Since both addition and subtraction are completely algebraic, and since either one can involve any of the four combinations of signs of the operands, they are truly very similar.

The division by 2 is handled in a rather different way from what one might expect and illustrates an interesting member of the floating-point instruction set. The Halve instruction (HER) divides the second operand by 2 and places the result in the first operand; both registers are the same here, as they so often are in using the RR-format instructions. What actually happens is that the fraction part is shifted right by one *binary* place, which is equivalent to dividing by 2. If the consequence of this is an intermediate result with all zeros in the first four bits of the fraction, the final result is postnormalized.

The next instruction is another Add Normalized, the details of which we discussed before. So far, however, we have not mentioned a feature of System/360 floating-point operations that is designed to increase the significance of final results. It is called the guard digit. As we know, the fractions in final results have six hexadecimal digits in short precision, and 14 in long precision. Intermediate results may have one additional significant low-order digit in the Add, Subtract, Compare, Halve, and Multiply operations, which participates in postnormalization of final results. This extra digit materializes when right-shifting into the guard digit position occurs during the operations named, as in Adding, for example, when the two operands are lined up with each other. When final results are subsequently shifted left in the process of postnormalization, the guard digit is simply included in the move.

At this point in our problem, we have the numerator in floating-point register 4 and the denominator in register 2. A Divide (RR, Short) places the quotient in register 4. Floating-point division works as follows. Both operands are prenormalized. Division of the fractions yields the quotient fraction. The characteristic of the denominator (or divisor) is subtracted from that of the numerator (or dividend), and then 64_{10} is added to get the characteristic back into excess-64 form. The arithmetic process here is similar, but opposite, to the Multiply instruction. In short-precision Divide, the low-order half of the registers is ignored, and the fraction of the result is six digits in length. Division of two normalized six-digit fractions will always yield either six or seven digits, never more or less. Postnormalization is never necessary, but the quotient fraction may need to be shifted right by one position and the characteristic increased correspondingly by 1.

Our program now requires us to square the result of the



```
          FLOATING POINT REGISTER 2                            FLOATING POINT REGISTER 4
OP        CONTENTS IN HEX          DEC EQUIVALENT              CONTENTS IN HEX          DEC EQUIVALENT

LE        3E.2D3EFD 00000000       +.6903999E-03               00.000000 00000000       +.0000000E+00
ME        3E.5A7DFA 00000000       +.1380800E-02               00.000000 00000000       +.0000000E+00
LCER      BE.5A7DFA 00000000       -.1380800E-02               00.000000 00000000       +.0000000E+00
AE        41.32B2AA 00000000       +.3168619E+01               00.000000 00000000       +.0000000E+00

LE        41.32B2AA 00000000       +.3168619E+01               43.356800 00000000       +.8545000E+03
SE        41.32B2AA 00000000       +.3168619E+01               43.104200 00000000       +.2601250E+03
HER       41.32B2AA 00000000       +.3168619E+01               42.821000 00000000       +.1300625E+03
AE        41.32B2AA 00000000       +.3168619E+01               42.833345 00000000       +.1312003E+03

DER       41.32B2AA 00000000       +.3168619E+01               42.2967F8 00000000       +.4140613E+02
MER       41.32B2AA 00000000       +.3168619E+01               43.6B277A 98040000       +.1714467430129645E+04
STE       41.32B2AA 00000000       +.3168619E+01               43.6B277A 98040000       +.1714467430129646E+04
```

Figure 9-8. The contents of floating-point registers 2 and 4 after execution of each of the short-precision instructions in the program in Figure 9-7 division, which is standing in register 4. A Multiply (RR, Short), in which the quantity in register 4 is specified for both operands, does the job. Finally, a Store puts the result in the fullword storage location Y.

Figure 9-9 shows a listing of the same program, with identical decimal values, rewritten to do all processing in long precision. Step-by-step changes in the contents of the registers may be seen in Figure 9-10. Here the full capacity of the registers is used in each operation, and the increase in precision of the arithmetic results can readily be seen. Except for the length of operands and results and the fact that in short precision the low-order halves of the registers are generally ignored, there is no difference in the execution of the instructions for long and short precision.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT

                                     1           PRINT NOGEN
000000                               2 LONGFP    START 0
000000 05B0                          3 BEGIN     BALR  11,0
000002                               4           USING *,11
000002 6820 B03E            00040    5           LD    2,D                  LOAD FLOATING POINT REGISTER 2 WITH D
000006 6C20 B046            00048    6           MD    2,FTWO               MULTIPLY D IN REGISTER 2 BY 2
00000A 2322                          7           LCDR  2,2                  REVERSE SIGN OF PRODUCT
00000C 6A20 B04E            00050    8           AD    2,CON1               ADD CONSTANT 3.17
000010 6840 B02E            00030    9           LD    4,B                  LOAD FLOATING POINT REGISTER 4 WITH B
000014 6B40 B036            00038   10           SD    4,C                  SUBTRACT C
000018 2444                         11           HDR   4,4                  USE HALVE INSTRUCTION TO DIVIDE BY 2
00001A 6A40 B026            00028   12           AD    4,A                  ADD A
00001E 2D42                         13           DDR   4,2                  DIVIDE NUMERATOR BY DENOMINATOR
000020 2C44                         14           MDR   4,4                  SQUARE THE QUOTIENT
000022 6040 B056            00058   15           STD   4,Y                  STORE THE FINAL RESULT
                                    16           EOJ
000028                              19           DS    0D
000028 41123455F31E11B0             20 A         DC    D'1.1377772805'
000030 4335680000000000             21 B         DC    D'854.50'
000038 4325260000000000             22 C         DC    D'594.3750'
000040 3E2D3EFD6BD10972             23 D         DC    D'6.904E-4'
000048 4120000000000000             24 FTWO      DC    D'2'
000050 4132B851EB851EB8             25 CON1      DC    D'3.17'
000058                              26 Y         DS    D
000000                              27           END   BEGIN
```

Figure 9-9. Assembly listing of the same program as in Figure 9-7, modified to perform all computations in long floating-point arithmetic

