IBM®

Student Text

A Programmer's Introduction to the IBM System/360 Architecture, Instructions, and Assembler Language

Preface

This text is intended to introduce to the student the characteristics of System/360 and its instruction set. Many sample programs are used to illustrate specific instructions and programming techniques. It is expected that the student has some knowledge of computing systems.

The following IBM System/360 Student Texts have been incorporated in this publication; however, the individual books are not obsoleted by this version:

Fixed-Point Operations (C20-1613)
Programming with Base Registers and the USING Instruction (C20-1614)
Introduction to Assembly Language Programming (C20-1615)
Decimal Operations (C20-1616)
Number Systems (C20-1618)*
Logical Operations on Characters and Bits (C20-1623)
Edit, Translate and Execute Instructions (C20-1624)
Subroutines and Subprograms (C20-1625)

The new material in this text includes the chapters on "Architecture," "Floating Point and Advanced Loops in Scientific Applications," and "Automatic Interrupts."

No attempt at completeness has been made and, therefore, it is expected that the student will refer to the appropriate Systems Reference Library (SRL) publications for additional detail.

*Number Systems (C20-1618) will continue to be available as a separate publication.

Minor Revision (May, 1966)
This addition, form C20-1646-1, is a minor revision of, but does not obsolete the preceding edition, form C20-1646-0.

Copies of this and other IBM publications can be obtained through IBM branch offices. Address comments concerning the contents of this publication to
IBM, Technical Publication Department, 112 East Post Road, White Plains, N.Y. 10601

Contents

Chapter 10: Floating Point and Advanced Loops in Scientific Applications

Chapter 11: Automatic Interrupts

Answers to Questions and Exercises

Introduction

Chapters 1 and 2 provide the student with an introduction to the architecture of System/360 and to the numbering systems that are of some significance to System/360. This knowledge provides a background for later chapters in which many of the instructions in the System/360 instruction set are introduced as well as illustrated by sample assembler language programs. (These samples were prepared by the 7090/ 7094 Support Package for IBM System/360.) In addition, some chapters discuss programming techniques that will be valuable to those studying the assembler language. One chapter discusses in detail the characteristics and use of the System/360 interrupt feature, which is introduced in the first chapter.

Questions and exercises are provided at the end of each chapter to help the student review the material; answers may be found at the back of this text.

This text is not directed to any one of the levels of programming systems support available for System/ 360 (Basic Programming Support, Basic Operating System/360, Operating System/360, and the 7090/ 7094 Support Package for IBM System/360). Therefore, IOCS programming is not covered.

It is assumed that the student, while studying this text, has access to *IBM System/360 Principles of Operation* (A22-6821), and to one of the SRL publications on the assembler language. Also, the student may wish to refer to the appropriate SRL publications for specific details on one or more of the programming systems available for System/360.

Chapter 1: Architecture

This chapter introduces the student who has some knowledge of computing systems to the overall structure of System/360 and the implications of its structure for new application areas. An introduction to such System/360 features as channels, automatic interrupts and the general purpose registers, and to instruction formats, data formats, and the various types of arithmetic operations, provides the student with a background that is prerequisite to an understanding of the later chapters of this text. This chapter also provides some insight into the need for a supervisory program.

System Features for New Application Approaches

The demands made upon a data processing system normally increase in the volume of processing to be done and in the scope of applications for which the system is utilized. To allow for growth in volume, System/360 was designed for implementation over a wide cost and performance range and to maintain program compatibility among the various models. For growth in application scope, the logical structure is that of a general purpose system for commercial, scientific, communication, and control applications.

To the user, a concern more immediate than growth considerations is cost versus performance. Before selecting higher-performance equipment, it is important to achieve maximum throughput from a lower-performance (and lower-priced) system. Achieving maximum throughput means decreasing the time required to process a total number of jobs so that the backlog of jobs is reduced. There is often, however, an opposing objective of decreasing the turnaround, or response, time for a given job. A report that takes three minutes of processor time is needed within an hour, but another four-hour run in progress requires two more hours for completion. Can we disrupt the program in progress? The answer has depended on the system and the programmed facilities available for restarting an interrupted program.

Because System/360 was designed to encompass solutions to such problems in all areas of data processing, it is helpful to further examine some of these conventional problems and to consider recent application approaches.

The most basic concept of computing, with which we are all familiar, is a program of sequential instructions. The processing unit fetches an instruction, decodes and executes it, increments an instruction counter, and then repeats this sequence of operations. A branch causes the contents of the instruction counter to be replaced with another address, and processing is continued from this address. This machine instruction fetch-execute-increment cycle is still basic to digital computers. In programming, however, we have come a long way from routines that read a card, process the data from the card, and write the results with no concurrent or overlapped operation.

The degree of concurrent operation that can be achieved depends not only on machine facilities but also on the programming employed.

The processing unit may be used for some portion of time and encounters recurring delays while awaiting input/output operations. Then the I/O equipment may be idled while processing takes place. Further, a system must often be configured for the largest job at the installation. That largest job may be run infrequently and the many smaller jobs that use the processing unit's time may utilize only a small portion of the total system's capacity. Lost time on the processing unit, lost time on I/O equipment, and less-than-maximum storage utilization are all wasteful.

The designers of System/360 sought solutions to these problems with a design that allows and encourages maximum utilization of available system resources. First, this design philosophy recognizes that data processing systems and programming systems should be integrated and not developed independently. New and sophisticated control techniques incorporated into the equipment for maximum utilization of resources take over many functions that previously were the concern of the problem programmer or of programming systems programmers. This last statement is not intended to imply that programming systems are not essential to utilize the system, but rather that there is a larger degree of interplay between equipment and program. In fact, the equipment was designed to run with a monitor program in control. System/360 and its control program are indistinguishable to the problem programmer.

Another consideration in the system's design was to facilitate the newer application approaches to computing, such as communications and multiprogramming.

Communications applications include time sharing, message switching, and the whole area of teleprocessing. Time sharing or conversational mode is the use of a number of remote terminals where each terminal has access to the computer. Here each terminal may be regarded as a personal computer, and all the independent users have access to a single computer virtually simultaneously because of ultra-high processing and switching speeds.

Message switching involves a telecommunications network where messages from remote points are sent to a central location for routing to their destination.

A common teleprocessing application is the processing of inquiries from remote terminals. Each terminal user introduces data to the system, and programs residing in the system perform whatever processing is required. The message may be simply a query for information stored within the system or it may be data to be entered and processed (with or without an answerback).

The program that handles the messages is called the foreground program. Other processing may take place between the servicing of messages. This "background" program is interrupted and the "foreground" program assumes control upon the receipt of a message. When the message is processed and no other messages are held pending, the foreground program relinquishes control to the background program.

Maximum utilization of system resources becomes particularly vital to a communication (or teleprocessing) system where input is unscheduled, where jobs are stacked (that is, where a series of jobs is run under the control of a supervisory program with a minimum of operator intervention), and in multiprogramming.

In applications involving multiprogramming optimum use is made of all facilities by having the system operate upon multiple programs or routines (tasks). While one task awaits data from an I/O device, another task utilizes the processing unit, and still other tasks utilize other I/O devices. As soon as a task utilizing the processing unit must wait for an I/O operation, it relinquishes control of the processing unit, and a waiting task assumes control. (The size, speed, and configuration of the system determine whether multiprogramming is practicable.)

Channel Concept

One of the system features that facilitate the simultaneous operations necessary for maximum utilization of the system's resources is channel circuitry. The electronic circuitry of a channel may be regarded as a small, independent computer that responds to its own set of commands. Channels provide the ability to read, write, and compute concurrently.

Each channel has its own program in main storage, and this program must be initiated by the supervisory program. A Start I/O instruction, for example, has the effect of selecting a specified I/O device and channel, and, if the device is available, starting the operation or operations specified by the channel program. In addition to the Start I/O instruction, there are three other instructions for communication between the processing unit and the channel: Test I/O, Halt I/O, and Test Channel.

These instructions are issued by the supervisory program, which contains an Input/Output Control System (IOCS). The address part of the instruction specifies the channel and the I/O device. When the channel and the device verify that the operation can be executed, the processing unit is released. The channel fetches its program from main storage and executes it. The transfer of data to or from main storage and the initiation of new operations by the channel program do not prevent processing of instructions by the processing unit.

Communications from the processing unit to the channels and I/O devices are discussed under "Channel Organization".

Selector and Multiplexor Channels

There are two types of channels: selector and multiplexor. Selector channels are used for the attachment of high-speed devices such as magnetic tapes, files, and drums. Multiplexor channels are intended primarily for low-speed devices. More than one device is usually attached to either a multiplexor or selector channel through one or more control units. The control unit's functions are indistinguishable to the user from the functions of the I/O device, and in fact, some control units are physically housed within the I/O device. A control unit functions only with the type of device for which it is designed. One control unit can have more than one of the same devices attached. Multiple tape units, for example, may be attached to a single tape control unit (see Figure 1).

When multiple slow-speed devices such as card readers are attached to a multiplexor channel, they can operate simultaneously through a time-sharing (interleaved) principle and processing can take place concurrently. When high-speed devices are attached to a multiplexor channel, only one device can operate at a time and the channel is said to be operating in burst mode. Operation of the Model 30 or 40 multiplexor channels in burst mode inhibits all other activity on the system. Selector channels always operate in burst mode and processing and I/O overlapping occur on all models (except a high-speed channel on Model 50, which inhibits processing).

As many as six selector channels can be operating concurrently with processing on Models 65 and 75.

Only one multiplexor channel can be connected to a system. The number of selector channels that can be attached varies from two on a Model 30 to six on Models 65 and 75. The important thing to remember is that channels all appear to function identically to the user; it is only the degree of simultaneity of channel operations and overlapped processing that differs among the various models.

Interrupts

We have seen that the processing unit may initiate an input/output operation and resume processing while the channel proceeds independently. The processing unit must, however, maintain control over I/O operations. When an I/O operation is completed and a channel is free, another operation in the channel should be begun, if possible, to gain maximum channel utilization. Instead of having the problem program repeatedly interrogating channels to see whether they are free, the channels themselves signal the processing unit when they become free – that is, upon completion of a channel program. The channel signals cause the supervisory program to take appropriate action such as starting another I/O operation. These signals belong to one class of interrupts that the processing unit must be prepared to handle.

Here we begin to see how the circuitry takes over functions that were formerly the programmer's concern. The automatic interrupt system may be contrasted with a programmed branch in which the contents of the instruction counter are replaced rather than incremented. These branches are the programmer's concern. With the automatic interrupt system, however, an application program is written to include conventional testing and branching, but ignores those branches that will be handled as automatic interrupts. When an interrupt occurs, the contents of the equivalent of an instruction counter are automatically replaced. This suspends the operation of the program in progress temporarily. In addition the control and status information needed to restart the program are automatically stored by the interrupt system itself.

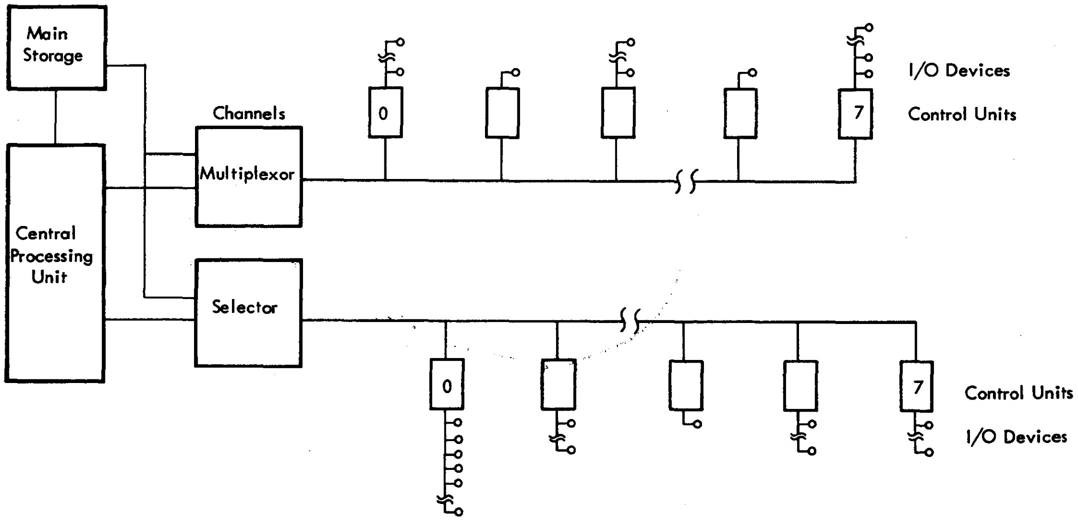


Figure 1. IBM System/360 basic logical structure

There are five classes of interrupts: input/output, program, supervisor call, external, and machine check.

- *Input/output interrupts.* The signal to the processing unit that a channel is free is typical of the class of interrupts called I/O interrupts. Special conditions in the channel or in an I/O unit cause the processing unit to take appropriate action.
- *Program interrupts.* Unusual conditions encountered in a problem program create program interrupts. Eight of the 15 possible conditions involve overflow, improper divides, lost significance, and exponent underflow. (Lost significance and exponent underflow may occur in floating-point arithmetic operations.) The remaining seven deal with improper addresses, attempted execution of invalid instructions, and similar conditions.
- *Supervisor call interrupts.* The significance of supervisor call interrupts will become apparent when we examine in more detail the effects of interrupts. Suffice it to say that Supervisor Call is an instruction that the program uses to cause an interrupt.
- *External interrupts.* Through an external call interrupt, the processing unit can respond to signals from the interrupt key on the system control panel, a built-in timer system, other processing units, or special devices.
- *Machine check interrupts.* A machine check condition initiates an automatic recording of the status of the system into a special scan-out area of main storage and then causes a machine check interrupt. A machine check can be caused only by a hardware malfunction and not by invalid data or instructions.

Some classes of interrupts can be ignored or held pending under program control. This prevents the interrupt from occurring and the interrupt is said to be "masked". An anticipated overflow is an example of an interrupt that the programmer would mask.

When the system is executing instructions of a problem program, it operates in what is called the problem program state. Interrupts that occur while the system is operating in the problem program state cause the processing unit to switch to the supervisory state. To ensure that the system has control over I/O functions, the control program takes control when an I/O instruction is required by a problem program. The control program operates in the supervisory state and includes a resident IOCS. Instructions that are executable only in the supervisory state are called "privileged".

A Supervisor Call instruction in a program is one method of causing a switch from the problem state to the supervisory state; that is, the problem program passes control to the supervisory program. An interruption code within the instruction may be used to convey messages from the calling program to the supervisory program. Two messages that the supervisory program would require are: (1) notification from the problem program that it is finished so that the supervisor can read in a new program, and (2) notification of requests to start I/O operations for the problem program. As soon as the I/O operation has begun, the supervisor program returns control to the problem program, which can continue processing while the I/O operation is taking place. Upon completion of the I/O operation, an I/O interrupt occurs. The supervisor program now determines whether any abnormal conditions were detected during the operation and takes appropriate action. The overall status of the processing unit is determined by alternatives other than the supervisor or problem state. These alternatives provide control of system resources by preventing a problem program from stopping the operation of the processing unit. There is no Halt instruction. In the problem state, processing instructions are valid but all I/O instructions and a group of control instructions are invalid. In the supervisory state, all instructions are valid.

The other alternative states are: running versus waiting state, masked versus interruptible state, and stopped versus operating state (see Figure 2).

In the running state, instruction fetching and execution proceeds in the normal manner. The wait state is typically entered by the program to await an interrupt – for example, an I/O interrupt or operator intervention from the console. In the wait state, no instructions are processed, the timer is updated, and I/O and exernal interrupts are accepted unless masked.

The processing unit may be interruptible or masked for the system (I/O or external), program, and machine interruptions. When the processing unit is interruptible for a class of interruptions, these interruptions are accepted. When the processing unit is masked, the system interruptions remain pending, but the program and machine-check interruptions are ignored. Instructions that alter the overall status of the processing unit are privileged.

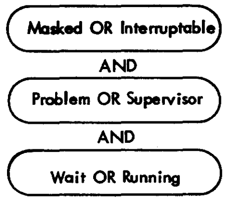


Figure 2. Alternative states of the processing unit in operation

Program Status Words and their Control of Interrupts

Passing control between problem programs and the supervisory program and returning to the right place in a program following an interrupt is accomplished with program status words (PSW's). Traditionally when information was required at some later point in a program, it was the programmer's responsibility to store it. With System/360, since the problem programmer cannot anticipate many interrupts, they become the responsibility of the system. Two storage locations are associated with each of the five classes of interrupts. One of the locations contains the address of the routine in the supervisory program that handles this class of interrupt. When an interrupt occurs, the system automatically replaces the current or active PSW, which contains an instruction counter plus other machine status information, with the PSW appropriate to this interrupt. This "new" PSW indicates among other things that the system is operating in the supervisory state and specifies the address of the routine that handles this class of interrupt. The PSW of the interrupted program is automatically stored as the "old" PSW (see Figure 3). The routine in the supervisory program that handles this interrupt will be run. Its last processing step will be to restore the old PSW as the active or current PSW, and the interrupted program will resume processing at the point where it was interrupted. Unlike the automatic switching of PSW's when an interrupt occurs, the replacement of the current PSW with the old PSW is accomplished by an instruction in the supervisory program. This programmed, rather than automatic, function was a deliberate design choice. Why, we may ask, does the Load PSW instruction need to address storage, since the system could readily determine the cause of the last interrupt? The answer is that in multiprogramming we frequently do not wish to return to the "task" last interrupted, but prefer that the control program stack up and control a sequence of PSW's.

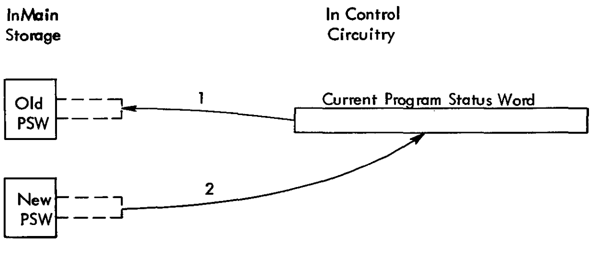


Figure 3. Interrupt program switching

Because the principle of the interrupt system is best understood in terms of the various PSW's, let us take a moment to examine their place and function. The old and new PSW's have permanent address assignments in main storage. The current PSW is contained in the control circuitry of the processing unit and, like an instruction counter, is updated as the program progresses. The new PSW locations contain the address of a routine to handle their particular class of interrupts. The addresses of these routines are not normally changed, and for a particular interrupt the same address will be read out each time this interrupt occurs. For each new PSW there is an old PSW that acts simply as temporary storage for the current PSW when an interrupt occurs. The interrupt causes the current PSW to be stored as the old PSW, and the new PSW becomes the current PSW. At the conclusion of the interrupted routine, the old PSW replaces the current PSW, restoring the system to its prior state and allowing the continuation of the interrupted program.

Old and new PSW's contained in storage are identical in format to the current PSW, since they are called upon and become "current". The location of old and new PSW's is shown in Figure 4. In the next topic, "Data Representation", we shall see that PSW's are doublewords with individual bits labeled 0-63. We can see now from the table that a machine check will cause the current PSW to be placed in storage locations beginning at 0048 and a new PSW will be brought out from locations beginning at 0112.

	Address	Length	Purpose
0	0000 0000	double word	Initial program Loading PSW
8	0000 1000	double word	Initial program Loading CCW1
16	0001 0000	double word	Initial program Loading CCW2
24	0001 1000	double word	External old PSW
32	0010 0000	double word	Supervisor call old PSW
40	0010 1000	double word	Program old PSW
48	0011 0000	double word	Machine check old PSW
56	0011 1000	double word	Input/output old PSW
64	0100 0000	double word	Channel status word
72	0100 1000	word	Channel address word
76	0100 1100	word	Unused
80	0101 0000	word	Timer
84	0101 0100	word	Unused
88	0101 1000	double word	External new PSW
96	0110 0000	double word	Supervisor call new PSW
104	0110 1000	double word	Program new PSW
112	0111 0000	double word	Machine check new PSW
120	0111 1000	double word	Input/output new PSW
128	1000 0000		Diagnostic scan-out area*

Figure 4. Permanent storage assignments

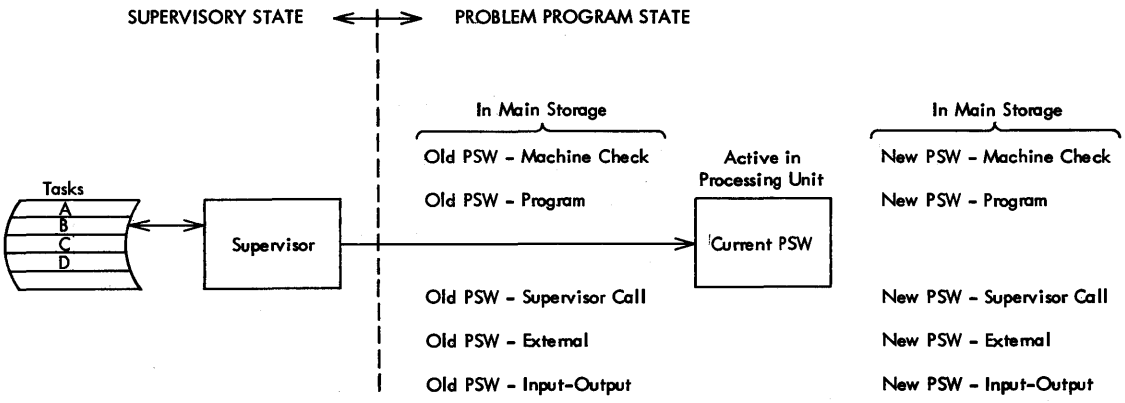


Figure 5. Problem program PSW active in processing unit contrasted with input/output operations in supervisory state

In a typical System/360 environment, more than one task is contending for time on the processing unit, and while one interrupt is being serviced, perhaps another interrupt occurs, while still another interrupt is held pending.

In Figure 5, the current PSW would reflect the status of a task B, which is being executed in the problem state.

In Figure 6, an interrupt has caused the processing unit to switch to the supervisory state. A new I/O PSW is replacing the active PSW and the active PSW is being stored as the old I/O PSW. Upon leaving the I/O routine (which is executed using the resident I/O supervisory program), the old I/O PSW will again become the current PSW, unless other interrupts occur.

We have seen that an interrupt causes a type of branch. What, we may ask, is the difference between a program branch and one caused by an interrupt? The portion of the PSW that has been compared with an instruction counter is called the instruction address. When a branch occurs, only the contents of the instruction address within the PSW are changed. On an interrupt the entire PSW is replaced. The PSW contains other status and control information in addition to the instruction address, which the processing unit requires. This includes such information as program status (supervisor versus problem state, masked versus interruptible state, stopped versus operating state, and running versus waiting state).

When interrupts occur is not the concern of the problem programmer. With reference to machine cycle time, it is interesting to note that the machine designers chose an optimum economic "interruptible" point, since status information must be saved and restored. This turns out to be after an instruction has completed "E" time. In the case of I/O, external, or supervisor call interrupts, then, the current instruction will be completed before the interrupt is taken. However, in the case of program and machine errors, the end may be forced by suppressing the instruction's execution.

Other aspects of the automatic interrupt system are discussed under the chapter entitled "Automatic Interrupts", which includes a discussion of simultaneous interrupts. The details of interrupts are found in the appropriate SRL publications.

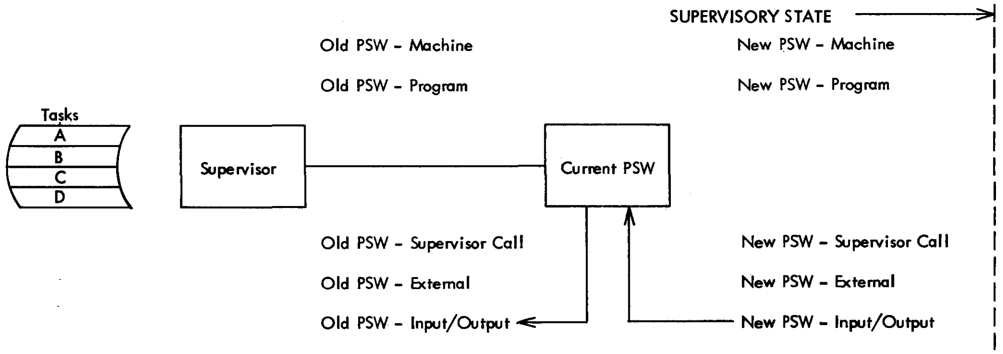


Figure 6. Switching of PSW's during an input/output interrupt

Data Representation

The most familiar method of data representation in commercial applications of computers has been binary coded decimal in which six bits are used to represent 64 alphameric and special characters. Records consist of many fields of widely different lengths. Scientific computers, on the other hand, generally operate upon fixed-word-length fields of binary data.

Several data formats can be used for processing with the System/360 to accommodate commercial and scientific applications. An eight-bit unit of information, called a byte, is fundamental to the formats. An initial byte may be addressed as an operand of an instruction, with the number of bytes used specified by the instruction. Because eight rather than six bits are used to represent a character, up to 256 possible characters could be represented in the Extended Binary Coded Decimal Interchange Code (EBCDIC) shown in Figure 7. Except for certain teleprocessing equipment, the code that makes use of characters is either EBCDIC or an eight-bit extension of a seven-bit code proposed by the International Standards Organization.

The chart shows bit positions, which determine bit patterns, at the top and to the left of each table.

The hole pattern of punched cards is shown at the bottom and to the right of each table in dark gray shading.

The table at the upper left shows control characters. The explanation of their meaning is given in a separate listing. The characters PF, for example, indicate "punch off".

Exceptions to the tabular representation of hole patterns to specify a binary bit pattern, a control character, or a graphic character are identified by numbers circled in the table, and the proper hole patterns are shown in a separate listing below the tables. The examples given opposite the tables are self-explanatory and serve to ensure correct reading of the tables. To illustrate this, the last example in the list is an exception indicated by the number 4 circled in the table at the upper left.

For further practice, translate the name John Doe into EBCDIC and use initial capitals and lowercase letters. The results should be:

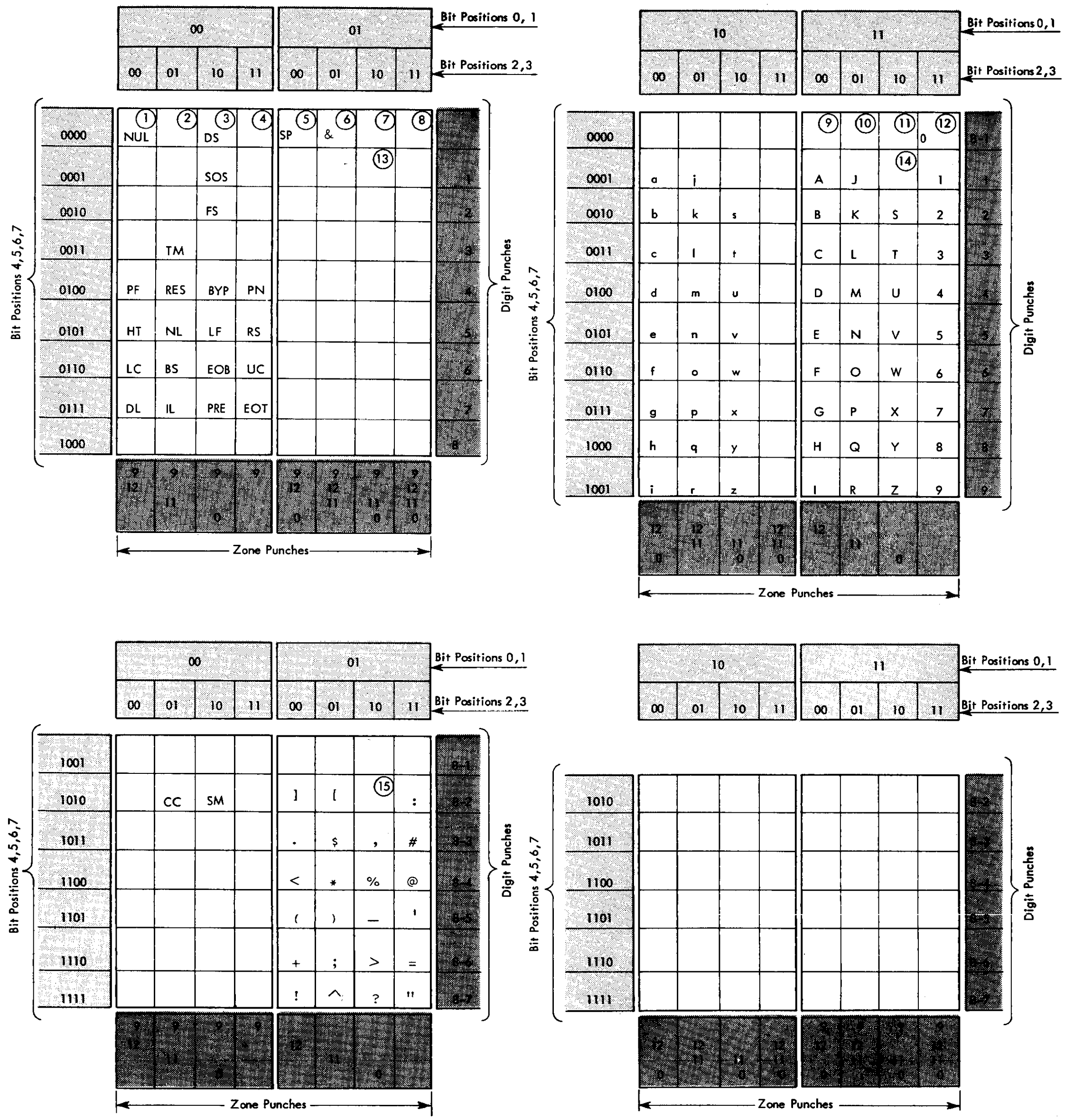
11010001	10010110	10001000	10010101
J	o	h	n

11000100	10010110	10000101
D	o	e

Note that in the tables the digits 0-9 have these bit configurations:

0	11110000	5	11110101
1	11110001	6	11110110
2	11110010	7	11110111
3	11110011	8	11111000
4	11110100	9	11111001

We may well ask what purpose the four leading 1s serve. The answer is that they provide a collating sequence in which numbers are higher than alphabetics in alphameric fields, but they are not used in arithmetic operations. Instead, an instruction is provided that "packs" two decimal digits into a byte by eliminating the leading 1s (see Figure 8). The decimal digits 0-9 are represented in the four-bit binary coded decimal form by 0000 through 1001. The elimination of the leading 1s (or zone portion) is accomplished with the Pack instruction.



	00				01				Bit Positions 0, 1
Bit Positions 4,5,6,7	00	01	10	11	00	01	10	11	Bit Positions 2,3
0000	① NUL	②	③ DS	④	⑤ SP	⑥ &	⑦	⑧	8-1
0001			SOS				⑬		1
0010			FS						2
0011		TM							3
0100	PF	RES	BYP	PN					4
0101	HT	NL	LF	RS					5
0110	LC	BS	EOB	UC					6
0111	DL	IL	PRE	EOT					7
1000									8
Zone Punches	9 12	9 11	9 0	9	9 12 0	9 12 11	9 11 0	9 12 11 0	Digit Punches

	10				11				Bit Positions 0,1
Bit Positions 4,5,6,7	00	01	10	11	00	01	10	11	Bit Positions 2,3
0000					⑨	⑩	⑪	0 ⑫	8-1
0001	a	j			A	J	⑭	1	1
0010	b	k	s		B	K	S	2	2
0011	c	l	t		C	L	T	3	3
0100	d	m	u		D	M	U	4	4
0101	e	n	v		E	N	V	5	5
0110	f	o	w		F	O	W	6	6
0111	g	p	x		G	P	X	7	7
1000	h	q	y		H	Q	Y	8	8
1001	i	r	z		I	R	Z	9	9
Zone Punches	12 0	12 11	11 0	12 11 0	12	11	0		Digit Punches

	00				01				Bit Positions 0,1
Bit Positions 4,5,6,7	00	01	10	11	00	01	10	11	Bit Positions 2,3
1001									8-1
1010		CC	SM		]	[	⑮	:	8-2
1011					.	$	,	#	8-3
1100					<	*	%	@	8-4
1101					(	)	_	'	8-5
1110					+	;	>	=	8-6
1111					!	^	?	"	8-7
Zone Punches	9 12	9 11	9 0	9	12	11	0		Digit Punches

	10				11				Bit Positions 0,1
Bit Positions 4,5,6,7	00	01	10	11	00	01	10	11	Bit Positions 2,3
1010									8-2
1011									8-3
1100									8-4
1101									8-5
1110									8-6
1111									8-7
Zone Punches	12 0	12 11	11 0	12 11 0	9 12 0	9 12 11	9 11 0	9 12 11 0	Digit Punches

①	12-0-9-8-1	⑤	No Punches	⑨	12-0	⑬	0-1
②	12-11-9-8-1	⑥	12	⑩	11-0	⑭	11-0-9-1
③	11-0-9-8-1	⑦	11	⑪	0-8-2	⑮	12-11
④	12-11-0-9-8-1	⑧	12-11-0	⑫	0		

EBCDIC chart explanation continued on next page.

Control Characters

NUL	Null	BS	Backspace	EOB	End of Block
PF	Punch Off	IL	Idle	PRE	Prefix
HT	Horizontal Tab	CC	Cursor Control	PN	Punch On
LC	Lower Case	DS	Digit Select	RS	Reader Stop
DL	Delete	SOS	Start of Significance	UC	Upper Case
TM	Tape Mark	FS	Field Separator	EOT	End of Transmission
RES	Restore	BYP	Bypass	SM	Set Mode
NL	New Line	LF	Line Feed	SP	Space

Special Graphic Characters

]	Right Bracket	*	Asterisk	>	Greater-than Sign
.	Period, Decimal Point	)	Right Parenthesis	?	Question Mark
<	Less-than Sign	;	Semicolon	:	Colon
(	Left Parenthesis	^	Circumflex, Logical NOT	#	Number Sign
+	Plus Sign	-	Minus Sign, Hyphen	@	At Sign
!	Exclamation Point, Logical OR	/	Slash	'	Prime, Apostrophe
&	Ampersand	,	Comma	=	Equal Sign
[	Left Bracket	%	Percent	"	Quotation Mark
$	Dollar Sign	_	Underscore		

Example	Type	Bit Pattern Bit Positions 01 23 4567	Hole Pattern	
			Zone Punches	Digit Punches
PF	Control Character	00 00 0100	12 -9	4
%	Special Graphic	01 10 1100	0	8 - 4
R	Upper Case	11 01 1001	11	9
a	Lower Case	10 00 0001	12 -0	1
	Control Character, function not yet assigned	00 11 0000	12 - 11 - 0 -9	8 - 1

Figure 7. Extended Binary Coded Decimal Interchange Code

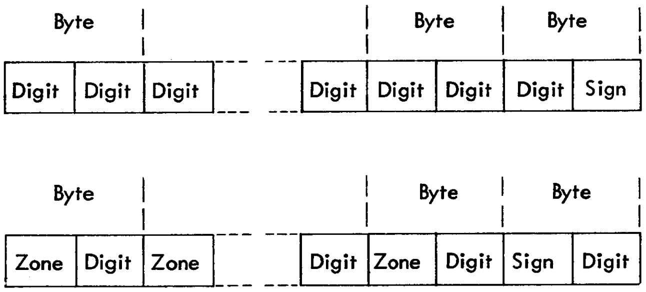


Figure 8. Packed and zoned decimal number formats

Arithmetic Operations

There are four classes of processing operations: fixed-point arithmetic, floating-point arithmetic, logical operations, and decimal arithmetic. Fixed-point arithmetic and logical operations are part of the standard instruction set. The decimal option is intended primarily for commercial applications and the floating-point arithmetic option is intended for engineering and scientific applications.

Fields of two, four, and eight bytes are called halfwords, words, and doublewords respectively (see Figure 9).

In fixed-point arithmetic the basic arithmetic operand is a signed value recorded as a binary integer, that is, a whole number (positive or negative) as contrasted with a fraction. It is called fixed-point because the machine interprets the number as a binary integer; that is, the point is to the right of the least significant position. The programmer has the responsibility for keeping track of an assumed point within a field.

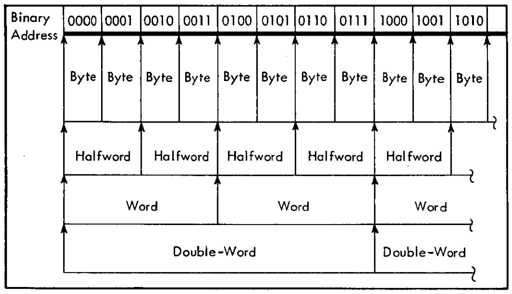


Figure 9. Halfwords, words, and doublewords as they appear in main storage

Fixed-point numbers occupy a fixed-length format consisting of a one-bit sign followed by the 31-bit integer field; alternatively, some operations may be performed on halfwords, and some multiply, divide, and shift instructions use a doubleword.

Until numeric data is ready for output on a device that uses characters, such as a printer or punch (character-set oriented), storage is most economically used by holding the data in binary or packed decimal digits.

In the following example of fixed-point arithmetic we shall, for the sake of simplicity, ignore the sign and fixed-length requirement.

Assume that a card reader has read the number 4096. The number itself will be transferred to main storage as four bytes of EBCDIC:

11110100 11110000 11111001 11110110

If this number is to be processed using fixed-point arithmetic, the PACK instruction is first issued and the number takes the binary coded decimal form:

0100 0000 1001 0110

A Convert to Binary instruction is then issued and, after its execution, the number takes the pure binary form:

1000000000000

which is 2^{12}.

Note that the decimal values of bit positions are:

128	64	32	16	8	4	2	1
7	6	5	4	3	2	1	0

The number itself is now ready for processing in fixed-point format. (Note that we have not illustrated the sign and length requirement.) After processing, a Convert to Decimal instruction and either an Unpack or an Edit instruction are used to prepare the output for a device using characters such as a printer or punch. If the results of processing are to be stored for further processing in binary form, the Convert to Decimal instruction and the Unpack or Edit instruction are omitted. If the results are to be stored as packed decimal digits, the Unpack instruction is omitted. Figure 10 shows this processing sequence.

No conversion from packed decimal to binary is necessary if the decimal instruction set is used. Instead, addition, subtraction, multiplication, division, and comparison are performed on packed binary coded decimal digits (see Figure 11). While fixed-point operations are performed on fixed-length fields, all decimal operations are performed on variable-length fields, the length of which is specified in the instruction. The address tells where the data is located, and the length specification tells how much data the instruction is to operate upon. From 0 to 15 bytes may be specified, so that, in effect, a 16-byte field may be addressed in arithmetic operations. A length specification of zero will address only the byte designated in the instruction address.

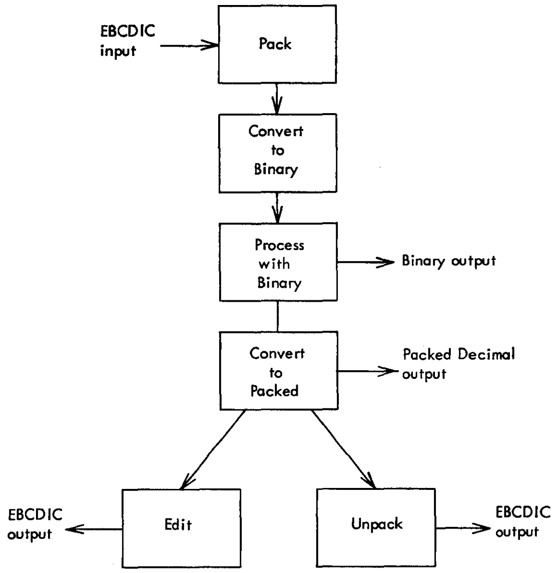


Figure 10. Fixed-point arithmetic processing sequence on EBCDIC input

Where numerical information such as a part number is not operated upon arithmetically, it may be processed in the zoned format – that is, without packing the digits.

Now consider the facts that lead the programmer to decide whether to use decimal or binary arithmetic operations. Decimal arithmetic can make the programmer and the system more productive when processing requires relatively few computational steps between input and output. When extensive processing is required, as in many scientific applications, storage and circuitry are more efficiently utilized with binary numbers.

Note that the number 4096 requires 32 bit positions in EBCDIC, 16 bit positions in packed binary coded decimal, and 13 bit positions in pure binary. Does not the economy of the binary configuration suggest the efficiency of binary operations? Figure 12, however, demonstrates that the decimal instruction set is a more direct route from input to output. The criterion for selection is the amount of processing to be done in the blocks labeled "process with binary" and "process with decimal".

As shown in Figure 12, the system will accept as input any code that is eight bits or less. For these other codes, such as a teletype code, tables are set up in storage, and translate instructions permit conversion of entire records of up to 256 characters with a single instruction. The figure lists output as binary, packed decimal digits, or EBCDIC. Actually, as with input, the output could be in any code up to eight bits through the use of translation tables.