```
        FLOATING POINT REGISTER 2                                 FLOATING POINT REGISTER 4
OP      CONTENTS IN HEX       DEC EQUIVALENT                      CONTENTS IN HEX        DEC EQUIVALENT

LD      3E.2D3EFD 6BD10972    +.6903999999999997E-03              00.000000 00000000     +.0000000000000000E+00
MD      3E.5A7DFA D7A212E4    +.1380799999999998E-02              00.000000 00000000     +.0000000000000000E+00
LCDR    BE.5A7DFA D7A212E4    -.1380799999999998E-02              00.000000 00000000     +.0000000000000000E+00
AD      41.32B2AA 0BD7A496    +.3168619199999997E+01              00.000000 00000000     +.0000000000000000E+00

LD      41.32B2AA 0BD7A496    +.3168619199999997E+01              43.356800 00000000     +.8545000000000000E+03
SD      41.32B2AA 0BD7A496    +.3168619199999997E+01              43.104200 00000000     +.2601250000000000E+03
HDR     41.32B2AA 0BD7A496    +.3168619199999997E+01              42.821000 00000000     +.1300625000000000E+03
AD      41.32B2AA 0BD7A496    +.3168619199999997E+01              42.833345 5F31E11B     +.1312002772804998E+03

DDR     41.32B2AA 0BD7A496    +.3168619199999997E+01              42.2967F8 86170C30     +.4140613592207608E+03
MDR     41.32B2AA 0BD7A496    +.3168619199999997E+01              43.6B277D 4E08C861     +.1714468091997438E+04
STD     41.32B2AA 0BD7A496    +.3168619199999997E+01              43.6B277D 4E08C861     +.1714468091997438E+04
```

Figure 9-10. The contents of floating-point registers 2 and 4 after execution of each of the long-precision instructions in the program in Figure 9-9

QUESTIONS AND EXERCISES

1. Write the DC instructions for the following short floating-point numbers:

3.14159265
-2.78
38754×10^{6}
$.00000278$
$-.000236 \times 10^{-7}$

2. Write the DC instructions for the following long floating-point numbers:

3.141592653589793
-2.78
-0.003×10^{-3}
3.8×10^{30}
0.000000008

3. Show the hexadecimal form that the following DC entries will generate in storage. (Note that 16777216 equals 16^{6} and that $.59604644 \times 10^{-7}$ equals 16^{-7}.)

```
DC   E′32′
DC   D′32′
DC   E′16777216′
DC   E′.59604644E–7′
DC   E′–.59604644E–7′
DC   E′–16777216′
```

4. After execution of each of the following sets of instructions, what will be in the registers used?

```
a.    LE     2,A
      AE     2,B
      HER    4,2
```

given A = 41789ABC and B = 41876544 in hexadecimal short floating-point.

b. What would be the results of the same instructions if A = 41200000 and B = 44600044?

```
c.    LE     6,A
      SR     6,6
      .      .
      .      .
    A DC     E′15′
d.    L      3,A
      A      3,B
      .      .
      .      .
    A DC     E′1.0′
    B DC     X′01000000′
```

5. Write a program segment to calculate the value of X in short floating-point arithmetic and put it into storage:

$$X = \frac{A - (B \times C)}{A + (B \times C)}$$

Chapter 3: Fixed-Point Arithmetic

1. Fullword
2. Receives
3. Sends

   Exception
4. No. The first operand must specify an even-numbered register for an even-odd pair.
5. An even-numbered register of an even-odd pair that contains the dividend

   Divisor

   The quotient is in the odd-numbered register.

   The remainder is in the even-numbered register.
6.

```
          START   256
BEGIN     BALR    11,0
          USING   *,11
          L       2,XANDY
          SRDL    2,12
          SRL     3,20
          ST      2,X
          STH     3,Y
```

(Continued in next column)