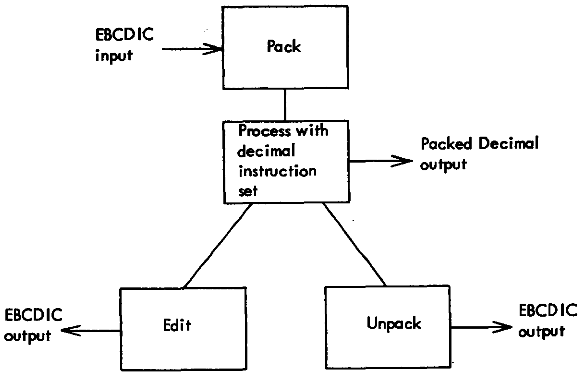


Figure 11. Processing sequence using the decimal instruction set on EBCDIC input

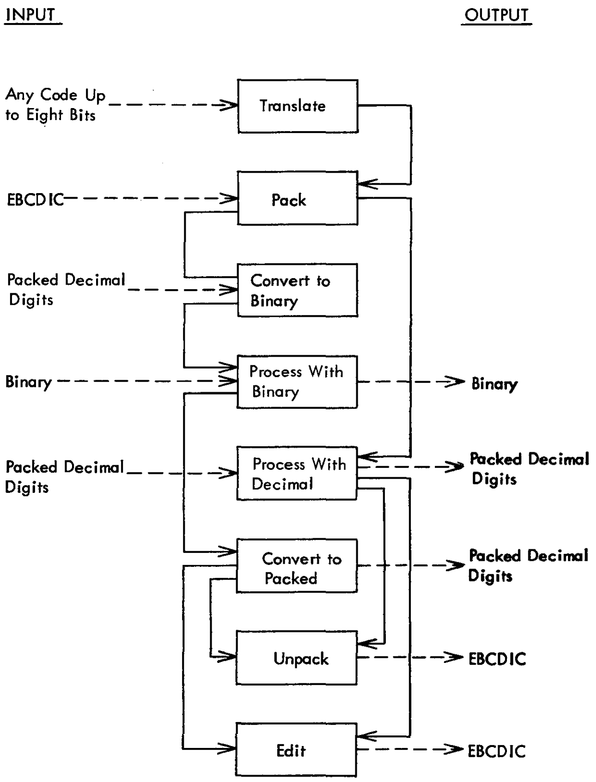


Figure 12. Various input processing sequences involving arithmetic

Sign Codes

When digits are read from cards, all unsigned digits are assigned the zone 1111 for EBCDIC. The sign patterns generated for EBCDIC are 1100 for plus and 1101 for minus. The usual case is that the sign occupies the zone positions of the least significant digit of a field. A three-digit field, then, would have this format:

zone digit zone digit sign digit

In EBCDIC a minus 123 would appear as:

1111 0001 1111 0010 1101 0011
1 2 – 3

After a Pack instruction is issued, the four-bit sign pattern occupies the four least significant bit positions of the field, and other zone bits are eliminated. A packed three-digit signed field, then, has this format:

digit digit digit sign

The digits and the sign code occupy four bit positions each. A minus 123, for example, has this bit configuration:

0001 0010 0011 1101
1 2 3 –

After a Convert to Binary instruction, a fixed-point operand occupies 31 bits of a word or 15 bits of a halfword. Another bit in the most significant position carries the sign, which is 0 for plus and 1 for minus (see Figure 13). Recall now that fixed-point operands are fixed in length. When the integer represented occupies less than a word or halfword, the sign bit is used to fill the unused high-order bit positions. The decimal number 4096, which we have seen is 1000000000000 in binary, can be represented in a halfword as 0001000000000000 if the sign is plus, or as 1111000000000000 in two's complement notation if the sign is minus. For a further explanation of complement notation see "Number Systems".

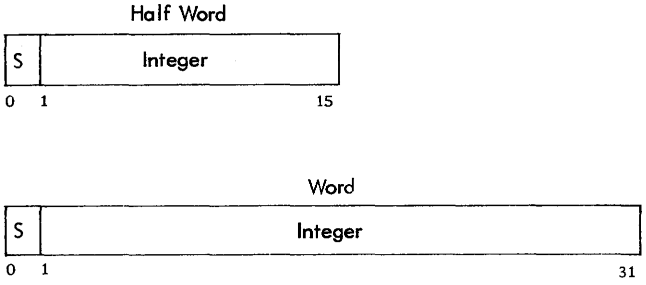


Figure 13. Fixed-point number format

Boundary Alignment

We have seen that System/360 can be used as a fixed-point binary computer with fixed-length operands and that it can perform decimal arithmetic on records characterized by many fields of varying length. A consecutive group of n bytes constitutes a field of length n. We need these variable- and fixed-length capabilities for the most efficient handling of both commercial and scientific applications. It should be emphasized that storage is addressable to the byte. Some instructions that address a byte always operate upon that byte and the next three consecutive bytes, so that a four-byte word is the operand. Other instructions require that the programmer specify as part of the instruction the number of bytes that constitute the operand.

Mention has been made of bytes, halfwords, and doublewords. Actually, as many as 256 bytes can be specified as operands in some instructions, such as data transfers.

Storage addresses within the system are represented by binary integers starting at zero. The location of a stored field is specified by the address of the leftmost byte of the field.

Boundary alignment is a programming restriction on fixed-length operands that requires some explanation. A variable-length field of data may start at any byte location. A fixed-length field of two, four, or eight bytes must have an address whose decimal equivalent is a multiple of two, four, or eight bytes respectively. A word address, for example, must be divisible by four. These are called integral boundaries. In binary, it turns out that the address must have:

- One low-order zero bit for a halfword
- Two low-order zero bits for a word
- Three low-order zero bits for a doubleword

Because the operation code is examined to determine whether fixed-length data is a halfword, word, or doubleword, the system can check to see that data is aligned on proper boundaries. A violation will cause a program interrupt that can be identified by the interruption code of the program status word as being "specification". Figure 14 shows various alignment possibilities.

The assembler language processor provides facilities that automatically position or allow us to force the required boundary alignments.

Boundary alignment restrictions were designed to force us to place words at consecutive integral boundaries to guarantee efficient machine operation when a program written for one model of System/360 is run on another model.

To illustrate, suppose that we correctly stored a halfword in location 512 and 513 and then incorrectly stored a series of fullwords beginning at location 514 (which is not divisible by 4). When we reference this data on a Model 50, which accesses a fullword on a single storage fetch, here is what would have to happen without boundary restrictions. An instruction that references the halfword at location 512 would also access half of the fullword beginning at location 514. Another storage access would be necessary to reference the other half of the fullword, and each successive fullword access would then fetch only half of the word we are seeking.

Thus, to guarantee efficiency and to maintain program compatibility among the various models, boundaries are identical for each model.

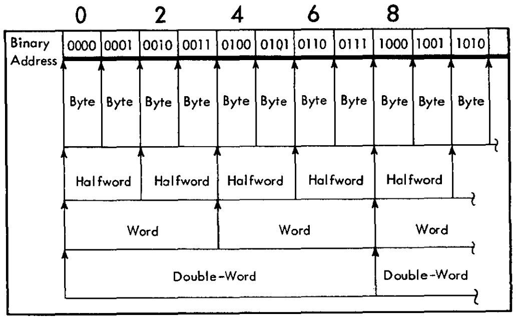


Figure 14. Integral boundaries for halfwords, words, and doublewords

General Registers and Storage Addressing

A set of 16 general purpose registers is standard. General registers can be used as index registers, relocation registers, accumulators for fixed-point arithmetic, and for logical operations.

Only four bits in an instruction are required to designate a register. Each register has a capacity of one 32-bit word.

Before considering the details of how these registers are utilized, it is helpful to see *why* registers were designed as part of the system.

Access time to storage increasingly limits performance as processor speeds improve. Using a single faster-access accumulator decreases overall processing time compared with the time required for storage-to-storage arithmetic. To efficiently utilize the single faster accumulator, however, it is necessary that data be refetched whenever it is reused and that results be stored temporarily for later use. Many of these fetch and store operations can be eliminated when multiple accumulators are available as registers.

Just as multiple registers improve the efficiency of arithmetic and logical operations, they can also provide a means of efficient address specification and modification.

Because the ability to address vast amounts of main storage is a desirable feature, an internal address of 24 binary bits is used. This permits up to 16,777,216 unique bytes to be addressed ($2^{24} = 16,777,216$).

An instruction, then, that involves a storage address would appear to require 24 bits to address the operand. Instead, instructions that designate a main storage location specify a register. A four-bit field in the instruction allows the specification of one of the registers numbered 0-15 as shown in Figure 15. The low-order 24 bits of this register contain an address referred to as the base address (B). The instruction must also contain a twelve-bit number called the displacement (D), which provides for relative addressing of up to 4095 bytes beyond the base address. The base and displacement are added together to produce an effective address.

Recall now that four bits of the instruction specify a register and twelve bits specify a displacement. With 16 bits we are able to specify a 24-bit address.

In addition to the base register, many System/360 instructions designate another general register called an index register. In these cases, the effective address is calculated by adding together the contents of the base register, the contents of the index register, and the displacement field (see Figure 16).

The contents of all general registers and storage locations participating in the addressing or execution part of an operation remain unchanged, except for the storing of the final result. This permits multiple instructions to reference a register containing the same base or index value.

Economy in instruction length through the use of the base-displacement addressing approach is one advantage of register utilization in addressing. Another significant advantage is the relocation facility provided. Since the instructions of a program reference registers, the contents of these registers can be specified at load time, so that programs and data can be located in main storage almost at will. When the pro-

R Field	Reg. No.	General Registers	Floating-Point Registers
0000	0	← 32 Bits →	← 64 Bits →
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 15. General purpose registers

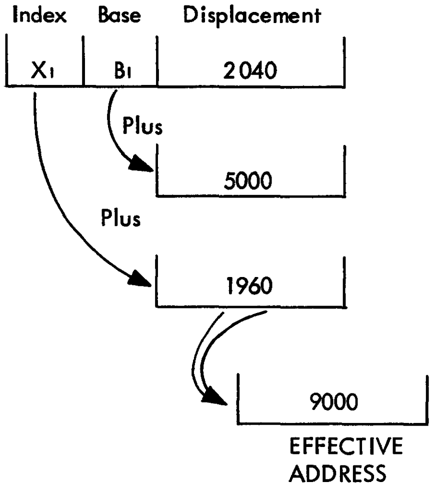


Figure 16. Address generation

gram is to be used at another time, other values can be specified in the base and index registers, so that the program can be executed from another segment of storage.

If, during the processing of a program, it is desirable to use these registers for other purposes, their contents can be stored in core storage. The registers would then be loaded with some other value, and processing continued. Note that the registers must be reloaded with their appropriate base values before executing a segment of the program that assumes the registers contain these values.

This approach of saving the contents of the registers and then restoring them as they are needed removes any limitation problem that might result from the fact that the system has only 15 registers usable for addressing. Register 0 cannot be used for address modification. A specification of 0 in either the base or index of an instruction means no base or index reference. This approach was taken to avoid the waste of having a register permanently filled with 0s when no indexing or when a base of 0 was desired. Certain instructions allow this register to be used as an accumulator, but when 0 is used in the base or index field, the system interprets it as meaning no base or index register.

There are multiple load and multiple store register instructions that make saving and restoration relatively simple operations.

The time spent in storing and restoring registers is quite small when compared with the time saved by having each instruction that references core storage contain only a 16-bit address field rather than a 24-bit address field. Similarly, the space used to preserve the contents of the registers is small compared with the space saved by reducing the instruction length.

Note that when we refer to a "base" or "index" we are referring to the use to which one of the 16 general purpose registers is being put, and not to a specialized register.

General registers are an important aspect of System/360. However, it is not only possible, but normal practice, to delegate to the assembly program almost all the clerical work of assigning base registers and computing displacements. Registers are used for addressing in a variety of ways. Some of the methods used in connection with the assembler language are examined under "Programming with Base Registers and the USING Instruction".

Relocation has been mentioned as one of the advantages of base-displacement addressing. Let us consider a simple situation in which we benefit from the ability to relocate programs. Assume that programs A and B are to be run together. Program A is located in 2000 consecutive storage locations as shown in Figure 17a. The next 3000 storage locations are occupied by program B. The following 2000 locations are unused, but, except for these locations, we shall consider that no other storage is available.

The next day program C, which requires 4000 bytes of storage, is to be run with program B. After looking at yesterday's storage map, we see that we have only 2000 consecutive locations available (either in the locations previously occupied by program A or in the unused area).

The register used on the previous day to load program B can have its contents modified by a load register instruction, so that today the base value is 2000 bytes higher than yesterday. Upon reloading program B, its starting address and all subsequent addresses will be 2000 positions higher. Thus we have relocated program B, and the last 2000 positions of program B will now occupy the storage segment previously unused. Four thousand consecutive locations are now available for program C, as shown in Figure 17b.

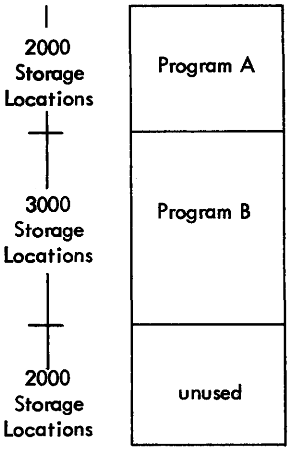


Figure 17a. Consecutive ascending locations in storage when program B is run with program A

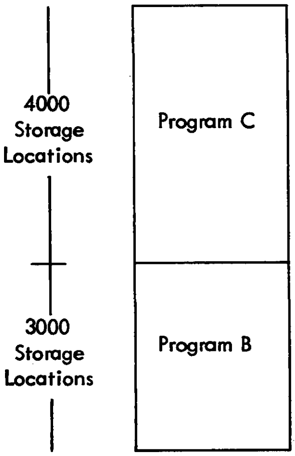


Figure 17b. Consecutive ascending locations in storage after relocation of program B to run with program C

Instruction Formats

We have seen that variable-length fields as well as words can be addressed. Instruction length is also variable. Some instructions cause no reference to main storage; others cause one or more references to main storage. To conserve storage space and save time in instruction execution, instruction length is variable and can be one, two, or three halfwords. Instructions specify the operation to be done and the location of data. Data may be located in main storage, registers, or a combination of the two. Instruction length is related to the number of storage addresses necessary for the operation. As a result, instructions will be of different lengths depending on the location of data. Instructions of different lengths can be arbitrarily combined in the same program.

When both operands are in registers, only eight binary bits are needed for register addresses. Since eight binary bits are used for the operation and eight bits for operands, the shortest instruction consists of one halfword and there is no reference to main storage.

When both operands are in main storage, a total of 32 bits are needed for the addresses (one four-bit base and one twelve-bit displacement for each of the two addresses) and, because the operation code and length specification(s) will require additional bits, the longest instruction (three halfwords in length) is used.

Figure 18 shows the five basic instruction formats. The format codes are RR, RX, RS, SI, and SS, which indicate the general locations of the operand or operands. RR denotes a register-to-register operation; RX, a register-to-indexed storage operation; RS, a register-to-storage operation; SI, a storage and immediate operand operation; and SS, a storage-to-storage operation. An "immediate operand" is a byte of data used as an operand that is carried in the instruction itself.

In the formats shown in Figure 18, R_1 specifies the address of the register containing the first operand. The second operand location, if any, is defined differently for each format.

In the RR format, the R_2 field specifies the address of the general register containing the second operand.

In the RX format, the contents of the general registers specified by the X_2 and B_2 fields are added to the contents of the D_2 field to form an address designating the storage location of the second operand.

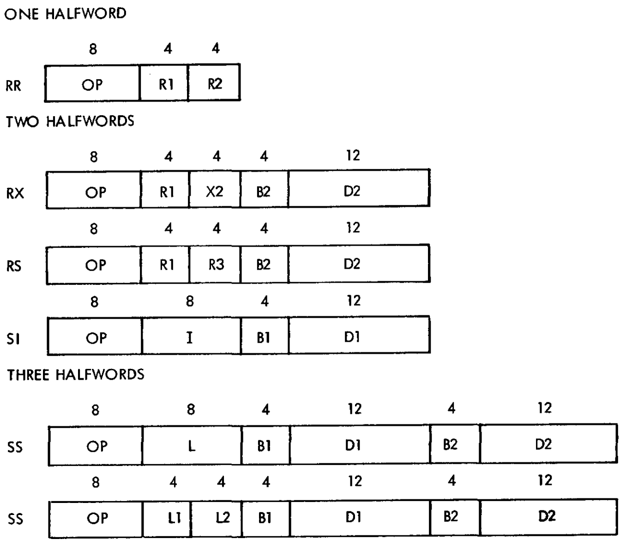


Figure 18. Instruction formats

The symbology employed in the RS format is explained with the example shown below. In shift operations employing the RS format, the designations of fields differ from the example shown, but this does not concern us here.

In most cases the results replace the first operand except for the Store instruction, and the Convert to Decimal instruction, where the result replaces the second operand.

The contents of all registers and storage locations participating in the addressing or execution part of an operation remain unchanged, except for the storing of the final result.

In the following examples of the instruction formats, the operands are expressed as decimal numbers, and the operation codes are expressed in the symbolic assembly language explained in this publication. Printouts of assembled programs (shown later in the text) are expressed hexadecimally. (The hexadecimal number system is explained under "Number Systems".)

RR Format

OP Code	R_1	R_2
AR	7	9
0 7	8 11	12 15

Execution of this Add instruction adds the contents of general register 9 to the contents of general register 7, and the sum is placed in general register 7.

RX Format

OP Code	R_1	X_2	B_2	D_2
ST	3	10	14	300
0 7	8 11	12 15	16 19	20 31

Execution of this Store instruction stores the contents of general register 3 at a main storage location addressed by the sum of 300 and the low-order 24 bits of general registers 14 and 10.

RS Format

OP Code	R_1	R_3	B_2	D_2
LM	3	9	11	300
0 7	8 11	12 15	16 19	20 31

This Load Multiple instruction causes the set of general registers starting with the register specified by R_1 and ending with the register specified by R_3 to be loaded from the locations designated by the second operand address.

The storage area from which the contents of the general registers are obtained starts at the location designated by the second operand address and continues through as many words as needed. The general registers are loaded in the ascending order of their addresses, starting with the register specified by R_1 and continuing up to and including the register specified by R_3.

It was pointed out earlier that the storing and restoration of registers is a relatively simple matter. There is also a multiple store instruction that provides for the storing of the registers, while this multiple load instruction provides for their restoration.

SI Format

OP Code	I_2	B_1	D_1
MVI	$	12	100
0 7	8 15	16 19	20 31

With this Move Immediate instruction in the example shown, a dollar sign ($) is to be placed in location 2100, leaving locations 2101-2105 unchanged. Let Z represent a four-bit zone. Assume that:

Register 12 contains	00	00	20	00	
Location 2100-2105 (before)	Z0	Z1	Z2	Z3	Z0
Location 2100-2105 (after)	$	Z1	Z2	Z3	Z0

SS Format

OP Code	L_1	L_2	B_1	D_1	B_2	D_2
AP	4	4	6	64	6	68
0 7	8 11	12 15	16 19	20 31	32 35	36 47

With this Add Decimal instruction, the second operand is added to the first operand, and the sum is placed in the first operand location. If necessary, high-order zeros are supplied for either operand. Note that in the register-to-register (RR) instruction example, the addition is on fixed-length binary fields.

The decimal arithmetic instruction in the SS format operates on data in the packed format with two decimal digits placed in one eight-bit byte. The length of the fields is specified explicitly in the instruction rather than implied in the operation code.

In each format (RR, RX, RS, SI, or SS) the first byte contains the operation code in the binary code, which is the actual machine language. In binary, the length and format of an instruction are specified by the first two bits of the operation code.

BIT POSITION	INSTRUCTION LENGTH	INSTRUCTION FORMAT
00	One halfword	RR
01	Two halfwords	RX
10	Two halfwords	RS or SI
11	Three halfwords	SS

During instruction decoding, the processing unit examines these first two bits of the operation code and determines how many bytes to fetch for this instruction. These bit configurations are part of the machine instruction, so that when, for example, we specify an Add register-to-register instruction, we are not concerned with specifying the instruction length.

We have seen that for fixed-length instructions the length of the *operand* is implicit in the instruction, and for variable-length operands the length is specified in the instruction. We have also seen that the length of the *instruction* itself is part of the operation code.

Storage Protection

System/360 was designed for operation with a supervisory program that schedules and governs the execution of multiple programs, handles exceptional conditions, and coordinates and issues input/output instructions.

In addition, the computing system and the supervisory programs are designed to prevent one program, such as a problem program, from modifying another program, such as the supervisor program. A means is provided by which the supervisor program can change any area of main storage, while the problem program can change only its own assigned areas. It is desirable, for example, that the supervisor program be able to change the main storage locations containing the new program status words. However, we would not want the problem program to be able to modify this same area. It is undesirable to have any part of the supervisor program changeable by the problem program. The feature that prevents the read-in of data into a protected area of core and thus prevents one program from destroying another is called storage protection.

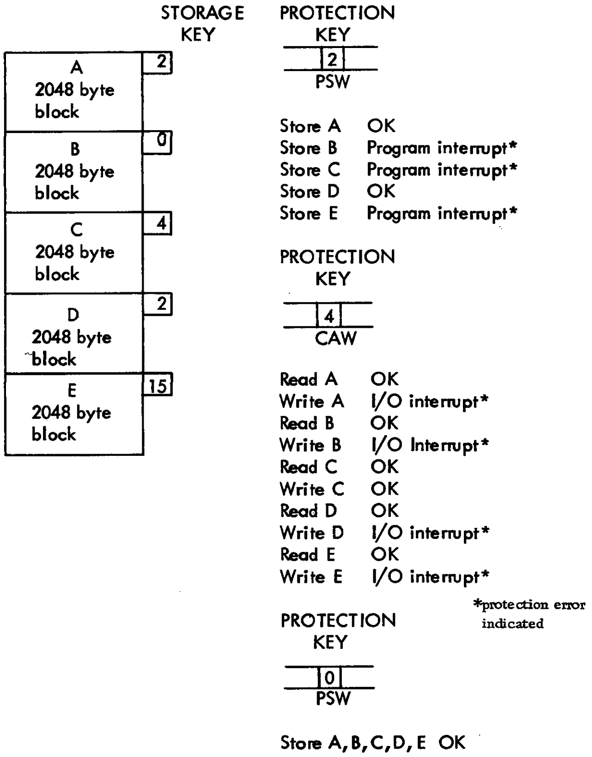
Storage protection is an optional feature on Models 30 and 40 and is standard on the larger systems. It has been pointed out that medium to large-scale systems are utilized most efficiently in a multiprogrammed environment and that the system is adept at handling communications applications involving more than one program. For such applications the supervisor program utilizing the storage protection feature assigns programs to particular areas of storage.

For protection purposes, main storage is divided into blocks of 2048 bytes each. Each 2048-byte block of storage has a four-bit register associated with it. The supervisory program may store any four-bit combination into any one of these registers. (Note that the supervisory program and not the problem program has access to these registers.) The four-bit combinations may be thought of as locks. Each block of storage, then, has its own lock.

The same lock may be assigned to more than one block and these blocks of 2048 bytes need not be contiguous.

The current PSW, as we have seen, acts as an instruction counter. Another of its functions is to keep track of the protection key of the program with which each instruction is associated. When a store operation is attempted by an instruction, the protection key of the current PSW is compared with the storage key of the affected block. When storing is specified not by a program instruction but by channel operation, a protection key supplied to the channel from the channel address word (CAW) is compared with the storage lock of the area in which the data is to be stored. Figure 19 illustrates a protection key for the channel address word in addition to the PSW protection key. The CAW is explained later under "Channel Organization". It has already been pointed out, however, that channels have their own programs, and to understand storage protection we should be aware that the protection key in the CAW provides protection on input operations from channels similar to that provided by the PSW on internal operations.

Storage takes place only if the key and lock combinations match or when the protection key is zero. Here *storage key* refers to the key stored in the register associated with a 2048-byte block of storage. *Pro-*



Store A	OK
Store B	Program interrupt*
Store C	Program interrupt*
Store D	OK
Store E	Program interrupt*

Read A	OK
Write A	I/O interrupt*
Read B	OK
Write B	I/O Interrupt*
Read C	OK
Write C	OK
Read D	OK
Write D	I/O interrupt*
Read E	OK
Write E	I/O interrupt*

*protection error indicated

Store A,B,C,D,E OK

Figure 19. Storage protection

tection key refers to the key contained in the PSW or channel. If the PSW, then, contains a nonzero protection key, a store operation will not occur in an area of storage with the zero key. If, on the other hand, the protection key is zero, a store operation can be executed using any area of storage without regard for its storage key. The supervisory program will sometimes require this zero master key in its PSW. The protection key of the current PSW in the problem program cannot be changed by the problem programmer, so interference with the supervisory program or with other programs is prevented.

When an instruction causes a protection mismatch, execution of the instruction is suppressed or terminated, and program execution is altered by an interrupt as shown in Figure 19.

Floating-Point Arithmetic

In fixed-point computation the position of digits must be aligned for each operand to express their integral or fractional value. The separation of the integral and fractional portion of a number denoted by a point in written notation is the programmer's responsibility.

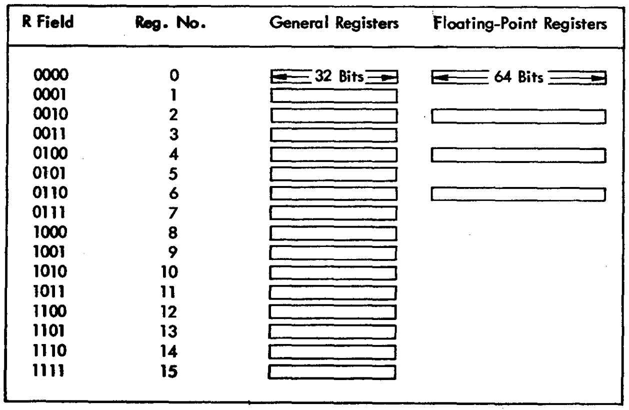
Scientific and engineering computations often involve multiplications and divisions where the magnitude of the quantities involved varies from very small fractions to large integers.

To relieve the programmer of the responsibility of shifting to position intermediate and final results, floating-point notation and circuitry to operate upon it have been characteristics of scientific computers. Floating-point arithmetic is an optional feature on Models 30 and 40 and is standard on the higher-performance models.

Four 64-bit floating-point registers identified by the numbers 0, 2, 4, and 6 are provided, as shown in Figure 20. The operation code determines whether a general purpose or floating-point register is to be used in an operation. An attempt to execute a floating-point instruction on a system not equipped with the feature will result in a program interrupt.

The notation used permits representation of numbers whose decimal equivalents have magnitudes in the range of 10^{-78} to 10^{+75}.

In this introduction to the system's structure, we shall not go into the details of floating-point arithmetic. It is interesting to note that either a short (32-bit) or long (64-bit) format operand may be specified. The short-length, equivalent to seven decimal places of precision, permits a maximum number of operands to be placed in storage and gives the shortest execution time. The long-length, used when higher precision is desired, gives up to 17 decimal places of precision. The formats differ only in the length of the fraction, as shown in Figure 21.



R Field	Reg. No.	General Registers	Floating-Point Registers
0000	0	32 Bits	64 Bits
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 20. General and floating-point registers

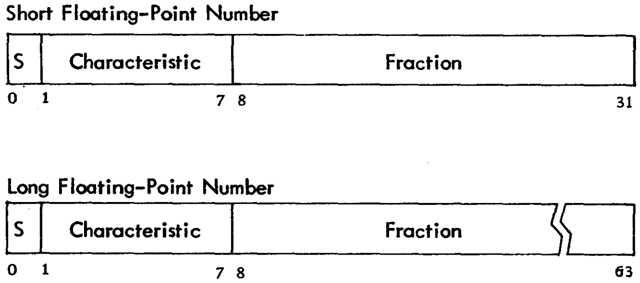


Figure 21. Short and long floating-point number formats

Channel Organization

In the section entitled "Channel Concept" mention was made of communications between the processing unit and the channel. We shall now examine in more detail the ways in which the processing unit, the channels, the control units, and the I/O devices communicate with each other.

System/360 is designed for use in conjunction with a supervisor program that allocates equipment to multiple programs and also monitors the execution of each problem program. The supervisor program must also monitor I/O operations. To permit unrelated problem programs to execute I/O operations concurrently, the channel hardware together with the supervisor program provides a means of assigning to each program the required I/O facilities. This assignment consists of establishing a path not only for transferring data between the I/O device and the designated area of main storage, but also for exchanging control and status information between the program and the I/O facility.

Input/output control units are attached to the channel by a standard connection, called the I/O interface. This interface is common to all channels and control units. It provides an information and signal sequence that is common to all types of I/O control units. The interface has nine one-way lines for input and nine lines for output to accommodate one byte including parity. Other lines carry status and control information. The important thing to remember is that identical lines are used for all control units including those for tape, disk, card, etc. The channel operates the control unit, and the control unit is designed to meet the interface requirements.

The control unit operates the actual device. Examples of control units are tape control, communications control, card control, and printer control. The channel, in turn, operates the control unit. The processing unit controls channel activity by means of four *instructions*:

Start I/O
Test I/O
Halt I/O
Test Channel

Commands constitute the channel program. The channel programs are held in main storage until an I/O operation is initiated by a Start I/O instruction. A channel address word (CAW) is permanently assigned to contain the address of the initial channel command word (CCW) (see Figures 22, 23, and 24). CCW's are decoded by the channel, which issues *orders* to the I/O device.

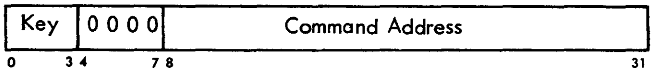
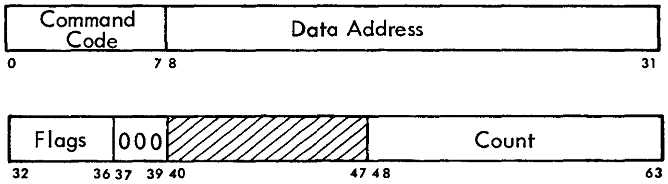


Figure 22. Channel address word format



Bits 0-7 specify the command code.
Bits 8-31 specify the location of a byte in main storage.
Bits 32-36 are flag bits.
Bit 32 causes the address portion of the next CCW to be used.
Bit 33 causes the command code and data address in the next CCW to be used.
Bit 34 causes a possible incorrect length indication to be suppressed.
Bit 35 suppresses the transfer of information to main storage.
Bit 36 causes an interruption as Program Control Interrupt
Bits 37-39 must contain zeros.
Bits 40-47 are ignored
Bits 48-63 specify the number of bytes in the operation.

Figure 23. Channel command word format

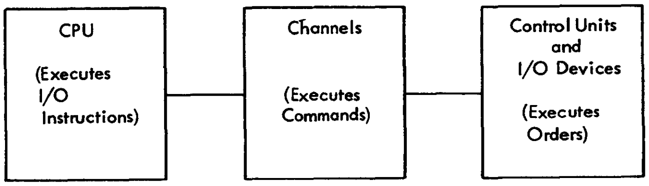


Figure 24. Relationship of I/O instructions, commands, and orders

The CCW contains the command to be executed, and for commands that initiate I/O operations it designates the storage area associated with the operation and the action to be taken whenever transfer to or from the area is completed. The CCW's can be located anywhere in main storage on doubleword boundaries, and more than one can be associated with a Start I/O instruction. The channel refers to a CCW in main storage only once, whereupon the pertinent information is stored in the channel.

The first CCW is fetched during the execution of Start I/O. Each additional CCW in the chain is obtained when the operation has progressed to the point where the additional CCW is needed.

The CCW has the format shown in Figure 23.

Bits 0-7 specify the operation to be performed. There are six valid commands:

Sense
Transfer in Channel
Read Backward
Write
Read
Control

The data address specifies the location of an eight-bit byte in main storage. It is the first location referred to in the area designated in the CCW.

The count specifies the number of eight-bit byte locations beyond the initial byte designated by the address.

It has been mentioned that channels function much like small independent computers. As such they contain registers. Bits 32 through 36 of the CCW are labeled "flags" (see Figure 23). The channel registers include a flag register that indicates command modes. These flags serve to chain data or commands for this series of CCW's, interrupt the processing unit, skip a portion of a record, suppress length indication, or terminate the operation.

These flags may be set on or off in each of the channel control words and the flag register is updated with each new CCW. Other registers within the channel circuitry are (1) a command counter, which tells the channel where to get the next command in a manner similar to that of an instruction counter in a processing unit, (2) a command register, which tells the channel which command is to be performed, (3) an address register, which tells the channel where to get or put data into core storage, (4) a count register, which indicates how many characters are to be read or written, and (5) a key register, which contains the storage protection key for the current operation.

The generalized CCW commands listed earlier apply to all devices. Read, Write, and Read Backward are self-explanatory. The Sense command is a request to the I/O control unit for device-dependent status information, such as the position of magnetic tape, the condition of the card stacker and hopper, or the detailed conditions detected in the last operation. This status information is transferred to the channel as data and is placed in the main storage area designated by the CCW.

Normally the detailed information provided by the sense command is not required, and an eight-bit status byte is provided to the channel (upon completion of an I/O operation) indicating the general conditions detected during the operation. This status byte is common to all I/O devices and cannot convey the detail conditions of termination provided by the sense command.

A control command causes the control unit to initiate at the I/O device an operation not involving the transfer of data — such as backspacing or rewinding magnetic tape, or positioning a disk access mechanism.

The Transfer in Channel command causes the next CCW to be fetched from the location designated by the data address field of this command instead of fetching the next sequential CCW. In effect, then, the Transfer in Channel command causes a branch from one sequence of CCW's to another.

When command chaining is specified by a flag bit in the CCW, the channel uses the new CCW to initiate a new operation at the device and permits the processor program to start with a single I/O instruction such sequences as printing multiple lines or reading multiple tape blocks. With command chaining it is possible for the channel to execute I/O programs of any number of I/O operations.

When data chaining is specified by a flag bit in the CCW, the channel uses the new CCW to designate another data area for the original I/O operation and the device continues to execute this operation. Only the allocation of storage areas is affected. Data chaining permits the reorganization of information as it is transferred between main storage and the I/O device.

The proper use of the available channel command words permits the following types of I/O functions:

Scatter-read — reading one physical record into multiple, noncontiguous areas of storage.

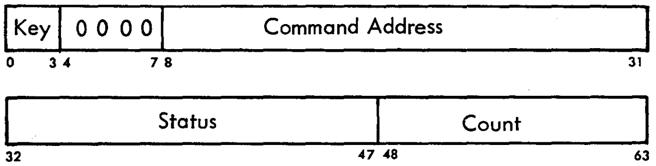
Extraction — reading only selected portions of a record into storage.

Control nondata I/O operations — for example, backspace, rewind, etc.

Command chaining — for sequentially performing operations on the same device, for example, reading over an interrecord gap.

Upon completion of the channel program, an I/O interrupt occurs; that is, the channel interrupts the processing unit. The channel makes available in main storage a channel status word (CSW). This doubleword contains an address that is eight bytes higher than the address of the last CCW used, and indicates in the count field the difference between the count in the last CCW and the amount of data transferred. The format of the channel status word is shown in Figure 25. The storage protection key is the key used in the operation. It is first supplied to the channel from the CAW as a result of a Start I/O instruction.

Bits 32-47 of the channel status word contain an eight-bit I/O device-status byte and a channel status byte. These two bytes provide such information as data-check, chaining check, and control unit end. The channel status word has a permanent storage assignment of locations 64 through 71 in main storage as shown in Figure 4.



Bits 0-3 contains the storage-protection key used in the operation.

Bit 4-7 contain zeros.

Bits 8-31 specify the location of the last CCW used.

Bits 32-47 contain an I/O device-status byte and a channel-status byte. The status bytes provide such information as data-check, chaining check, control-unit end, etc.

Bits 48-63 contain the residual count of the last CCW used.

Figure 25. Channel status word format

With the command address, status, and count fields of the CSW, the program can determine the status of an I/O device or the conditions under which an I/O operation has been terminated.

The processing unit's program depends on I/O interrupts for information concerning the progress of I/O operations. So that the processor program can tell in advance when conditions in the channel or in the device should alert the program, a mask bit is associated with each channel. A masked channel cannot cause an I/O interrupt, and consequently the supervisor program can suppress I/O interrupts by masking the channels. The conditions in the channels and devices are preserved until accepted by the processor program. The program can determine whether an interrupt condition is pending in the channel by issuing the instruction Test Channel.

Channel masking allows the processor program to accept I/O interrupts selectively by channel. However, on a given channel more than one I/O control unit can contain pending conditions that cause program interruption. The instruction Test I/O allows a program to accept interrupts selectively by I/O device. This instruction gives the program the status of the designated device and clears any interrupt condition pending in the device. Test I/O provides the same information as an I/O interrupt, since the channel status word is stored. Keeping the channels masked and interrogating devices by the Test I/O instruction prevents the program from being interfered with by conditions unrelated to the program being run.

In a real-time or communications environment, on the other hand, the processor program would keep all channels unmasked and depend on I/O interrupts for information concerning the progress of I/O events as they occur.

Summary

System/360 includes provisions for large storage capacity, simple program relocation, flexible protection, and general supervisory facilities. Provisions are also included for a variety of data formats, an extensive set of processing operations, and machine language compatibility among the various models.

To compensate for higher computational speeds relative to human reaction time, and to adapt the system to online and real-time multiprogramming tasks, the system is more highly automated by having the system resources controlled by a supervisory program. Provision for this control is embodied in these concepts:

- Supervisory mode with associated privileged instructions
- Storage protection
- Hardware monitoring
- The ability to perform interrupts
- A wait state available to the supervisor program, rather than a stop or halt instruction available to the problem programmer.

Questions and Exercises

1. Can a tape unit be attached to a multiplexor channel?
2. If the problem program issues a Load PSW instruction to cause the new I/O PSW to be loaded, can the problem program cause an I/O operation to be executed?
3. The instruction address contained in the new Supervisor Call PSW addresses a routine to handle this class of interrupts. What action must this routine first take?
4. A program interrupt will occur if the Convert to Binary instruction attempts to operate upon data that contains invalid codes for packed decimal. What are the valid four-bit codes for packed decimal?
5. Is data punched in an IBM card as Hollerith code acceptable as input to a System/360 equipped with a card reader?
6. If floating-point arithmetic is intended for scientific and engineering applications, while the decimal instruction set is primarily for commercial applications, by whom are fixed-point arithmetic instructions used?
7. In what position is the sign of a number located?
8. What storage location is addressed by an instruction with zeros in the index and displacement fields and the number 5 in the base register field?
9. Why does the programmer select a particular instruction length?

Chapter 2: Number Systems

Numeric symbols, or numbers, were invented to facilitate counting. Various number systems differ in the arrangement and type of number symbols used. Early number systems frequently employed cumbersome symbols and inconvenient rules, which hindered the advance of systematic mathematical thought. The slowly increasing rationality of systems of numerical notation and the arithmetical rules built upon them bears a close relation to the progress of mathematics and science, in general.

The following sections present a brief review of the basis of modern positional number systems and the arithmetical manipulation and conversion of the binary and hexadecimal systems of notation, both of which have been found useful in electronic data processing.

Positional Notation

The Arabs invented the numerical symbols and system of positional notation on which our present decimal system and other number systems are based. Each of the symbols has a fixed value one higher than that of the symbol before it in the progression from smallest to largest: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. When several symbols (or digits) are combined, the value of the number depends upon the relative positions of the individual digits, as well as on the digit values. In any system of positional notation, the digit position on the extreme right is the one of least value, or lowest order, and is called the "least significant digit" (LSD); the digit on the extreme left is the one of highest value and is called the "most significant digit" (MSD). The increase in value of each digit position depends on the *base*, or *radix*, of the number system. Thus, in the decimal system, with base 10, the value of the digit positions to the left of the least significant (or unit) digit, increases by a power of 10 for each position. The decimal system has the base (radix) 10 because it has ten discrete number symbols (0-9) available for counting.

As an example of positional notation, consider the decimal number 6,594. Although its value is immediately apparent, the notation 6,594 actually signifies

$$\begin{aligned} & 6 \text{ thousands} + 5 \text{ hundreds} + 9 \text{ tens} + 4 \text{ units} \\ \text{or}\quad & 6000 + 500 + 90 + 4 = 6{,}594 \end{aligned}$$

The positional value of each digit is made even clearer when the number is expressed in powers of ten:

$$\begin{aligned} 6{,}594 &= 6 \times 10^3 + 5 \times 10^2 + 9 \times 10^1 + 4 \times 10^0 \\ &= 6 \times 1000 + 5 \times 100 + 9 \times 10 + 4 \times 1 \end{aligned}$$

Positional notation is not possible without the zero. Its presence within a number simply means that the power of the base represented by the 0 digit position is not used. Thus, the decimal number 8,003 signifies

$$\begin{aligned} & 8 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 3 \times 10^0 \\ &= 8 \times 1000 + 0 \times 100 + 0 \times 10 + 3 \times 1 \\ &= 8{,}000 + 3 = 8{,}003 \end{aligned}$$

Fractions and mixed numbers are treated in positional notation in just as simple a fashion. Each digit position to the *right of the point* is assigned a negative power of the base, starting with –1, in ascending sequence. Thus, in the decimal system, the first digit to the right of the decimal point is multiplied by 10^{-1}, the second digit by 10^{-2}, the third by 10^{-3}, and so on. For example, the mixed decimal number 436.578 may be expressed as

$$\begin{aligned} & 4 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 5 \times 10^{-1} + \\ & \qquad 7 \times 10^{-2} + 8 \times 10^{-3} \\ &= 400 + 30 + 6 + 5/10 + \\ & \qquad 7/100 + 8/1000 \\ &= 436.578 \end{aligned}$$

These rules of positional notation are generally applicable to all number systems, regardless of the base, or radix, used.

The binary (base 2) number system uses only two distinct symbols, 0 and 1, which signify "no units" and "one unit", respectively. In contrast to the decimal system, however, the place value of binary digits to the left of the least significant digit (LSD) increases by a power of 2 each time, rather than by powers of 10. For example, the binary number 101101 signifies:

$$\begin{aligned}101101 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\\ &= (1 \times 32) + 0 + (1 \times 8) + (1 \times 4) + 0 + (1 \times 1)\\ &= 32 + 0 + 8 + 4 + 0 + 1\\ &= 45 \text{ (in the decimal system).}\end{aligned}$$

Expressing a binary number in powers of 2, thus, is one way (though not usually the best) of finding its decimal equivalent. To avoid confusion when several systems of notation are employed, it is customary to enclose each number in parentheses and to write the base as a subscript, in decimal notation. Using the previous example:

$$(101101)_2 = (45)_{10}$$

Fractions are handled in the same way by assigning negative powers of 2 to the right of the *binary point* in ascending sequence. For instance, the binary number 0.1011 means:

$$\begin{aligned}(0.1011)_2 &= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}\\ &= 1/2 + 0 + 1/8 + 1/16 = 11/16\\ &= 0.5 + 0 + 0.125 + 0.0625 = (0.6875)_{10}\end{aligned}$$

Again, a literal expansion of the binary number in powers of 2 yields the decimal equivalent. Simpler methods of conversion will be described later on.

For reference, the first 16 binary numbers and their decimal equivalents are:

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

Hexadecimal Numbers

Large binary numbers consist of long strings of zeros and ones, which are frequently awkward to interpret and handle. The hexadecimal (base 16) numbering system is used as a convenient way of representing such large binary numbers. Each hexadecimal digit stands for four binary digits.

Hexadecimal notation requires the use of 16 symbols to represent 16 number values. Since the decimal system provides only ten number symbols (0 – 9), six additional marks are needed to represent the remaining values. The letters A, B, C, D, E, F have been adopted for this purpose, though any other six marks could have served equally well. The entire list of hexadecimal symbols, thus, consists of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D E, and F, in ascending sequence of value. Table 1 shows equivalent decimal, hexadecimal, and binary numbers (through decimal 31). Note that upon reaching decimal 16, the hexadecimal symbols are exhausted, and a "1 carry" is placed in front of each hexadecimal symbol during the second cycle, from decimal 16 through 31.

To convert binary numbers to hexadecimal notation, simply divide the number into groups of four binary digits, starting from the right, and replace each group by the corresponding hexadecimal symbol. If the left-hand group is incomplete, fill in zeros as required. For example, the binary number

$$\begin{aligned}111110011011010011 &= 0011/1110/0110/1101/0011\\ &= \;3 \quad E \quad 6 \quad D \quad 3\\ &= (3E6D3)_{16}\end{aligned}$$

If the binary number is a fraction or a mixed number, care must be taken to mark off groups of four bits from each side of the binary point position. Thus, the binary number

$$\begin{aligned}1011001010.1011011 &= 0010/1100/1010.1011/0110\\ &= \;2 \quad C \quad A \quad B \quad 6\\ &= (2CA.B6)_{16}\end{aligned}$$

Similarly, to convert hexadecimal numbers into binary, substitute the corresponding group of four binary digits for each hexadecimal symbol and drop off any unnecessary zeros. For instance, the hexadecimal number

$$\begin{aligned}&(6C4F2E.7B8)_{16}\\ &= 6 \quad C \quad 4 \quad F \quad 2 \quad E \quad 7 \quad B \quad 8\\ &= 0110\ /1100/\ 0100/1111/0010/1110.0111/1011/1000\\ &= (11011000100111100101110.011110111\ )_2\end{aligned}$$

Decimal	*Hexadecimal*	*Binary*
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
18	12	10010
19	13	10011
20	14	10100
21	15	10101
22	16	10110
23	17	10111
24	18	11000
25	19	11001
26	1A	11010
27	1B	11011
28	1C	11100
29	1D	11101
30	1E	11110
31	1F	11111

Table 1. Decimal, hexadecimal, and binary notation

The meaning of hexadecimal numbers is made clear by expansion in powers of 16. For example, the hexadecimal number 2CA.B6, above, means (when decimals are substituted for hexadecimal symbols)

$$\begin{aligned}&2 \times 16^2 + 12 \times 16^1 + 10 \times 16^0 + 11 \times 16^{-1} + 6 \times 16^{-2}\\ &= 2 \times 256 + 12 \times 16 + 10 \times 1 + 11/16 + 6/256\\ &= 512 + 192 + 10 + 0.6875 + 0.0234375\\ &= 714 + 0.7109375 = (714.7109375)_{10}\end{aligned}$$

In working out an example of this type, it is best to arrange the products in a vertical column for convenient addition.

Binary and Hexadecimal Arithmetic

Arithmetic in bases other than 10 can always be carried out by converting all operands to the decimal system, doing the required arithmetic, and then reconverting the results to the original number base. This procedure is not recommended for binary arithmetic, which is extremely simple, but may be advisable for complicated hexadecimal arithmetic, particularly when a good hexadecimal-decimal conversion table is available. (See, for example, the conversion table in Appendix E of the manual *IBM System/360 Principles of Operation,* A22-6821.) Nevertheless, the programmer should be familiar, at least, with simple addition and subtraction in the binary and hexadecimal notations and, therefore, examples of these operations are included.

The rules of arithmetic are the same in all positional number systems. Thus, it is necessary only to recall the corresponding rules of decimal arithmetic to be able to do arithmetic in any other number base.

Binary Addition

Addition is essentially a shortcut to counting. We add two digits either by counting through the values of the two digits in sequence or, more simply, by memorizing the sum of the digits from an addition table. Whenever the sum of two digits exceeds the available number symbols of the notation (that is, the limit of any digit position), a 1 is carried to the next-higher-order digit position. Thus, in the decimal system, $3 + 5 = 8$, but $9 + 1 = 0$ with a carry of 1 (that is, 10).

In the binary system, there are only two symbols, 0 and 1. Hence, adding 1 plus 1 in binary notation exceeds the limit of counting (no symbol being available) and, therefore, the result is 0 with a 1 carried to the next-higher-order digit position. The complete rules of binary addition are given below.

Binary addition

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 0$ with a carry of 1.

(This may be written as 10, but is pronounced "one, zero".) The binary addition table below is a convenient way of summarizing these results.

Binary addition table

+	0	1
0	0	1
1	1	10

Three examples of binary addition are given below. The example on the left is self-explanatory. The center example develops a carry, which is indicated above the proper digit position. The example on the right consists of the addition of two eight-bit numbers and involves several carries, which are indicated. As a check, the binary operands also have been converted to decimals, and the addition has been carried out in both systems. The results check, as you can verify by conversion.

```
             1 ←Carries→ 1   11
  1010    101010      00111001 =   57
+  101  + 001001    + 00100011 = + 35
------  --------    ----------   ----
  1111    110011      01011100 =   92
```

It is frequently necessary to add two 1's and a 1 carry from a lower-order position. This results in a 1, with a carry of 1 to the next-higher-order position. In brief,

$1 + 1 + 1 = 1$ with a carry of 1 (which may be written 11). The following two examples illustrate the process

```
Carries  111     111
         1111   10111000 =   184
       +  111 + 00111011 = +  59
       ------ ----------   -----
        10110   11110011 =   243
```

When adding together several binary numbers, more than one carry may be developed to a single column.

```
Carries {  1  1
        { 11111
            1011
            1101
            1001
            0001
            1001
          ------
          101011
```

Additional exercises in binary addition can be found later.

Hexadecimal Addition

Addition in the hexadecimal system follows the same rules as decimal and binary addition. Working with alphameric symbols – numbers and letters – appears strange at first, particularly since results long familiar from decimal addition have a different meaning in hexadecimal notation. This requires a degree of reorientation and practice. For instance, while 4 + 5 = 9 in both the decimal and hexadecimal systems, 7 + 8 = F (*not* 15) in hexadecimal notation. Whenever the sum of two digits exceeds F – the highest-valued hexadecimal symbol – a carry of 1 is developed to the next-higher-order digit position. Thus, 7 + 9 = 10 (that is, 0 with a carry of 1), 9 + 9 = 12 (that is, 2 with a carry of 1), C + 9 = 15, and so on. (Refer to Table 1.)

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
1	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	1
2	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	2
3	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	3
4	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	4
5	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	5
6	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	6
7	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	7
8	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	8
9	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	9
A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	A
B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	B
C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	C
D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	D
E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	E
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	F
	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Table 2. Hexadecimal addition

The hexadecimal addition table (Table 2) has 16 × 16 = 256 entries, which one could hardly be expected to memorize. Hence, reference to the table is required during hexadecimal addition. The use of the table is simple.

Locate the two hexadecimal digits in the respective row and column of the table. (It makes no difference which digit is selected for a column and which for a row.) The sum of the two digits is given by the *intersection* of the row and column. Note that the highest entry, at the bottom right of the table, is 1E, which represents the sum of F plus F (decimal 15 + 15). If a carry of 1 needed to be added to this, the result would be 1F (equivalent to decimal 31); that is, F + F + 1 = 1F. In general, if a carry develops during hexadecimal addition, it is convenient to mentally add the 1 carry to the *lower-valued* of the two operands and then add in the other operand by use of the addition table. As an alternative, the two digits may first be added by use of the table; the carry is then added in by going to the next square to the right or below the intersection that represents the sum of the digits. For example, to add the digits 7 + C + 1 (carry), one may add 7 plus 1 (carry) equals 8, and then look up the result of adding 8 (column) plus C (row) at the intersection of the 8-column and the C-row, which shows 14. Alternatively, one can add 7 (column) plus C (row) by use of the table, finding 13 at the intersection; the 1 carry is then added in by going one square below (or to the right of) the intersection, which again yields 14 as the result. Three examples of hexadecimal addition follow. Additional exercises are given at the end of the chapter.

Hexadecimal addition

		11	1 1
9654 =	38,484	6AE	8F97
+ 4528 =	+ 17,704	+ 1FA	+ D44C
DB7C =	$(56,188)_{10}$	8A8	163E3

The example at left is straightforward and does not involve any carries. To verify the results, each of the operands has also been converted into decimals (by expansion in powers of 16) and the addition carried out in the decimal system. The center example, which does involve carries, may be verbalized as follows, using Table 2: A plus E equals 8 with a carry of 1 into the next-higher-order digit position. Adding the next-higher-order digits, F plus A equals 9 with a carry of 1, plus the lower-order carry equals A, with a carry of 1 into the next-higher-order digit position. Adding the next-higher-order digits, 1 plus 6 equals 7, plus the carry of 1 equals 8. This completes the addition. Similarly, adding the lowest-order digits of the example at right, we obtain C plus 7 equals 3, with a carry of 1 into the next-higher-order digit position. Adding the next-higher-order digits, 4 plus 9 equals D, plus the carry of 1 equals E with no carry. (Alternatively, 4 plus the 1 carry equals 5, plus 9 equals E with no carry.) Addition of the next-higher-order digits yields 4 plus F equals 3 with a carry of 1 to the next-higher-order position. Adding these digits, D plus

8 plus a 1 carry equals 6 with a carry of 1; since there is no higher-order position, this carry is placed next to the 6. This completes the addition.

Binary Subtraction

In checking out programs, it may be useful to be familiar with the conventional, direct method of subtraction, "borrowing" whenever necessary. The rules of direct binary subtraction are given below.

Binary subtraction

$$0 - 0 = 0$$
$$1 - 1 = 0$$
$$1 - 0 = 1$$
$$0 - 1 = 1 \text{ with a borrow of } 1$$

(from the next-higher-order digit position). By incorporating the borrow, the last rule above may be interpreted to mean

$$10 - 1 = 1$$

(which is equivalent to decimal $2 - 1 = 1$).

Borrowing is necessary whenever the subtrahend (the number on the bottom) is larger than the minuend (the number on top). It consists of subtracting a 1 from the next-higher-order digit to the left in the minuend and placing it next to the lower-order digit in the minuend. Since additional higher-order borrows are frequently required, the process is often confusing. An alternative, and frequently better, way is to "pay back", or carry, the 1 borrowed from the minuend digit at left to the subtrahend immediately beneath it. An example illustrates the method.

Conventional method *Payback method*

$$\begin{array}{lccccccccccccccc}
 & & & & 10 & & & & & & & & & & & \\
\text{Borrows} & & & 0 & \cancel{0} & 10 & & & & & & & 1 & 1 & & \\
 & 1 & 0 & \cancel{1} & \cancel{1} & \cancel{0} & 1 & 1 & \leftarrow\text{Minuend}\rightarrow & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
 & -1 & 0 & 0 & 1 & 1 & 1 & 1 & \leftarrow\text{Subtrahend}\rightarrow & -1 & 0 & 0 & 1 & 1 & 1 & 1 \\
\cline{2-8}
 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & \text{Carry} & & & 1 & 1 & & & \\
\cline{10-16}
 & & & & & & & & & 0 & 0 & 0 & 1 & 1 & 0 & 0
\end{array}$$

Note, in the conventional method (at left), the many changes necessary in the minuend to accommodate successive borrowing. This may become confusing at times and it is best to write down the minuend once again after all borrowing is completed.

In the payback method of subtraction, a borrow of 1 is simply placed next to any minuend digit that requires it. This 1 is then paid back as a carry of 1 to the next-higher-order subtrahend digit at the left. Although the method is easily carried out in one's head, the example above could be verbalized as follows. Starting with the lowest-order digit position, 1 from 1 leaves 0 (put down 0); 1 from 1 leaves 0 (put down 0); 1 from 10 (after borrowing) leaves 1 (put down 1 and carry 1 to the next-higher-order subtrahend digit); the carry of 1 plus 1 equals 10, and 10 from 11 (after borrowing) leaves 1 (put down 1 and carry 1 to the next-higher-order subtrahend digit); the carry of 1 plus 0 equals 1, and 1 from 1 leaves 0 (put down 0); next, 0 from 0 leaves 0 (put down 0), and finally, 1 from 1 leaves 0 (put down 0). This completes the subtraction.

Three more examples of binary subtraction are given below. The method has not been indicated, since either one can be used, according to individual preference. The additional exercises at the end of this chapter should also be completed at this time.

Binary subtraction

$$\begin{array}{rrrcr}
1000 & 10110001 & 110011 & = & 51 \\
-\ \ 1 & -01010101 & -011101 & = & -29 \\
\hline
111 & 01011100 & 010110 & = & 22
\end{array}$$

Hexadecimal Subtraction

Subtraction in the System/360 is actually carried out by complementing the subtrahend and adding it to the minuend (see "Complements" later in this manual). It is useful for the programmer to know, however, how to perform direct subtraction of hexadecimal numbers.

Hexadecimal subtraction follows the same rules as decimal and binary subtraction with the proviso that a carry or borrow of 1 in hexadecimal notation represents decimal 16. To obtain the difference of two hexadecimal digits, refer to Table 2. Locate the column heading that represents the digit to be subtracted (subtrahend). Go down this column to the digit(s) that represents the minuend. The heading of the row horizontally across from the minuend represents the difference between the two digits. When the subtrahend digit is greater than the minuend digit, it will be necessary, of course, to add in a borrow of 1 to the minuend digit before looking up the difference in the table. Either the conventional or the payback method of subtraction can be used, as is illustrated in the following two examples:

Hexadecimal subtraction

1. *Conventional method*

$$\begin{array}{lcccl}
\text{Borrows:} & & 19 & & \\
 & 7 & \cancel{9} & 18 & \\
 & \cancel{8} & \cancel{A} & \cancel{8} & \text{Minuend} \\
 & -1 & F & A & \text{Subtrahend} \\
\cline{2-4}
 & 6 & A & E &
\end{array}$$

2. *Payback method*

$$\begin{array}{lccccccl}
 & & & 1 & 1 & & 1 & \\
 & & 1 & 6 & 3 & E & 3 & \text{Minuend} \\
 & - & & D & 4 & 4 & C & \text{Subtrahend} \\
\text{Carries:} & & 1 & 1 & & 1 & & \\
\cline{2-7}
 & & 0 & 8 & F & 9 & 7 &
\end{array}$$

Example 1, above, illustrates the borrowing method. Starting with the lowest-order digits at right, A cannot be subtracted from 8, since it exceeds 8. Hence, a 1 is borrowed from the next-higher-order digit at left, A, reducing that digit to 9 (since A − 1 = 9) and increasing the minuend digit to 18. To carry out the subtraction 18 − A, Table 2 is consulted. Under the A-column (the subtrahend), the minuend digits, 18, appear in the E-row. Hence, 18 minus A equals E. Put down E. Proceeding to the next-higher-order digit position, F cannot be subtracted from 9; hence a 1 is borrowed from the 8 at left, reducing that digit to 7, and increasing the minuend to 19. In the table, going down in the F-column (subtrahend), the minuend digits, 19, appear in the A-row. Therefore, 19 minus F equals A. Put down A. Finally, the difference between the high-order digits at left, 7 minus 1, equals 6. Put down 6. This completes the subtraction.

Example 2, above, illustrates the payback method of hexadecimal subtraction. Starting with the low-order (right) digits, C cannot be subtracted from 3; hence, add in a 1 (actually a 10, of course), making it 13. From Table 2, 13 minus C equals 7. Put down 7. In the next-higher-order digit position, a carry of 1 is first added to the subtrahend; 4 plus 1 equals 5. Then, from the table, E minus 5 equals 9. Put down 9. In the next digit position at left, 4 cannot be subtracted from 3, but after borrowing, 13 minus 4 equals F (from the table). Put down F. The 1 previously borrowed is added to the subtrahend of the next digit position; D plus 1 equals E. After borrowing, 16 minus E equals 8 (from the table). Finally, 1 minus the carry of 1 equals 0. Put down 0. Alternatively, the two high-order minuend digits, 1 and 6, could have been taken together as 16. Then, by subtracting E from 16, the difference of 8 is obtained at once.

Three additional examples of hexadecimal subtraction are given below. Work out these examples, as well as the exercises in back, and verify the results by adding back.

Hexadecimal subtraction

F9D5	D935F	FDE74B.2C6A5
− EB63	− 8E7C2	− 7B 3AF4.95C09
E72	4AB9D	82AC56.96A9C

Binary Multiplication

The three rules of binary multiplication are:

$0 \times 0 = 0$
$0 \times 1 = 0$ (or $1 \times 0 = 0$)
$1 \times 1 = 1$

In practice, it is not necessary to remember these rules. Simply *copy the multiplicand* (the number on top) whenever the multiplier digit (on the bottom) is 1 and shift an extra place to the left (or copy 0's) for each multiplier digit that is a 0. The following examples illustrate the method:

Binary multiplication

```
1.                                      2.
   011  Multiplicand                       1100110
 × 011  Multiplier                       ×    1000
 -----                                  ----------
   011  1st partial product             1100110000  (copy 0's in
  011   2nd partial product                          multiplier)
 -----
  1001  Product

3.                 4.
   1.01                 110110  Multiplicand
 × 10.1               × 110011  Multiplier
 ------               --------
    101                 110110  Copy multiplicand
   1010                110110   Shift once and copy
 ------               110110    Shift 3 times and copy
  11.001             110110     Shift and copy
                  ------------
                  101011000010  Add partial products
```

Hexadecimal Multiplication

The rules of multiplication in the hexadecimal system are the same as those in the decimal and binary systems. However, since the process is fairly complicated, it will be necessary to refer to Table 3 to determine the product of multiplying two hexadecimal digits. In the decimal and binary systems it is customary to display the partial products of the multiplier digits with the multiplicand on a single line, each. Because of the carries, this is not usually convenient in hexadecimal multiplication; each product of two digits is separately written down and added, allowing for necessary shifts.

The following two examples illustrate the process of hexadecimal multiplication, using Table 3:

Hexadecimal multiplication

Example 1:

```
           9D7
        ×   5A
        ------
            46 ←A × 7 ⎫  Partial products of
           82 ←—A × D ⎬  first multiplier digit
         5A ←———A × 9 ⎭
        ------
          23 ←——5 × 7 ⎫  Partial products of
         41 ←———5 × D ⎬  second multiplier digit
        2D ←————5 × 9 ⎭
Carries 11
        ------
        37596
        ↑↑
        │└————————— 15 hexadecimal, carry 1
        └—————————— 17 hexadecimal, carry 1
```

Note that each of the partial products of a multiplier digit is shifted one place to the left with respect to the previous product. Care must be taken, however, to shift the *first* partial product of the *second* multiplier digit (23, above) only one place with respect to the first partial product of the first multiplier digit (46, above), as in decimal multiplication. In adding up the partial products, use is made again of Table 2. Any resulting carries are applied to the next-higher-order digit position, as is indicated in the example.

1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
3	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
4	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Table 3. Hexadecimal multiplication

Example 2:

```
             ABCD
           × ABCD
           ------
               A9        D × D     Partial
              9C         D × C     products
             8F          D × B
            82           D × A
           ------
              9C         C × D
             90          C × C
            84           C × B
           78            C × A
           ------
             8F          B × D
            84           B × C
           79            B × B
          6E             B × A
          ------
            82           A × D
           78            A × C
          6E             A × B
         64              A × A
Carries: 133232
         ↑↑↑↑↑↑
        734B8229
         ↑↑↑↑↑↑
13, carry 1      22, carry 2
34, carry 3      32, carry 3
3B, carry 3      28, carry 2
```

Answer: $(73,4B8,229)_{16}$

The carries resulting from the addition of the partial products are indicated above. The computation can be verified by converting the operands to decimal notation, as is later shown, carrying the arithmetic through in decimals, and converting the result back to hexadecimal notation.

Number Base Conversion

It may be necessary to ascertain the equivalent of a number in a base different from the one in which it is expressed. It may, for example, be occasionally desired to enter decimals into storage without using the conversion instruction of the machine.

A great number of conversion methods exist, of which only a few are useful in practice. As has been shown earlier, the literal method of expanding the number in powers of its base readily yields the *decimal* equivalent of the number, provided all arithmetic is carried out in the decimal system. The method also has the advantage that it works equally well for whole numbers, fractions, and mixed numbers; other methods generally require separate treatment for whole numbers (integers) and fractions. For conversions into bases other than ten, however, the method of literal expansion into a power series (sometimes called *transliteration*) frequently becomes difficult to carry out.

Another direct conversion technique is the method of *subtraction*, or *casting out.* To convert from base A to base B, first find the largest multiple of the highest power of B contained in the base A number (the number to be converted). This multiple represents the most significant digit (MSD) of the new base B number. Subtract this highest-power multiple from the original number and determine the largest multiple of the next-highest power of B contained in the remainder. This forms the next-most significant digit of the new number. Subtract again and continue the process until every power of base B has been exhausted. The multiple of the lowest power of B contained in the number is the least significant digit (LSD) of the new number.

The process is relatively simple for conversions to the binary system, since the multiples of the powers of 2 are either 1 or 0. For example, to convert decimal 25 into the binary system, proceed as follows:

	Decimal	*Binary Digits*
	25	
Highest power of 2 in 25:	$\underline{16} = \underline{\mathit{1}} \times 2^4$	1 (MSD)
Remainder:	9	
Next-highest power of 2:	$\underline{-8} = \underline{\mathit{1}} \times 2^3$	1
Remainder:	1	
Next-highest power of 2:	$\underline{0} = \underline{\mathit{0}} \times 2^2$	0
Remainder:	1	
Next-highest power of 2:	$\underline{0} = \underline{\mathit{0}} \times 2^1$	0
Remainder:	1	
Lowest power of 2:	$\underline{-1} = \underline{\mathit{1}} \times 2^0$	1 (LSD)
Remainder:	0	

Hence, $(25)_{10} = (11001)_2$

The method of subtraction becomes tedious for larger numbers and requires memorization (or a table) of the powers of the base into which the number is to be converted. A more rapid and convenient technique is the *division/multiplication method.* Here division is used for conversion of integers from one base to another, while multiplication is used for conversion of fractions, as will be described in the following paragraphs. For the conversion of *integers,* the method of division may be summarized as follows:

> To convert from base A to base B, divide repeatedly by the base A equivalent of B, until the quotient comes out zero. (Use base A arithmetic.) The remainder of the first division is the rightmost or least significant digit. The remainder of the last division is the leftmost or most significant digit. If B is greater than A, convert the remainder digits into base B.

Most of the conversion examples that follow will illustrate this method, which is the most useful in practice.

Decimal to Binary Integer Conversion

Conversion of decimal integers into binary notation by the division method can be done mentally, since it involves division by 2. Proceed as follows:

> Divide the decimal number repeatedly by 2, until a quotient of 0 is obtained. The equivalent binary number is composed of the remainders, the first remainder being the rightmost or least significant digit, while the last remainder is the leftmost, or most significant digit.

Since the divisions can be performed mentally, the quotients may be placed directly beneath the dividend and the remainders opposite the quotients, as illustrated in the examples below:

Decimal to binary integer conversion

Quotients	*Remainders*	*Quotients*	*Remainders*
$(27)_{10}$		$(568)_{10}$	
13	1 (LSD)	284	0 (LSD)
6	1	142	0
3	0	71	0
1	1	35	1
0	1 (MSD)	17	1
$(27)_{10}$ =	$(11011)_2$	8	1
		4	0
		2	0
		1	0
		0	1 (MSD)
		$(568)_{10}$ =	$(1000111000)_2$

Note, in each case, that the divisions are carried out through a zero quotient. The remainders are then written horizontally, left to right, beginning with the *bottom* of the column.

Decimal to Hexadecimal Integer Conversion

The division method is used, as follows:

> Divide the decimal number repeatedly by 16, until a zero quotient is obtained. Convert decimal remainders 10–15 into hexadecimal symbols A–F. The first remainder is the least significant hexadecimal digit; the last remainder is the most significant digit.

Here the divisions usually have to be done longhand in all but the simplest cases. Record the quotient and remainder of each division, as shown in the examples below:

Example 1: Convert decimal 195 into hexadecimals.

Divide by		*Base 16*	=	*Quotient*	+	*Remainder*	(=	*Hex Digits)*
195	÷	16	=	12	+	3		3
12	÷	16	=	0	+	12	=	C

$(195)_{10} = (C3)_{16}$

Example 2: Convert decimal 1710 into hexadecimals.

Divide by		*Base 16*	=	*Quotient*	+	*Remainder*		*(Hex Digits)*
1710	÷	16	=	106	+	14	=	E
106	÷	16	=	6	+	10	=	A
6	÷	16	=	0	+	6		6

$(1710)_{10} = (6AE)_{16}$

The hexadecimal equivalents of decimal numbers in the range of 0–4095 (hexadecimal 0–FFF) may also be looked up directly in the hexadecimal-decimal conversion table in Appendix E of the manual *IBM System/360 Principles of Operation* (A22-6821).

Binary to Decimal Integer Conversion

The decimal equivalent of a binary number is easily obtained by the method of direct expansion in powers of 2, as was described earlier. This works well for all binary numbers – integers, fractions, and mixed numbers – and can frequently be done mentally by inspection. Simply write down the powers of 2 of the binary number in column format (starting with the high-order digit) and add up the column to obtain the equivalent decimal.

A shortcut method of binary-to-decimal conversion is known as the *double-dabble* method (*dabble* means double and add). The method consists of the following procedure:

> Double the highest-order (leftmost) binary digit and add it to the digit at its right. Double the sum and add 1 or 0, depending upon whether the next digit to the right is a 1 or a 0. Repeat until the sum contains the lowest-order digit at right.

For example, to convert 1011 into a decimal, start with the 1 bit at left, double (making it 2) since the next bit is 0, then *dabble* since the next bit is a 1 (that is, 2 × 2 = 4; adding 1 makes 5), and finally, dabble again since the last bit at right is a 1, making it a total of decimal 11 (that is, 2 × 5 = 10; adding 1 makes 11).

The double-dabble technique can be done very quickly with a little practice, by simply remembering to double each time if the next digit is 0, and to double and add 1 if the next digit is 1. Additional exercises are presented below and in the back.

Example 1: Convert $(110101)_2$ into decimals.

Binary digits:	1	1	0	1	0	1
Decimal sums:		2 + 1 = 3	6	13	26	53
		dabble	double	dabble	double	dabble

Therefore, $(110101)_2 = (53)_{10}$

Example 2: Convert $(1110011)_2$ into decimals.

		Dabble	Dabble	Double	Double	Dabble	Dabble
Binary:	1	1	1	0	0	1	1
Decimal sums:		3	7	14	28	57	115

Hence, $(1110011)_2 = (115)_{10}$

In most cases the procedure can be done mentally without bothering to write down the partial sums each time.

Hexadecimal to Decimal Integer Conversion

Although the division method can be used for the conversion of hexadecimal integers into decimals, the ~~two methods pre~~sented below will be found to give results more rapidly. The direct method consists of expansion of the hexadecimal numbers in powers of 16, using decimal arithmetic for the calculations. This method can be formulated by the following rule:

> Multiply the decimal equivalent of each hexadecimal digit by the place value of the digit, expressed in decimals (that is, by the appropriate power of 16). Add all such products to obtain the equivalent decimal.

Using the previous examples, but in reverse:

Example 1: Convert $(C3)_{16}$ into decimals.

$$C3 = 12 \times 16^1 + 3 \times 16^0 = 192 + 3 = (195)_{10}$$

Example 2: Convert $(6AE)_{16}$ into decimals.

$$\begin{aligned} 6AE &= 6 \times 16^2 + 10 \times 16^1 + 14 \times 16^0 \\ &= 6 \times 256 + 10 \times 16 + 14 \times 1 \\ &= 1536 + 160 + 14 \quad = 1710 \\ &= (1710)_{10} \end{aligned}$$

The method requires memorization, or a table, of powers of 16 and becomes unwieldy for larger-size hexadecimal numbers. Another technique, which consists of a combination of multiplication and addition, is frequently easier than direct expansion. It may be stated as follows:

> Multiply the decimal equivalent of the high-order (leftmost) hexadecimal digit by 16. Add in the decimal equivalent of the next lower-order hexadecimal digit (at right) and multiply the sum again by 16. Continue this process until the last (rightmost) hexadecimal digit is added to the product. The last sum is the decimal sought for. Do not multiply it by 16!

Using again the previous example, to convert $(6AE)_{16}$ into decimals:

$$\begin{array}{lrl}
 & 6 \;\; A \;\; E & \text{hexadecimal} \\
\text{Multiply} & \times 16 & \\
\cline{2-2}
 & 96 & \\
\text{Add in A} & +10 & \\
\cline{2-2}
 & 106 & \\
\text{Multiply} & \times 16 & \\
\cline{2-2}
 & 636 & \\
 & 106 & \\
\cline{2-2}
 & 1696 & \\
\text{Add in E} & +14 & \\
\cline{2-2}
\text{Sum} & 1710 & \text{decimal. Hence, } (6AE)_{16} = (1710)_{10}
\end{array}$$

The conversion of large hexadecimal numbers by either direct expansion or the multiplication-addition method described above becomes quite tedious and difficult. The use of conversion tables makes possible rapid conversion of hexadecimal integers into equivalent decimals, so that necessary arithmetic can be performed in the decimal system. The hexadecimal-decimal conversion table in Appendix E of *IBM System/360 Principles of Operation* permits direct conversion of three-position hexadecimal integers into decimals, and vice versa. This includes the range of hexadecimal numbers 000 to FFF, which is equivalent to decimal 0000 to 4095. Hexadecimal integers of four to eight positions can be converted by using the extended hexadecimal-decimal integer conversion table (Table 4) in conjunction with the three-position table in the Principles of Operation Manual. (Alternatively, the three low-order digits of a hexadecimal number may be converted by using one of the methods described earlier.) Table 4 consists essentially of powers of 16 (16^3 through 16^7), multiplied by the range of hexadecimal digits from 1 through F. Thus, all the arithmetic necessary for expansion of a hexadecimal integer in powers of 16 is already performed in the table.

The following two examples illustrate the use of Table 4, as well as a comparison with the multiplication-addition method of conversion described earlier.

Example 1: Convert $(FA9C4D)_{16}$ into decimals, using tables.

	X000	X0000	X00000	X000000	X0000000
1	4,096	65,536	1,048,576	16,777,216	268,435,456
2	8,192	131,072	2,097,152	33,554,432	536,870,912
3	12,288	196,608	3,145,728	50,331,648	805,306,368
4	16,384	262,144	4,194,304	67,108,864	1,073,741,824
5	20,480	327,680	5,242,880	83,886,080	1,342,177,280
6	24,576	393,216	6,291,456	100,663,296	1,610,612,736
7	28,672	458,752	7,340,032	117,440,512	1,879,048,192
8	32,768	524,288	8,388,608	134,217,728	2,147,483,648
9	36,864	589,824	9,437,184	150,994,944	2,415,919,104
A	40,960	655,360	10,485,760	167,772,160	2,684,354,560
B	45,056	720,896	11,534,336	184,549,376	2,952,790,016
C	49,152	786,432	12,582,912	201,326,592	3,221,225,472
D	53,248	851,968	13,631,488	218,103,808	3,489,660,928
E	57,344	917,504	14,680,064	234,881,024	3,758,096,384
F	61,440	983,040	15,728,640	251,658,240	4,026,531,840

Table 4. Hexadecimal-decimal integer conversion

$$\begin{array}{rrcr}
\textit{From Table 4:} & \text{F00000} & = & 15{,}728{,}640 \\
 & \text{A0000} & = & 655{,}360 \\
 & 9000 & = & 36{,}864 \\
\textit{From 3-position table (or below):} & \text{C4D} & = & 3{,}149 \\
\cline{2-4}
\text{Sum:} & (FA9C4D)_{16} & = & 16{,}424{,}013 \\
 & & & \text{decimal}
\end{array}$$

$$(C4D)_{16} = 12 \times 16^2 + 4 \times 16 + 13 \times 1 = 3072 + 64 + 13 = 3{,}149$$

Example 2: Check the hexadecimal multiplication ABCD × ABCD = 734B8229 by converting to decimal arithmetic. Use both tables and longhand methods.

Solution:

$$\begin{aligned}
(ABCD)_{16} &= 10 \times 16^3 + 11 \times 16^2 + 12 \times 16^1 + 13 \times 16^0 \\
&= 10 \times 4096 + 11 \times 256 + 12 \times 16 + 13 \times 1 \\
&= \quad 40{,}960 \quad + \quad 2{,}816 \quad + \quad 192 \quad + 13 \\
&= (43{,}981)_{10}
\end{aligned}$$

Carrying out the multiplication in decimals:

$$\begin{array}{rl}
43981 & \text{decimal} \\
\times\ 43981 & \text{decimal} \\
\hline
43981 & \\
351848 & \\
395829 & \\
131943 & \\
175924 & \\
\hline
1934328361 & \quad \text{Answer: } (1{,}934{,}328{,}361)_{10}
\end{array}$$

To check the hexadecimal multiplication, convert $(73{,}4B8{,}229)_{16}$ into decimals:

1. *By use of conversion tables* (Table 4 or the table in the Principles of Operation Manual):

$$\begin{array}{rrcr}
\text{From Table 4:} & 70000000 & = & 1{,}879{,}048{,}192 \\
 & 3000000 & = & 50{,}331{,}648 \\
 & 400000 & = & 4{,}194{,}304 \\
 & \text{B0000} & = & 720{,}896 \\
 & 8000 & = & 32{,}768 \\
\text{From 3-position table:} & 229 & = & 553 \\
\cline{2-4}
\text{Sum:} & (734B8229)_{16} & = & (1{,}934{,}328{,}361)_{10}
\end{array}$$

This checks the answer obtained above by decimal multiplication.

2. *By use of multiplication-addition method:*

```
Hexadecimal          7 3 4 B 8 2 2 9
Multiply          × 16
                   112
Add in 3          + 3←
                   115
Multiply          × 16
                   690
                  115
                  1840
Add in 4          + 4←
                  1844
Multiply          × 16
                 11064
                 1844
                 29504
Add in B          + 11←
                 29515
Multiply          × 16
                177090
                29515
                472240
Add in 8          +   8←
                472248
Multiply          ×  16
               2833488
               472248
               7555968
Add in 2          +    2←
               7555970
Multiply          ×   16
              45335820
              7555970
             120895520
Add in 2          +     2←
             120895522
Multiply          ×    16
             725373132
            120895522
            1934328352
Add in 9          +      9←
```

Final Sum: 1934328361 Answer: $(1{,}934{,}328{,}361)_{10}$

The conversion again checks the previous results. Note, however, the excessive length of this method compared with direct use of the tables for conversion.

Conversion of Fractions

In general, the fractional part of a number must be converted separately from its integer part, since the process of conversion is different for each. Frequently, the conversion of a fraction is the *inverse* of integer conversion. Where the integer part is converted by a process of repeated division, the fractional part is converted by repeated multiplication; where the integer is converted by multiplication, the fraction is converted by division.

Conversion of Decimal Fractions to Binary

Decimal fractions are converted into the binary system by successively *multiplying the fraction by 2.* The integer parts formed during multiplication are the successive binary digits, the first integer being the most significant digit of the binary fraction. Ignore the integer parts during each multiplication and continue multiplying by 2 until the fraction either has been reduced to zero or a sufficient number of binary digits have been generated, in the event of a nonterminating fraction. The scheme of the example below may be used.

Example: Convert the decimal fraction 0.828125 into binary.

Decimal Fraction	×	2	=	*Product*	*Integer Part*	*(Binary Digit)*
0.828125	×	2	=	1.65625	1	(MSD)
0.65625	×	2	=	1.3125	1	
0.3125	×	2	=	0.625	0	
0.625	×	2	=	1.25	1	
0.25	×	2	=	0.50	0	
0.5	×	2	=	1.00	1	(LSD)

Collecting digits from top to bottom and placing them to the right of the binary point, the answer is $(0.110101)_2$.

Conversion of Decimal Fractions to Hexadecimal

The same procedure is used as for conversion to binary, except that the fraction is repeatedly multiplied by 16. The integer part of the first product is the high-order or most significant digit of the hexadecimal fraction. Convert decimal integers between 10 and 15 into corresponding hexadecimal digits (A through F). Continue multiplication by 16 until either the fraction is removed or a sufficient number of hexadecimal digits have been generated.

Example: Convert $(0.828125)_{10}$ into a hexadecimal fraction.

Decimal Fraction	× 16 =	*Product*	*Integer Part*	= *Hex Digit*
0.828125	× 16 =	13.25	13	= D (MSD)
0.25	× 16 =	4.00	4	= 4 (LSD)

Collecting digits and placing to right of hexadecimal point: 0.D4

Therefore $(0.828125)_{10} = (0.D4)_{16}$

Conversion of Binary Fractions to Decimals

The method of repeated multiplication can be used for converting binary fractions into equivalent decimals. The technique is somewhat clumsy, however, since all arithmetic must be done in binary notation; that is, the fraction must be multiplied by the binary equivalent of decimal 10 = $(1010)_2$ and the binary digits developed must then be converted back into decimals. In most cases, a literal expansion of the binary fraction in *ascending negative powers of* 2 will give the decimal equivalent more quickly and less laboriously. A table of the decimal equivalents of negative powers of 2 will prove handy for conversion. (See Appendix D in the above-mentioned manual *IBM System/360 Principles of Operation.*)

Example: Convert $(0.110101)_2$ into a decimal fraction.

$$\begin{aligned} 0.110101 &= 1 \times 2^{-1} + 1 \times 2^{-2} + 0 + 1 \times 2^{-4} + 0 + 1 \times 2^{-6} \\ &= 1/2 + 1/4 + 0 + 1/16 + 0 + 1/64 = 53/64 \\ &= 0.5 + 0.25 + 0 + 0.0625 + 0 + 0.015625 \\ &= (0.828125)_{10} \quad \text{Answer} \end{aligned}$$

Conversion of Hexadecimal Fractions to Decimals

The conversion of a hexadecimal fraction to decimals is the inverse of hexadecimal-to-decimal integer conversion and, accordingly, requires repeated *division* by 16. The rule is:

Divide the decimal equivalent of the low-order (rightmost) hexadecimal digit by 16. Add the quotient to the next-higher-order hexadecimal digit (at left) and again divide by 16. Repeat the process until the high-order (leftmost) digit of the hexadecimal fraction has been used. The last quotient is the answer.

Example: Convert $(0.D4)_{16}$ to a decimal fraction. First divide the low-order digit, 4, by 16:

$$4.00 \div 16 = 0.25$$

Add in the decimal equivalent of the high-order digit, D (=13):.

$$0.25 + 13 = 13.25$$

Divide 13.25 again by 16, as shown below:

```
                                  0.828125
                             16 / 13.250000
                                  12 8
                                  ----
                                    45
                                    32
                                    --
                                    130
                                    128
                                    ---
                                      20
                                      16
                                      --
                                       40
                                       32
                                       --
                                       80
Answer: (0.828125)10                   80
                                       --
                                       00
```

Since the conversion of hexadecimal fractions proves frequently lengthy and tedious, as the example illustrates, Table 5 has been included for the direct conversion of up to three-place hexadecimal fractions into equivalent decimal fractions. The example below illustrates the use of Table 5: conversion table.

Example: Convert the hexadecimal fraction 0.ABC into an equivalent decimal fraction, using the table.

From Table 5:	0.A	= 0.625000000000
	0.0B	= 0.042968750000
	0.00C	= 0.002929687500
Sum:	0.ABC	= 0.670898437500

Hence, $(0.ABC)_{16} = (0.6708984375)_{10}$

	0.X	0.0X	0.00X
1	0.0625	0.00390625	0.000244140625
2	0.1250	0.00781250	0.000488281250
3	0.1875	0.01171875	0.000732421875
4	0.2500	0.01562500	0.000976562500
5	0.3125	0.01953125	0.001220703125
6	0.3750	0.02343750	0.001464843750
7	0.4375	0.02734375	0.001708984375
8	0.5000	0.03125000	0.001953125000
9	0.5625	0.03515625	0.002197265625
A	0.6250	0.03906250	0.002441406250
B	0.6875	0.04296875	0.002785546875
C	0.7500	0.04687500	0.002929687500
D	0.8125	0.05078125	0.003273828125
E	0.8750	0.05468750	0.003517968750
F	0.9375	0.05859375	0.003662109375

Table 5. Hexadecimal-decimal fraction conversion

It has been mentioned earlier that subtraction is carried out by the addition of complements. The present section is concerned with the use of complements to represent negative numbers and for the addition and subtraction of *signed* (positive or negative) numbers. The meaning and use of complements must be thoroughly understood to avoid mishandling of arithmetic instructions. The definitions and examples which follow should therefore be studied carefully.

Subtraction with Ten's Complement

The meaning of the phrase "subtraction by addition of the complement" is made clear by an illustration from decimal arithmetic. Suppose the six-digit decimal number 235,481 is to be subtracted from 584,673. This may be done conventionally:

$$584{,}673 - 235{,}481 = 349{,}192$$

Alternatively, the subtraction may be done by the addition of the ten's complement as follows:

$$584{,}673 + \underbrace{(1{,}000{,}000 - 235{,}481)}_{\text{ten's complement}} - 1{,}000{,}000$$

The term in parentheses (1,000,000 − 235,481) is called the *ten's complement* of 235,481. It turns out to be

$$\begin{array}{r} 1{,}000{,}000 \\ -\ \ 235{,}481 \\ \hline 764{,}519 \end{array} \quad \text{(ten's complement of 235,481)}$$

Note that the ten's complement can be written down by inspection, by subtracting each digit of the number from 9, and then adding 1 to the low-order (least significant) digit at right. (Alternatively, the ten's complement of a number may be formed by subtracting each digit from 9, except the rightmost, which is subtracted from 10.)