```
          EOJ
XANDY     DS      F
X         DS      F
Y         DS      H
          END     BEGIN
```

7. (c) Condition code is 1 or 3.
8. BC 15,NEWONE

   The extended mnemonic equivalent is B NEWONE.
9. LM 2,5,X1
10. SR 5,5
11. It will be the sum of the contents of register 3 (the base register), register 11 (the index register), and the displacement.
12. BXLE 5,6,NEWONE

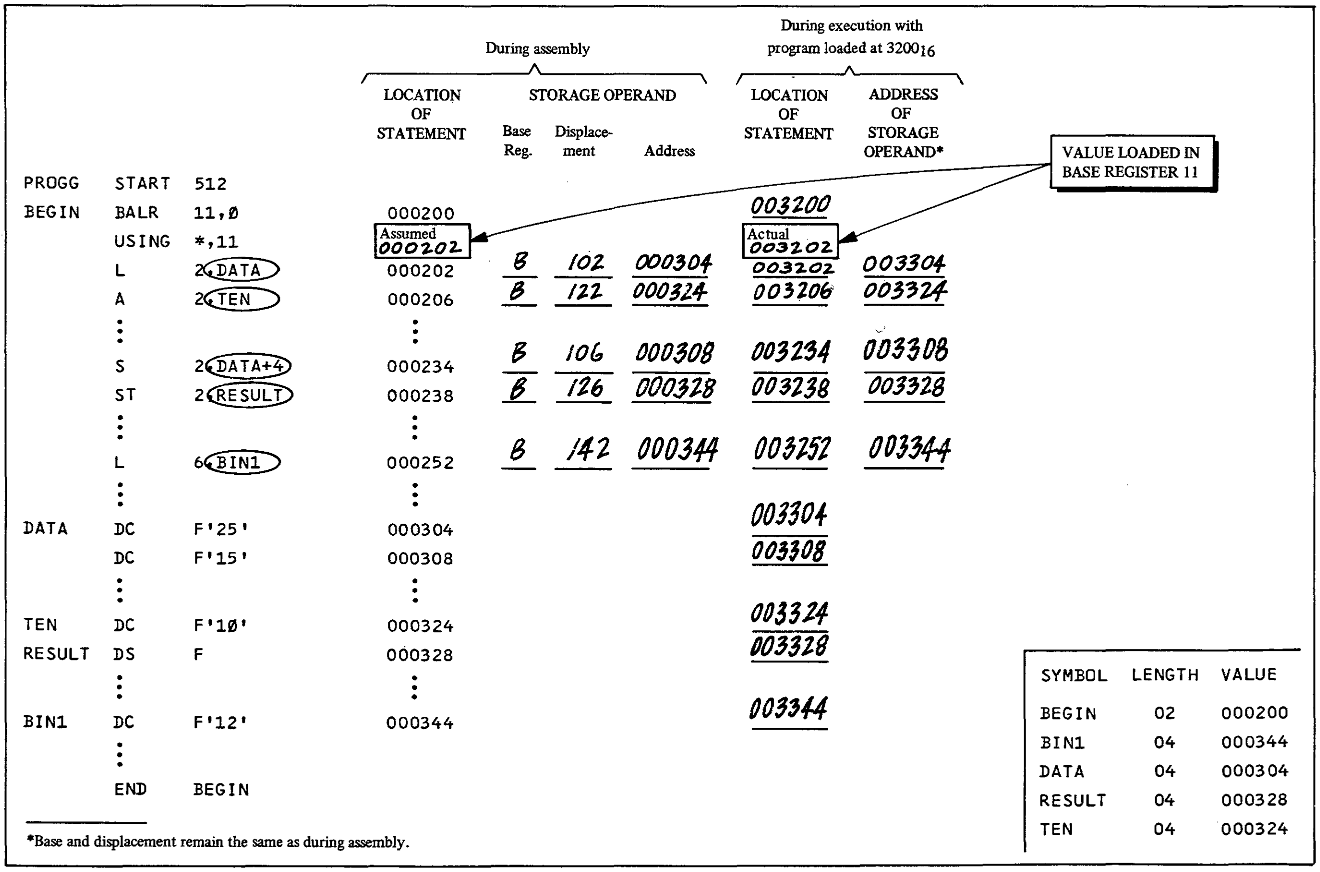
Chapter 4: Programming with Base Registers and the USING Instruction

1a. USING *,11

b. BALR 11,0

2, 3, and 4. See illustration below.

5. See illustration on next page.



			During assembly				During execution with program loaded at 3200_{16}	
			LOCATION OF STATEMENT	STORAGE OPERAND Base Reg.	Displacement	Address	LOCATION OF STATEMENT	ADDRESS OF STORAGE OPERAND*
PROGG	START	512						
BEGIN	BALR	11,0	000200				003200	
	USING	*,11	Assumed 000202				Actual 003202	
	L	2,DATA	000202	B	102	000304	003202	003304
	A	2,TEN	000206	B	122	000324	003206	003324
	⋮		⋮					
	S	2,DATA+4	000234	B	106	000308	003234	003308
	ST	2,RESULT	000238	B	126	000328	003238	003328
	⋮		⋮					
	L	6,BIN1	000252	B	142	000344	003252	003344
	⋮		⋮					
DATA	DC	F'25'	000304				003304	
	DC	F'15'	000308				003308	
	⋮		⋮					
TEN	DC	F'10'	000324				003324	
RESULT	DS	F	000328				003328	
	⋮		⋮					
BIN1	DC	F'12'	000344				003344	
	⋮							
	END	BEGIN						

SYMBOL	LENGTH	VALUE
BEGIN	02	000200
BIN1	04	000344
DATA	04	000304
RESULT	04	000328
TEN	04	000324

---

*Base and displacement remain the same as during assembly.

Answers to questions 2, 3, and 4

			During assembly				During execution with program loaded at 1000_{16}	
			LOCATION OF STATEMENT	STORAGE OPERAND Base Reg.	Displace-ment	Address	LOCATION OF STATEMENT	ADDRESS OF STORAGE OPERAND*
PROGH	START	0						
BEGIN	BALR	11,0	000000				001000	
	USING	FIRST,11						
FIRST	BC	15,SKIP	000002				001002	
DATA	DC	F'3472'	000008				001008	
	⋮		⋮					
BASE1	DC	A(FIRST+4096)	000024				001024	
BASE2	DC	A(FIRST+8192)	000028				001028	
	⋮		⋮					
SKIP	L	10,BASE1	000104	B	022	000024	001104	001024
	USING	FIRST+4096,10						
	L	9,BASE2	000108	B	026	000028	001108	001028
	USING	FIRST+8192,9						
	⋮		⋮					
	BC	15,CK8	001504	9	902	002904	002504	003904
	⋮		⋮					
LOOP	A	4,DATA	001898	B	006	000008	002898	001008
	⋮		⋮					
LOOPB	S	5,DATA	002204				003204	
	⋮		⋮					
	BC	8,LOOP	002508	A	896	001898	003508	002898
	⋮		⋮					
CK8	BC	8,LOOPB	002904	9	202	002204	003904	003204
	END	BEGIN						

REGISTER	VALUE LOADED INTO BASE REGISTERS During assembly (assumed)	During execution (actual)
11	000002	001002
10	001002	002002
9	002002	003002

SYMBOL	VALUE
BASE1	000024
BASE2	000028
BEGIN	000000
CK8	002904
DATA	000008
FIRST	000002
LOOP	001898
LOOPB	002204
SKIP	000104

---

*Base and displacement remain the same as during assembly.

Answers to questions 5

Chapter 5: Decimal Arithmetic

1a. CON3 DC PL5'3'
b. 000000003C
2. Assembler
Data definitions
Programmer
3. Equal to
4. One less than
5a. The multiplicand in the low-order positions and zeros in the high-order positions
b. In the storage area specified by the first operand

6a.

00	02	48	9C	10	3C
158	159	15A	15B	15C	15D

b. 15A
7. Storage area containing the dividend
Divisor