Now adding the ten's complement to the minuend,

$$\begin{array}{r} 584{,}673 \\ +\ 764{,}519 \\ \hline 1{,}349{,}192 \end{array}$$

Finally, to subtract 1,000,000 (indicated above), it is necessary only to drop the high-order 1 at left. Thus the final result is 349,192, which checks with the result of conventional subtraction. Note that in a computer with fixed-length arithmetic registers – say, six digits each – the high-order 1 (appearing in the sum of the minuend and the complement) would have been *dropped automatically,* since it could not have been contained in the register. Thus, with *fixed-length arithmetic* (as is the case in a computer), subtraction can always be done by adding the complement of the subtrahend to the minuend, and simply ignoring the high-order carry (off-register).

Definition of Radix Complement

It has been shown that the ten's complement of a number is obtained by subtracting it from 1000000 . . . (as many zeros as there are digits in the number). More precisely, the ten's complement of a number (N) is obtained by subtracting it from the base, or radix, of 10 *raised to a power equal to the number (n) of digit positions:*

Ten's complement of $N = 10^n - N$,
where n = number of digit positions in N

Note that 10^n is *one more* than the largest fixed-length decimal number that can be formed with *n* digit positions; that is, the largest possible decimal of *n* digits is

$$10^n - 1$$

More generally, the *radix* or *base (b) complement* of a fixed-length number (N) is obtained by subtracting the number from the base raised to a power equal to the number (*n*) of digit positions. Expressed in mathematical form:

Radix (base) complement of $N = b^n - N$
where *b* is the base, or radix, of the number system and *n* is the number of digit positions in N.

Again, the largest fixed-length number of radix *b* that can be formed with *n* digit positions is equal to $b^n - 1$. Thus, when the original number is *added back* to the complement, the resulting number

$$[(b^n - N) + N] \text{ is} = b^n = 10000\ldots$$

This is 1 more than the largest possible number of length *n*. For this reason the high-order carry of 1 can be ignored in fixed-length arithmetic. Since the extra 1 may be ignored, an *alternative definition for the radix complement of a fixed-length number* is as follows:

> The radix complement of any number in a fixed-length arithmetic system is that number which when added to the original number produces all zeros.

Signed Numbers

Fixed-point numbers carry a *sign bit* (the first bit position), which indicates whether the number is positive or negative. Positive numbers are represented in true (binary) form with a zero sign bit, while negative numbers are represented in complement form with a one bit in the sign position. The handling of signed numbers will be described later under "Two's Complement Notation". Since the rules for handling signed numbers are the same, however, in any numerical system, they will be briefly reviewed at this time. Two rules should be recalled:

1. To add numbers of *like* sign (both positive or both negative), find the sum of the numbers and give it the common sign. In other words, the sum of two positive numbers is positive, and the sum of two negative numbers is negative.
2. To add numbers of *unlike* sign, find the difference between the numbers and give it the sign of the larger number.

It is of interest to note that in a computer using complement notation and sign bits, the signs pretty much take care of themselves. The only case that causes concern is the addition of a large negative number to a smaller positive number, or equivalently, the subtraction of a larger number from a smaller number. How does the computer know that the result is negative in this case? An example with decimal arithmetic shows how complement notation takes care of this. Assume that 584,673 is to be subtracted from 235,481. Using the conventional method and rule 2, above, it is found that

$$\begin{array}{r} +\,235{,}481 \\ -\,584{,}673 \\ \hline -\,349{,}192 \end{array}$$

Using the method of adding the complement, first find the ten's complement of 584,673, which is 415,327 (by inspection). Now adding the complement,

$$\begin{array}{rl} 235{,}481 & \\ +\,415{,}327 & \text{(ten's complement of 584,673)} \\ \hline 650{,}808 & \text{(complement of correct result)} \end{array}$$

This is, of course, the wrong result. Note, however, that no high-order carry (off-register) has been generated this time, as was the case in the earlier example of 584,673 – 235,481. This provides the clue to the computer: whenever there is no carry off-register after adding the complement, the *result is negative and should be recomplemented.* (Alternatively, whenever there is a carry of 1 off-register, the result is *positive and recomplementing is unnecessary.*) Thus, recomplementing the earlier result,

$$1{,}000{,}000 - 650{,}808 = 349{,}192$$

which is the correct answer after a negative sign has been affixed (that is, – 349,192).

Finally, the steps required in subtraction of decimals with the ten's complement may be summarized as follows:

> Compute the ten's complement of the subtrahend and add it to the minuend. If there is a carry of 1 (off-register), the result is positive and recomplementing is not necessary. If there is no carry, the result is negative and must be recomplemented.

With two's complement notation, the sign bits generally indicate the sign of the result. Since negative numbers are always represented in complement form, recomplementation is not necessary, as will be shown later.

Two's Complement

The two's complement and one's complement in binary notation are analogous to the ten's complement and nine's complement of the decimal system. According to the definition of the radix complement, the two's complement of a binary number (N) with *n* digit positions is:

$$\text{Two's Complement of } N = 2^n - N$$

Thus, for an eight-bit binary number (N) the two's complement is

$$2^8 - N = 100000000 - N$$

The two's complement of the binary number 00111001, for example, is $2^8 - 00111001$, or

$$\begin{array}{r} 100000000 \\ -\;\;00111001 \\ \hline 11000111 \end{array}$$

Note that actual subtraction is not required, since the two's complement can be obtained by inspection of the number. Each bit of the number is simply *inverted* (that is, a 1 is changed to a 0, and a 0 is changed to a 1) and a 1 is then added to the low-order (least significant) bit at right. Thus, the

binary number	00111001	
is inverted	11000110	
1 is added	+ 1	
to obtain	11000111	= two's complement of 00111001,

which checks with the result obtained above.

Summing up, the *two's complement of a binary number is obtained by inverting each bit of the number and adding a 1 in the low-order (least significant) bit position.*
Examples of addition and subtraction of negative numbers in two's complement form are given in the next section.

Two's Complement Notation

The first bit position (0) of a fixed-point binary integer holds the *sign* of the binary number; the remaining bit positions designate the *magnitude* of the number. *Positive* numbers are represented in true binary notation with a *zero sign* bit. *Negative* numbers are represented in two's complement notation with a *one* bit in the sign position. The representation of negative numbers in complement notation makes recomplementation unnecessary.

In addition to making recomplementation unnecessary, two's complement notation facilitates extension of the operands for high-precision multiplications and divisions. In the type of number representation described above, the halfword or fullword operands may be considered the low-order portion of an infinitely long representation of the number. The bits between the sign position and the leftmost significant bit of the integer are always the same as the sign bit. When the number is positive, all bits to the left of the most significant bit, including the sign bit, are zeros. When the number is negative, all bits to the left of the MSD, including the sign bit, are ones. Therefore, to extend an operand with high-order bits (to a fullword or two words in length), a field of the proper length is prefixed, in which each bit equals the high-order (sign) bit of the operand. For example, the 16-bit positive number

S

0 1101011 00011010

may be extended to 32 bits by prefixing a field of 16 zeros to the high-order zero bit:

S

0 0000000 00000000 01101011 00011010

Similarly, the 16-bit negative number

S

1 1011001 10011010

is extended to 32 bits by prefixing 16 ones to the one sign bit:

S

1 1111111 11111111 11011001 10011010

Zero and Maximum Numbers

It is important to note that two's complement notation does not include the representation of a *negative zero.* A zero is always positive and includes a zero (positive) sign bit. Thus,

S

0 = 0 0000000 00000000

Because of the one sign bit in negative numbers, the range of negative numbers in two's complement notation is *one larger* than the total set of positive numbers. Since the sign bit is occupied by a zero, the largest positive 16-bit number consists of 15 one bits and a zero sign bit, which is equal to $2^{15} - 1$, or decimal 32,767 (since $2^{15} = 32{,}768$):

S

$2^{15} - 1 = 32{,}767 =$ 0 1111111 11111111 (max. + number)

The largest 32-bit positive number consists of a zero sign bit, followed by 31 one bits. This comes out to $2^{31} - 1$, or decimal 2,147,483,647 (since $2^{31} =$ 2,147,483,648).

In contrast, the largest negative number consists of an all-zero integer field with a sign bit of 1. Thus, the largest 16-bit negative number is

S

$-2^{15} = -32{,}768 =$ 1 0000000 00000000

and the largest 32-bit negative number is

$-2^{31} = -2{,}147{,}483{,}648 =$

1 0000000 00000000 00000000 00000000

The CPU of the System/360 cannot represent the *complement* of the largest negative number. When an operation, such as subtraction from zero, produces the complement of the largest negative number, the number remains unchanged, and a fixed-point overflow exception is recognized. (This causes a program interruption when the fixed-point overflow mask bit is 1.) When the final result of the complemented number is within the representable range, however, an overflow does not occur. An example would be a subtraction from − 1. The product of the two largest negative numbers also does not cause an overflow, since it is representable as a *double-length positive number.* Additional examples of overflow conditions are described in the next section.

Overflow

When the result of an add, subtract, or shift operation exceeds the capacity of the register or field containing the result, an overflow condition results. Since an overflow carries into the leftmost, or sign-bit, position, it changes the sign. (However, in shift operations the sign of the shifted number remains *unchanged,* even if significant high-order bits are shifted out.) Thus, in a *positive* overflow, the final sum or difference comes out *negative,* while a *negative* overflow results in a *positive* sum.

The CPU recognizes an overflow condition by comparing the carries out of the sign-bit (leftmost) position with the high-order bit position (MSD) of the binary number. (No carry out is equivalent to a carry of zero.) If the carries out and the high-order bit are the same, the result is satisfactory and no overflow occurs; if they are different, an overflow is recognized and the sign bit changes. (This is not corrected after the overflow.) Moreover, an overflow causes a program interruption when the fixed-point overflow mask bit is 1.

The presence or absence of an overflow condition may be recognized by the condition of the carries. (As an alternative, the indicated operation may be performed in decimals to check whether the result exceeds the capacity of the register.) The result of an operation does *not* overflow if there is either *no carry into* the high-order bit position and *no carry out,* or a *carry of 1 into* the high-order position and also a *carry out.* In contrast, an operation overflows if there is either a *carry into* the high-order position and *no carry out,* or a *carry out* but *no carry into* the high-order bit position. The following examples, which have only eight bit positions for convenience, illustrate the four possible cases. Decimal equivalents are given for comparison purposes. Note that the results of the two operations which result in overflow exceed decimal 127 (that is, $2^7 - 1$), which is the maximum number that can be contained in an eight-bit register in two's complement notation.

Examples of Two's Complement Notation

```
1. + 62 =   00111110
   + 27 = + 00011011          No overflow
   ____   __________
   + 89 =   01011001
```

In this straightforward addition, there is neither a carry into the high-order bit position nor a carry out. Since the carries agree (both are zero), there is no overflow.

```
2. + 62 =     00111110
   - 27 = +   11100101  (two's complement of 00011011 = 27)
              1 ← Carry in
   ____     __________
   + 35 = ←①  00100011  No overflow
         Carry out
```

In this example, there is both a carry into the high-order position and a carry out, which is ignored. There is no overflow.

```
3. + 27 =     00011011  No overflow
   - 62 = +   11000010  (two's complement of 00111110 = 62)
   ____       ________
   - 35 =     11011101  (two's complement of 00100011 = 35)
```

In this subtraction of a larger number from a smaller number, there is neither a carry in nor a carry out, but the *sign* changes to 1, indicating that the result (—35) is in complement form. There is no overflow.

```
4. - 62 =     11000010  (two's complement of 00111110 = 62)
   - 27 = +   11100101  (two's complement of 00011011 = 27)
              1
   ____     __________
   - 89 = ←①  10100111  (two's complement of 89 = 01011001)
```

Here there is both a carry into the high-order position and a carry out (which is ignored). There is no overflow, but the sign bit of 1 indicates that the result is in two's complement form and, hence, negative.

```
5. + 62 =    00111110
   + 89 = +  01011001       Overflow!
             1
   _____   __________
   + 151 =   10010111
```

In this addition example, there is a carry into the high-order bit position, but no carry out, indicating an overflow condition. (This is also evident from the decimal result, 151, which exceeds 127, the maximum number that can be contained in an eight-bit register.) Note also that the sign bit has changed to 1, indicating a negative result for positive overflow.

```
6. - 62 =     11000010  (two's complement of 00111110 = 62)
   - 89 = +   10100111  (two's complement of 01011001 = 89)
   ____     __________
                          Overflow!
   - 151 = ←① 01101001  (two's complement of 10010111 = 151)
```

Here no carry into the high-order bit position is developed, but there is a carry out, indicating an overflow. This is also evident from the decimal result, since —151 exceeds the capacity (—127) of an eight-bit register. Note further that the sign bit has changed to zero, indicating a positive result for a negative overflow condition.

Questions and Exercises

1. Express the following numbers in successive powers of the radix and evaluate their decimal equivalents (where applicable): $(547)_{10}$; $(3{,}289.6375)_{10}$; $(121{,}001)_{3}$; $(3213)_{5}$; $(110{,}110{,}001)_{2}$; $(100011)_{2}$; and $(0.111111)_{2}$.

2. What is the meaning of the hexadecimal number 9B4D.3A7 in positional notation? What is its decimal equivalent?

3. Convert the following binary numbers into hexadecimal notation:

10001010; 1001110100; 1110101.001110101.

4. Convert the following hexadecimal numbers into binary notation:

A72B; 39BF4D; ABCDEF; 52.A7EF98; 123.ABC.

5. Add the binary numbers 1100011 and 0111001; 111101111 and 111101111; 1000 1111 1010 0001 and 0001 0011 1110 0101.

6. Add the following hexadecimal numbers, using Table 2 when necessary: 12345 and 56789; 8FB5 and CD69; 345.789 and 832.BDE; ABCD.09EF and 1234.5698.

7. Give the rules of binary subtraction and subtract the following binary numbers, using either the conventional or payback method, as desired: 100000 – 1; 111010 – 100100; 111111111 – 100000000; 10001.11001 – 1101.00110.

8. Perform the following hexadecimal subtractions, using Table 2:

F865 – 9AB7; E73F.A983 – A9CD.87FE.

9. Perform the following binary multiplications:

1100 × 11; 1010 × 1001; 10.001 × 1.01.

10. Perform the following hexadecimal multiplications, using Table 3:

3E7 ×5B9; D.38 × 6.EF.

11. Explain the subtraction (casting out) method of number conversion and convert to binary the following decimal numbers: 39; 583; 7946. Also, describe the division method and perform the conversions above by this method. Which method is more rapid?

12. Using the division method, convert to hexadecimal notation the following decimal numbers: 89; 438; 999; 5793; 875,472,925.

13. Describe the double-dabble method for converting binary integers into decimals and convert the folowing binaries: 110101; 1110110001; 111111111.

14. Explain the direct and the multiplication-addition method for converting hexadecimal integers into decimals and perform the following conversions (check the results by use of Table 4 of this manual and the table referred to earlier in *IBM System/360 Principles of Operation*): 7E5; F8D; 89F7.

15. Convert the following decimal fractions and mixed numbers first into binaries and then into hexadecimals (check the results by converting each of the fractions back into decimals): 0.79; 0.6666666 . . .; 0.123; 34.675.

Chapter 3: Introduction to Assembler Language Programming

This chapter is an expository introduction to System/360 assembler language programming. Later chapters in this text make considerable use of assembler language program samples, and thus require the background that this chapter provides.

Review and Terminology

Programming in an assembler language offers a number of important advantages over programming in the actual language of the computer:

1. Mnemonic operation codes are provided. For instance, the actual operation code for the instruction Store in hexadecimal is 50; in the assembler language we can write the mnemonic operation code ST. Most programmers never learn the actual codes.

2. Addresses of data and instructions can be written in symbolic form, and in practice almost all addresses are so written. The programmer is thereby relieved of severe problems in the effective allocation of storage, and the resulting program is far easier to modify. Furthermore, the use of symbolic addresses reduces the clerical aspects of programming and eliminates many programming errors. If the symbols are chosen to be meaningful, the program is also much easier to read and understand than if written with numerical addresses.

3. Data may be introduced into the program structure, and space reserved for results, by the use of suitable *assembler instructions.* These are written in somewhat the same form as machine instructions but are treated quite differently by the assembler.

4. Many other assembler instructions direct the assembler in various other matters of concern. Among the most important of these are the techniques for letting the assembler assign base registers and compute displacements.

The sum effect of these advantages is so great that it is virtually out of the question to program in actual machine language, that is, to write actual operation codes and numerical addresses, and, in the case of the System/360, to write actual base register numbers and displacements.

An assembler language program is not directly executable by the computer. The mnemonic operation codes and symbolic addresses must be translated into the form the machine expects of instructions. This is the function of the *processor program,* also called the *assembler.*

The assembly process begins with a *source program* written by the programmer. Ordinarily, a special coding form is used such as that shown in Figure 27. (We shall study this program in detail later.) Cards are punched from this form, making up the *source program deck.* This source program deck becomes the primary input to the assembly process, as shown in Figure 26.

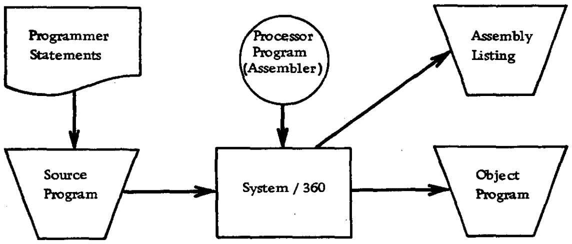


Figure 26. Schematic representation of the assembly process

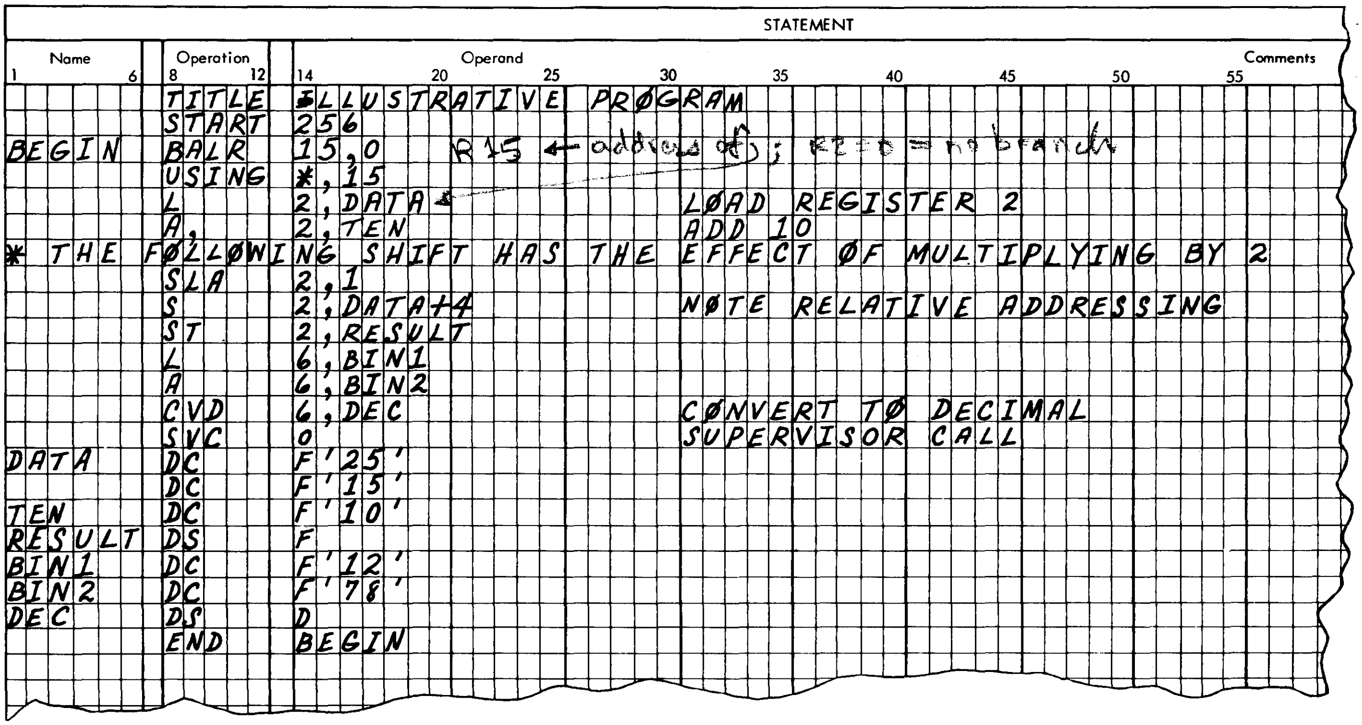
The assembly is done, in our case, by the System/360 under control of a *processor program.* The processor program is supplied by IBM; it consists of many thousands of machine instructions.

There are two outputs from the processor run. The first is an *object program* consisting of actual machine instructions corresponding to the source program statements written by the programmer. In many cases the object program is punched into cards; in other cases it is left on magnetic tape or magnetic disks. The second output is a *program listing* or *assembly listing.* This important document shows the original source program statements side by side with the object program instructions created from them. Many programmers work from the assembly listing as soon as it is available, hardly ever referring to their coding sheets again. An example appears as Figure 28, which we shall also consider in detail later.

In this chapter we shall consider the broad outlines of the assembler language for the System/360. The presentation will be largely in terms of examples that bring out most of the matters needed for writing simple programs.

There will be no attempt to cover every feature of the assembler language. Some topics will be treated in other chapters in this text; others are of such a specialized nature as to be best left to the assembler language reference manual.

It is assumed that the reader has access to one of the SRL publications on the assembler language.



```
                                  STATEMENT
  Name       Operation   Operand                                              Comments
1      6    8      12   14         20        25        30        35        40        45        50        55
        TITLE   ILLUSTRATIVE PRØGRAM
        START   256
BEGIN   BALR    15,0      R15 ← address of ; R0 = no branch
        USING   *,15
        L       2,DATA                 LØAD REGISTER 2
        A       2,TEN                  ADD 10
* THE FØLLØWING SHIFT HAS THE EFFECT ØF MULTIPLYING BY 2
        SLA     2,1
        S       2,DATA+4               NØTE RELATIVE ADDRESSING
        ST      2,RESULT
        L       6,BIN1
        A       6,BIN2
        CVD     6,DEC                  CØNVERT TØ DECIMAL
        SVC     0                      SUPERVISOR CALL
DATA    DC      F'25'
        DC      F'15'
TEN     DC      F'10'
RESULT  DS      F
BIN1    DC      F'12'
BIN2    DC      F'78'
DEC     DS      D
        END     BEGIN
```

Figure 27. A program to illustrate assembler language concepts. The "processing" performed by the program is not intended to be realistic, and it is not necessary to understand the functons of the various instructions for the purposes of this publication.

A First Example

Our first example of a program written in the assembler language for the System/360 (shown in Figure 27), and written on a coding form, does some elementary processing, the details of which do not concern us.

The coding form permits a maximum of four fields on each line. These are called name, operation, operand, and comments. The *name* provides a reference identification for the line, if other parts of the program need to refer to it. Names are not required for every line. They are most commonly given to data elements and to instructions to which branches are made. It is also permissible to use a name simply for identification on an instruction to which a branch is not made.

An *operation* is required on every line, with the exception of a comment line, which begins with an asterisk. The operation is most commonly a mnemonic operation code for a System/360 instruction, but it may also be any one of a number of *assembler instructions*. Examples of assembler instructions in this program are TITLE, START, USING, DC, DS, and END. We shall study each of these shortly.

For the purposes of this chapter, it is not necessary that the reader understand the functioning of the various instructions used in the sample programs. All instructions are explained in other chapters of this text.

The *operand* is most often a part of a System/360 instruction, including an address part.

The *comments* field may be used freely, at the programmer's discretion, to document the purpose and the methods of the program. A comment for a given line may begin anywhere following the operand, as long as there is at least one blank space between operand and comment. Many programmers prefer to begin all comments in some fixed column, such as column 30 or column 40, but this is discretionary.

The column assignments printed on this form are acceptable for our purposes and will be used throughout this text. We may note in passing, however, that considerable flexibility is permitted in the format of the source program. (The reader is referred to the SRL publication.)

Let us now turn to the program itself.

The first line contains an assembler instruction, TITLE. Whatever is written in the operand field, ILLUSTRATIVE PROGRAM in this case, will be printed at the top of every page of the assembly listing.

The START instruction is used to dictate the starting address of the assembled object program. The value used in this example is 100_{16}.

The next two lines are important ones, which should be written in every program. BALR is the mnemonic operation code for Branch and Link Register, which has the general format BALR R1,R2, where R1 and R2 are registers. The action of the instruction is to place in R1 the address of the instruction following the BALR, and to branch to the address contained in R2 unless R2 is zero, in which case there is no branch. Our instruction, BALR 15,0, will therefore place the address of the next machine instruction in register 15 and there will be no branch. Register 15 is to be the base register for this program.

With the assembler instruction USING we now inform the assembler that register 15 is the base register. This instruction also tells the assembler what will be in register 15: the asterisk refers to the address of the next instruction.

Let us review the effect of the BALR-USING combination. The BALR places in register 15 the address of the next machine instruction, which is the L 2, DATA, since USING is not a machine instruction and takes no space in the object program. The USING then informs the assembler that register 15 is available as a base register and what its contents will be.

As in so many other aspects of programming, it is important to understand when these actions occur. The BALR is an object program instruction and is not executed until the assembled object program is loaded. The assembler, of course, does not execute it. The USING is strictly an assembler instruction, in this case giving information to the assembler about what will be done later by the object program. Once the assembly is finished, the USING has no further function.

Now we come to the body of the program, starting with the L 2,DATA instruction. L is the mnemonic operation code for Load, which in this case places in register 2 the contents of the fullword having the symbolic address DATA. Looking down the page, we see that DATA is in the name field of a DC assembler instruction, which, as we shall study in a moment, stands for Define Constant. The A 2,TEN is a similar type of instruction, adding to register 2 the contents of a fullword having the symbolic address TEN.

The next instruction, SLA 2,1, is a little different. SLA stands for 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 Subtract instruction that comes next, S 2,DATA +4, exhibits *relative addressing*: the address is given "relative to" another address. The 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, introduces no new assembler ideas.

The following three instructions present no new assembler concepts either. The Load and Add are familiar, forming a sum in register 6. The Convert to Decimal (CVD) converts the contents of register 6, which are binary, to a decimal number in the location DEC. DEC is required by the machine design to be a doubleword, aligned on a doubleword boundary. We shall see how the alignment is handled in considering the data definition assembler instructions, the DC's and DS's.

The final System/360 instruction is the Supervisor Call, which returns control to the "supervisor" program that runs the System/360 between jobs. (There is no Stop instruction in the system.)

The DC assembler instruction, which stands for Define Constant, allows us to introduce data into the program structure. Specification of the type of data, and the amount of space needed for it, is the function of the *type designation*, in this case the F. F stands for Fullword, in binary format. The number written in quotes following the F will be assembled and entered into storage along with the assembled instructions.

Down at RESULT we have another assembler instruction, DS, which stands for Define Storage. This is used to allocate space without entering anything into storage. We use a type specification F to indicate the amount of storage and what boundary alignment should be performed, if any. That is, the assembled address of an F-type DC or DS must be a multiple of 4; the assembler will skip over a few bytes, if required, to reach such an address.

The DS at DEC will allocate an eight-byte space, aligned on a doubleword boundary; that is, the assigned address will be a multiple of 8.

The END assembler instruction specifies that nothing further follows; the assembly process may be completed. If an operand is written, as we have done, it directs the assembler to set up the object program so that when it is executed, the first instruction will be the one named. In our case, we have said that when the object program is carried out the BALR at BEGIN should be the first instruction.

Now we may turn to Figure 28, the assembly listing for this program, to see how things were handled.

We see that the source program we wrote has been reproduced without change on the right-hand side of the listing. The object program instructions created from the source program statements are shown at the left, with the storage location of each line appearing in the leftmost column in hexadecimal. Source program lines from which no object program entries were created are blank in the object program position; in this program, TITLE, START, USING, and END are in this category.

The TITLE line has indeed caused the words ILLUSTRATIVE PROGRAM to be printed at the top of the page. The START 256 instruction has caused the first word of the program to be placed in $100_{16} = 256_{10}$. We can read the assembled BALR instruction; 05 is the actual operation code, F is R1, and 0 is R2. The assembled object program instructions and constants are given exclusively in hexadecimal. For instance, base register 15 is shown as F, in hexadecimal.

BALR is a representative of the RR format instructions: a one-byte operation code, and a second byte containing two register numbers.

The USING has generated no object program entry; its work was finished when it informed the assembler that register 15 was available for use as a base register and that base register 15 would contain the address of the next machine instruction. We see that the address of the next machine instruction has been printed: 000102.

That next instruction is the first actual processing operation, the Load. Load is an example of an RX format instruction. Reading the assembled bytes from left to right, we have: 58 is the operation code, 2 is the register to be loaded, 0 means there is no index register, F is the base register, and 022 is the displacement. We remember that in an RX format instruction the effective address is formed from the sum of the base register contents, the index register contents, if any, and the displacement. In this case the base register contains 102, there is no index, and the displacement is 022; the sum of these is 124. Looking down to the assembled location of the symbol DATA, we see that it is 124, as it should be.

In the Add instruction that follows, the pattern is the same. The base register contents of 102, plus the displacement of 02A, gives a sum of 12C, which is the location of TEN.

The SLA instruction is an example of an RS format instruction; the major difference in format between an RX and an RS instruction is that the RS does not permit an index register. In the instruction at hand, there is as a matter of fact no reference to storage, but a base register may still be specified if we wish, in order to provide for a variable number of positions of shifting. In this case, however, such is not desired. The index register and base register positions are simply zero; the effective address (the shift amount) is just the displacement of 1.

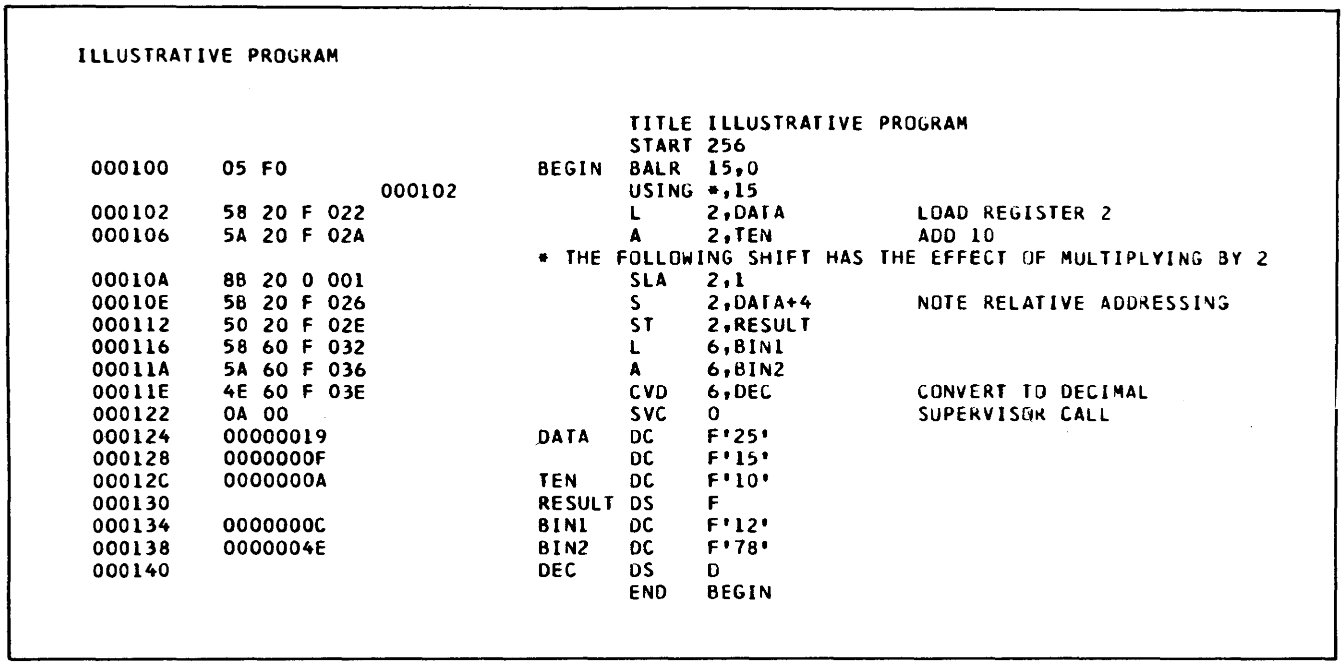
The next five instructions are further examples of RX formats, offering no new concepts. The reader may wish to check that displacements have been computed correctly, taking into account the relative addressing on the Subtract.

We see even in this simple example that the assembler has assumed a great deal of the clerical burden that would be required to program the System/360 in actual machine language. To the familiar advantage of an assembler language (mnemonic operation codes and symbolic addressing), which are important enough, we have the added feature of automatic base register assignment and displacement computation.

The assembled entries for the DC's are simply the requested constants, in hexadecimal. We note that the DS enters nothing, but simply reserves space. A study of the address for the doubleword constant, DEC, shows that boundary alignment has been performed. The fullword constant BIN2 was placed at 138; a fullword is four bytes, so 13C was available for DEC. But 13C is not on a doubleword boundary, so four bytes were skipped over and DEC was assigned to 140.

The action of the assembler instruction END, in causing the first executed instruction to be the one named BEGIN, is not exhibited on the assembly listing. This is done as a part of the makeup of the object program deck.

As it happens, the designation of BEGIN as the first instruction is not actually required. If the instruction is written simply as END, which is permitted, the assembler arranges to start executing instructions with the first word of the object deck. This would be the BALR instruction anyway. We have written the operand BEGIN as a matter of good programming habit.



```
ILLUSTRATIVE PROGRAM

                                                   TITLE ILLUSTRATIVE PROGRAM
                                                   START 256
000100    05 F0                            BEGIN   BALR  15,0
                             000102                USING *,15
000102    58 20 F 022                              L     2,DATA          LOAD REGISTER 2
000106    5A 20 F 02A                              A     2,TEN           ADD 10
                                           * THE FOLLOWING SHIFT HAS THE EFFECT OF MULTIPLYING BY 2
00010A    8B 20 0 001                              SLA   2,1
00010E    5B 20 F 026                              S     2,DATA+4        NOTE RELATIVE ADDRESSING
000112    50 20 F 02E                              ST    2,RESULT
000116    58 60 F 032                              L     6,BIN1
00011A    5A 60 F 036                              A     6,BIN2
00011E    4E 60 F 03E                              CVD   6,DEC           CONVERT TO DECIMAL
000122    0A 00                                    SVC   0               SUPERVISOR CALL
000124    00000019                         DATA    DC    F'25'
000128    0000000F                                 DC    F'15'
00012C    0000000A                         TEN     DC    F'10'
000130                                     RESULT  DS    F
000134    0000000C                         BIN1    DC    F'12'
000138    0000004E                         BIN2    DC    F'78'
000140                                     DEC     DS    D
                                                   END   BEGIN
```

Figure 28. Assembly listing of the program of Figure 27

A simplification of the writing of a program in the first place is only one of the major advantages of assembler language programming. Once written, the program is a great deal easier to modify than if it had been written with actual machine addresses. To illustrate this fact, we now make a few minor changes in the program of the preceding section.

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

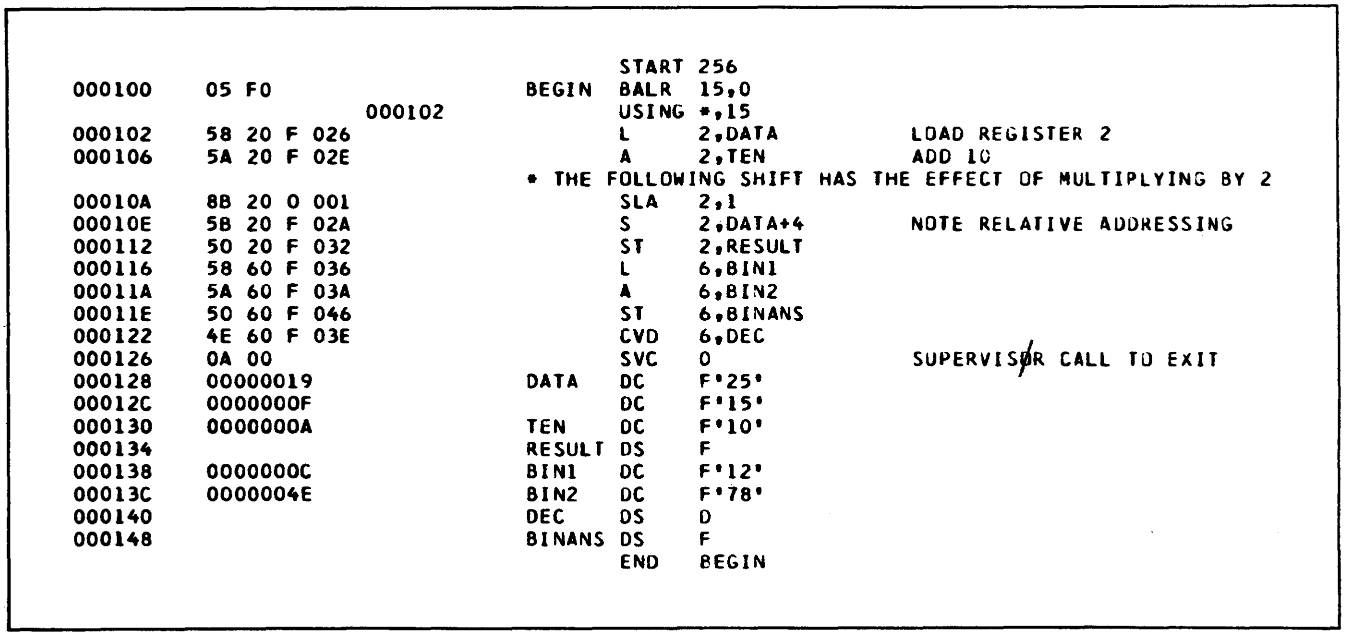
just before the CVD.

This is a rather simple sort of change and one that is representative of the kind of modification that is made with routine frequency on most 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 29 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 the displacements 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.

The intended lesson in this example is that the assembler handled clerical details of address computations in a more or less automatic manner that relieves the programmer of work and concern.



```
                                                   START 256
000100    05 F0                           BEGIN    BALR  15,0
                          000102                   USING *,15
000102    58 20 F 026                              L     2,DATA            LOAD REGISTER 2
000106    5A 20 F 02E                              A     2,TEN             ADD 10
                                          * THE FOLLOWING SHIFT HAS THE EFFECT OF MULTIPLYING BY 2
00010A    8B 20 0 001                              SLA   2,1
00010E    5B 20 F 02A                              S     2,DATA+4          NOTE RELATIVE ADDRESSING
000112    50 20 F 032                              ST    2,RESULT
000116    58 60 F 036                              L     6,BIN1
00011A    5A 60 F 03A                              A     6,BIN2
00011E    50 60 F 046                              ST    6,BINANS
000122    4E 60 F 03E                              CVD   6,DEC
000126    0A 00                                    SVC   0                 SUPERVISØR CALL TO EXIT
000128    00000019                        DATA     DC    F'25'
00012C    0000000F                                 DC    F'15'
000130    0000000A                        TEN      DC    F'10'
000134                                    RESULT   DS    F
000138    0000000C                        BIN1     DC    F'12'
00013C    0000004E                        BIN2     DC    F'78'
000140                                    DEC      DS    D
000148                                    BINANS   DS    F
                                                   END   BEGIN
```

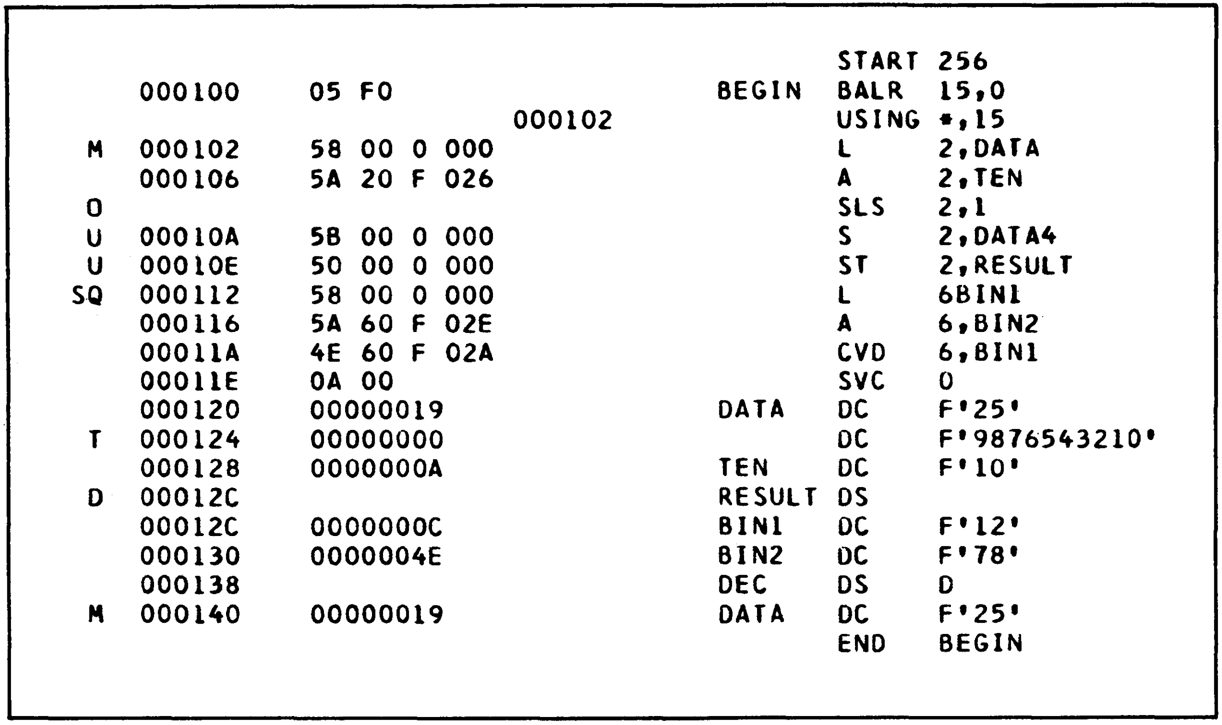
Figure 29. Assembly listing of a slightly modified version of the program of Figure 27

Error Analysis by the Assembler

Certain kinds of programming errors can be detected rather simply by the assembler. Some errors make it impossible to generate an instruction, and thereby disable the entire program. Others cannot be identified as definite errors, but only as possible or probable errors.

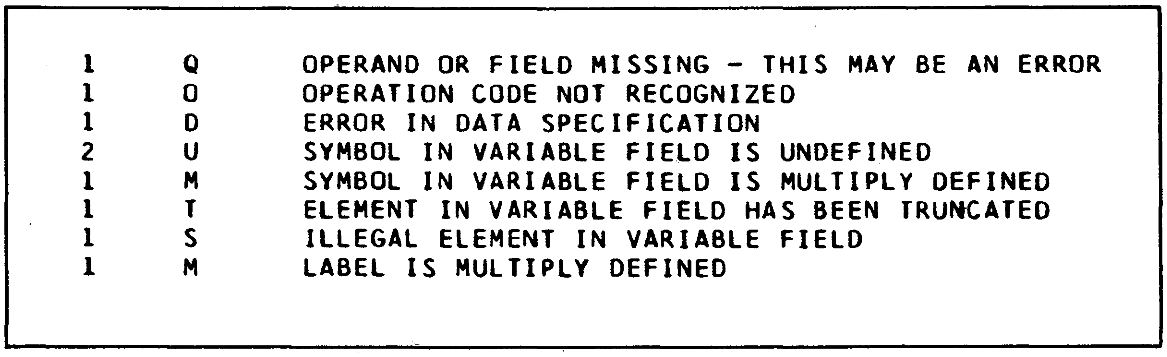
The assembler for the System/360 carries out the complete assembly, if possible, regardless of the number of errors, and regardless of the fact that the first error detected may have made it impossible to execute the object program. The idea is that if there are many errors the programmer would like to know about all of them, not just the first one the assembler encountered.

Figure 30 is the assembly listing of a program written specially with a number of errors in it, to demonstrate what the assembler can do and how it announces its findings. It will be noted in Figure 30 that many instructions have letters at the left end of the line. Figure 31 is the key to the meaning of these let-



```
                                                   START 256
    000100    05 FO                         BEGIN  BALR  15,0
                               000102              USING *,15
M   000102    58 00 0 000                          L     2,DATA
    000106    5A 20 F 026                          A     2,TEN
O                                                  SLS   2,1
U   00010A    5B 00 0 000                          S     2,DATA4
U   00010E    50 00 0 000                          ST    2,RESULT
SQ  000112    58 00 0 000                          L     6BIN1
    000116    5A 60 F 02E                          A     6,BIN2
    00011A    4E 60 F 02A                          CVD   6,BIN1
    00011E    0A 00                                SVC   0
    000120    00000019                      DATA   DC    F'25'
T   000124    00000000                             DC    F'9876543210'
    000128    0000000A                      TEN    DC    F'10'
D   00012C                                  RESULT DS
    00012C    0000000C                      BIN1   DC    F'12'
    000130    0000004E                      BIN2   DC    F'78'
    000138                                  DEC    DS    D
M   000140    00000019                      DATA   DC    F'25'
                                                   END   BEGIN
```

Figure 30. Assembly listing of a program with a number of deliberate errors



```
1    Q    OPERAND OR FIELD MISSING - THIS MAY BE AN ERROR
1    O    OPERATION CODE NOT RECOGNIZED
1    D    ERROR IN DATA SPECIFICATION
2    U    SYMBOL IN VARIABLE FIELD IS UNDEFINED
1    M    SYMBOL IN VARIABLE FIELD IS MULTIPLY DEFINED
1    T    ELEMENT IN VARIABLE FIELD HAS BEEN TRUNCATED
1    S    ILLEGAL ELEMENT IN VARIABLE FIELD
1    M    LABEL IS MULTIPLY DEFINED
```

Figure 31. Key to the error codes in Figure 30. This listing was produced by the assembler

ters; it was printed by the assembler as part of the output. Let us see what the assembler has told us.

The M is identified as meaning "label is multiply defined". If we try to take this too literally, it is a bit confusing: this instruction has no label. How about the address, DATA? Scanning down the page, we see the M again on the last DC, and we then notice that DATA has been used twice as a label. In such a case the assembler cannot know which was the intended one and which was the error, so it marks the error and does not assemble the instruction.

The next error is O, for "operation code not recognized". SLS was presumably written for SLA, with the programmer remembering the operation name Shift Left Single and writing SLS. The assembler makes no guesses on incorrect operation codes, and assembles no instruction.

The U on the next error means "symbol in variable field is undefined". The undefined symbol is DATA4. Remembering the earlier version of the program, we know that DATA+4 was meant—but the assembler cannot know that. The symbol DATA4 is simply undefined, since it never appears anywhere as a label.

The next instruction also has an undefined symbol. Why? We certainly have an entry for RESULT. But looking at the entry for RESULT, we see that *it* has an error: D for "error in data specification". The error is the complete absence of any kind of type specification. Without a type specification, the assembler cannot know how many bytes to allocate to the DS.

The next instruction is tagged as having two errors. S is for "illegal element in variable field", and Q means "operand or field missing—this may be an error". This is instructive: the whole trouble is the absence of a comma; the assembler did not tell us that, but came back with two rather different comments. This is no failure of the writers of the assembler; in many cases it is a very difficult matter to guess the writer's intentions when something does not meet specifications. The errors detected were that the first element in the operand field was something other than a number between zero and 15, and that there was only one field where there should be two. It will usually happen that even where the error comment is not directly indicative of the error, once the existence of an error has been pointed out we can see it fairly readily.

The T on the DC means "element in variable field has been truncated". What this refers to is the fact that the decimal number 9876543210 cannot be contained in a 32-bit word. We see that zero was established as the value to be loaded.

The other two errors have already been discussed.

There are several other errors that the assembler can detect. It does not seem worthwhile to devise error programs to either illustrate or list all of them. Programmers making the errors will discover soon enough the capabilities of the assembler. It must be noted, however, that there are many kinds of errors that are beyond the power of the assembler to analyze. If we incorrectly write DATA4 where we mean DATA+4, the assembler can detect it—unless DATA4 is itself a legitimate symbol!

In short, the error analysis capabilities of the System/360 assembler can be much help to us in producing a correct program, but the *absence* of error indications should never be taken to mean that the program is correct.

Decimal Instructions in Assembler Language

We have seen that the System/360 assembler language can automatically assign base registers and compute displacements. These functions are extensions of the ordinary assembler to handle special features of the computer. In assembling programs using the System/360 decimal arithmetic instructions, the assembler also goes somewhat beyond normal practice in allowing for lengths of operands to be *implied* rather than stated explicitly.

In the case of implied base registers, it is possible but seldom necessary to state a base register explicitly. In the case of the lengths of decimal operands, it is possible and frequently essential to state explicit lengths. For this reason, among others, the discussion that follows will be somewhat more involved than what has preceded. The reader who wishes to proceed to the study of fixed-point operations immediately will not be handicapped by skipping over this section and returning to it later, in connection with a study of decimal arithmetic. In fact, the reader without previous contact with System/360 decimal instructions is advised to postpone study of the rest of this chapter until he has become at least slightly familiar with decimal arithmetic.

Figure 32 is the assembly listing of a simple representative program to do some elementary processing of information with the decimal arithmetic feature of the System/360.

The START, BALR, and USING are much as before. Base register assignment is completely automatic in this program, as it was before.

The first processing instruction is:

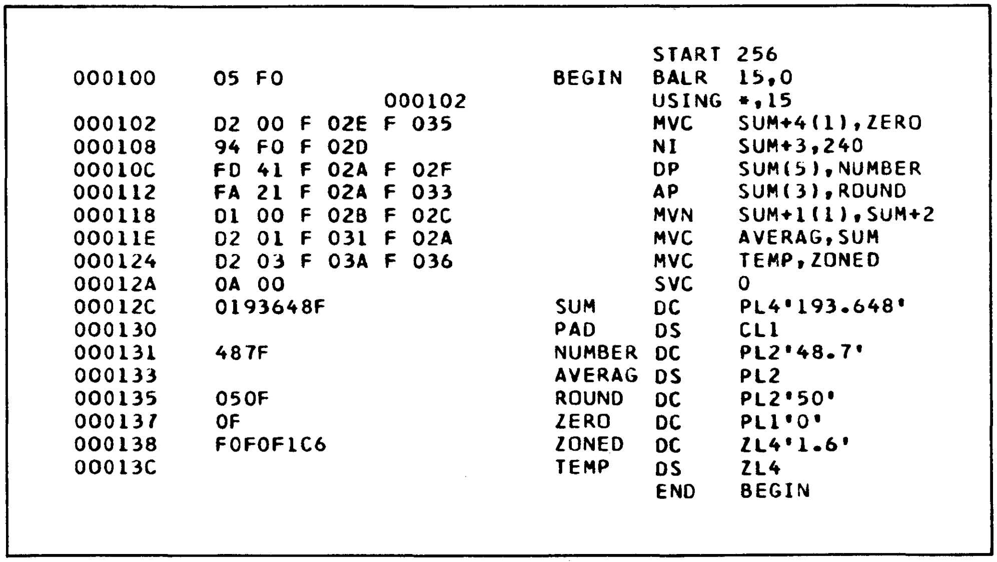
MVC SUM+4(1),ZERO

MVC stands for Move Characters. The instruction calls for the field at ZERO to be moved to a location the first byte of which is at SUM+4, and the field moved is one character long. Move Characters is an SS (Storage to Storage) format instruction, so there are two core storage addresses. The instruction is six bytes long. Reading across it, we have: the actual operation code is D2; one less than the length of the field moved is zero – that is, the field is one character in length; the base register for the first operand address is F; the displacement for the first operand address is 02E; the base register for the second operand address is also F; the displacement for the second operand address is 035.

This instruction tells us a number of things about how the assembler handles decimal instructions.

The length code in a decimal instruction is always one less than the length of the field to be moved. When we specify an explicit length, however, we are able to write the actual length; the assembler subtracts one to get the length code.

The symbol SUM, we see from the constants area of the program, is equivalent to 12C. The address in the



```
                                                    START 256
000100     05 F0                          BEGIN     BALR  15,0
                           000102                   USING *,15
000102     D2 00 F 02E F 035                        MVC   SUM+4(1),ZERO
000108     94 F0 F 02D                              NI    SUM+3,240
00010C     FD 41 F 02A F 02F                        DP    SUM(5),NUMBER
000112     FA 21 F 02A F 033                        AP    SUM(3),ROUND
000118     D1 00 F 02B F 02C                        MVN   SUM+1(1),SUM+2
00011E     D2 01 F 031 F 02A                        MVC   AVERAG,SUM
000124     D2 03 F 03A F 036                        MVC   TEMP,ZONED
00012A     0A 00                                    SVC   0
00012C     0193648F                       SUM       DC    PL4'193.648'
000130                                    PAD       DS    CL1
000131     487F                           NUMBER    DC    PL2'48.7'
000133                                    AVERAG    DS    PL2
000135     050F                           ROUND     DC    PL2'50'
000137     0F                             ZERO      DC    PL1'0'
000138     F0F0F1C6                       ZONED     DC    ZL4'1.6'
00013C                                    TEMP      DS    ZL4
                                                    END   BEGIN
```

Figure 32. Assembly listing of a program involving decimal instructions

instruction was SUM+4, which is equivalent to 130. The base register contains 102 and the displacement is 02E, which add to 130 as required. (It is essential never to forget that variable-length fields in the System/360 are always addressed by the *left*most byte.) The base register contents of 102 plus the displacement of 035 also correctly lead to the address of 137 for ZERO.

The next instruction is an example of the final type of instruction format, the SI (Storage-Immediate). In this class of instruction, a part of the instruction contains data. (This is the meaning of "immediate", as distinguished from a reference to a storage location addressed by the instruction.) In the example at hand, a certain logical operation is to be performed using the one character at SUM+3 and the part of the instruction represented by the decimal number 240.

Looking at the assembled instruction, we have: the actual operation code is 94; the "immediate" part, in hexadecimal, is F0; the base register for the one storage reference is F; the displacement is 02D.

The next instruction is a Divide Decimal, which is an SS format instruction, but with a slight difference in that there are two length codes in the actual instruction, one for each field. In the symbolic instruction, we have written one of the lengths explicitly and left the other implied. The length of 5 with SUM was done to extend the field by one character, to include an extra byte for division. In the assembled instruction, after the operation code of FD, we see 41. The 4 is the length code for SUM(5), the code being one less than the actual length. The length code for the second operand is 1, and we see that NUMBER is in fact a two-byte field. The assembled address for SUM(5) is 102 + 02A = 12C, the address of the leftmost byte of the field.

The Add Decimal instruction that follows introduces no new assembler concepts. It is also an SS format instruction with two length codes. Note in this case that we needed an explicit length of 3 to be associated with SUM, so the length code in the instruction came out 2; this did not affect the assembled address.

The Move Numeric (MVN) again presents no new assembler ideas.

The Move Characters that follows contains two addresses, both written without explicit lengths. The two implied lengths are different, in an instruction in which there is only one length code. We see that the assembler has picked the implied length of the first operand, AVERAG.

The final Move Characters and the Supervisor Call should present no difficulties.

In the DC assembler instruction for SUM we have a new type specification, P, for Packed. "Packed" refers to the two-digits-in-one-byte form in which numbers must appear for use with the decimal arithmetic instructions. Since decimal arithmetic deals with variable-length data, we use a *length modifier* to indicate how many bytes are to be assigned. (Actually, the length modifier may be omitted if we are satisfied to accept the length that would be implied by the number of digits to be stored. In this case, the length assigned would be 4 even without the L4 in the operand.)

The assembled constant shows that the sign (F) has been put into the rightmost four bits of the rightmost byte, as required, that the one unused digit position has been filled with zero and placed at the left, and that the decimal point in the constant as we wrote it has been ignored. (We may write decimal points in a P-type DC or not, as desired, for our convenience. If written, they have no effect on the assembler.)

The symbol PAD goes with a DS instruction to reserve one character position, that is, one byte, in which we place nothing. The type designation this time is C, for Character. This is the type designation that may be used whenever variable-length space is reserved with a DS instruction, and it may be used on a DC to enter data in character form, such as alphabetic information.

The next four entries provide further examples of the same types of DC and DS instructions we have already seen.

The DC for ZONED introduces one more type designation, Z, for Zoned. This refers to the zoned format for numerical data, in which each digit takes up a complete byte, and the sign is contained in the zone portion (leftmost four bits) of the rightmost byte. In the assembled constant, the F's are the zones attached to the nonsign digits, and the C is the plus sign. The decimal point in the constant as we wrote it has again been ignored.

The length modifier here is essential, if we really want the assembled constant to have four bytes; without a length indication, the assembler would assign the two bytes in which the constant can be stored.

Summary

The techniques presented in this chapter will help the programmer write useful System/360 programs in assembler language. Further aspects of the language appear in other chapters in this text, and the interested reader can study the appropriate SRL document on the assembler language.

Questions and Exercises

1. The name of an instruction refers to the (leftmost, rightmost) byte of the instruction. The name of a constant refers to the (leftmost, rightmost) byte of the constant.

2. On an assembly listing, the storage location of each instruction, constant or area, as well as the object instruction and constant are printed in (decimal/ hexadecimal).

3. Consider Figure 28:

a. State the location of the instruction ST 2,RESULT.

b. State the location of the field RESULT.

c. Recall that an object instruction (including the one for ST 2,RESULT) refers to a storage location through the specification of a base register, a displacement, and sometimes an index register. At execution time System/360 develops the address (the effective address) of the storage location by developing the sum of the contents of the specified base register, the contents of the specified index register (if any), and the displacement. Now consider how, for the object instruction for ST 2,RESULT, the effective address of RESULT is formed (no indexing is specified).

(1) What base register is specified?

(2) What are the indicated contents of the base register?

(3) What is the displacement?

(4) What is the effective address?

4. State the difference between the DC and DS Assembler instructions.

5. Consider the constants set up in Figure 28 and assume that the instruction beginning at location 112 is ST 2,DATA+12 instead of as shown in Figure 28. State the effective address of DATA+12.

6. Consider Figure 28:

a. State the locations that are allocated by the statement RESULT DS F.

b. State the locations that are occupied by the constant BIN2.

c. State the locations that are allocated by the statement DEC DS D.

7. Assume the DS statements given below and also assume that the assembler assigned a location of 138 to AREA1 and 140 to AREA2. What locations will the assembler assign to the next four areas?

Location			
138	AREA1	DS	F
140	AREA2	DS	D
—	AREA3	DS	F
—	AREA4	DS	D
—	AREA5	DS	F
—	AREA6	DS	D

8. In answering question 7, you will have observed that some locations were skipped to align fields on their proper boundaries. Resequence the DS statements in question 7 so that storage is used more efficiently. Assume that the first DS will start at 138.

Chapter 4: Fixed-Point Operations

This chapter introduces and discusses some of the fixed-point operations 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.

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.

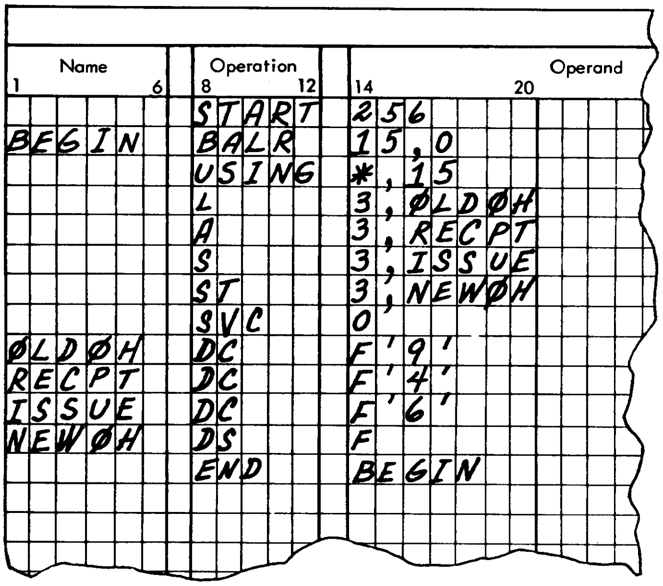
For a first example we shall consider a simple inventory calculation. We begin the calculation with an on-hand quantity, a receipt quantity, and an issue quantity. We are required to compute the new on-hand, according to the formula:

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 33. We shall be concentrating on the four actual processing instructions, but at the outset we shall display all programs in logically complete form.


Name
1
6
Operation
8
12
14
20
Operand

```
       START 256
BEGIN  BALR  15,0
       USING *,15
       L     3,ØLDØH
       A     3,RECPT
       S     3,ISSUE
       ST    3,NEWØH
       SVC   0
ØLDØH  DC    F'9'
RECPT  DC    F'4'
ISSUE  DC    F'6'
NEWØH  DS    F
       END   BEGIN
```

Figure 33. A program, written in assembler language, to perform a simple arithmetic computation in binary

The first 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 15 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.

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.

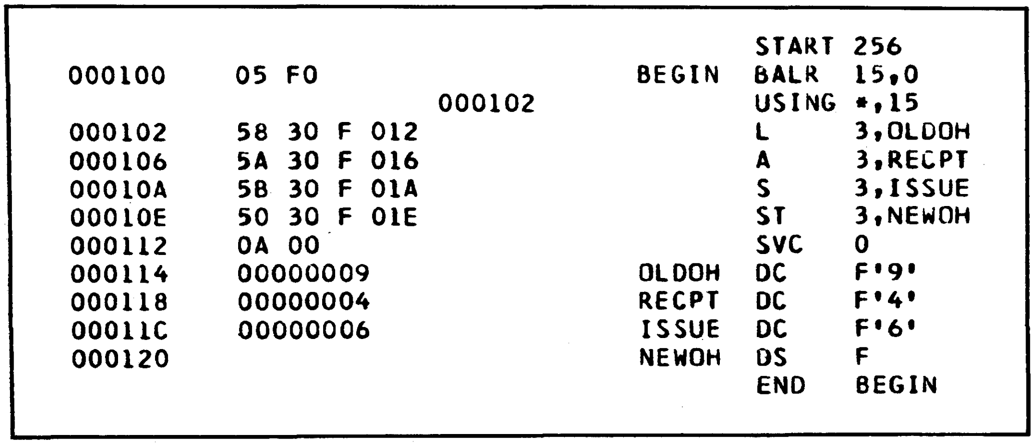
4. The effective address of a byte in storage is formed by adding the contents of the base register, the contents of the index register, and the displacement. If register zero is specified for an index register or a base register, zero is used in the address computation, rather than whatever register 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 34. 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,



```
                                                   START 256
000100     05 F0                           BEGIN   BALR  15,0
                           000102                  USING *,15
000102     58 30 F 012                             L     3,OLDOH
000106     5A 30 F 016                             A     3,RECPT
00010A     5B 30 F 01A                             S     3,ISSUE
00010E     50 30 F 01E                             ST    3,NEWOH
000112     0A 00                                   SVC   0
000114     00000009                        OLDOH   DC    F'9'
000118     00000004                        RECPT   DC    F'4'
00011C     00000006                        ISSUE   DC    F'6'
000120                                     NEWOH   DS    F
                                                   END   BEGIN
```

Figure 34. The assembly listing for the program of Figure 33

the index register is zero, the base register is F ($= 15_{10}$),and the displacement is 012_{16}. Since the base register contains 102, the effective address is 114, which we see is the address associated with 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 15 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 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 somehow indicate what we want done next. The System/360 provides no Stop instruction, to force a program organization that keeps the machine in operation as much of the time as possible. What we have shown here is a Supervisor Call instruction. The use of this instruction assumes that there is in storage, at the time of execution of this program, a supervisor program that runs the machine between jobs. We here indicate to the supervisor program that this program has no further need for the machine; the operand of zero specifies that no further actions on the part of the supervisor program are needed by this program.

We have not written the result onto any output device. In actual practice, previous parts of the program would have read the values used in the calculation, and subsequent parts would use the result. We are not prepared at this time to write the input and output instructions and, indeed, will not do so throughout this publication. We have simply entered illustrative values for the input quantities 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. (There are corresponding halfword instructions, as we shall see in later examples.)

The END instruction informs the assembler that the termination of the program has been reached and (in this case) specifies that the first instruction to be executed when the program is loaded should be the one having the name BEGIN, namely the BALR.

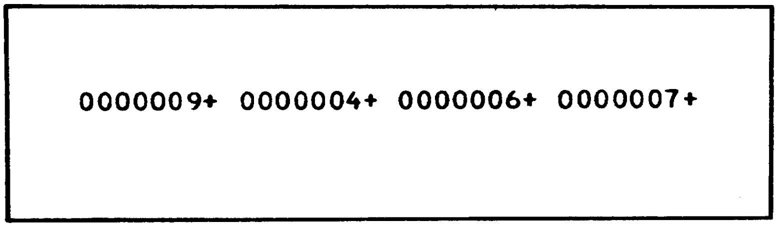
With a suitable program it is possible, and rather simple once the methods are clear, to get a "dump" of the contents of selected areas of storage. With such a dump routine we can get our data and results out of the machine without writing any output instructions.

This was done, leading to the numbers reproduced in Figure 35. The four items, in sequence, are OLDOH, RECPT, ISSUE, and NEWOH.

It might be interesting to run this program again, but with the illustrative value for ISSUE being, say, 16. This will result in a negative quantity for NEWOH. We know that negative fixed-point numbers are represented in 2's complement form. The dump routine used above will make a conversion to true-number-and-sign, as shown in the first line of Figure 36. In the second line, the same numbers are printed also in hexadecimal; this was done by a slightly different dump call to the supervisor program.

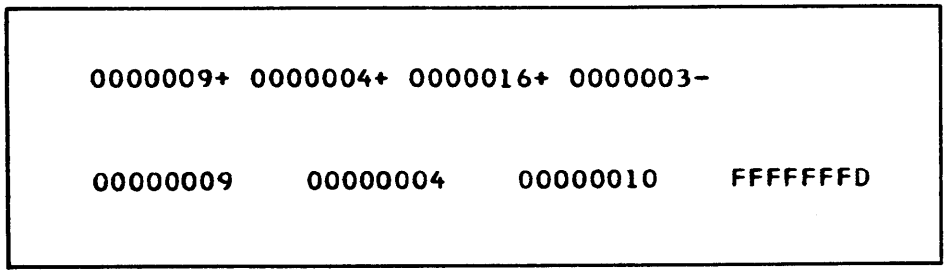
We see that the first three numbers, which are positive, have zeros before the significant digits. The last number, however, being negative, is shown in hexadecimal form to have 1's to the left of the significant digits, since $F_{16} = 1111_2$. If we were to write out this hexadecimal number, FFFFFFFD, in its binary form, we would have thirty 1's followed by 01. Referring to the rules for formation of a 2's complement, we see that the complement of this number is 11_2, which is 3. Checking with the given data and the formula, we see that this is the correct answer and, of course, –3 was printed as the decimal value for a further check, if one was needed.

Naturally, if such a result were actually produced 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 you received, which is what the negative result would imply. A complete program would include a test for the possibility of a negative result, with some kind of corrective action.



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

Figure 35. Output of the program of Figures 33 and 34. The four numbers are OLDOH, RECPT, ISSUE, and NEWOH, in that order.



```
0000009+ 0000004+ 0000016+ 0000003-

00000009     00000004     00000010     FFFFFFFD
```

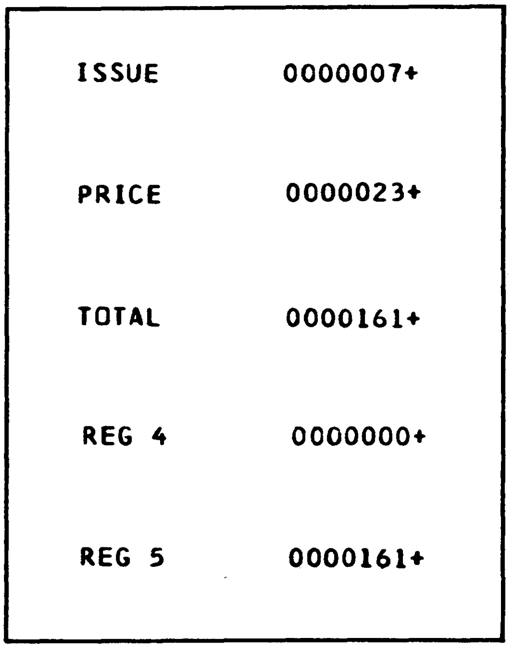
Figure 36. Output of the program of Figures 33 and 34, with a value for ISSUE that causes NEWOH to be negative. The values of OLDOH, RECPT, ISSUE, and NEWOH are printed in the top line in decimal. In the second line they are printed in hexadecimal; the value for NEWOH is in complement form.

For a simple example of 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 37. The first three lines are standard. 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 multiplicand 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.

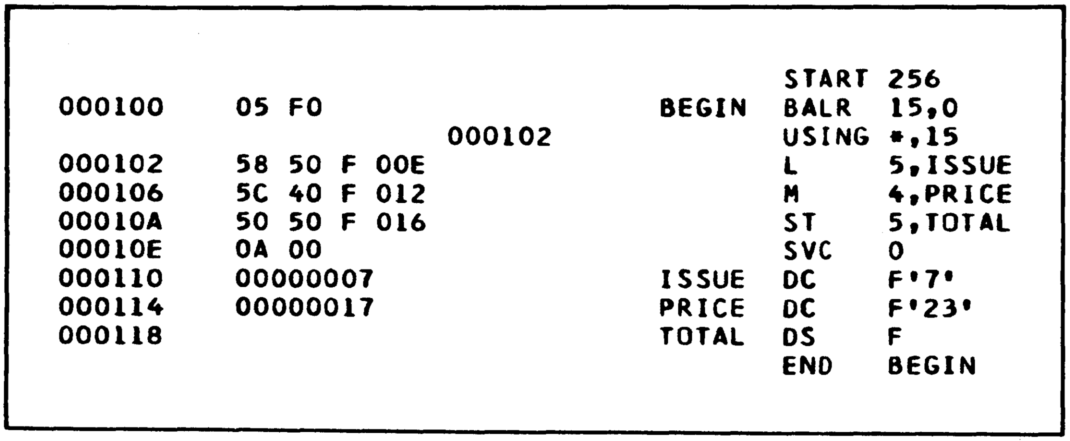
After the product has been formed, we store it in TOTAL on the assumption that the result does not exceed the length of one register. The validity of such an assumption, of course, is the responsibility of the programmer; if in fact the product extended over into register 4, there would be no automatic 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. The printout of Figure 38 gives ISSUE, PRICE TOTAL, and the contents of register 4 and 5 after the completion of the program. The identifications were produced with suitable instructions to the dump routine. 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          0000007+

PRICE          0000023+

TOTAL          0000161+

REG 4          0000000+

REG 5          0000161+
```

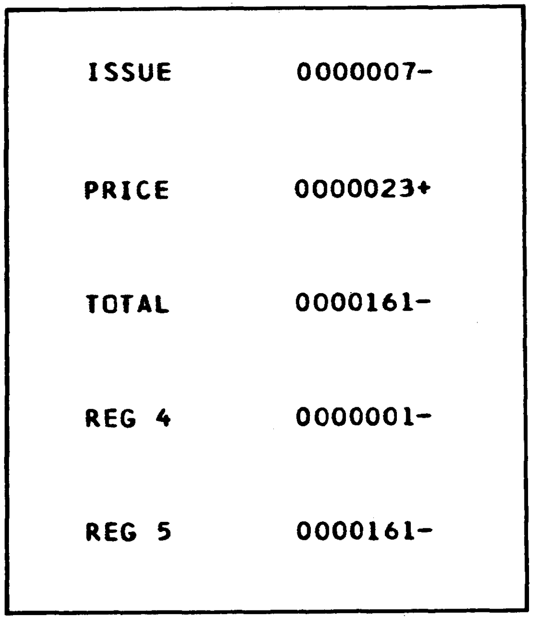
Figure 38. Output of the program of Figure 37



```
                                            START  256
000100    05 F0                      BEGIN  BALR   15,0
                         000102             USING  *,15
000102    58 50 F 00E                       L      5,ISSUE
000106    5C 40 F 012                       M      4,PRICE
00010A    50 50 F 016                       ST     5,TOTAL
00010E    0A 00                             SVC    0
000110    00000007                   ISSUE  DC     F'7'
000114    00000017                   PRICE  DC     F'23'
000118                               TOTAL  DS     F
                                            END    BEGIN
```

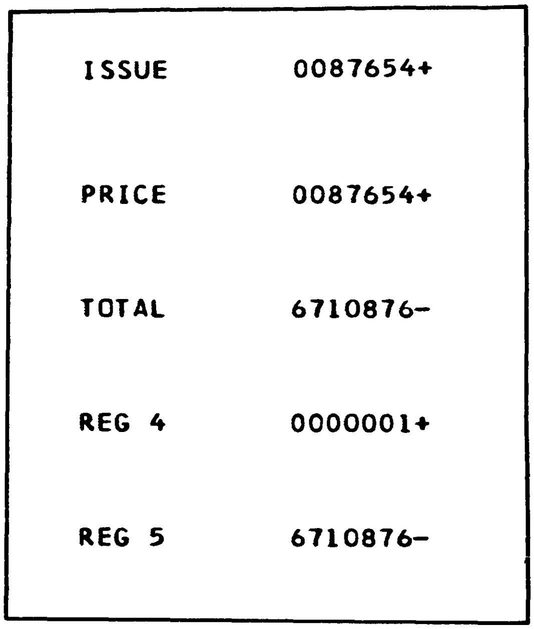
Figure 37. Assembly listing of a program to perform a binary multiplication

In Figure 39 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       0000007-

PRICE       0000023+

TOTAL       0000161-

REG 4       0000001-

REG 5       0000161-
```

Figure 39. Output of the program of Figure 37, with a negative value for ISSUE



```
ISSUE       0087654+

PRICE       0087654+

TOTAL       6710876-

REG 4       0000001+

REG 5       6710876-
```

Figure 40. Output of the program of Figure 37, with values for ISSUE and PRICE that lead to a TOTAL too large to fit in a fullword

In Figure 40 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 5858830849. 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?

The answer becomes apparent if we convert the product to hexadecimal and look at 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 signals a negative number when register 5 is taken as a word by itself!

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.

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 to 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.

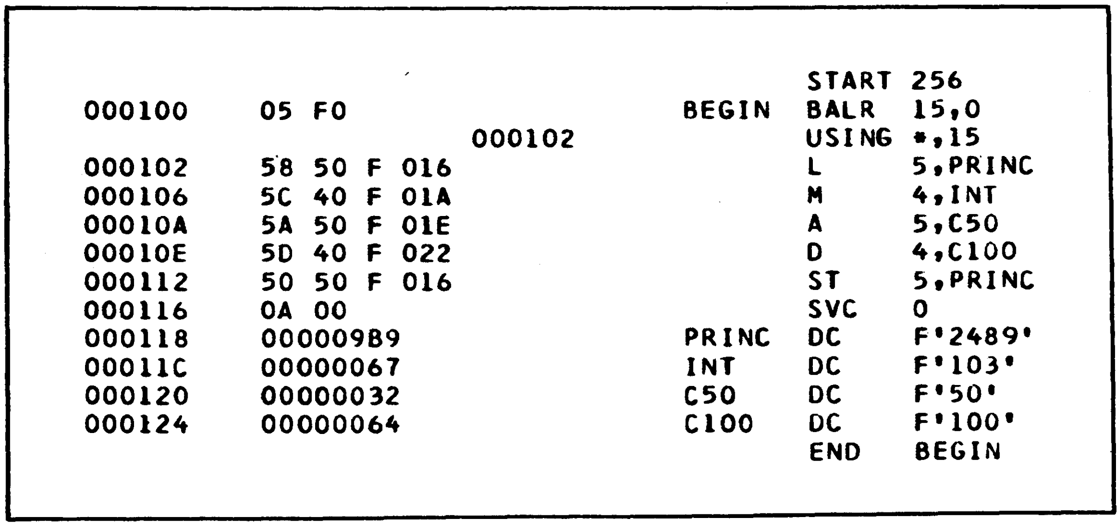
One possible program is shown in Figure 41. (There are other ways, as we shall see.) After the usual preliminaries we load the principal into an odd-numbered register preparatory to multiplying. The interest rate is shown as 103, which should be read as 1.03. This is a shortcut: instead of multiplying 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, 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 re-



```
                                               START 256
000100    05 F0                        BEGIN   BALR  15,0
                            000102             USING *,15
000102    58 50 F 016                          L     5,PRINC
000106    5C 40 F 01A                          M     4,INT
00010A    5A 50 F 01E                          A     5,C50
00010E    5D 40 F 022                          D     4,C100
000112    50 50 F 016                          ST    5,PRINC
000116    0A 00                                SVC   0
000118    000009B9                     PRINC   DC    F'2489'
00011C    00000067                     INT     DC    F'103'
000120    00000032                     C50     DC    F'50'
000124    00000064                     C100    DC    F'100'
                                               END   BEGIN
```

Figure 41. Assembly listing of a program involving binary multiplication and division, with binary rounding by a decimal amount

mainder 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.

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 42 to see how this much of the revised program looks.

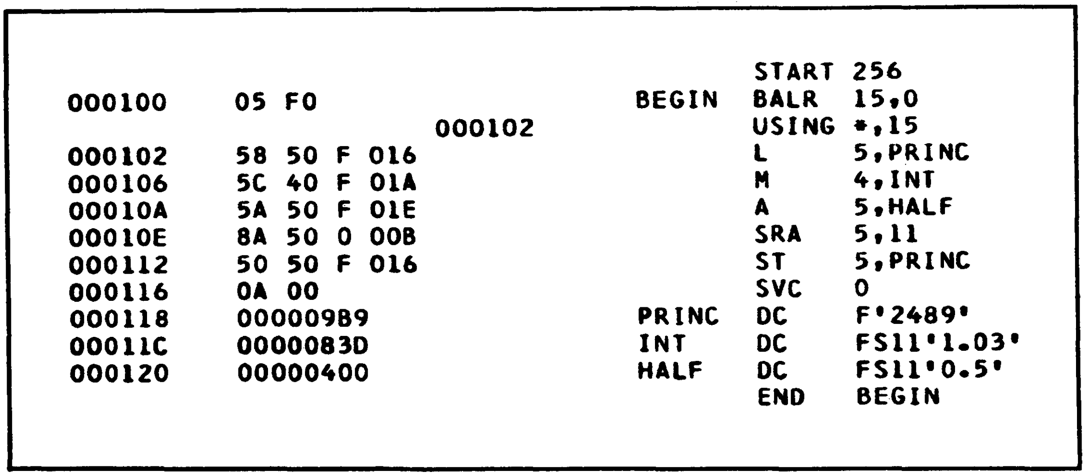
The Load is the same as before, as is the Multiply. The constant used for the multiplication is different, however. Down at INT we see that the DC is

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 wrote 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



```
                                                   START 256
000100     05 F0                          BEGIN    BALR  15,0
                          000102                   USING *,15
000102     58 50 F 016                             L     5,PRINC
000106     5C 40 F 01A                             M     4,INT
00010A     5A 50 F 01E                             A     5,HALF
00010E     8A 50 0 00B                             SRA   5,11
000112     50 50 F 016                             ST    5,PRINC
000116     0A 00                                   SVC   0
000118     000009B9                       PRINC    DC    F'2489'
00011C     0000083D                       INT      DC    FS11'1.03'
000120     00000400                       HALF     DC    FS11'0.5'
                                                   END   BEGIN
```

Figure 42. A different version of the program of Figure 41, using a scale factor to get a binary fraction in a DC constant

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 2463 ($24.63); 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 2464 ($24.64) 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.

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 we assume, in which three data items are packed 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.

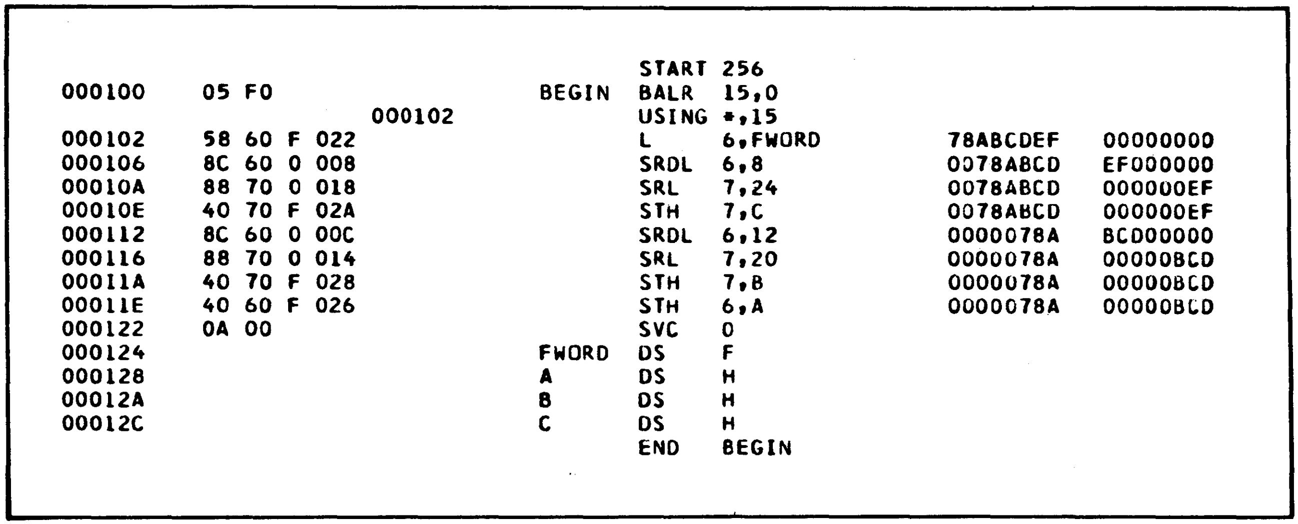
The program shown in Figure 43 is a more or less straightforward matter of shifting and storing, but a few notes are necessary to make clear what is happening at certain points.

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 packed 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 packed word 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 packed fullword are now properly positioned in register 7 to be stored in a halfword location with the Store Halfword (SH) 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

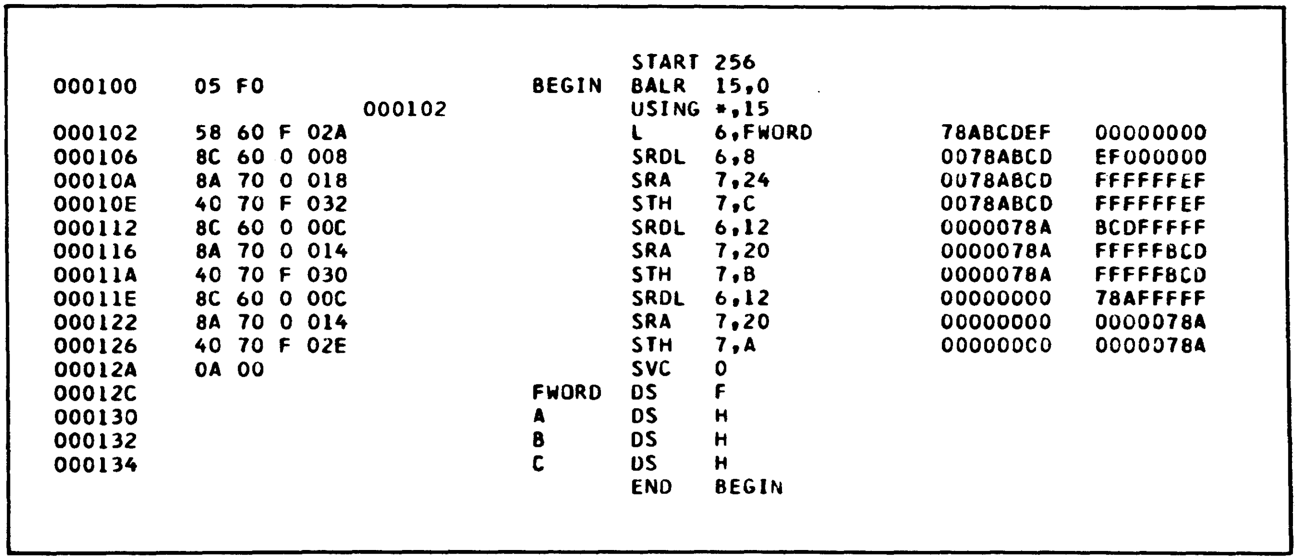


```
                                                START 256
000100   05 F0                          BEGIN   BALR  15,0
                           000102               USING *,15
000102   58 60 F 022                            L     6,FWORD           78ABCDEF   00000000
000106   8C 60 0 008                            SRDL  6,8               0078ABCD   EF000000
00010A   88 70 0 018                            SRL   7,24              0078ABCD   000000EF
00010E   40 70 F 02A                            STH   7,C               0078ABCD   000000EF
000112   8C 60 0 00C                            SRDL  6,12              0000078A   BCD00000
000116   88 70 0 014                            SRL   7,20              0000078A   00000BCD
00011A   40 70 F 028                            STH   7,B               0000078A   00000BCD
00011E   40 60 F 026                            STH   6,A               0000078A   00000BCD
000122   0A 00                                  SVC   0
000124                                  FWORD   DS    F
000128                                  A       DS    H
00012A                                  B       DS    H
00012C                                  C       DS    H
                                                END   BEGIN
```

Figure 43. Assembly listing of a program to separate three quantities packed in one fullword

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.