The quotient will be in the left portion of the dividend area, and the remainder in the right portion.

8a. SOURCE 66 55 44 33 22 11
DEST 11 22 66 55 44 6S
b. SOURCE 66 55 44 33 22 11
DEST 11 22 33 4S 55 6S
c. SOURCE 66 55 44 33 22 11
DEST 00 00 00 04 43 3S

9. No. The ZAP instruction, as all the decimal arithmetic instructions and the decimal compare instructions, requires a legitimate sign in the low-order byte of the "sending" field.

10a. MVN RESULT+5(1),FACTOR+4
MVO RESULT,FACTOR(4)
b. MVN FACTOR+3(1),FACTOR+4
ZAP RESULT,FACTOR(4)

11a. SI
b. NI HOLD,X'00'
c. NI HOLD+3,X'0F'

12. In both cases, each bit position of the referenced storage operand is analyzed against the corresponding bit position of the immediate portion of the instruction. The storage byte referenced by the first operand, after execution will be:

a. For the And Immediate instruction, a 1 in each bit position in which both operands had 1s, and zeros elsewhere.

b. For the Or Immediate instruction a 1 in the bit positions in which either or both operands had a 1, and a zero where both operands had zeros.

13. Packed decimal
14. PACK
15. UNPK (Unpack)
16a. DC F'578' or DC H'578'
b. DC ZL3'578'
c. DC PL2'578'

17. There are at least four ways to write the DC statement. Keep in mind that 4B is the hexadecimal equivalent of 75_{10}.

a. DC F'75' would generate the 4-byte constant: 00 00 00 4B.

b. DC H'75' would generate the 2-byte constant: 00 4B.

c. DC X'4B' would generate the 1-byte constant: 4B.

The advantage of methods a and b over method c is that the programmer does not have to convert from decimal to hexadecimal. A disadvantage is that more space is used than is perhaps necessary.

d. The statement DC FL1'75' would remove this disadvantage since the characters L1 specify that the length (L) of the constant is to be 1 byte. Thus a 1-byte field of 4B would be generated. A point to remember is that when a length is stated for an F-type constant no boundary alignment is performed by the assembler.

18. IC 6,OLD
19. STC 6,OLD

20a. No. MASK is not located on a fullword boundary. The N instruction requires the operand in storage to be on a fullword boundary.

b. The statement DS 0F could be inserted immediately before the DC defining MASK.

c. DC F'15'

Chapter 6: Logical Operations on Characters and Bits

1. XI KEY,15 (immediate data in decimal)
XI KEY,X'0F' (immediate data in hexadecimal)
XI KEY,B'00001111' (immediate data in binary)
2. TM ADDR,X'30'
BC 5,ANIMAL
3. TM ADDR,X'06'
BC 4,LIST2

There are many acceptable ways of performing tests such as 2 and 3. The TM instruction, where it can be used, has the advantages of leaving storage unchanged and obviating the need for registers or work areas.

4a. 05
b. 2C (the final C is the code for a plus sign)
c. 43
5. 8 bits, 1 byte
6. 2048 bits, 256 bytes

7. (b) Alphameric characters. Despite their plausibility, a and c are not correct in the general case because of possible difficulty with sign codes.

8. (c) An inequality. All codes are valid.

9. (a) 5, (b) 2, (c) 3 plus the contents of general register 1, (d) the computed effective address for FIELD, *not* the word stored at that address

10. Among the many ways to solve this are the following:

```
                CLC     FIELD(1),FIVE
                BC      6,NOT5
                 . .     .
        FIVE    DC      X'05'
```

or:

```
                CLI     FIELD,X'05'
                BC      6,NOT5
```

or:

```
                TM      FIELD,X'05'
                BC      12,NOT5
                TM      FIELD,X'FA'
                BC      5,NOT5
```

11. The second byte of the BC instruction, containing the mask M1 and index X2 fields.

12. (d) The OI instruction changes the BC 0 instruction, which never branches, to a BC 15 instruction, which branches unconditionally. Hence, after the first time around, the sequence between the BC and symbolic address ADDR is always skipped.

13. The instruction sequence between the BC instruction and the address ADDR will be alternately executed and skipped.

14.

```
                N       5,MASK
                 . . .   . . .
        MASK    DC      X'FF000000'
```

Chapter 7: Edit, Translate, and Execute Instructions

1. BBBB1540
2. BBBB5721BB
3. BBBBBBB.01BCR
4. BBBBBBBBB
5. BB0,000.10BB
6. BBBB101.43CRBBBBBB1.07BCR
7a. PATRN DC X'40206B2020206B2020214B202040C3D9'
b. BBBB92,500.01BCR
c. BBBB92,500.01BCR
8. (c) PATRN+2
9. No
10. (e) ACBD
11. (d) Address of AREA+2 and X'01' respectively

12. 12345678991000000000
Area is first set to zeros by the MVI and MVC instructions. The EX instruction first causes the low-order 8 bits of register 2 (0A) to be Or'd with the 8-bit length code portion (00) of the Move instruction. The result of the Or'ing is a length code of 0A (10 in decimal). Since the object instruction length code is always one less than the number of bytes to be affected, the Move instruction will cause 11 bytes to be moved.

13. 10000000000000000000

Chapter 8: Subroutine Linkages and Program Relocation

1a. The return address is entered in Register 14, and an unconditional branch is made to the address in Register 15.
b. The return address is entered in Register 14, and an unconditional branch is made to the location designated by SUB.
c. BR 14
2. 1 d
13 c
14 a
15 b
3a. Assures alignment of address constants by use of a CNOP.
b. Places the address of subroutine in Register 15
c. Places the address of return in Register 14
d. Sets up parameter list address by use of a BALR 1,15
e. Defines as many address constants as there are in parameter list.
4a. 000 408
b. 000 404
c. 000 404
d. 000 406
e. 000 402
f. 000 402
5a. STM 14,12,12(13)
b. LM 14,12,12(13)
BR 14
6a. EXTRN assembler instruction
b. ENTRY assembler instruction

Chapter 9: Floating-Point Arithmetic

1. DC E'3.14159265'
DC E'-2.78'
DC E'38754E+6'
(DC E'3.8754E+10' is another possibility)
DC E'0.278E-5'
DC E'-2.36E-11'
2. DC D'3.141592653589793'
DC D'-2.78'
DC D'-3E-6'
DC D'3.8E+30'
DC D'8E-9'
3. 42200000
4220000000000000
47100000
3A100000
BA100000
C7100000
4a. Floating-point register 2: 42100000 XXXXXXXX
Floating-point register 4: 41800000 XXXXXXXX
b. Floating-point register 2: 44600244 XXXXXXXX
Floating-point register 4: 44300122 XXXXXXXX
c. Floating-point register 6: 41F00000 XXXXXXXX
General register 6: 00000000
d. General register 3: 42100000
5.