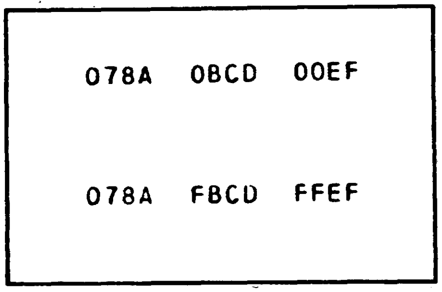
Actually, the restriction to positive numbers is not too difficult to remove. It would have to be agreed that the leftmost bit of each item was its sign bit, that is, that in generating the packed word each item was in complement form and of such length as to fit in the item size allotted. With this assumption, the program of Figure 44 properly expands the sign bits of the items and stores the three 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 changed to include a final two shifts to expand the sign of item A.



```
                                                     START 256
000100     05 F0                           BEGIN     BALR  15,0
                         000102                      USING *,15
000102     58 60 F 02A                               L     6,FWORD          78ABCDEF    00000000
000106     8C 60 0 008                               SRDL  6,8              0078ABCD    EF000000
00010A     8A 70 0 018                               SRA   7,24             0078ABCD    FFFFFFEF
00010E     40 70 F 032                               STH   7,C              0078ABCD    FFFFFFEF
000112     8C 60 0 00C                               SRDL  6,12             0000078A    BCDFFFFF
000116     8A 70 0 014                               SRA   7,20             0000078A    FFFFFBCD
00011A     40 70 F 030                               STH   7,B              0000078A    FFFFFBCD
00011E     8C 60 0 00C                               SRDL  6,12             00000000    78AFFFFF
000122     8A 70 0 014                               SRA   7,20             00000000    0000078A
000126     40 70 F 02E                               STH   7,A              000000C0    0000078A
00012A     0A 00                                     SVC   0
00012C                                     FWORD     DS    F
000130                                     A         DS    H
000132                                     B         DS    H
000134                                     C         DS    H
                                                     END   BEGIN
```

Figure 44. Modified version of the program of Figure 43, making it operate correctly if the three quantities are allowed to be negative

Figure 45 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 45 we see that these have been put into halfwords by the program of Figure 43 as 078A, 0BCD and 00EF, that is, as three positive numbers. In the second line we see that the program of Figure 44, on the other hand, interpreted the second and third numbers as negative since their leftmost bits were 1's. The three output halfwords are 078A, FBCD, and FFEF, showing that the sign bits of the second and third numbers were properly expanded.



```
078A   0BCD   00EF


078A   FBCD   FFEF
```

Figure 45. Output of the programs of Figure 43 (first line) and Figure 44 (second line) with the original word being hexadecimal 78ABCDEF

Branches and Decision Codes

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 values of the condition code is specified in the description of each instruction that affects the code. (Many instructions do not.) The four possible values are 0, 1, 2, and 3.

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 which values of the condition code are of interest to us. 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 as follows:

Mask bits	*Decimal value*	*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 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.

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 46 expresses the logic of the method that will be used here to perform the sort; other ways are possible.

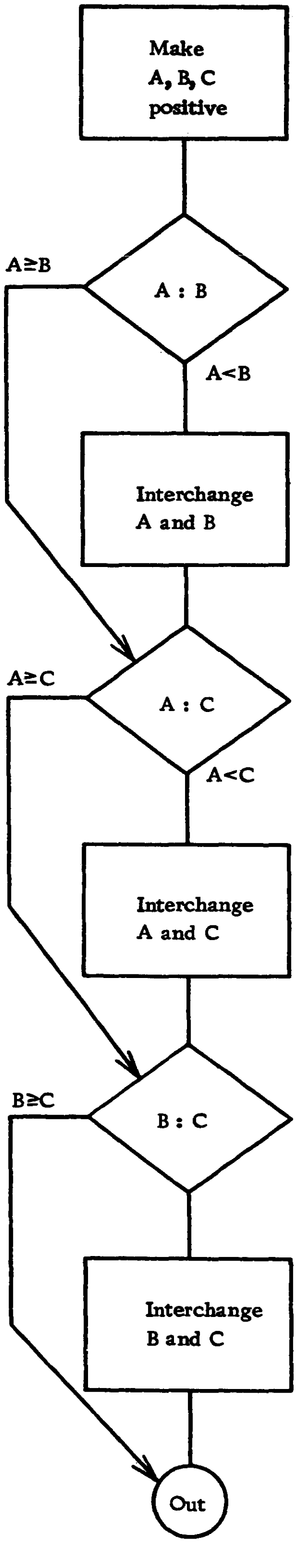


Figure 46. Program flowchart of a method of sorting three numbers into descending sequence. Any negative numbers are changed to positive before sorting.

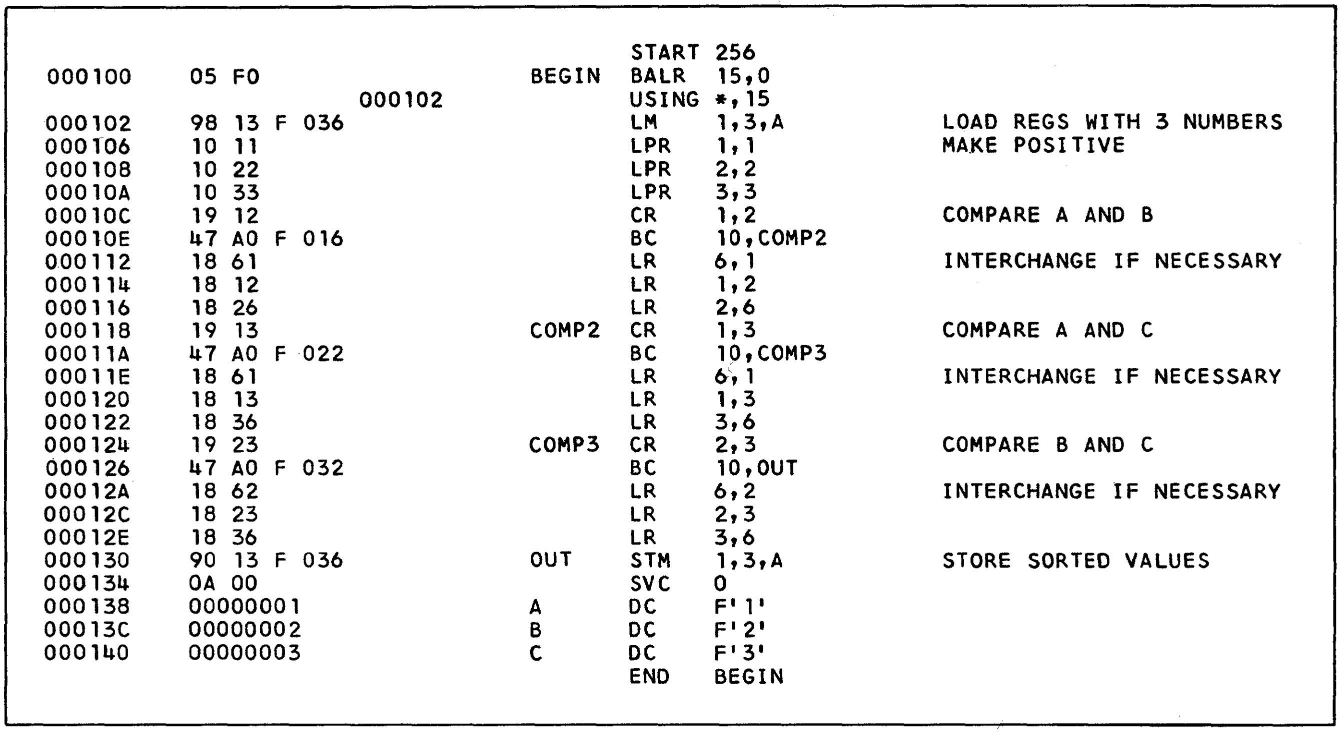
We first make all three numbers positive. A comparison is then made between A and B; if A is the smaller we interchange the two values. Now we know that what is in A is the larger of the two, whether because it originally was or because we interchanged it with the original B. A similar process compares 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 in B and C, and an interchange if necessary, gets the "middle" number in B and the smallest in C.

The program of Figure 47 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 1, B in 2, and C in 3.

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 was negative, and place the result back in the same register. If it were necessary, the two registers could of course be different.

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 1 is already equal to or greater than the number in register 2, we skip down to the second comparison, because A and B are already in correct sequence.



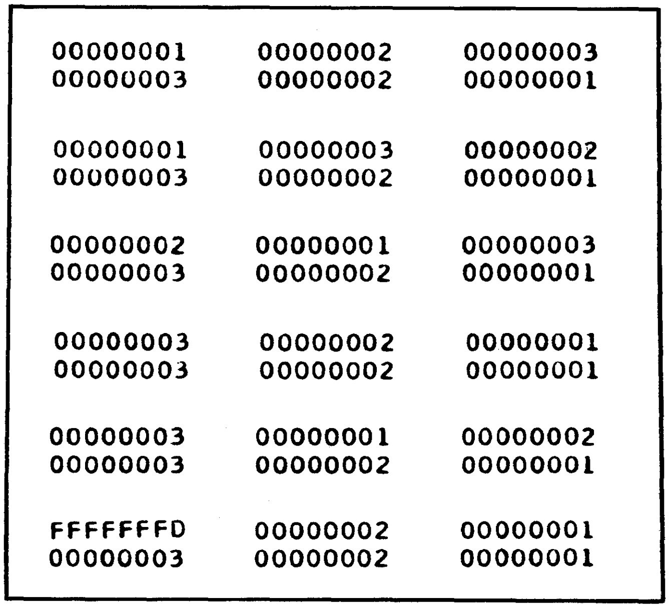
```
                                               START 256
000100    05 F0                         BEGIN  BALR  15,0
                          000102               USING *,15
000102    98 13 F 036                          LM    1,3,A           LOAD REGS WITH 3 NUMBERS
000106    10 11                                LPR   1,1             MAKE POSITIVE
000108    10 22                                LPR   2,2
00010A    10 33                                LPR   3,3
00010C    19 12                                CR    1,2             COMPARE A AND B
00010E    47 A0 F 016                          BC    10,COMP2
000112    18 61                                LR    6,1             INTERCHANGE IF NECESSARY
000114    18 12                                LR    1,2
000116    18 26                                LR    2,6
000118    19 13                         COMP2  CR    1,3             COMPARE A AND C
00011A    47 A0 F 022                          BC    10,COMP3
00011E    18 61                                LR    6,1             INTERCHANGE IF NECESSARY
000120    18 13                                LR    1,3
000122    18 36                                LR    3,6
000124    19 23                         COMP3  CR    2,3             COMPARE B AND C
000126    47 A0 F 032                          BC    10,OUT
00012A    18 62                                LR    6,2             INTERCHANGE IF NECESSARY
00012C    18 23                                LR    2,3
00012E    18 36                                LR    3,6
000130    90 13 F 036                   OUT    STM   1,3,A           STORE SORTED VALUES
000134    0A 00                                SVC   0
000138    00000001                      A      DC    F'1'
00013C    00000002                      B      DC    F'2'
000140    00000003                      C      DC    F'3'
                                               END   BEGIN
```

Figure 47. Assembly listing of a program to carry out the sorting procedure charted in Figure 46

The interchange, if it is necessary, is performed by moving the contents of register 1 to register 6, moving 2 to 1, and finally moving 6 to 2. 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 48 shows before-and-after values of A, B, and C for the 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.



```
00000001    00000002    00000003
00000003    00000002    00000001

00000001    00000003    00000002
00000003    00000002    00000001

00000002    00000001    00000003
00000003    00000002    00000001

00000003    00000002    00000001
00000003    00000002    00000001

00000003    00000001    00000002
00000003    00000002    00000001

FFFFFFFD    00000002    00000001
00000003    00000002    00000001
```

Figure 48. Sets of sample input and output for the program of Figure 47. Each pair of lines represents three input values (first line) and the sorted output (second line).

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

We are given a man's earning for a week (EARN), his previous ("old") year-to-date earnings (OLDYTD), and his previous year-to-date Social Security tax (OLDSS). We are to compute his Social Security tax for this week (TAX), his new year-to-date earnings (NEWYTD) and new Social Security tax (NEWSS). Assume the Social Security tax is computed as 3⅝% of earnings (with certain exclusions such as sick pay, which we shall ignore) up to an annual limit on taxable income of $4800. The program must decide whether the employee has yet earned $4800 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 $4800 before this week's pay but, counting this week's pay, goes over $4800, only the portion of this week's pay that takes him up to the $4800 limit is taxable.

The flowchart of Figure 49 expresses the logic we have just described. Figure 50 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. Now we compare the *old* year-to-date with $4800. 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 $4800. If the branch is not taken, the old year-to-date was already over $4800, 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 $4800. The Branch on Condition with a mask of 2 asks whether the first operand – the new year-to-date – was greater than $4800. If so, it is necessary to compute the tax on just that part of this week's pay that takes the total up to $4800. At OVER48, accordingly, we load register 7 with $4800 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.

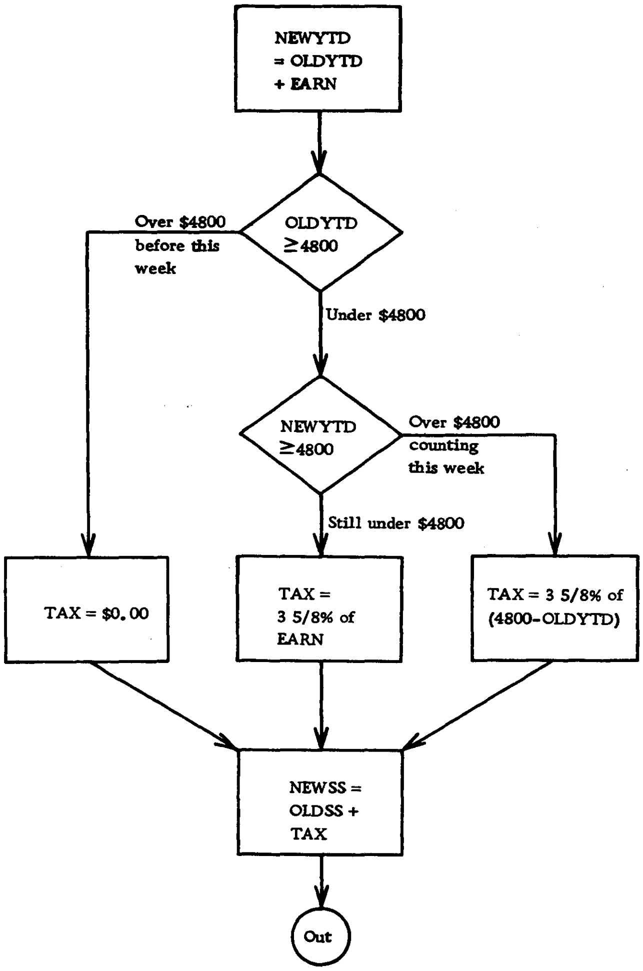
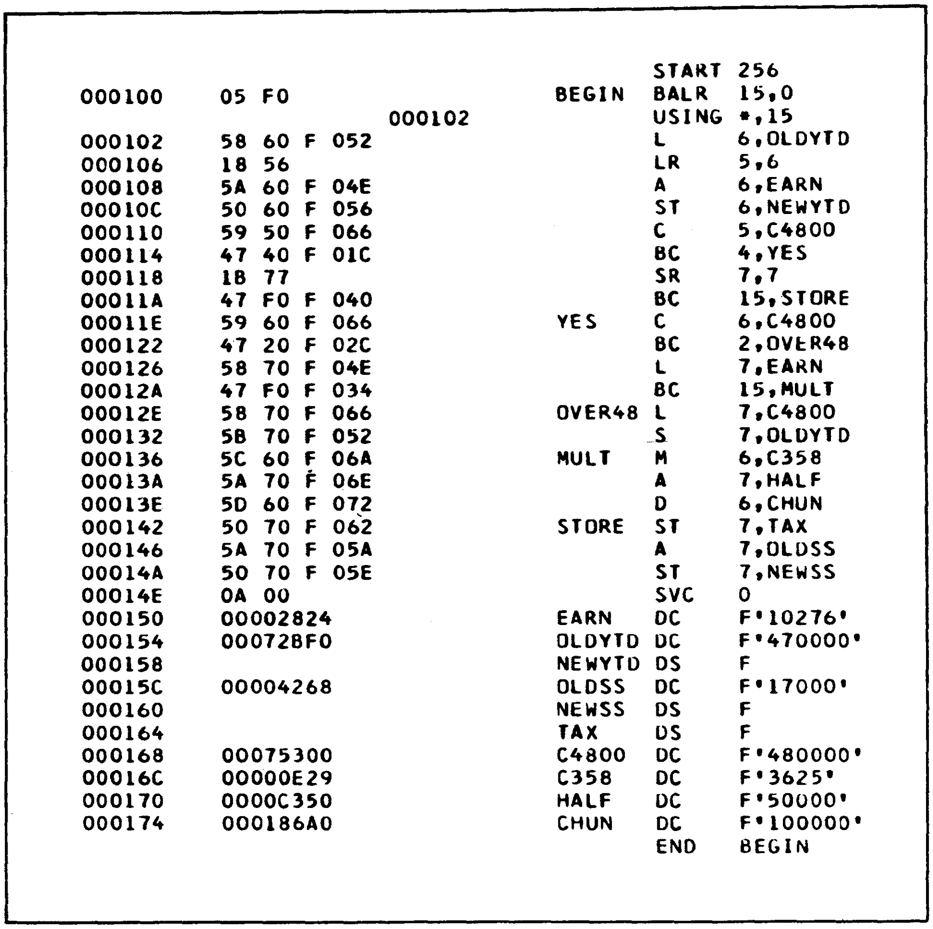


Figure 49. Program flowchart of a procedure for computing a Social Security tax

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 3⅝%. This constant is entered as the integer 3625, which is the decimal fraction form of the percentage. We must think of this number as



```
                                                    START 256
000100    05 F0                            BEGIN    BALR  15,0
                           000102                   USING *,15
000102    58 60 F 052                               L     6,OLDYTD
000106    18 56                                     LR    5,6
000108    5A 60 F 04E                               A     6,EARN
00010C    50 60 F 056                               ST    6,NEWYTD
000110    59 50 F 066                               C     5,C4800
000114    47 40 F 01C                               BC    4,YES
000118    1B 77                                     SR    7,7
00011A    47 F0 F 040                               BC    15,STORE
00011E    59 60 F 066                      YES      C     6,C4800
000122    47 20 F 02C                               BC    2,OVER48
000126    58 70 F 04E                               L     7,EARN
00012A    47 F0 F 034                               BC    15,MULT
00012E    58 70 F 066                      OVER48   L     7,C4800
000132    5B 70 F 052                               S     7,OLDYTD
000136    5C 60 F 06A                      MULT     M     6,C358
00013A    5A 70 F 06E                               A     7,HALF
00013E    5D 60 F 072                               D     6,CHUN
000142    50 70 F 062                      STORE    ST    7,TAX
000146    5A 70 F 05A                               A     7,OLDSS
00014A    50 70 F 05E                               ST    7,NEWSS
00014E    0A 00                                     SVC   0
000150    00002824                         EARN     DC    F'10276'
000154    00072BF0                         OLDYTD   DC    F'470000'
000158                                     NEWYTD   DS    F
00015C    00004268                         OLDSS    DC    F'17000'
000160                                     NEWSS    DS    F
000164                                     TAX      DS    F
000168    00075300                         C4800    DC    F'480000'
00016C    00000E29                         C358     DC    F'3625'
000170    0000C350                         HALF     DC    F'50000'
000174    000186A0                         CHUN     DC    F'100000'
                                                    END   BEGIN
```

Figure 50. Assembly listing of one version of a program to calculate Social Security tax

0.03625, however, remembering that it is a fraction. The constant for rounding, HALF, is therefore 50,000, and we remove all the excess decimals by dividing by 100,000. 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 $102.76 per week. Multiplying by 0.03625 and rounding to the nearest cent, we get a Social Security tax of $3.73. In 46 weeks of working at this rate, the man will accumulate a year-to-date earnings of $4726.96 and a year-to-date Social Security tax of $171.58. Now in the next week his full earnings are not taxable, but only the part that takes him up to $4800, or $73.04; the tax on this amount is $2.65. Adding $2.65 to his previous year-to-date Social Security, we get $174.23, which is more than 3⅝% of the $4800 maximum

The difficulty is in the computation of the tax on one week's earnings. Before rounding, the product of $102.76 and 0.03625 is $3.725050. When we round this to $3.73 we add nearly half a cent. For each of the 46 weeks we are adding nearly half a cent – which adds up to 23 cents.

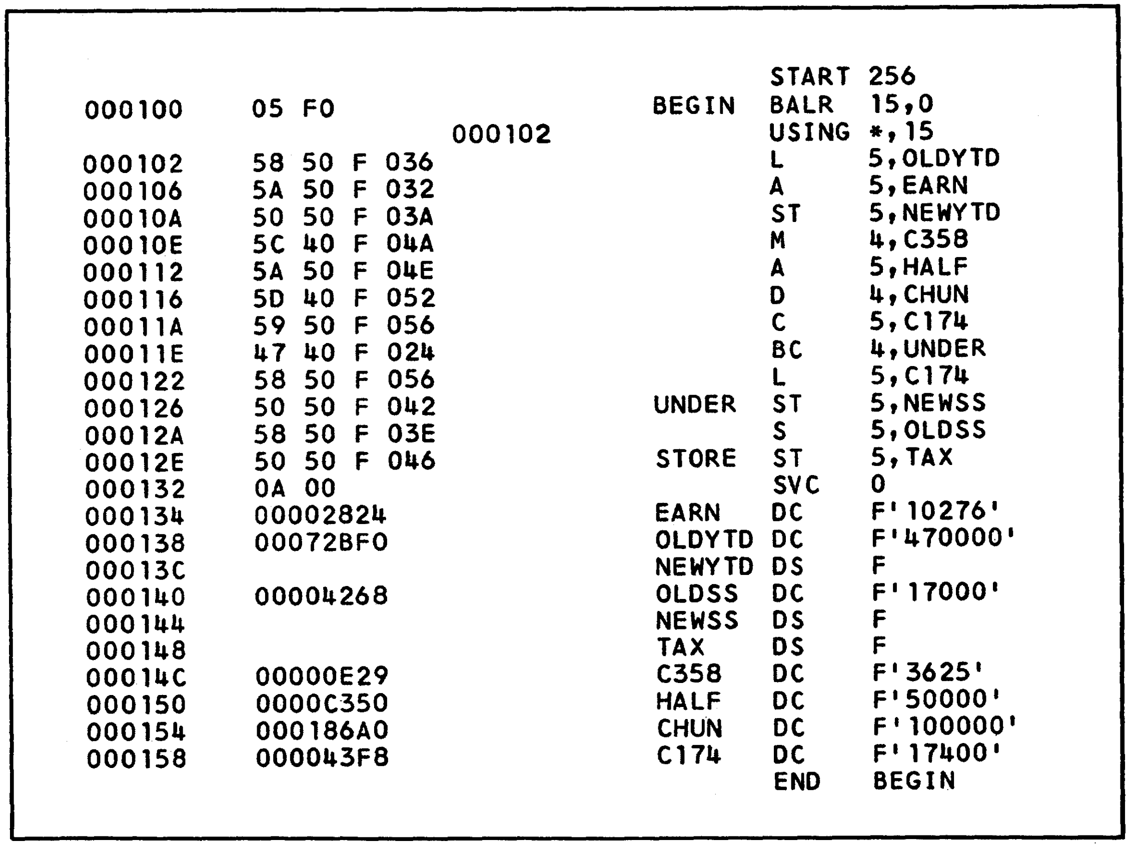
This would be sloppy; the employee would have a right to claim a refund, if he wanted to bother; the government would be annoyed. Social Security tax is very seldom computed the way we have shown.

Fortunately, correcting the trouble is not only fairly easy, but leads to a shorter program. The general approach is to compute 3⅝% 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 $3.73 as the tax. The second week, we begin by computing 0.03625 times $205.52, the new year-to-date gross; this gives us $7.45 as the new year-to-date Social Security, which we store. This week's tax is $7.45 minus the previous year-to-date Social Security of $3.73, or $3.72. In other words, where last week we were a half a cent high, now we are half a cent low; the two cancel each other. The offset will not always be so simple; 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 $174. If so, the result is replaced by $174 and the tax is computed as before, by subtracting the previous year-to-date Social Security. If that was already $174, 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 is taking him over the limit, the tax is the difference between the maximum tax and the amount already paid, which is correct.

The program shown in Figure 51 should not be too difficult to follow after the description of the process that has just been given. The program is eight instructions 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.



```
                                                 START 256
000100     05 F0                          BEGIN  BALR  15,0
                           000102                USING *,15
000102     58 50 F 036                           L     5,OLDYTD
000106     5A 50 F 032                           A     5,EARN
00010A     50 50 F 03A                           ST    5,NEWYTD
00010E     5C 40 F 04A                           M     4,C358
000112     5A 50 F 04E                           A     5,HALF
000116     5D 40 F 052                           D     4,CHUN
00011A     59 50 F 056                           C     5,C174
00011E     47 40 F 024                           BC    4,UNDER
000122     58 50 F 056                           L     5,C174
000126     50 50 F 042                    UNDER  ST    5,NEWSS
00012A     58 50 F 03E                           S     5,OLDSS
00012E     50 50 F 046                    STORE  ST    5,TAX
000132     0A 00                                 SVC   0
000134     00002824                       EARN   DC    F'10276'
000138     00072BF0                       OLDYTD DC    F'470000'
00013C                                    NEWYTD DS    F
000140     00004268                       OLDSS  DC    F'17000'
000144                                    NEWSS  DS    F
000148                                    TAX    DS    F
00014C     00000E29                       C358   DC    F'3625'
000150     0000C350                       HALF   DC    F'50000'
000154     000186A0                       CHUN   DC    F'100000'
000158     000043F8                       C174   DC    F'17400'
                                                 END   BEGIN
```

Figure 51. Assembly listing of a much better version of a 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 52. 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.

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 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.

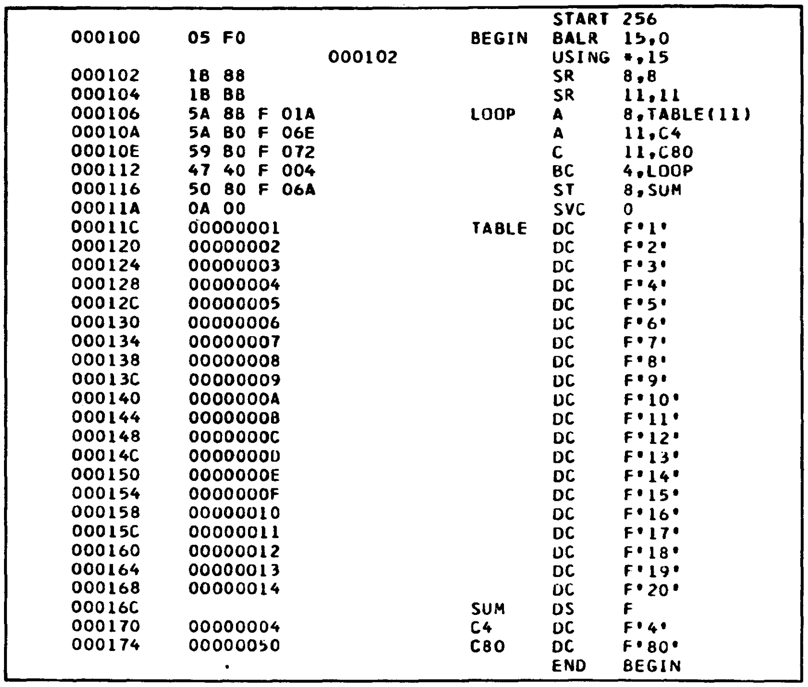
The second version shortens the repeated section of the loop by one instruction. Normally, we do not worry too much about trying to get the last microsecond out of programs, but in heavily repeated parts it is worth some effort.

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 53 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.

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.

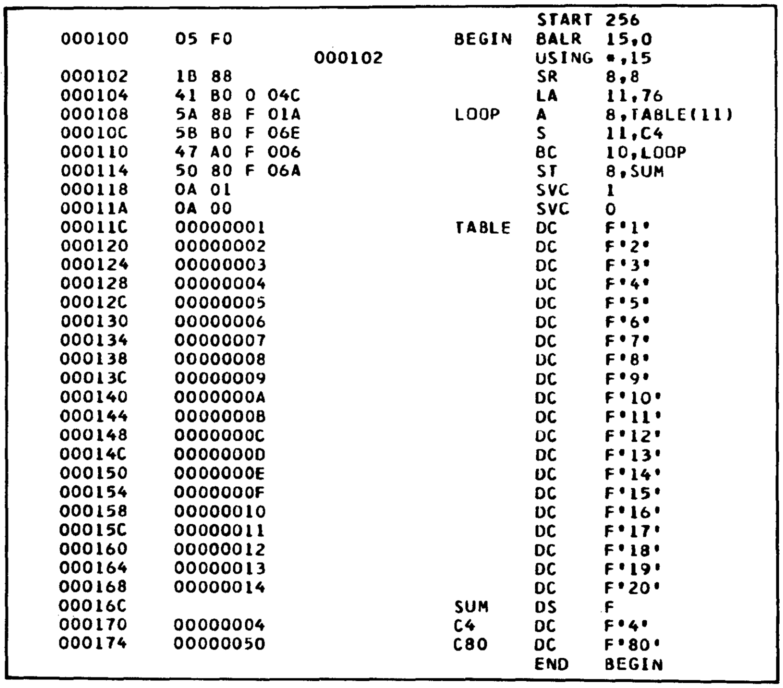
The Store is as before.

Where in the first version there were four instruc-



```
                                             START 256
000100    05 F0                      BEGIN   BALR  15,0
                        000102               USING *,15
000102    1B 88                              SR    8,8
000104    1B BB                              SR    11,11
000106    5A 8B F 01A                LOOP    A     8,TABLE(11)
00010A    5A B0 F 06E                        A     11,C4
00010E    59 B0 F 072                        C     11,C80
000112    47 40 F 004                        BC    4,LOOP
000116    50 80 F 06A                        ST    8,SUM
00011A    0A 00                              SVC   0
00011C    00000001                   TABLE   DC    F'1'
000120    00000002                           DC    F'2'
000124    00000003                           DC    F'3'
000128    00000004                           DC    F'4'
00012C    00000005                           DC    F'5'
000130    00000006                           DC    F'6'
000134    00000007                           DC    F'7'
000138    00000008                           DC    F'8'
00013C    00000009                           DC    F'9'
000140    0000000A                           DC    F'10'
000144    0000000B                           DC    F'11'
000148    0000000C                           DC    F'12'
00014C    0000000D                           DC    F'13'
000150    0000000E                           DC    F'14'
000154    0000000F                           DC    F'15'
000158    00000010                           DC    F'16'
00015C    00000011                           DC    F'17'
000160    00000012                           DC    F'18'
000164    00000013                           DC    F'19'
000168    00000014                           DC    F'20'
00016C                               SUM     DS    F
000170    00000004                   C4      DC    F'4'
000174    00000050                   C80     DC    F'80'
                                             END   BEGIN
```

Figure 52. First version of a program to form the sum of 20 numbers



```
                                             START 256
000100    05 F0                      BEGIN   BALR  15,0
                        000102               USING *,15
000102    1B 88                              SR    8,8
000104    41 B0 0 04C                        LA    11,76
000108    5A 8B F 01A                LOOP    A     8,TABLE(11)
00010C    5B B0 F 06E                        S     11,C4
000110    47 A0 F 006                        BC    10,LOOP
000114    50 80 F 06A                        ST    8,SUM
000118    0A 01                              SVC   1
00011A    0A 00                              SVC   0
00011C    00000001                   TABLE   DC    F'1'
000120    00000002                           DC    F'2'
000124    00000003                           DC    F'3'
000128    00000004                           DC    F'4'
00012C    00000005                           DC    F'5'
000130    00000006                           DC    F'6'
000134    00000007                           DC    F'7'
000138    00000008                           DC    F'8'
00013C    00000009                           DC    F'9'
000140    0000000A                           DC    F'10'
000144    0000000B                           DC    F'11'
000148    0000000C                           DC    F'12'
00014C    0000000D                           DC    F'13'
000150    0000000E                           DC    F'14'
000154    0000000F                           DC    F'15'
000158    00000010                           DC    F'16'
00015C    00000011                           DC    F'17'
000160    00000012                           DC    F'18'
000164    00000013                           DC    F'19'
000168    00000014                           DC    F'20'
00016C                               SUM     DS    F
000170    00000004                   C4      DC    F'4'
000174    00000050                   C80     DC    F'80'
                                             END   BEGIN
```

Figure 53. Second version of a program to form the sum of 20 numbers

tions 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 11 will be the index; it initially contains zero. Register 12 will contain the amount by which the index is to be incremented each time around the loop, 4. Register 13 will contain the limit value, the value of the index which is not to be exceeded, 76. If we have the instruction:

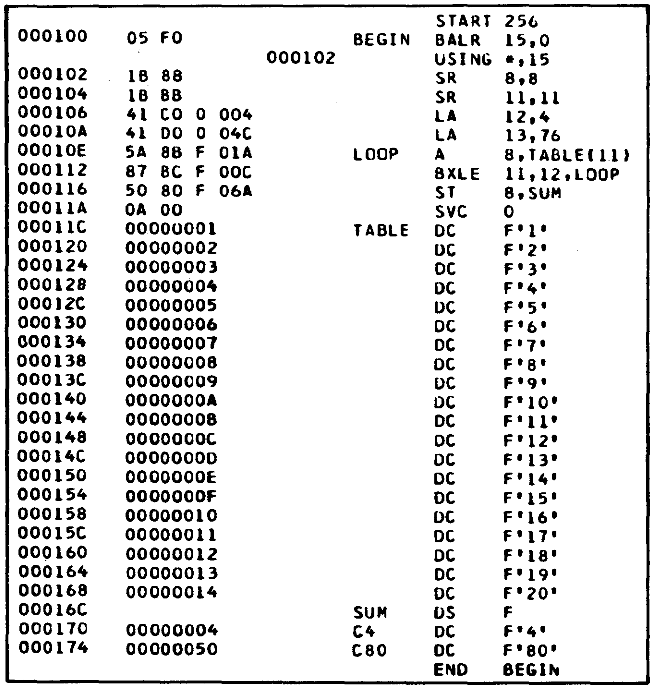
BXLE 11,12,LOOP

the action will be as follows: The contents of register 12 (4) are added to register 11, which is the index and initially contains zero. If the sum is less than or equal to the contents of register 13, the limit, the branch to LOOP is taken; otherwise the next instruction in sequence is taken.

The instruction is written in assembly 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:



```
                                              START 256
000100   05 F0                        BEGIN   BALR  15,0
                           000102             USING *,15
000102   1B 88                                SR    8,8
000104   1B BB                                SR    11,11
000106   41 C0 0 004                          LA    12,4
00010A   41 D0 0 04C                          LA    13,76
00010E   5A 8B F 01A                  LOOP    A     8,TABLE(11)
000112   87 BC F 00C                          BXLE  11,12,LOOP
000116   50 80 F 06A                          ST    8,SUM
00011A   0A 00                                SVC   0
00011C   00000001                     TABLE   DC    F'1'
000120   00000002                             DC    F'2'
000124   00000003                             DC    F'3'
000128   00000004                             DC    F'4'
00012C   00000005                             DC    F'5'
000130   00000006                             DC    F'6'
000134   00000007                             DC    F'7'
000138   00000008                             DC    F'8'
00013C   00000009                             DC    F'9'
000140   0000000A                             DC    F'10'
000144   0000000B                             DC    F'11'
000148   0000000C                             DC    F'12'
00014C   0000000D                             DC    F'13'
000150   0000000E                             DC    F'14'
000154   0000000F                             DC    F'15'
000158   00000010                             DC    F'16'
00015C   00000011                             DC    F'17'
000160   00000012                             DC    F'18'
000164   00000013                             DC    F'19'
000168   00000014                             DC    F'20'
00016C                                SUM     DS    F
000170   00000004                     C4      DC    F'4'
000174   00000050                     C80     DC    F'80'
                                              END   BEGIN
```

Figure 54. Third version of a program to form the sum of 20 numbers. This shortest version uses the Branch Index Low or Equal (BXLE) instruction.

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 11,12,LOOP

   the limit value is in register 13, 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.

We shall see in later chapters how it can be useful to have the second and third registers the same – for now we shall use R3 operands that are even-numbered.

This instruction at first glance seems more complicated than it is. Let us turn to an example to see how it works. Figure 54 is the final version of our summing loop.

We begin the program by loading the 3 registers that will be used by the BXLE instruction (registers 11, 12, and 13), 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 situations 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.

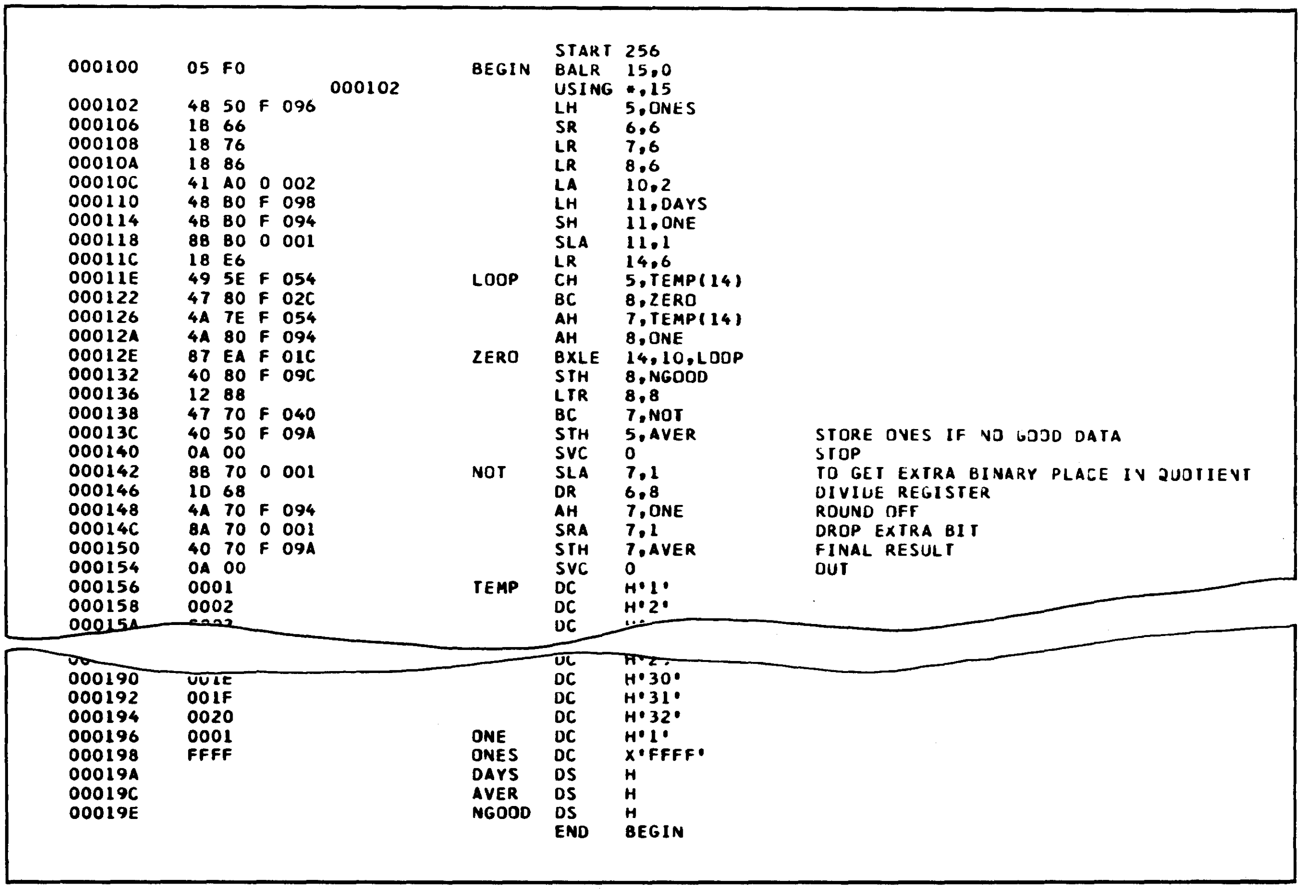
In an attempt to draw together some of the things that have been discussed in this chapter, we shall now consider a final problem that involves several different concepts.

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 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 55 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*
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
14	Index register
15	Base register



```
                                            START 256
000100    05 F0                      BEGIN  BALR  15,0
                         000102             USING *,15
000102    48 50 F 096                       LH    5,ONES
000106    18 66                             SR    6,6
000108    18 76                             LR    7,6
00010A    18 86                             LR    8,6
00010C    41 A0 0 002                       LA    10,2
000110    48 B0 F 098                       LH    11,DAYS
000114    4B B0 F 094                       SH    11,ONE
000118    8B B0 0 001                       SLA   11,1
00011C    18 E6                             LR    14,6
00011E    49 5E F 054                LOOP   CH    5,TEMP(14)
000122    47 80 F 02C                       BC    8,ZERO
000126    4A 7E F 054                       AH    7,TEMP(14)
00012A    4A 80 F 094                       AH    8,ONE
00012E    87 EA F 01C                ZERO   BXLE  14,10,LOOP
000132    40 80 F 09C                       STH   8,NGOOD
000136    12 88                             LTR   8,8
000138    47 70 F 040                       BC    7,NOT
00013C    40 50 F 09A                       STH   5,AVER         STORE ONES IF NO GOOD DATA
000140    0A 00                             SVC   0              STOP
000142    8B 70 0 001                NOT    SLA   7,1            TO GET EXTRA BINARY PLACE IN QUOTIENT
000146    1D 68                             DR    6,8            DIVIDE REGISTER
000148    4A 70 F 094                       AH    7,ONE          ROUND OFF
00014C    8A 70 0 001                       SRA   7,1            DROP EXTRA BIT
000150    40 70 F 09A                       STH   7,AVER         FINAL RESULT
000154    0A 00                             SVC   0              OUT
000156    0001                       TEMP   DC    H'1'
000158    0002                              DC    H'2'
00015A                                      DC
                                            DC
000190    001E                              DC    H'30'
000192    001F                              DC    H'31'
000194    0020                              DC    H'32'
000196    0001                       ONE    DC    H'1'
000198    FFFF                       ONES   DC    X'FFFF'
00019A                               DAYS   DS    H
00019C                               AVER   DS    H
00019E                               NGOOD  DS    H
                                            END   BEGIN
```

Figure 55. A program to compute the average of a set of temperatures, taking into account the possibility of missing readings

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 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 (14) 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).

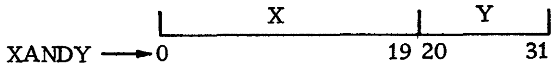
2. The first operand of an instruction *usually* specifies the operand that (sends, receives) information.

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 core called X, and Y in a halfword area in core 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 52. 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 5: Programming with Base Registers and the USING Instruction

A major programming feature of the System/360 is the use of base registers, which provide three important advantages. First, 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. Second, 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. Third, if proper care is exercised, base registers may still be used for indexing through storage addresses without destroying their effectiveness for the first two purposes.

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 of the clerical work of assigning 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 the remainder of this publication 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 the timing of various actions: during assembly, program loading, or program execution.

The USING Instruction

Automatic assignment of base registers and the automatic computation of displacement require the programmer to supply two items of information to the assembler and one to the object machine. 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: assign base registers and compute 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  15,0
```

This places in register 15 the address of the next byte after the BALR, and there is no branch. The choice of register 15 was arbitrary.

An Example

These ideas may be made more concrete by considering an example. Figure 56 is an assembly listing of a program the processing details of which do not concern us.

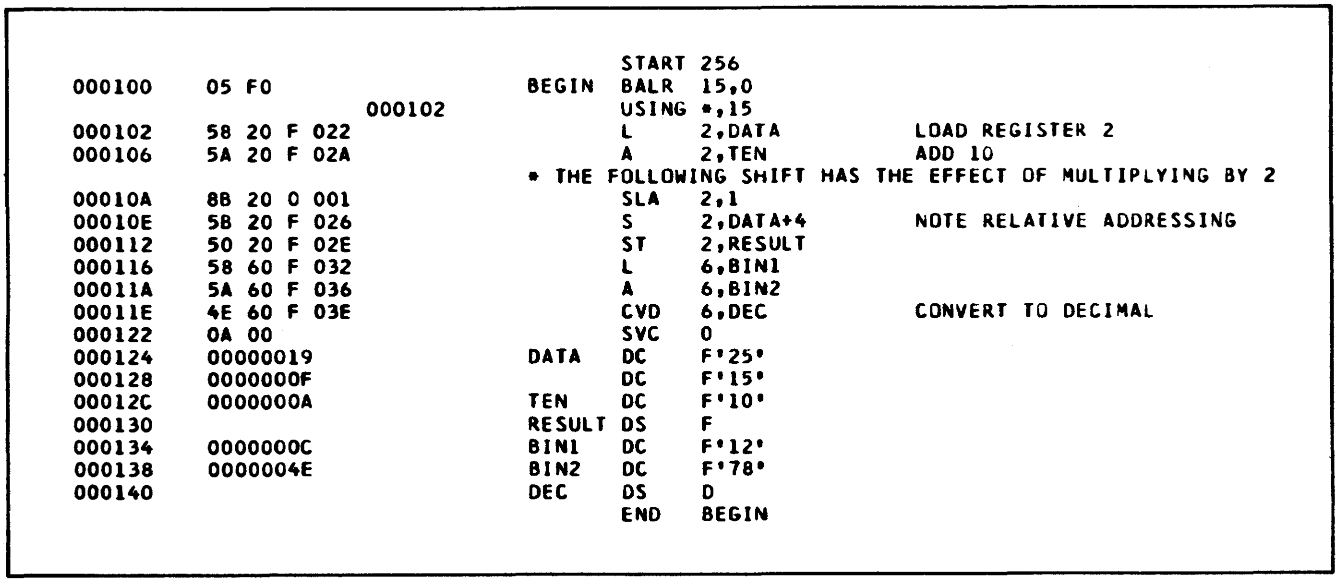
The START instruction specifies that the assembled first byte of the program is location $256_{10} = 100_{16}$. We see that the BALR instruction has in fact been placed in 100. (All numbers in the object program area of the assembly listing – on the left-hand side – are hexadecimal.) The BALR instruction specifies that general purpose register 15 is to be loaded with the address of the next machine instruction. This, of course, is done at execution time by the object machine. The USING instruction, which is an assembler instruction and takes no space in the object program, informs the assembler that general purpose register 15 will contain the address of the next machine instruction. Register 15 becomes the base register for this program. The BALR is a two-byte instruction, so the next instruction, the Load, is placed at 102. This number, printed to the left of the USING, indicates what the assembler assumed would be the contents of base register 15.

Let us now 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 $F_{16} = 15_{10}$, and the displacement is 022_{16}. With base register 15 containing 102 and with a displacement of 22, we get an actual address of 124_{16}. Looking down the listing we see that 124 is in fact the absolute address corresponding to the symbol DATA, as it should be.

The Add instruction is similar. With base register 15 again automatically designated, we have a base address of 102 and a displacement of 2A for an effective address of 12C, which is the absolute equivalent 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 of 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



```
                                                 START 256
000100    05 F0                          BEGIN   BALR  15,0
                           000102                USING *,15
000102    58 20 F 022                            L     2,DATA          LOAD REGISTER 2
000106    5A 20 F 02A                            A     2,TEN           ADD 10
                                         * THE FOLLOWING SHIFT HAS THE EFFECT OF MULTIPLYING BY 2
00010A    8B 20 0 001                            SLA   2,1
00010E    5B 20 F 026                            S     2,DATA+4        NOTE RELATIVE ADDRESSING
000112    50 20 F 02E                            ST    2,RESULT
000116    58 60 F 032                            L     6,BIN1
00011A    5A 60 F 036                            A     6,BIN2
00011E    4E 60 F 03E                            CVD   6,DEC           CONVERT TO DECIMAL
000122    0A 00                                  SVC   0
000124    00000019                       DATA    DC    F'25'
000128    0000000F                               DC    F'15'
00012C    0000000A                       TEN     DC    F'10'
000130                                   RESULT  DS    F
000134    0000000C                       BIN1    DC    F'12'
000138    0000004E                       BIN2    DC    F'78'
000140                                   DEC     DS    D
                                                 END   BEGIN
```

Figure 56. Assembly listing of a program to illustrate base register assignment and displacement computation. The "processing" performed is not intended to be realistic.

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.

Actually, there is a third "time" that should be considered: loading time of the object program. We said with our START instruction that the first byte of the program should be placed in $256_{10} = 100_{16}$. Everything said so far has assumed that the program is actually loaded starting in 100_{16}. But what if it were not? Suppose we were to decide after assembling the program that in order to avoid conflict with other programs the program should be loaded into 1200_{16}. With a suitable control card we inform the relocatable loader that the first byte of the assembled object program is to go into 1200. What would have to be changed in order to make the object program operate correctly from the new location?

The answer is that *nothing* need be changed. When the program has been loaded we begin by executing the BALR instruction. Now, what is the address of the next instruction after the BALR? Answer: 1202. This value goes into register 15 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 $1202 + 22 = 1224$. With the new starting location, 1224 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.

A complete relocation of the program after assembly is thus a simple matter of changing a control card in loading. In more complex program structures the loader 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.

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

It is possible, of course, to circumvent the system partially, by designating base register zero, in which case no base register is used at all. The program is then restricted to the use of addresses that can be contained in the displacement portion of the instructions, namely zero through 4095_{10}. Although there may conceivably be circumstances that justify this attack, it must be strongly discouraged as a general practice. The lower parts of storage of course have special functions (in connection with the PSW's) that cannot be disturbed. Even if these locations were avoided, however, it is inadvisable to discard the relocation feature.

The techniques of program relocation are heavily involved in a discussion of subroutines and subprograms. For the remainder of this presentation we shall assume that each program is loaded where indicated by the START.

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}. This means that an effective address cannot be less than the base register contents, nor more than 4095 greater (assuming no indexing, of course). If a program must reference a range of addresses greater than 4095, either the base register contents must be changed or more than one base register must be used. For routine programming, the latter solution is much more common.

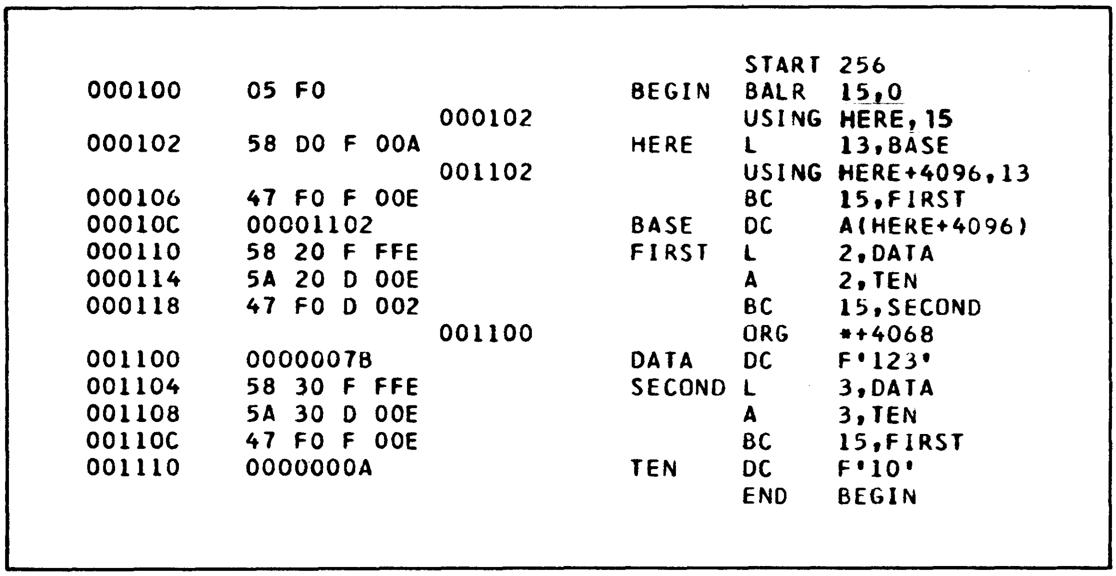
It should be noted, however, that it takes a rather large program segment to exhaust the range of displacements using one base register. With average length instructions, it takes a full pad of coding paper to use up 4096 bytes. It will usually be desirable to break a program this large into smaller 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. It may fairly frequently be advantageous to assign one base register to the program and another to data. This is done in the example in the last section of this presentation.

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 naturally 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 program shown in Figure 57 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 15 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 15, and permits us to refer to the contents of 15 in terms of a symbol, which we shall need for loading register 13. (The choice of register 13 was arbitrary.)

In loading the second base register, we cannot use a BALR: we want register 13 to contain not the address of the next instruction, but 4096 more than whatever went into 15. 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: 1102 is 1000 hexadecimal greater than the value of the symbol HERE, and $1000_{16} = 4096_{10}$.

Base register 13 will thus be loaded with 1102_{16} at



```
                                                   START 256
000100    05 F0                           BEGIN    BALR  15,0
                         000102                    USING HERE,15
000102    58 D0 F 00A                     HERE     L     13,BASE
                         001102                    USING HERE+4096,13
000106    47 F0 F 00E                              BC    15,FIRST
00010C    00001102                        BASE     DC    A(HERE+4096)
000110    58 20 F FFE                     FIRST    L     2,DATA
000114    5A 20 D 00E                              A     2,TEN
000118    47 F0 D 002                              BC    15,SECOND
                         001100                    ORG   *+4068
001100    0000007B                        DATA     DC    F'123'
001104    58 30 F FFE                     SECOND   L     3,DATA
001108    5A 30 D 00E                              A     3,TEN
00110C    47 F0 F 00E                              BC    15,FIRST
001110    0000000A                        TEN      DC    F'10'
                                                   END   BEGIN
```

Figure 57. Assembly listing of an illustrative program that has an Origin assembler instruction to make the program appear to have more than 4096 bytes, thus requiring two base registers

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 13: we see that the base register is 15 (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 13 is being loaded, the only base register available is 15; the effective address of the instruction that loads 13 therefore cannot be more than 4096 greater than the contents of 15; 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.

(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 15, which means that the following instructions and data are referenced by base register 13.

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

The Branch on Condition to FIRST involves a location under the control of base register 15; if base register 13 were specified, the displacement would have to be negative. The Load at FIRST refers to DATA. The base is 15, 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 15 cannot be used. We see that 13 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 13, so register 13 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. We shall see later an instance in which this rule for choosing base registers is important.

Separate Base Registers for Instructions and Data

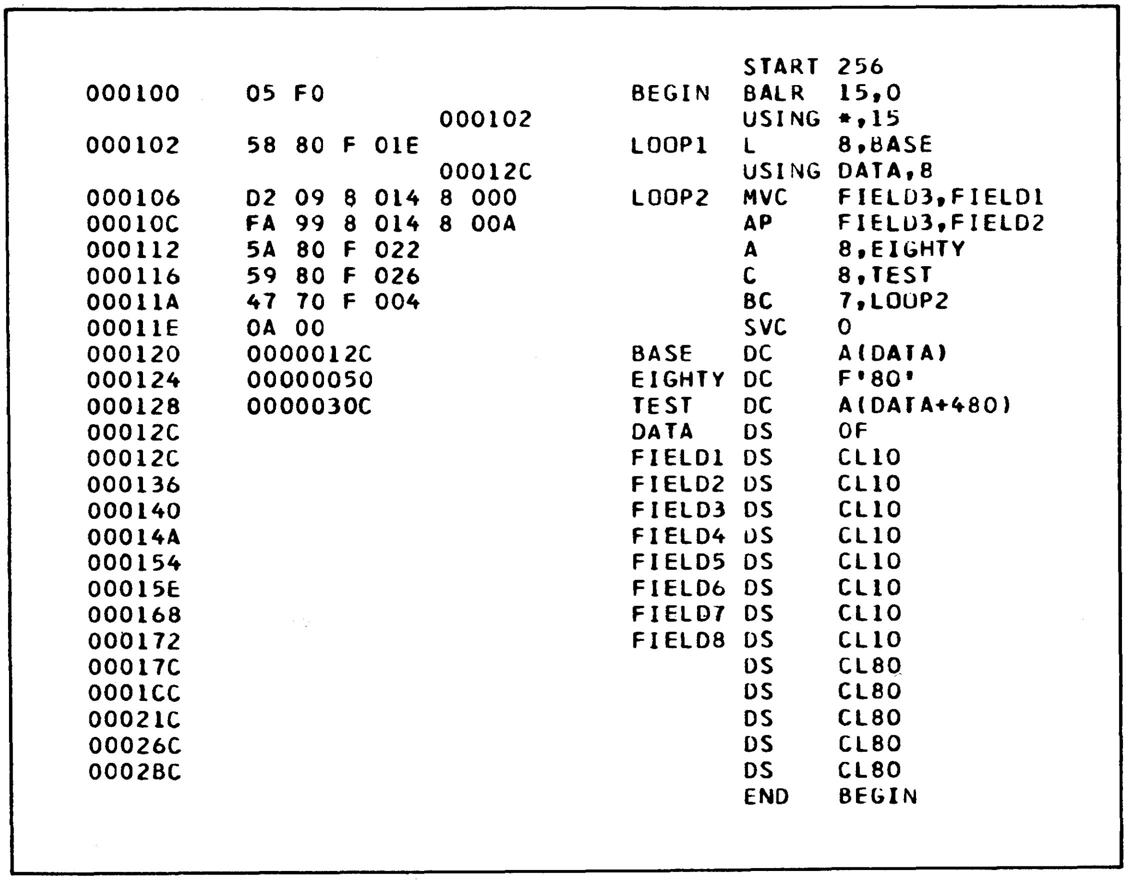
We have suggested that it will be rare for a program segment to be so long as to require more than one base register. On the other hand, it may be fairly common to want separate base registers for instructions and data, even though the instructions take far fewer than 4096 bytes. How this can happen 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 are not indexable. 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 so as 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 58. The loading of base registers is much as it was in Figure 57, except



```
                                                START 256
000100   05 F0                          BEGIN   BALR  15,0
                              000102            USING *,15
000102   58 80 F 01E                    LOOP1   L     8,BASE
                              00012C            USING DATA,8
000106   D2 09 8 014 8 000              LOOP2   MVC   FIELD3,FIELD1
00010C   FA 99 8 014 8 00A                      AP    FIELD3,FIELD2
000112   5A 80 F 022                            A     8,EIGHTY
000116   59 80 F 026                            C     8,TEST
00011A   47 70 F 004                            BC    7,LOOP2
00011E   0A 00                                  SVC   0
000120   0000012C                       BASE    DC    A(DATA)
000124   00000050                       EIGHTY  DC    F'80'
000128   0000030C                       TEST    DC    A(DATA+480)
00012C                                  DATA    DS    0F
00012C                                  FIELD1  DS    CL10
000136                                  FIELD2  DS    CL10
000140                                  FIELD3  DS    CL10
00014A                                  FIELD4  DS    CL10
000154                                  FIELD5  DS    CL10
00015E                                  FIELD6  DS    CL10
000168                                  FIELD7  DS    CL10
000172                                  FIELD8  DS    CL10
00017C                                          DS    CL80
0001CC                                          DS    CL80
00021C                                          DS    CL80
00026C                                          DS    CL80
0002BC                                          DS    CL80
                                                END   BEGIN
```

Figure 58. Assembly listing of a program that has separate base registers for program and data, with the base register for data being used for looping

that this time register 8 is loaded with the address corresponding to DATA, rather than with 4096 more than what 15 contained. As a matter of fact, it turns out that 15 contains 102_{16}, and 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 15, 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 used later by some other. We therefore have provided 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 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 have been to use DATA as the name of the first field, DATA+10 for the second, DATA+20 for the third, etc. The loss of meaningful names is a disadvantage. Another alternative would have been to omit the entry for DATA and use FIELD1 wherever DATA appears earlier. This would also be a little less meaningful, perhaps.

The Move Characters instruction at LOOP2 moves the first field to the third field location. Reading across the assembled instruction, we have: the actual operation code 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; 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 FIELD1, 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 Condition here is in effect a "Branch if Unequal". If the Branch is not executed, we are finished and the next instruction is a Supervisor Call.

If the program had been written to use only one base register, we would be in trouble with the address of the Branch on Condition. The assembler would have assumed a certain value for the base register and computed 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 assembler would be there. What we told the assembler correctly led to the first record processed; by the time we changed the contents (during execution) the assembler is no longer 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 readily done simply by re-executing the Load instruction at LOOP1.

If this program were ever relocated, 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 to change the initial program loading location, without notifying the address constants of the change. This matter is properly handled by an automatic flagging of all address constants in the deck or tape produced by the assembler, and by suitable modifications performed by the relocatable loader.

In order to illustrate one last facet, suppose that there were some compelling reason to place additional instructions *after* DATA. (It is assumed that there would be 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 15 or 8 could supply a displacement of acceptable size; the assembler could pick the one leading to the smaller displacement: 8. But the contents of 8 change as the loop is executed; how can we tell the assembler that 15 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 15, so it is the one used, as desired.

Summary

Base registers, when used effectively, provide some of the most powerful and flexible processing features of the System/360. From addressing compatibility between systems, to program relocation, to indexing in SS format instructions – these are only some of the ways in which base registers add to the power of the system. In other chapters in this text we shall see how they come into play in program segmentation and in double indexing through tables, to mention only two others.

With a thorough understanding of what is done at each of the three "times" – assembly, loading, and execution – the inventive programmer will find many other ways to make good use of base registers.

Questions and Exercises

Consider the following program. Note that some of the program statements have been omitted from the listing. 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 when answering questions 1-4, which refer to this program.

	Value Assumed by Assembler	Value Loaded at Execution Time	
		Program loaded at 200$_{16}$	Program loaded at 1200$_{16}$
Reg 15			

Location (relocated)	Location	Object Instruction		Execution Time Effective Address				
		Base Register	Displace- ment	Program Loaded at 200$_{16}$	Program Loaded at 1200$_{16}$			
							START	512
1200	200					BEGIN	BALR	15,0
							USING	*,15
	202						L	2,DATA
	206						A	2,TEN
	⋮						⋮	
	234						S	2,DATA+4
	238						ST	2,RESULT
	⋮						⋮	
	252						L	6,BIN1
	⋮						⋮	
	304					DATA	DC	F'25'
	308						DC	F'15'
	⋮						⋮	
	324					TEN	DC	F'10'
	328					RESULT	DS	F
	⋮						⋮	
	344					BIN1	DC	F'12'
							⋮	
							END	BEGIN

Symbol Table

	Location	Length
BEGIN	200	2
BIN1	344	4
DATA	304	4
RESULT	328	4
TEN	324	4

1a. What instruction informs the assembler that register 15 is to be used as a base register, and tells the assembler what value it must assume to be in that base register at execution time?

b. What instruction causes register 15 to be loaded with the base address at execution time?

2. The assembly process can be explained in terms of two phases. In the first phase, the assembler, among other functions, determines the length and location for each instruction, area, and constant. While doing this it constructs a symbol table (note the symbol table at the bottom of the assembly listing). The symbol table consists of one entry for each symbol appearing in the name field of the source program. Each entry contains the symbol, the storage address assigned to it, and the length (in bytes) of the storage area associated with it.

In the second phase, the assembler again processes the source program, developing for each symbolic operand the base register and displacement that will appear in the object instruction. To develop the base register specification and displacement, the assembler builds and uses a base register table, containing one entry for each USING instruction. Each entry notes the base register and the value the assembler assumes will be loaded into that base register at execution time (both of which are specified by the USING instruction).

a. In the space provided in the assembly listing, write the value the assembler assumes to be in base register 15 at execution time.

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

3. Assuming the program is not relocated:

a. In the space provided, write the value placed in register 15 at execution time.

b. Using the specified base register and displacement, write (in the space provided) the effective address developed at execution time for each encircled operand.

4. Assume that the program, when loaded, is relocated starting at 1200_{16} instead of 200_{16}. In the spaces provided, list:

a. The locations into which each instruction, area, and constant is loaded.

b. The value placed in register 15 at execution time.

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

5. Consider the following program. As in the previous program, some of the program statements have been omitted, the locations are listed in hexadecimal, and the locations are such that you should have no difficulty with hexadecimal arithmetic. In the spaces provided, list:

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

b. The contents of base registers 13, 14, and 15 assumed by the assembler.

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

d. The value actually placed in registers 13, 14, and 15 at execution time (assuming the program is not relocated).

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

	Value Assumed by Assembler	Value Loaded at Execution Time	
		Program loaded at 1000_{16}	
Reg 15			
Reg 14			
Reg 13			

	Location	Object Instruction		Execution Time Effective Address				
		Base Register	Displacement	Program Loaded at 1000_{16}				
							START	4096
	1000					BEGIN	BALR	15,0
							USING	FIRST,15
	1002					FIRST	BC	15,SKIP
	1006					DATA	DC	F'3472'
	⋮						⋮	
	1024					BASE1	DC	A(FIRST+4096)
	1028					BASE2	DC	A(FIRST+8192)
	⋮						⋮	
	1104					SKIP	L	14,BASE1
							USING	FIRST+4096, 14
	1108						L	13,BASE2
							USING	FIRST+8192, 13
	⋮						⋮	
	2504						BC	15,CK8
	⋮						⋮	
	2898					LOOP	A	4,DATA
	⋮						⋮	
	3204					LOOPB	S	5,DATA
	⋮						⋮	
	3508						BC	8,LOOP
	⋮						⋮	
	3904					CK8	BC	8,LOOPB
							END	BEGIN

Symbol Table

Symbol	Location

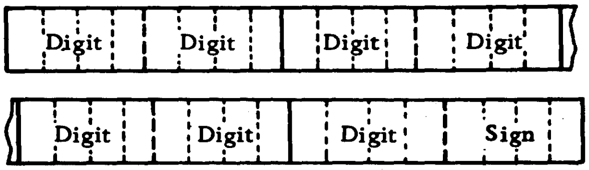
Chapter 6: Decimal Operations

The decimal instruction set is an optional feature of the 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 (preparing data for printing by the insertion of characters such as , . $). The decimal instruction set permits operations on variable length data and includes the following instructions:

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

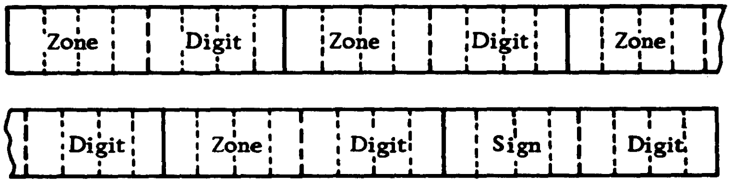
Data operated upon by instructions in the decimal set must be in one of two forms, *packed* or *zoned,* depending on the instruction. As a rough generalization, we can say that the packed format is required for arithmetic and the zoned for input/output.

In the packed format, two decimal digits are placed in each byte except the rightmost, 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. 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.*



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 the zone bits is 1111.

Decimal instructions have precise requirements that operands be in packed or zoned format. The Pack and Unpack instructions, standard instructions on 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 between binary and packed 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:

Op Code L_1 L_2 B_1 D_1 B_2 D_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 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 instruction. 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 instruction (one less than whatever we write) is the function of the assembler.

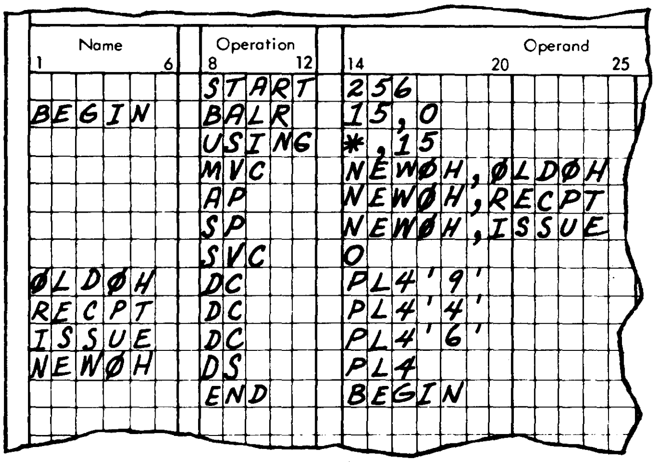
With these preliminaries in mind, let us turn to an example.

*These are the EBCDIC codes, which we shall use throughout. See the SRL manuals for a discussion of ASCII codes.

Addition and Subtraction in Decimal

Let us take the first example used in the chapter on "Fixed-Point Operations" 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 59 let us look first at the data definitions.


Name
1
6
Operation
8
12
Operand
14
20
25

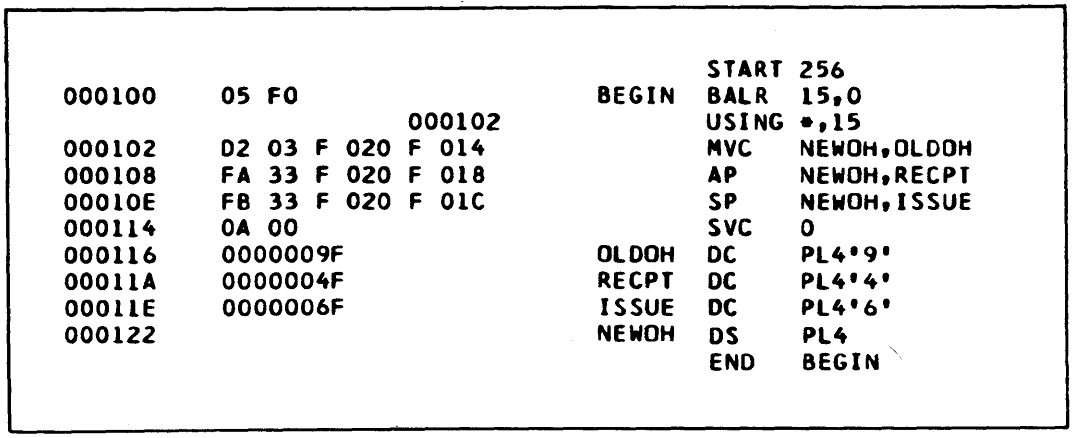
```
       START 256
BEGIN  BALR  15,0
       USING *,15
       MVC   NEWØH,ØLDØH
       AP    NEWØH,RECPT
       SP    NEWØH,ISSUE
       SVC   0
ØLDØH  DC    PL4'9'
RECPT  DC    PL4'4'
ISSUE  DC    PL4'6'
NEWØH  DS    PL4
       END   BEGIN
```

Figure 59. An assembler language program to perform a simple arithmetic calculation in decimal, using the System/360 decimal instruction set

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 actually permitted for other types of data, too, although we have had no previous occasion to use them.)

Looking at the assembly listing in Figure 60, 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 F (binary 1111), which is what the assembler used for a plus in this case.

Turning back to the instructions of the program, we see that the START, BALR, and USING are standard. 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*



```
                                              START 256
000100   05 F0                         BEGIN  BALR  15,0
                        000102                USING *,15
000102   D2 03 F 020 F 014                    MVC   NEWOH,OLDOH
000108   FA 33 F 020 F 018                    AP    NEWOH,RECPT
00010E   FB 33 F 020 F 01C                    SP    NEWOH,ISSUE
000114   0A 00                                SVC   0
000116   0000009F                      OLDOH  DC    PL4'9'
00011A   0000004F                      RECPT  DC    PL4'4'
00011E   0000006F                      ISSUE  DC    PL4'6'
000122                                 NEWOH  DS    PL4
                                              END   BEGIN
```

Figure 60. Assembly listing of the program of Figure 59

from the data definitions. It is also possible, and frequently necessary, to write *explicit* lengths to override what the assembler would imply.

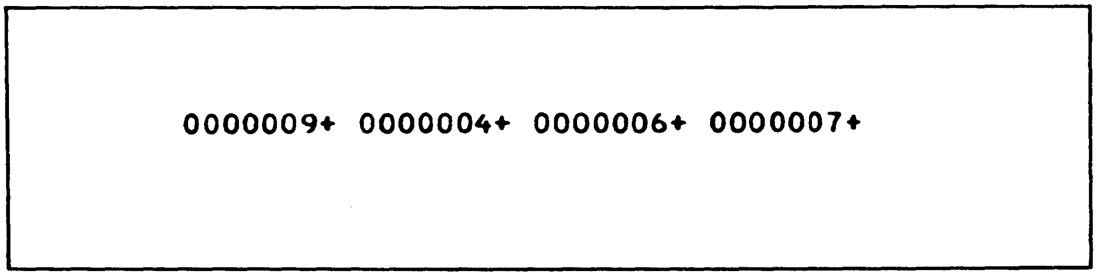
The generation of an address from the base register contents and the displacement is as before: for instance, for OLDOH the base register contains 102, the displacement is 014; the sum of these is 116, which we see is the address for OLDOH.

The purpose of the Move Characters instruction was 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 references to the general registers (except, of course, to get the base), so we must provide storage locations for all operations. 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 storage dump of Figure 61 shows that the result has been correctly computed.



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

Figure 61. Output of the program of Figures 59 and 60. The four quantities are 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 62.

The decimal multiply 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 00000256367+, 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.

Always bear in mind that the rightmost byte of an operand in decimal arithmetic must have a sign. We might be tempted, for instance, to set up a constant consisting of a 5 without a zero, and add directly into the position where we want to get the rounding. This would be illegal.

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

```
                                            START 256
000100    05 F0                       BEGIN BALR  15,0
                      000102                USING *,15
                                      *
                                      *   THE NUMBERS IN THE COMMENTS FIELD ARE THE CONTENTS
                                      *   OF PROD AFTER THE EXECUTION OF EACH INSTRUCTION
                                      *   THE C IS A PLUS SIGN IN THE PACKED FORMAT
                                      *
000102    F8 53 F 026 F 020                 ZAP   PROD,PRINC           00 00 00 02 48 9C
000108    FC 51 F 026 F 024                 MP    PROD,INT             00 00 02 56 36 7C
00010E    FA 51 F 026 F 02C                 AP    PROD,ROUND           00 00 02 56 41 7C
000114    D1 00 F 02A F 02B                 MVN   PROD+4(1),PROD+5     00 00 02 56 4C 7C
                                      *   THIS IS NOW THE CONTENTS OF PRINC
00011A    D2 03 F 020 F 027                 MVC   PRINC,PROD+1         00 02 56 4C
000120    0A 00                             SVC   0
000122    0002489F                    PRINC DC    PL4'2489'
000126    103F                        INT   DC    PL2'103'
000128                                PROD  DS    PL6
00012E    050F                        ROUND DC    PL2'50'
                                            END   BEGIN
```

Figure 62. Assembly listing of a program involving a decimal multiplication

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: 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 and systems analyst.) The rightmost byte of PROD contains a digit and sign that we now wish to drop, since they are to the right of the product after rounding. 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.

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; the equivalent of shifting, considered later in "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 were integers, that is, 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 had 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 had one and the divisor had 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 had been 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:

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 as desired to give the necessary number of decimal places.

Shifting of Decimal Fields

Shifting *as such* is not provided in the System/360 decimal operations. As in other variable-field-length computers with which the reader may be familiar, 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 simply the price we 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.

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 one character. Finally, we move one byte of the 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. The relevant statement from the Principles of Operation Manual (A22-6821) is: "The first and second operand fields may overlap when the rightmost byte of the first operand is coincident with or to the right of the rightmost byte 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 shifted to the left by four bit positions in moving it to 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.)

A shift to the left presents slightly different problems. Suppose that we have a source field of three bytes this time 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 us) be implied 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 write it in hexadecimal, X'FO'. 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+2,240	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, with the And Immediate 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:

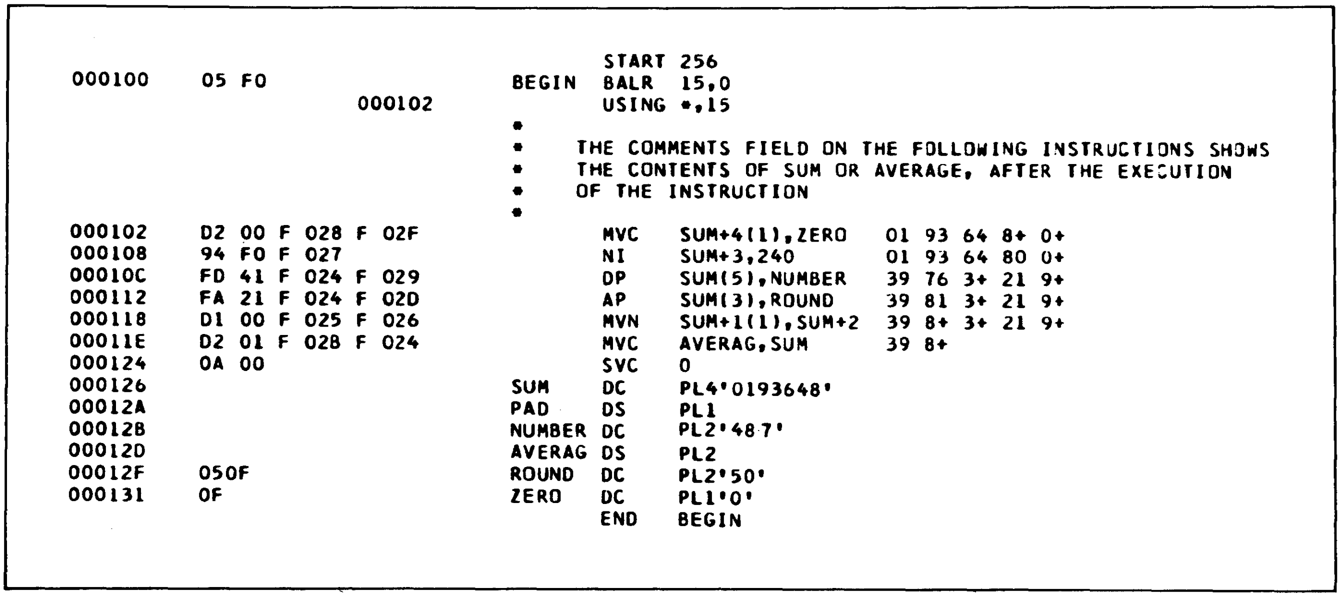
XX XX XX X0 0+

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 63. 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



```
                                          START 256
000100    05 F0                    BEGIN  BALR  15,0
                      000102              USING *,15
                                   *
                                   *    THE COMMENTS FIELD ON THE FOLLOWING INSTRUCTIONS SHOWS
                                   *    THE CONTENTS OF SUM OR AVERAGE, AFTER THE EXECUTION
                                   *    OF THE INSTRUCTION
                                   *
000102    D2 00 F 028 F 02F               MVC   SUM+4(1),ZERO     01 93 64 8+ 0+
000108    94 F0 F 027                     NI    SUM+3,240         01 93 64 80 0+
00010C    FD 41 F 024 F 029               DP    SUM(5),NUMBER     39 76 3+ 21 9+
000112    FA 21 F 024 F 02D               AP    SUM(3),ROUND      39 81 3+ 21 9+
000118    D1 00 F 025 F 026               MVN   SUM+1(1),SUM+2    39 8+ 3+ 21 9+
00011E    D2 01 F 02B F 024               MVC   AVERAG,SUM        39 8+
000124    0A 00                           SVC   0
000126                             SUM    DC    PL4'0193648'
00012A                             PAD    DS    PL1
00012B                             NUMBER DC    PL2'487'
00012D                             AVERAG DS    PL2
00012F    050F                     ROUND  DC    PL2'50'
000131    0F                       ZERO   DC    PL1'0'
                                          END   BEGIN
```

Figure 63. Assembly listing of a program involving decimal division and the equivalent of decimal shifting

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.

Format and Base Conversions

It is often necessary to convert from zoned to packed format and vice versa, and to convert between binary and decimal form. In this section we shall examine an illustrative problem that involves both types of conversion, and the special instructions available to make them relatively simple.

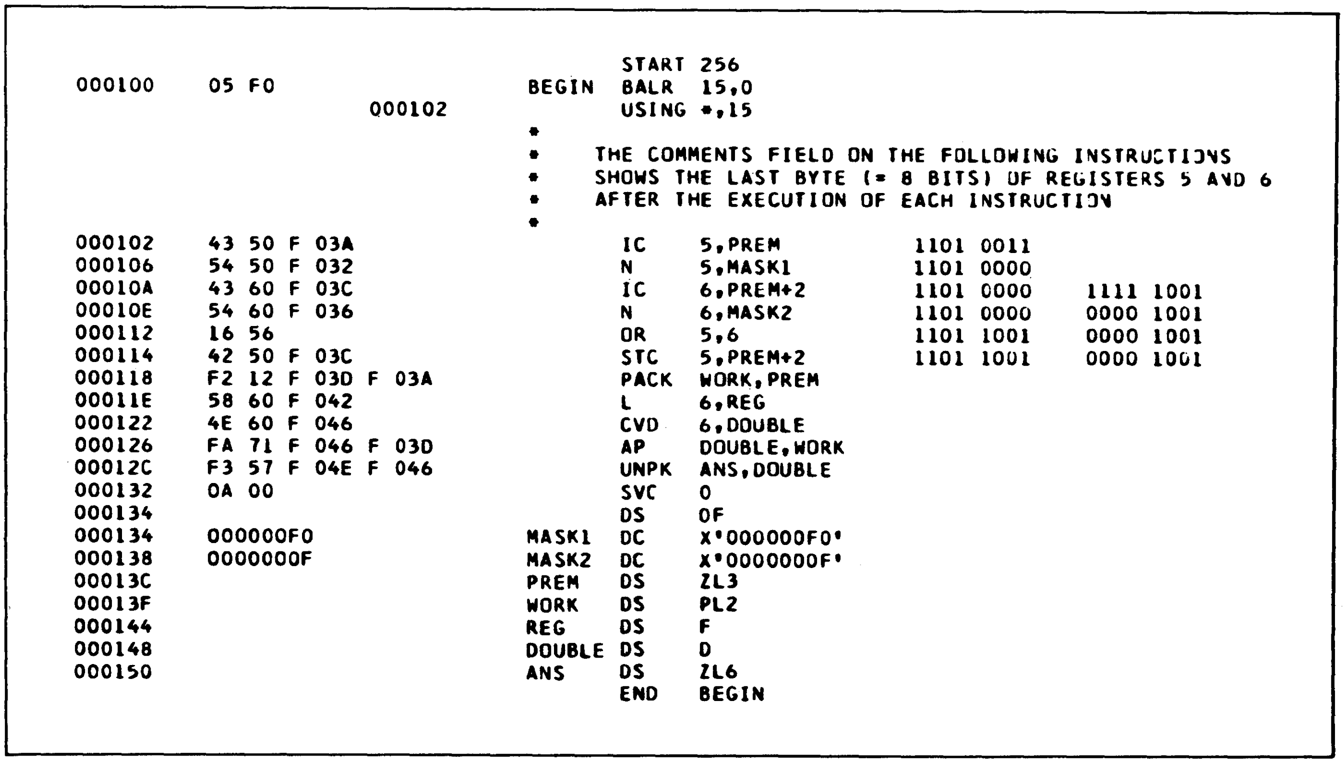
We are given a fullword named REG, in binary format. The three-byte field named PREM was read directly from a card on which the sign was in the high-order position, instead of the low-order. That is, a positive number was punched with a 12 zone over the leftmost digit, and a minus number was punched 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 64 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.