```
      .      .
      .      .
      .      .
      LE     2,B     B in Reg. 2
      ME     2,C     B x C in Reg. 2
      LCER   4,2     -(B x C) in Reg. 4
      AE     2,A     A+(B x C) in Reg. 2
      AE     4,A     A-(B x C) in Reg. 4
      DER    4,2     A-(B x C) ÷ A+(B x C)
      STE    4,X     Store final result
      .      .
      .      .
      .      .
A     DS     F
B     DS     F
C     DS     F
X     DS     F
```

Appendix

SYSTEM/360 MACHINE INSTRUCTIONS

STANDARD INSTRUCTION SET

NAME	MNEMONIC	TYPE	CODE	OPERANDS (Assembler Format)
* Add	AR	RR	1A	R1, R2
* Add	A	RX	5A	R1, D2 (X2, B2)
* Add Halfword	AH	RX	4A	R1, D2 (X2, B2)
* Add Logical	ALR	RR	1E	R1, R2
* Add Logical	AL	RX	5E	R1, D2 (X2, B2)
* AND	NR	RR	14	R1, R2
* AND	N	RX	54	R1, D2 (X2, B2)
* AND	NI	SI	94	D1 (B1), I2
* AND	NC	SS	D4	D1 (L, B1), D2 (B2)
Branch and Link	BALR	RR	05	R1, R2
Branch and Link	BAL	RX	45	R1, D2 (X2, B2)
Branch on Condition	BCR	RR	07	M1, R2
Branch on Condition	BC	RX	47	M1, D2 (X2, B2)
Branch on Count	BCTR	RR	06	R1, R2
Branch on Count	BCT	RX	46	R1, D2 (X2, B2)
Branch on Index High	BXH	RS	86	R1, R3, D2 (B2)
Branch on Index Low or Equal	BXLE	RS	87	R1, R3, D2 (B2)
* Compare	CR	RR	19	R1, R2
* Compare	C	RX	59	R1, D2 (X2, B2)
* Compare Halfword	CH	RX	49	R1, D2 (X2, B2)
* Compare Logical	CLR	RR	15	R1, R2
* Compare Logical	CL	RX	55	R1, D2 (X2, B2)
* Compare Logical	CLC	SS	D5	D1 (L, B1), D2 (B2)
* Compare Logical	CLI	SI	95	D1 (B1), I2
Convert to Binary	CVB	RX	4F	R1, D2 (X2, B2)
Convert to Decimal	CVD	RX	4E	R1, D2 (X2, B2)
Diagnose		SI	83	
Divide	DR	RR	1D	R1, R2
Divide	D	RX	5D	R1, D2 (X2, B2)
* Exclusive OR	XR	RR	17	R1, R2
* Exclusive OR	X	RX	57	R1, D2 (X2, B2)
* Exclusive OR	XI	SI	97	D1 (B1), I2
* Exclusive OR	XC	SS	D7	D1 (L, B1), D2 (B2)
Execute	EX	RX	44	R1, D2 (X2, B2)
* Halt I/O	HIO	SI	9E	D1 (B1)
Insert Character	IC	RX	43	R1, D2 (X2, B2)
Load	LR	RR	18	R1, R2
Load	L	RX	58	R1, D2 (X2, B2)
Load Address	LA	RX	41	R1, D2 (X2, B2)
* Load and Test	LTR	RR	12	R1, R2
* Load Complement	LCR	RR	13	R1, R2
Load Halfword	LH	RX	48	R1, D2 (X2, B2)
Load Multiple	LM	RS	98	R1, R3, D2 (B2)
* Load Negative	LNR	RR	11	R1, R2
* Load Positive	LPR	RR	10	R1, R2
† Load PSW	LPSW	SI	82	D1 (B1)
Move	MVI	SI	92	D1 (B1), I2
Move	MVC	SS	D2	D1 (L, B1), D2 (B2)
Move Numerics	MVN	SS	D1	D1 (L, B1), D2 (B2)
Move with Offset	MVO	SS	F1	D1 (L1, B1), D2 (L2, B2)
Move Zones	MVZ	SS	D3	D1 (L, B1), D2 (B2)
Multiply	MR	RR	1C	R1, R2
Multiply	M	RX	5C	R1, D2 (X2, B2)
Multiply Halfword	MH	RX	4C	R1, D2 (X2, B2)
* OR	OR	RR	16	R1, R2
* OR	O	RX	56	R1, D2 (X2, B2)
* OR	OI	SI	96	D1 (B1), I2
* OR	OC	SS	D6	D1 (L, B1), D2 (B2)
Pack	PACK	SS	F2	D1 (L1, B1), D2 (L2, B2)
† Set Program Mask	SPM	RR	04	R1
Set System Mask	SSM	SI	80	D1 (B1)
* Shift Left Double	SLDA	RS	8F	R1, D2 (B2)
* Shift Left Single	SLA	RS	8B	R1, D2 (B2)
Shift Left Double Logical	SLDL	RS	8D	R1, D2 (B2)
Shift Left Single Logical	SLL	RS	89	R1, D2 (B2)
* Shift Right Double	SRDA	RS	8E	R1, D2 (B2)
* Shift Right Single	SRA	RS	8A	R1, D2 (B2)
Shift Right Double Logical	SRDL	RS	8C	R1, D2 (B2)
Shift Right Single Logical	SRL	RS	88	R1, D2 (B2)
* Start I/O	SIO	SI	9C	D1 (B1)
Store	ST	RX	50	R1, D2 (X2, B2)
Store Character	STC	RX	42	R1, D2 (X2, B2)
Store Halfword	STH	RX	40	R1, D2 (X2, B2)
Store Multiple	STM	RS	90	R1, R3, D2 (B2)
* Subtract	SR	RR	1B	R1, R2
* Subtract	S	RX	5B	R1, D2 (X2, B2)
* Subtract Halfword	SH	RX	4B	R1, D2 (X2, B2)
* Subtract Logical	SLR	RR	1F	R1, R2
* Subtract Logical	SL	RX	5F	R1, D2 (X2, B2)
Supervisor Call	SVC	RR	0A	I
* Test and Set	TS	SI	93	D1 (B1)
* Test Channel	TCH	SI	9F	D1 (B1)
* Test I/O	TIO	SI	9D	D1 (B1)
* Test Under Mask	TM	SI	91	D1 (B1), I2
Translate	TR	SS	DC	D1 (L, B1), D2 (B2)
* Translate and Test	TRT	SS	DD	D1 (L, B1), D2 (B2)
Unpack	UNPK	SS	F3	D1 (L1, B1), D2 (L2, B2)

---

* Condition code is set

† New condition code is loaded

DECIMAL FEATURE INSTRUCTIONS

NAME	MNEMONIC	TYPE	CODE	OPERANDS (Assembler Format)
* Add Decimal	AP	SS	FA	D1 (L1, B1), D2 (L2, B2)
* Compare Decimal	CP	SS	F9	D1 (L1, B1), D2 (L2, B2)
Divide Decimal	DP	SS	FD	D1 (L1, B1), D2 (L2, B2)
* Edit	ED	SS	DE	D1 (L, B1), D2 (B2)
* Edit and Mark	EDMK	SS	DF	D1 (L, B1), D2 (B2)
Multiply Decimal	MP	SS	FC	D1 (L1, B1), D2 (L2, B2)
* Subtract Decimal	SP	SS	FB	D1 (L1, B1), D2 (L2, B2)
* Zero and Add	ZAP	SS	F8	D1 (L1, B1), D2 (L2, B2)

DIRECT CONTROL FEATURE INSTRUCTIONS

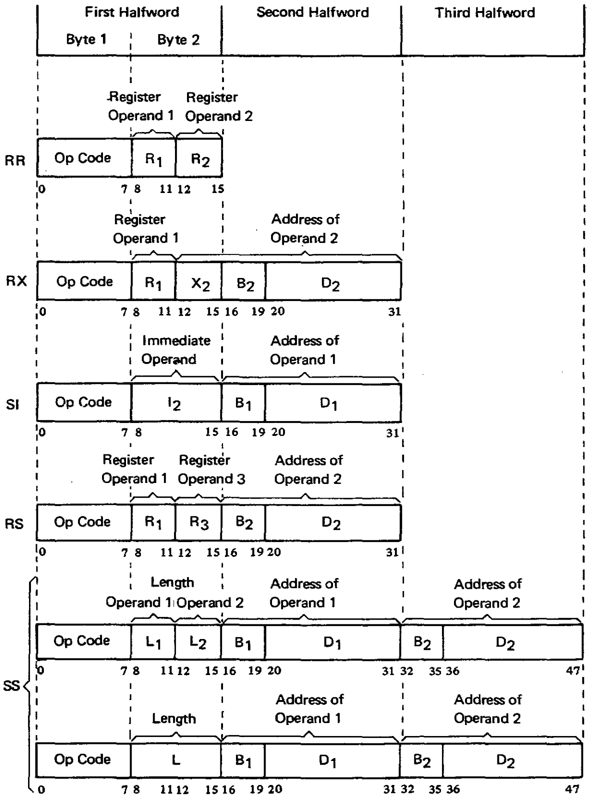
NAME	MNEMONIC	TYPE	CODE	OPERANDS (Assembler Format)
Read Direct	RDD	SI	85	D1 (B1), I2
Write Direct	WRD	SI	84	D1 (B1), I2

PROTECTION FEATURE INSTRUCTIONS

NAME	MNEMONIC	TYPE	CODE	OPERANDS (Assembler Format)
Insert Storage Key	ISK	RR	09	R1, R2
Set Storage Key	SSK	RR	08	R1, R2

Floating-point feature instructions are listed in Chapter 9.

MACHINE FORMAT



CONDITION CODE SETTINGS

Code State	0	1	2	3
Mask Bit Position	8	4	2	1
Fixed-Point Arithmetic				
Add H/F	zero	< zero	> zero	overflow
Add Logical	zero no carry	not zero no carry	zero carry	not zero carry
Compare H/F (A:B)	equal	A low	A high	--
Load and Test	zero	<zero	> zero	carry
Load Complement	zero	<zero	> zero	overflow
Load Negative	zero	<zero	--	--
Load Positive	zero	--	> zero	overflow
Shift Left Double	zero	<zero	> zero	overflow
Shift Left Single	zero	<zero	> zero	overflow
Shift Right Double	zero	<zero	> zero	--
Shift Right Single	zero	<zero	> zero	--
Subtract H/F	zero	<zero	> zero	overflow
Subtract Logical	--	not zero no carry	zero carry	not zero carry
Decimal Arithmetic				
Add Decimal	zero	<zero	> zero	overflow
Compare Decimal (A:B)	equal	A low	A high	--
Subtract Decimal	zero	< zero	> zero	overflow
Zero and Add	zero	<zero	> zero	overflow
Logical Operations				
And	zero	not zero	--	--
Compare Logical (A:B)	equal	A low	A high	--
Edit	zero	<zero	> zero	--
Edit and Mark	zero	<zero	> zero	--
Exclusive Or	zero	not zero	--	--
Or	zero	not zero	--	--
Test Under Mask	zero	mixed	--	one(s)
Translate and Test	zero	incomplete	complete	--

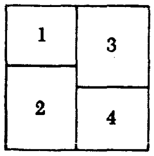
EXTENDED MNEMONIC CODES FOR THE BRANCH ON CONDITION INSTRUCTION

Assembler Code		Meaning	Machine Instruction Generated	
B	D2(X2,B2)	Branch Unconditional	BC	15,D2(X2,B2)
BR	R2	Branch Unconditional (RR format)	BCR	15,R2
NOP	D2(X2,B2)	No Operation	BC	0,D2(X2,B2)
NOPR	R2	No Operation (RR format)	BCR	0,R2
		Used after compare instructions (A:B)		
BH	D2(X2,B2)	Branch on High	BC	2,D2(X2,B2)
BL	D2(X2,B2)	Branch on Low	BC	4,D2(X2,B2)
BE	D2(X2,B2)	Branch on Equal	BC	8,D2(X2,B2)
BNH	D2(X2,B2)	Branch on Not High	BC	13,D2(X2,B2)
BNL	D2(X2,B2)	Branch on Not Low	BC	11,D2(X2,B2)
BNE	D2(X2,B2)	Branch on Not Equal	BC	7,D2(X2,B2)
		Used after arithmetic instructions		
BO	D2(X2,B2)	Branch on Overflow	BC	1,D2(X2,B2)
BP	D2(X2,B2)	Branch on Plus	BC	2,D2(X2,B2)
BM	D2(X2,B2)	Branch on Minus	BC	4,D2(X2,B2)
BZ	D2(X2,B2)	Branch on Zero	BC	8,D2(X2,B2)
BNP	D2(X2,B2)	Branch on Not Plus	BC	13,D2(X2,B2)
BNM	D2(X2,B2)	Branch on Not Minus	BC	11,D2(X2,B2)
BNZ	D2(X2,B2)	Branch on Not Zero	BC	7,D2(X2,B2)
		Used after Test under Mask instructions		
BO	D2(X2,B2)	Branch if Ones	BC	1,D2(X2,B2)
BM	D2(X2,B2)	Branch if Mixed	BC	4,D2(X2,B2)
BZ	D2(X2,B2)	Branch if Zeros	BC	8,D2(X2,B2)
BNO	D2(X2,B2)	Branch if Not Ones	BC	14,D2(X2,B2)

EBCDIC CHART

The 256-position chart at the right, outlined by the heavy black lines, shows the graphic characters and control character representations for the Extended Binary-Coded Decimal Interchange Code (EBCDIC). The bit-position numbers, bit patterns, hexadecimal representations and card hole patterns for these and other possible EBCDIC characters are also shown.

To find the card hole patterns for most characters, partition the chart into four blocks as follows:



Block 1: Zone punches at top of table; digit punches at left

Block 2: Zone punches at bottom of table; digit punches at left

Block 3: Zone punches at top of table; digit punches at right

Block 4: Zone punches at bottom of table; digit punches at right

Fifteen positions, indicated by circled numbers, are exceptions to the above arrangement. The card hole patterns for these positions are given below the chart.

Following are some examples of the use of the EBCDIC chart:

Character	Type	Bit Pattern	Hex	Hole Pattern	
				Zone Punches	Digit Punches
PF	Control Character	00 00 0100	04	12 - 9	- 4
%	Special Graphic	01 10 1100	6C	0	- 8 - 4
R	Upper Case	11 01 1001	D9	11	- 9
a	Lower Case	10 00 0001	81	12 - 0	- 1
	Control Character, function not yet assigned	00 11 0000	30	12 - 11 - 0 - 9	- 8 - 1

Bit Positions
01 23 4567

EBCDIC CHART

Bit Positions 4, 5, 6, 7	Second Hexadecimal Digit	Digit Punches	00				01				10				11				Bit Positions 0,1
			00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11	Bit Positions 2,3
			0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	First Hexadecimal Digit
			12				12	12		12	12	12		12	12				Zone Punches
				11				11	11	11		11	11	11		11			
					0		0		0	0	0		0	0			0		
			9	9	9	9	9	9	9	9									Digit Punches
0000	0	8-1	① NUL	② DLE	DS ③	④	SP ⑤	& ⑥	- ⑦	⑧					⑨	⑩	⑪	⑫ 0	8-1
0001	1	1	SOH	DC1	SOS				/ ⑬		a	j			A	J	⑭	1	1
0010	2	2	STX	DC2	FS	SYN					b	k	s		B	K	S	2	2
0011	3	3	ETX	TM							c	l	t		C	L	T	3	3
0100	4	4	PF	RES	BYP	PN					d	m	u		D	M	U	4	4
0101	5	5	HT	NL	LF	RS					e	n	v		E	N	V	5	5
0110	6	6	LC	BS	ETB	UC					f	o	w		F	O	W	6	6
0111	7	7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7	7
1000	8	8		CAN							h	q	y		H	Q	Y	8	8
1001	9	8-1		EM							i	r	z		I	R	Z	9	9
1010	A	8-2	SMM	CC	SM		¢	!	⑮	:									8-2
1011	B	8-3	VT	CU1	CU2	CU3	.	$	,	#									8-3
1100	C	8-4	FF	IFS		DC4	<	*	%	@									8-4
1101	D	8-5	CR	IGS	ENQ	NAK	(	)	_	'									8-5
1110	E	8-6	SO	IRS	ACK		+	;	>	=									8-6
1111	F	8-7	SI	IUS	BEL	SUB	|	¬	?	"									8-7
			12				12				12	12		12	12	12		12	Zone Punches
				11				11				11	11	11		11	11	11	
					0				0		0		0	0	0		0	0	
			9	9	9	9									9	9	9	9	

Card Hole Patterns (exceptions to punches shown in chart)

① 12-0-9-8-1
② 12-11-9-8-1
③ 11-0-9-8-1
④ 12-11-0-9-8-1
⑤ No Punches
⑥ 12
⑦ 11
⑧ 12-11-0
⑨ 12-0
⑩ 11-0
⑪ 0-8-2
⑫ 0
⑬ 0-1
⑭ 11-0-9-1
⑮ 12-11

Control Character Representations

ACK	Acknowledge
BEL	Bell
BS	Backspace
BYP	Bypass
CAN	Cancel
CC	Cursor Control
CR	Carriage Return
CU1	Customer Use 1
CU2	Customer Use 2
CU3	Customer Use 3
DC1	Device Control 1
DC2	Device Control 2
DC4	Device Control 4
DEL	Delete
DLE	Data Link Escape
DS	Digit Select
EM	End of Medium
ENQ	Enquiry
EOT	End of Transmission
ESC	Escape
ETB	End of Transmission Block
ETX	End of Text
FF	Form Feed
FS	Field Separator
HT	Horizontal Tab
IFS	Interchange File Separator
IGS	Interchange Group Separator
IL	Idle
IRS	Interchange Record Separator
IUS	Interchange Unit Separator
LC	Lower Case
LF	Line Feed
NAK	Negative Acknowledge
NL	New Line
NUL	Null
PF	Punch Off
PN	Punch On
RES	Restore
RS	Reader Stop
SI	Shift In
SM	Set Mode
SMM	Start of Manual Message
SO	Shift Out
SOH	Start of Heading
SOS	Start of Significance
SP	Space
STX	Start of Text
SUB	Substitute
SYN	Synchronous Idle
TM	Tape Mark
UC	Upper Case
VT	Vertical Tab

Special Graphic Characters

¢	Cent Sign
.	Period, Decimal Point
<	Less-than Sign
(	Left Parenthesis
+	Plus Sign
|	Logical OR
&	Ampersand
!	Exclamation Point
$	Dollar Sign
*	Asterisk
)	Right Parenthesis
;	Semicolon
¬	Logical NOT
-	Minus Sign, Hyphen
/	Slash
,	Comma
%	Percent
_	Underscore
>	Greater-than Sign
?	Question Mark
:	Colon
#	Number Sign
@	At Sign
'	Prime, Apostrophe
=	Equal Sign
"	Quotation Mark

SYSTEM/360 ASSEMBLER INSTRUCTIONS

Following is a representative list of assembler instructions, grouped according to use. The mnemonics used for conditional assembly and macro definition are included simply to clarify classification of assembler instructions as a whole. Information on these two subjects is given in the System/360 Assembler Language manuals (see Preface). The meaning of the extended mnemonics for the Branch on Condition machine instructions, and the machine code generated by each, appear elsewhere in this Appendix.

MNEMONIC	MEANING
For symbol definition	
EQU	Equate Symbol
For data definition	
DC	Define Constant
DS	Define Storage
CCW	Define Channel Command Word
For program sectioning and linking	
START	Start Assembly
CSECT	Identify Control Section
DSECT	Identify Dummy Section
ENTRY	Identify Entry-point Symbol
EXTRN	Identify External Symbol
COM	Identify Blank Common Control Section
For base register assignment	
USING	Use Base Address Register
DROP	Drop Base Address Register
For control of printed listings	
TITLE	Identify Assembly Output
EJECT	Start New Page
SPACE	Space Listing
PRINT	Print Optional Data
For program control	
ICTL	Input Format Control
ISEQ	Input Sequence Checking
ORG	Set Location Counter
LTORG	Begin Literal Pool
CNOP	Conditional No Operation
COPY	Copy Predefined Source Coding
END	End Assembly
PUNCH	Punch a Card
REPRO	Reproduce Following Card
For macro definition	
MACRO	
MNOTE	
MEXIT	
MEND	
For conditional assembly	
GBLA	
GBLB	
GBLC	
LCLA	
LCLB	
LCLC	
SETA	
SETB	
SETC	
AIF	
AGO	
Extended mnemonics for the BC and BCR machine instructions	
B	
BR	
NOP	
NOPR	
BH	
BL	
BE	
BNH	
BNL	
BNE	
BO	
BP	
BM	
BZ	
BNP	
BNM	
BNZ	
BNO	

TYPES OF ASSEMBLER LANGUAGE CONSTANTS

Code	Type	Machine Format
C	Character	8-bit code for each character
X	Hexadecimal	4-bit code for each hexadecimal digit
B	Binary	Binary
F	Fixed-point	Signed, fixed-point binary; normally a fullword
H	Fixed-point	Signed, fixed-point binary; normally a halfword
E	Floating-point	Short floating-point; normally a fullword
D	Floating-point	Long floating-point; normally a doubleword
P	Decimal	Packed decimal
Z	Decimal	Zoned decimal
A	Address	Value of address; normally a fullword
Y	Address	Value of address; normally a halfword
S	Address	Base register and displacement value; a halfword
V	Address	Space reserved for external symbol addresses; each address normally a fullword

Index

In this index, assembler and macro instructions are identified as such. Machine instructions are listed by name in capital letters.

READER'S COMMENT FORM

A Programmer's Introduction
to IBM System/360 Assembler Language

SC20-1646-6

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM. If you wish a reply, be sure to include your name and address.

COMMENTS

fold

fold

fold

fold

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.
FOLD ON TWO LINES, STAPLE AND MAIL.

YOUR COMMENTS PLEASE

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

Please note that requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or the IBM branch office serving your locality.

FOLD FOLD

FIRST CLASS
PERMIT NO. 142
POUGHKEEPSIE, NEW YORK

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY:

IBM Corporation
Education Center, Bldg. 005
South Road
Poughkeepsie, New York 12602

ATTENTION: Education Development - Publications Services, Dept. **78L**

FOLD FOLD

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York 10601
(USA only)

IBM World Trade Corporation
821 United Nations Plaza, New York, N.Y. 10017
(International)

SC20-1646-6

System/360 Assembler Language Printed in USA SC20-1646-6