```
                                                  START 256
000100     05 F0                           BEGIN  BALR  15,0
                          000102                  USING *,15
                                           *
                                           *    THE COMMENTS FIELD ON THE FOLLOWING INSTRUCTIONS
                                           *    SHOWS THE LAST BYTE (= 8 BITS) OF REGISTERS 5 AND 6
                                           *    AFTER THE EXECUTION OF EACH INSTRUCTION
                                           *
000102     43 50 F 03A                            IC    5,PREM         1101 0011
000106     54 50 F 032                            N     5,MASK1        1101 0000
00010A     43 60 F 03C                            IC    6,PREM+2       1101 0000    1111 1001
00010E     54 60 F 036                            N     6,MASK2        1101 0000    0000 1001
000112     16 56                                  OR    5,6            1101 1001    0000 1001
000114     42 50 F 03C                            STC   5,PREM+2       1101 1001    0000 1001
000118     F2 12 F 03D F 03A                      PACK  WORK,PREM
00011E     58 60 F 042                            L     6,REG
000122     4E 60 F 046                            CVD   6,DOUBLE
000126     FA 71 F 046 F 03D                      AP    DOUBLE,WORK
00012C     F3 57 F 04E F 046                      UNPK  ANS,DOUBLE
000132     0A 00                                  SVC   0
000134                                            DS    0F
000134     000000F0                        MASK1  DC    X'000000F0'
000138     0000000F                        MASK2  DC    X'0000000F'
00013C                                     PREM   DS    ZL3
00013F                                     WORK   DS    PL2
000144                                     REG    DS    F
000148                                     DOUBLE DS    D
000150                                     ANS    DS    ZL6
                                                  END   BEGIN
```

Figure 64. Assembly listing of a program that moves zone bits from one byte to another, converts a number to packed format, converts another number from binary to decimal, and does arithmetic in decimal

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 And (N) instructions, however, *do* require their operands to be on fullword boundaries. This is the purpose of the DS OF 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 an aligned on a doubleword.

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

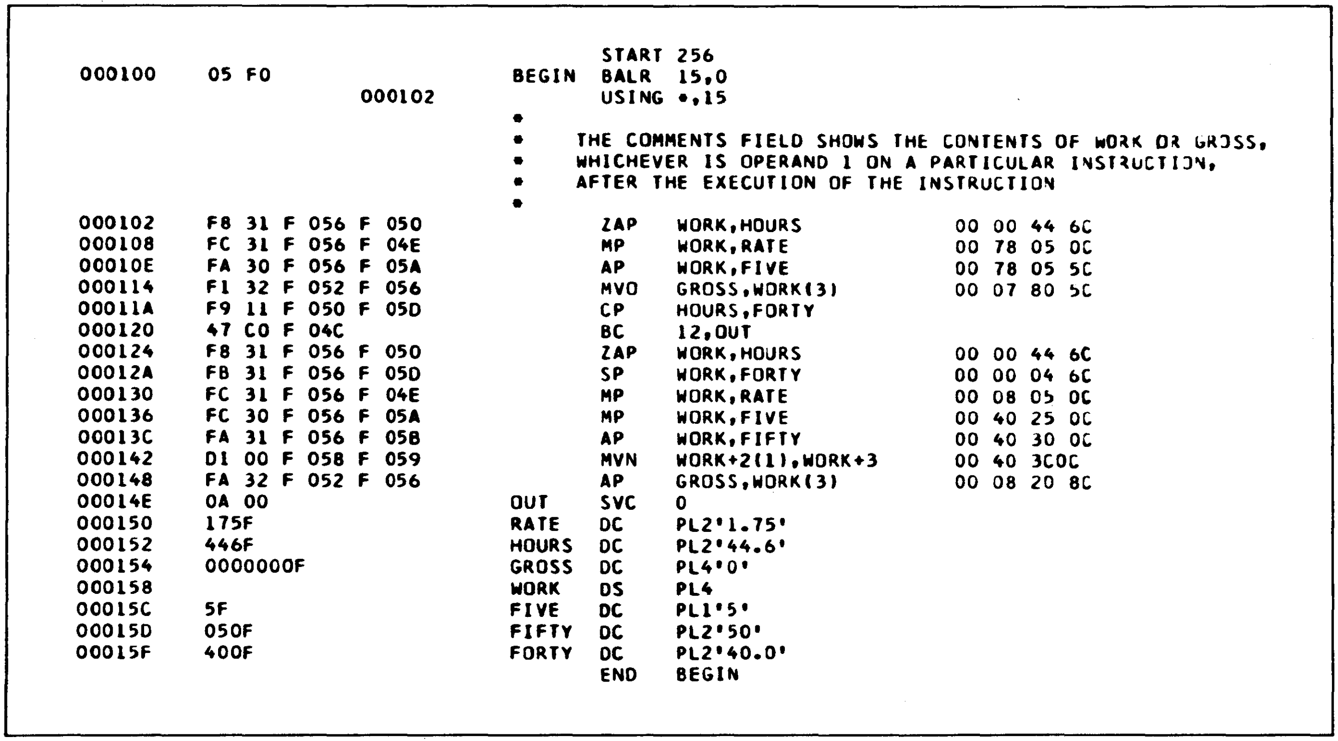
Logical tests and decisions are as necessary in decimal operations as elsewhere. The System/360 provides a Compare Decimal instruction, and the condition code is set as a result of this and three arithmetic instructions. In this section we shall explore an example that uses the Compare Decimal instruction.

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 in GROSS the total wages earned.

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 65 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 Con-



```
                                                  START 256
000100     05 F0                          BEGIN   BALR  15,0
                          000102                  USING *,15
                                          *
                                          *     THE COMMENTS FIELD SHOWS THE CONTENTS OF WORK OR GROSS,
                                          *     WHICHEVER IS OPERAND 1 ON A PARTICULAR INSTRUCTION,
                                          *     AFTER THE EXECUTION OF THE INSTRUCTION
                                          *
000102     F8 31 F 056 F 050                      ZAP    WORK,HOURS          00 00 44 6C
000108     FC 31 F 056 F 04E                      MP     WORK,RATE           00 78 05 0C
00010E     FA 30 F 056 F 05A                      AP     WORK,FIVE           00 78 05 5C
000114     F1 32 F 052 F 056                      MVO    GROSS,WORK(3)       00 07 80 5C
00011A     F9 11 F 050 F 05D                      CP     HOURS,FORTY
000120     47 C0 F 04C                            BC     12,OUT
000124     F8 31 F 056 F 050                      ZAP    WORK,HOURS          00 00 44 6C
00012A     FB 31 F 056 F 05D                      SP     WORK,FORTY          00 00 04 6C
000130     FC 31 F 056 F 04E                      MP     WORK,RATE           00 08 05 0C
000136     FC 30 F 056 F 05A                      MP     WORK,FIVE           00 40 25 0C
00013C     FA 31 F 056 F 05B                      AP     WORK,FIFTY          00 40 30 0C
000142     D1 00 F 058 F 059                      MVN    WORK+2(1),WORK+3    00 40 3C0C
000148     FA 32 F 052 F 056                      AP     GROSS,WORK(3)       00 08 20 8C
00014E     0A 00                          OUT     SVC    0
000150     175F                           RATE    DC     PL2'1.75'
000152     446F                           HOURS   DC     PL2'44.6'
000154     0000000F                       GROSS   DC     PL4'0'
000158                                    WORK    DS     PL4
00015C     5F                             FIVE    DC     PL1'5'
00015D     050F                           FIFTY   DC     PL2'50'
00015F     400F                           FORTY   DC     PL2'40.0'
                                                  END    BEGIN
```

Figure 65. Assembly listing of a program that computes a man's gross pay, including any overtime pay

dition 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. (In this example we do not show the continuation of the computation.)

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 there is sufficient space in WORK 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 Numerical 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.

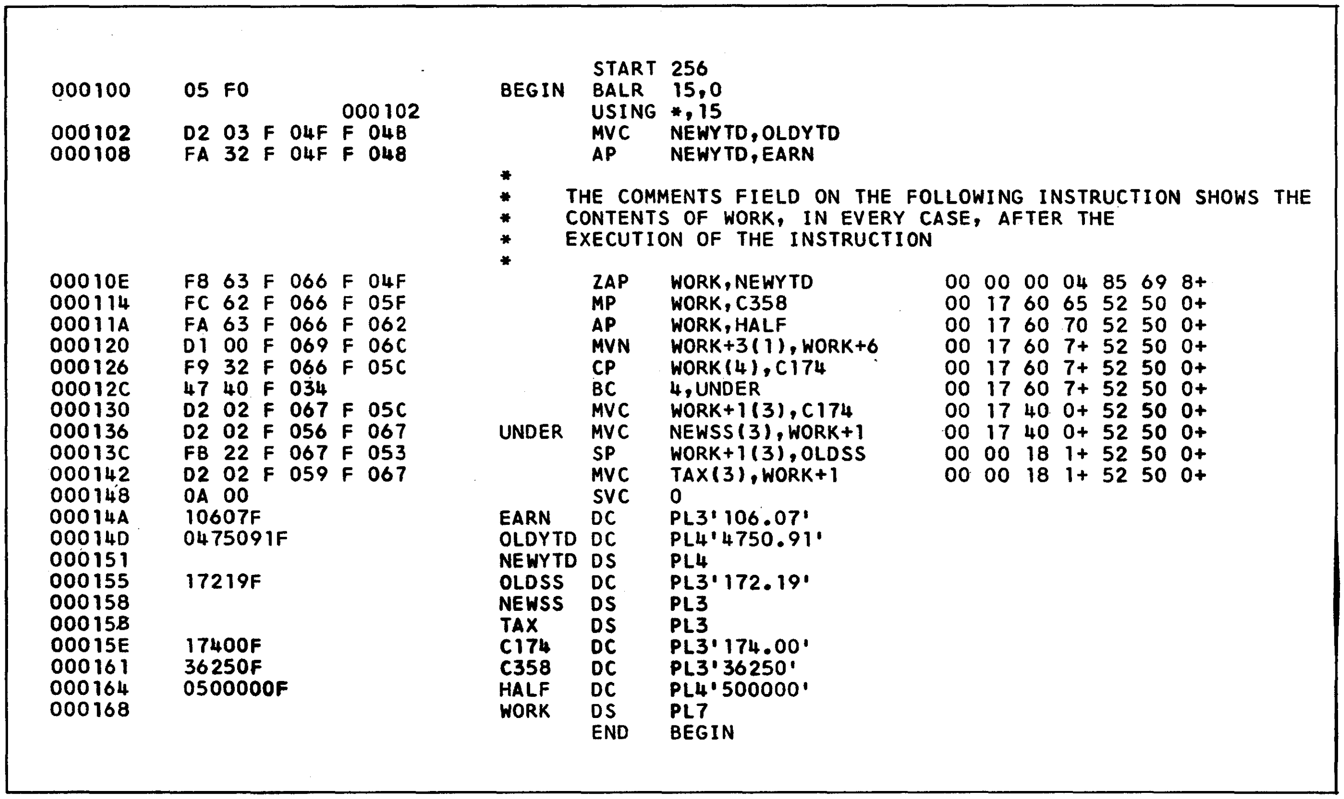
The Social Security Problem in Decimal

For a little further practice in applying decimal operations, we may rewrite the Social Security calculation of Figure 51 in the chapter on "Fixed-Point Operations". The logic of the decimal program shown in Figure 66 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 66 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 3⅝% 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 DC is 7, an explicit length must be given.

The remaining operations closely parallel the ones in the earlier version.



```
                                                 START 256
000100    05 F0                          BEGIN   BALR  15,0
                           000102                USING *,15
000102    D2 03 F 04F F 04B                      MVC   NEWYTD,OLDYTD
000108    FA 32 F 04F F 048                      AP    NEWYTD,EARN
                                         *
                                         *    THE COMMENTS FIELD ON THE FOLLOWING INSTRUCTION SHOWS THE
                                         *    CONTENTS OF WORK, IN EVERY CASE, AFTER THE
                                         *    EXECUTION OF THE INSTRUCTION
                                         *
00010E    F8 63 F 066 F 04F                      ZAP   WORK,NEWYTD          00 00 00 04 85 69 8+
000114    FC 62 F 066 F 05F                      MP    WORK,C358            00 17 60 65 52 50 0+
00011A    FA 63 F 066 F 062                      AP    WORK,HALF            00 17 60 70 52 50 0+
000120    D1 00 F 069 F 06C                      MVN   WORK+3(1),WORK+6     00 17 60 7+ 52 50 0+
000126    F9 32 F 066 F 05C                      CP    WORK(4),C174         00 17 60 7+ 52 50 0+
00012C    47 40 F 034                            BC    4,UNDER              00 17 60 7+ 52 50 0+
000130    D2 02 F 067 F 05C                      MVC   WORK+1(3),C174       00 17 40 0+ 52 50 0+
000136    D2 02 F 056 F 067              UNDER   MVC   NEWSS(3),WORK+1      00 17 40 0+ 52 50 0+
00013C    FB 22 F 067 F 053                      SP    WORK+1(3),OLDSS      00 00 18 1+ 52 50 0+
000142    D2 02 F 059 F 067                      MVC   TAX(3),WORK+1        00 00 18 1+ 52 50 0+
000148    0A 00                                  SVC   0
00014A    10607F                         EARN    DC    PL3'106.07'
00014D    0475091F                       OLDYTD  DC    PL4'4750.91'
000151                                   NEWYTD  DS    PL4
000155    17219F                         OLDSS   DC    PL3'172.19'
000158                                   NEWSS   DS    PL3
00015B                                   TAX     DS    PL3
00015E    17400F                         C174    DC    PL3'174.00'
000161    36250F                         C358    DC    PL3'36250'
000164    0500000F                       HALF    DC    PL4'500000'
000168                                   WORK    DS    PL7
                                                 END   BEGIN
```

Figure 66. Assembly listing of a program to compute Social Security tax in decimal

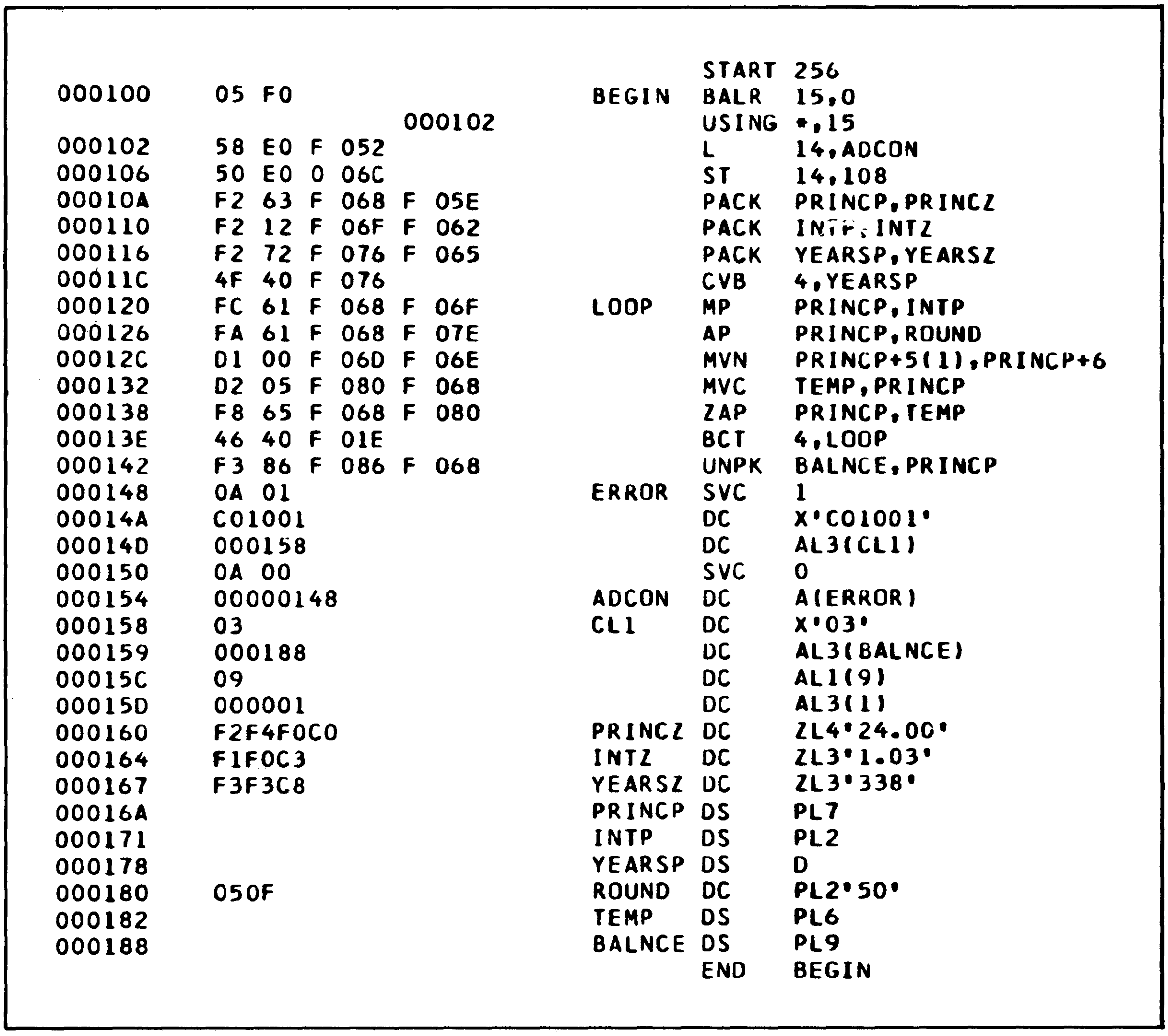
The "Indian" Problem

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 67 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, and using



```
                                                 START 256
000100    05 F0                           BEGIN  BALR  15,0
                         000102                  USING *,15
000102    58 E0 F 052                            L     14,ADCON
000106    50 E0 0 06C                            ST    14,108
00010A    F2 63 F 068 F 05E                      PACK  PRINCP,PRINCZ
000110    F2 12 F 06F F 062                      PACK  INTP,INTZ
000116    F2 72 F 076 F 065                      PACK  YEARSP,YEARSZ
00011C    4F 40 F 076                            CVB   4,YEARSP
000120    FC 61 F 068 F 06F               LOOP   MP    PRINCP,INTP
000126    FA 61 F 068 F 07E                      AP    PRINCP,ROUND
00012C    D1 00 F 06D F 06E                      MVN   PRINCP+5(1),PRINCP+6
000132    D2 05 F 080 F 068                      MVC   TEMP,PRINCP
000138    F8 65 F 068 F 080                      ZAP   PRINCP,TEMP
00013E    46 40 F 01E                            BCT   4,LOOP
000142    F3 86 F 086 F 068                      UNPK  BALNCE,PRINCP
000148    0A 01                           ERROR  SVC   1
00014A    C01001                                 DC    X'C01001'
00014D    000158                                 DC    AL3(CL1)
000150    0A 00                                  SVC   0
000154    00000148                        ADCON  DC    A(ERROR)
000158    03                              CL1    DC    X'03'
000159    000188                                 DC    AL3(BALNCE)
00015C    09                                     DC    AL1(9)
00015D    000001                                 DC    AL3(1)
000160    F2F4F0C0                        PRINCZ DC    ZL4'24.00'
000164    F1F0C3                          INTZ   DC    ZL3'1.03'
000167    F3F3C8                          YEARSZ DC    ZL3'338'
00016A                                    PRINCP DS    PL7
000171                                    INTP   DS    PL2
000178                                    YEARSP DS    D
000180    050F                            ROUND  DC    PL2'50'
000182                                    TEMP   DS    PL6
000188                                    BALNCE DS    PL9
                                                 END   BEGIN
```

Figure 67. Assembly listing of a program to compute compound interest (the "Indian" problem)

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 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) to 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 either some preliminary calculations as to the possible size, or a guess.

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 (A22-6821), 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 is $523,998.22.

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. The MP instruction specifies the location of ______________ ______________ in the first operand, and the location of the ______________ ______________ in the second operand.

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. 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 three factors:

QUAN – 4 whole numbers
TCOST – 6 whole numbers and 2 decimal places
AVCOST – 6 whole numbers and 2 decimal places

The problem is to divide QUAN into TCOST to develop a quotient AVCOST, which is not to be rounded.

a. How many decimal places must the dividend contain to develop a proper quotient?

b. What must be the minimum size (in bytes) of the area in which the dividend is located at the time the DP instruction is executed?

9. 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)

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

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

11. 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.

12a. 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.

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

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

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

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

17. 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.

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

19. 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.

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

21. Consider the following excerpts from an assembly listing. MASK is located at 136

```
        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 7: Logical Operations on Characters and Bits

This chapter discusses the subject of logical operations through the medium of several illustrative programs. These illustrative programs were designed to bring out various aspects of the use of logical operations, with the logic being the primary focus of the example. The reader will realize, of course, that, in practical applications, logic is one of many tools and techniques used in a complete program.

The first example demonstrates the logic involved in sorting three items into ascending sequence. Two sections show numerous examples of tests on combinations of bits in a byte and the setting of bit combinations. Another major example uses the computation of a check digit in a self-checking number to illustrate logical operations on a sequence of characters. A final example involves a series of bit and byte operations on input data fields.

Instructions emphasized in this chapter include the various types of comparisons, Insert Character, Store Character, Test Under Mask, the various forms of And and Or, and Branch on Condition.

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 on a binary basis. 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. (In ASCII coding, 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 the comparison instruction*; all we really need to know is that names will be correctly alphabetized and that digits are *consistently* ranked somewhere.

The word "character" in Compare Logical Character is meant to imply that the 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

*Called the machine's *collating sequence* 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 68. 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 SMALL, MEDIUM, and LARGE, where the results 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 fourth character from the leftmost character of the "record". 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

```
                                             START 256
00010C    05 F0                       BEGIN  BALR  15,0
                      0C0102                 USING *,15
000102    98 24 F 072                        LM    2,4,ADDRA        LOAD REGISTERS WITH ADDRESSES
000106    D5 04 2 004 3 004                  CLC   4(5,2),4(3)      COMPARE A AND B
00010C    47 C0 F 014                        BC    12,X             BRANCH IF A ALREADY LESS OR EQUAL
00011C    18 62                              LR    6,2              INTERCHANGE ADDRESSES OF A AND B
000112    18 23                              LR    2,3              X
000114    18 36                              LR    3,6              X
000116    D5 04 2 004 4 004           X      CLC   4(5,2),4(4),     COMPARE A AND C
00011C    47 C0 F 024                        BC    12,Y             BRANCH IF A ALREADY LESS OR EQUAL
000120    18 62                              LR    6,2              INTERCHANGE ADDRESSES OF A AND C
000122    18 24                              LR    2,4              X
000124    18 46                              LR    4,6              X
000126    D5 04 3 004 4 004           Y      CLC   4(5,3),4(4)      COMPARE B AND C
00012C    47 C0 F 034                        BC    12,MOVE          BRANCH IF B ALREADY LESS OR EQUAL
00013C    18 63                              LR    6,3              INTERCHANGE ADDRESSES OF B AND C
000132    18 34                              LR    3,4              X
000134    18 46                              LR    4,6              X
000136    D2 0C F 07E 2 0C0           MOVE   MVC   SMALL,0(2)       ADDRESS OF SMALLEST OF THREE IS NOW IN 2
00013C    D2 0C F 088 3 000                  MVC   MEDIUM,0(3)      LIKEWISE FOR MEDIUM, IN REGISTER 3
000142    D2 0C F 098 4 0C0                  MVC   LARGE,0(4)       LIKEWISE FOR LARGEST, IN REGISTER 4
000148    0A 00                              SVC   0                PROGRAM TERMINATION
00014A                                A      DS    CL13
000157                                B      DS    CL13
000164                                C      DS    CL13
000174    0000014A                    ADDRA  DC    A(A)
000178    00000157                    ADDRB  DC    A(B)
00017C    00000164                    ADDRC  DC    A(C)
000180                                SMALL  DS    CL13
00018C                                MEDIUM DS    CL13
00019A                                LARGE  DS    CL13
                                             END   BEGIN
```

Figure 68. A program to sort three 13-character items into ascending sequence on keys in the middle five characters of each item. The three items are in A, B, and C; the sorted items are placed in SMALL, MEDIUM, and LARGE.

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 guaranteed 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.

Figure 69 shows the contents of registers 2, 3, and 4 at four points in the program: at the beginning, at X, at Y, and at MOVE. The three original data items, in order, were 1111CCCCC1111, 2222BBBBB2222, and 3333AAAAA3333. In other words, the items were in reverse order according to their keys.

	Register 2	Register 3	Register 4
Before	0000014A	00000157	00000164
X	00000157	0000014A	00000164
Y	00000164	0000014A	00000157
MOVE	00000164	00000157	0000014A

Figure 69. The contents of registers 2, 3, and 4 at four points during the execution of the program of Figure 68. The original 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.

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, which is intended to be illustrative only. Suppose that 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 would mean a style with orange only; 1010 would describe 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,10
BC  5,YES
```

In the Test Under Mask instruction, the 10 is the mask, written here in decimal. 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 11 in decimal. 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,11
BC  4,YES
```

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,9
BC  1,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,3
BC  8,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 9 (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 was 00001000 after the And Immediate, there will be *all* zeros after the Exclusive Or Immediate. If, on the other hand, there was a zero in the position for red, there will now be a 1. And if there was a 1 in the position for orange, there will 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 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,9
XI   WORK,8
BC   8,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 instead of the four we had in the wallpaper situation, 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). (The bits of a character are numbered from zero to 7 from the left.) 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 (83 decimal). 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,83
```

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 (172 decimal). The instruction is therefore

```
NI   PATTRN,172
```

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,83
```

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.

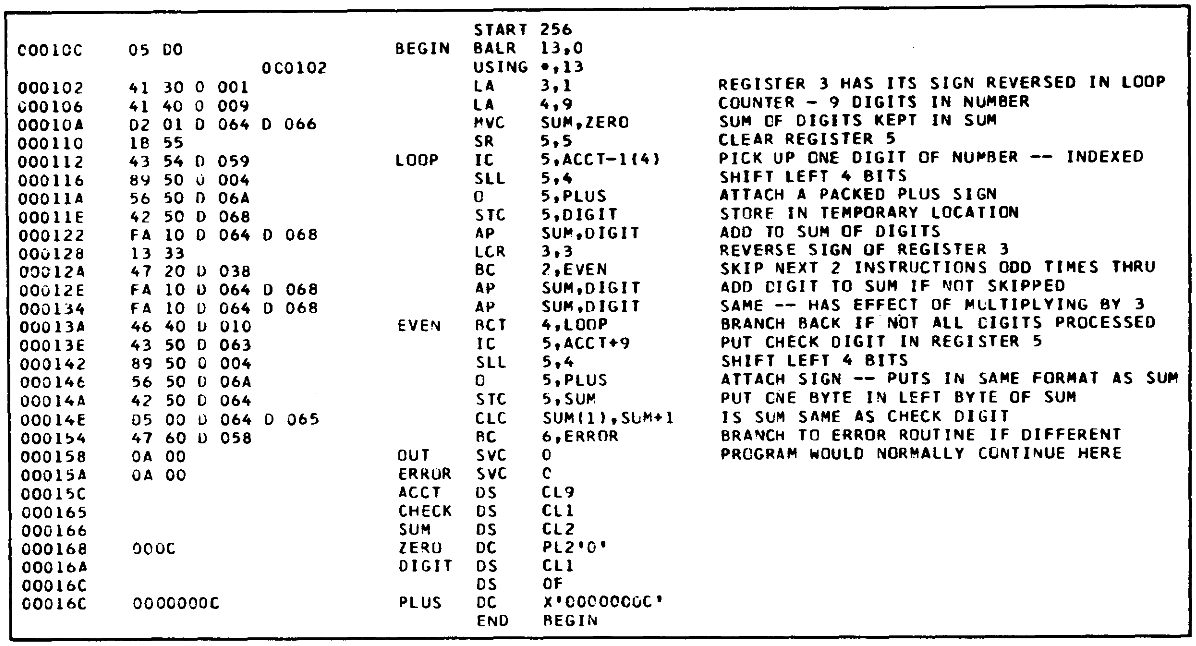
More important, perhaps, there is considerable protection against clerical error. If any one digit is miscopied, the erroneous account number will not pass the check. Furthermore, most transpositions of two adjacent digits will cause the check to fail. For instance, the check digit for 132456789 would 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 70 we begin by loading register 3 with 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 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.



```
                                            START 256
C0010C    05 D0                      BEGIN  BALR  13,0
                          0C0102            USING *,13
000102    41 30 0 001                       LA    3,1             REGISTER 3 HAS ITS SIGN REVERSED IN LOOP
000106    41 40 0 009                       LA    4,9             COUNTER - 9 DIGITS IN NUMBER
00010A    D2 01 D 064 D 066                 MVC   SUM,ZERO        SUM OF DIGITS KEPT IN SUM
000110    1B 55                             SR    5,5             CLEAR REGISTER 5
000112    43 54 D 059                LOOP   IC    5,ACCT-1(4)     PICK UP ONE DIGIT OF NUMBER -- INDEXED
000116    89 50 0 004                       SLL   5,4             SHIFT LEFT 4 BITS
00011A    56 50 D 06A                       O     5,PLUS          ATTACH A PACKED PLUS SIGN
00011E    42 50 D 068                       STC   5,DIGIT         STORE IN TEMPORARY LOCATION
000122    FA 10 D 064 D 068                 AP    SUM,DIGIT       ADD TO SUM OF DIGITS
000128    13 33                             LCR   3,3             REVERSE SIGN OF REGISTER 3
00012A    47 20 D 038                       BC    2,EVEN          SKIP NEXT 2 INSTRUCTIONS ODD TIMES THRU
00012E    FA 10 D 064 D 068                 AP    SUM,DIGIT       ADD DIGIT TO SUM IF NOT SKIPPED
000134    FA 10 D 064 D 068                 AP    SUM,DIGIT       SAME -- HAS EFFECT OF MULTIPLYING BY 3
00013A    46 40 D 010                EVEN   BCT   4,LOOP          BRANCH BACK IF NOT ALL DIGITS PROCESSED
00013E    43 50 D 063                       IC    5,ACCT+9        PUT CHECK DIGIT IN REGISTER 5
000142    89 50 0 004                       SLL   5,4             SHIFT LEFT 4 BITS
000146    56 50 D 06A                       O     5,PLUS          ATTACH SIGN -- PUTS IN SAME FORMAT AS SUM
00014A    42 50 D 064                       STC   5,SUM           PUT ONE BYTE IN LEFT BYTE OF SUM
00014E    D5 00 D 064 D 065                 CLC   SUM(1),SUM+1    IS SUM SAME AS CHECK DIGIT
000154    47 60 D 058                       BC    6,ERROR         BRANCH TO ERROR ROUTINE IF DIFFERENT
000158    0A 00                      OUT    SVC   0               PROGRAM WOULD NORMALLY CONTINUE HERE
00015A    0A 00                      ERROR  SVC   C
00015C                               ACCT   DS    CL9
000165                               CHECK  DS    CL1
000166                               SUM    DS    CL2
000168    000C                       ZERO   DC    PL2'0'
00016A                               DIGIT  DS    CL1
00016C                                      DS    0F
00016C    0000000C                   PLUS   DC    X'0000000C'
                                            END   BEGIN
```

Figure 70. A program to compute the check digit for a self-checking number, and compare the computed value with the check digit contained in the last position of the number

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, so we pick up the digits one at a time, from right to left, as stated.

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, 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 multipled 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 even-numbered digits 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 with a Supervisor Call, as well as the normal path. We will not attempt to indicate what steps might be taken to reject the record of which the invalid account number would have been a part.

There are, of course, many other techniques for computing check digits which give greater protection or make the check digit operations simpler. For a more complete discussion of this topic, see *Disk Storage Concepts* (F20-8161).

We turn now to a hypothetical example of the type of thing that is sometimes necessary in working with involved input formats.

We are given two numbers. 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 all seven characters contain valid digits, we simply go on to the next test; if any one contains numeric bits not valid for a digit, we shall go to a Supervisor Call.

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 a Supervisor Call.

Next, we start with an eight-byte field named COMB. We shall assume for the purposes here that the numeric portions 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 71 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 zones) 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.

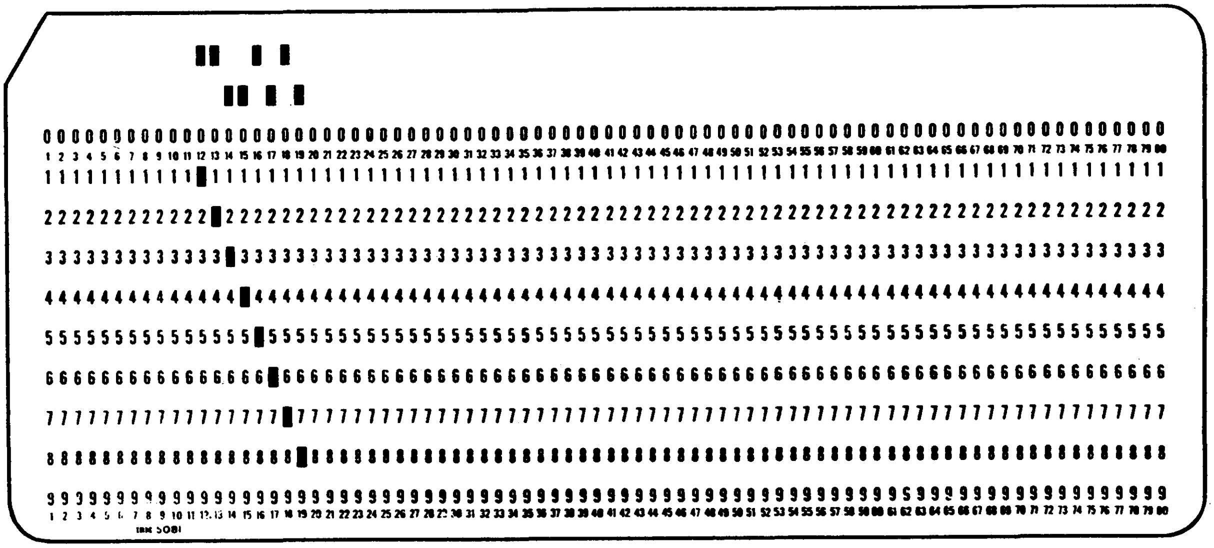


Figure 71. An illustrative card field for COMB, an area used in th e program in Figure 73

A flowchart showing the logic of this problem is shown in Figure 72.

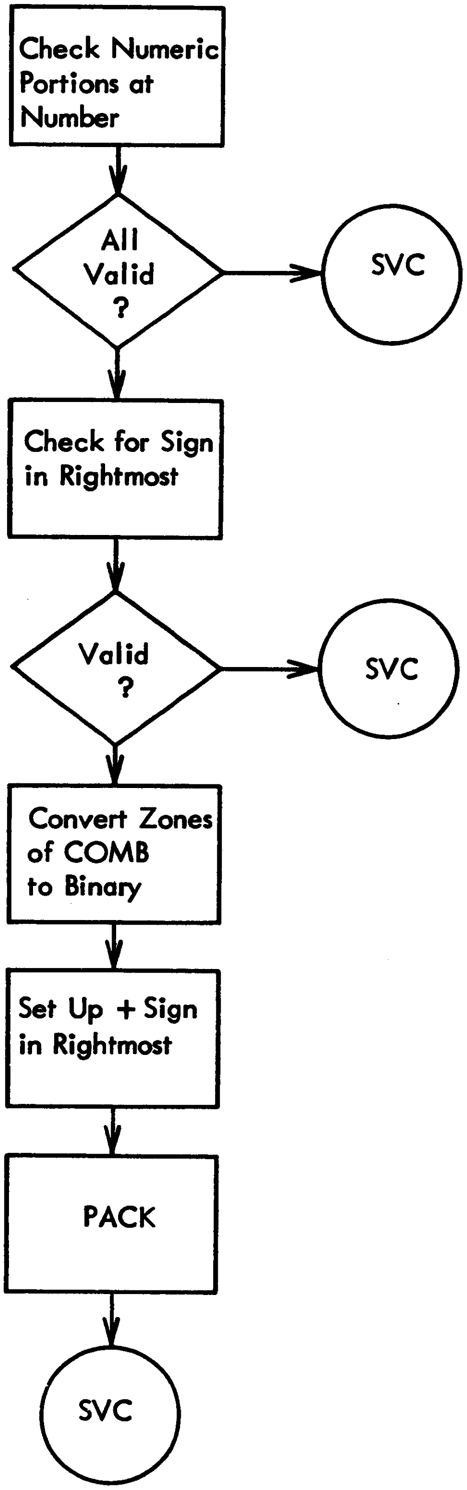

Check Numeric Portions at Number
All Valid ?
SVC
Check for Sign in Rightmost
Valid ?
SVC
Convert Zones of COMB to Binary
Set Up + Sign in Rightmost
PACK
SVC

Figure 72. A flowchart of the processing carried out by the program in Figure 73

The program in Figure 73 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.

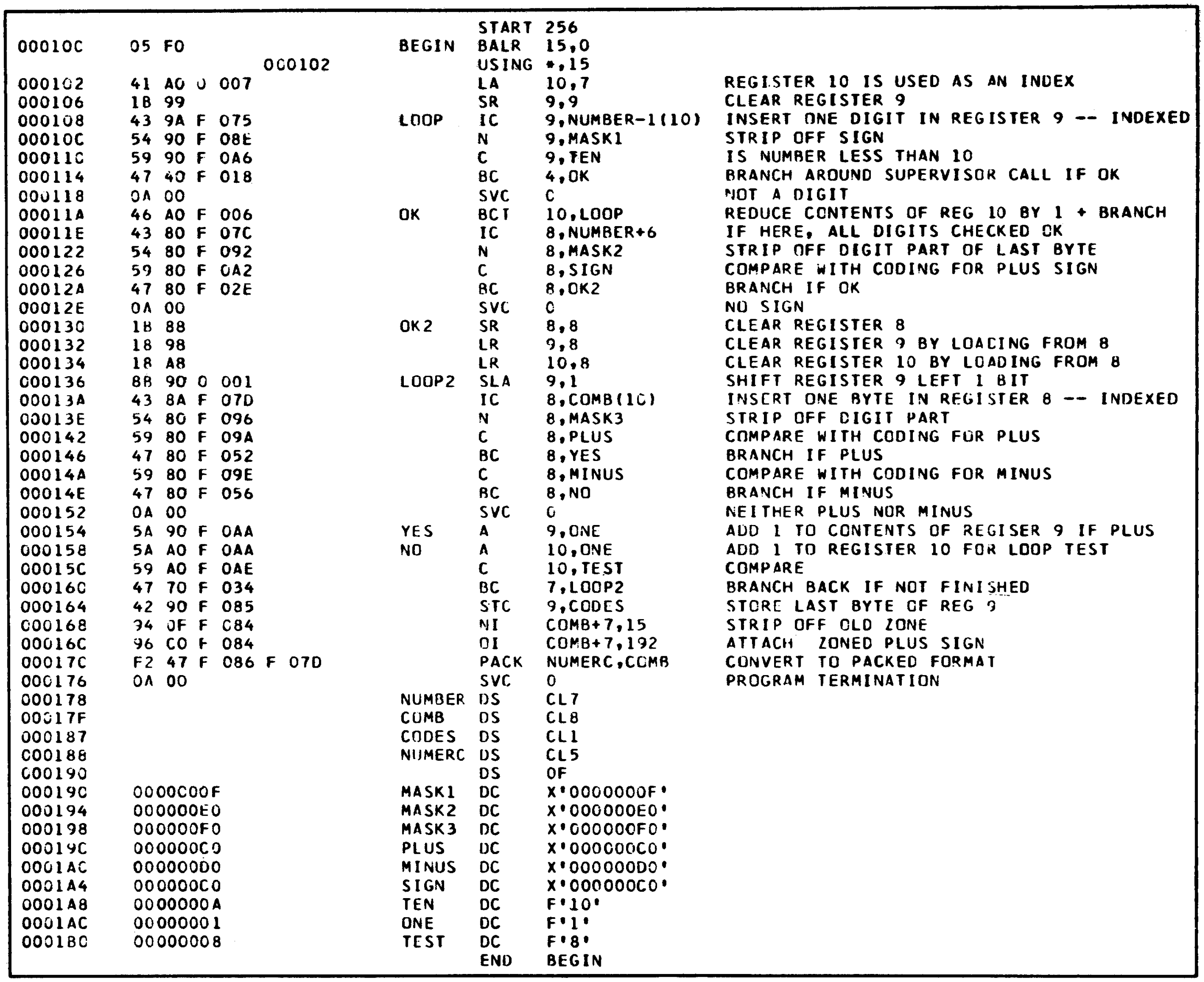
The instruction after OK picks up the rightmost byte of NUMBER; this should have either a plus sign or a minus sign. Another And, but with a different mask, strips off the numeric portion and the rightmost bit of the sign; we do not care whether the sign is plus or minus, a distinction which is made 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 eight or less.

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.



```
                                                START 256
00010C    05 F0                         BEGIN   BALR  15,0
                           000102               USING *,15
000102    41 A0 0 007                           LA    10,7              REGISTER 10 IS USED AS AN INDEX
000106    1B 99                                 SR    9,9               CLEAR REGISTER 9
000108    43 9A F 075                   LOOP    IC    9,NUMBER-1(10)    INSERT ONE DIGIT IN REGISTER 9 -- INDEXED
00010C    54 90 F 08E                           N     9,MASK1           STRIP OFF SIGN
000110    59 90 F 0A6                           C     9,TEN             IS NUMBER LESS THAN 10
000114    47 40 F 018                           BC    4,OK              BRANCH AROUND SUPERVISOR CALL IF OK
000118    0A 00                                 SVC   C                 NOT A DIGIT
00011A    46 A0 F 006                   OK      BCT   10,LOOP           REDUCE CONTENTS OF REG 10 BY 1 + BRANCH
00011E    43 80 F 07C                           IC    8,NUMBER+6        IF HERE, ALL DIGITS CHECKED OK
000122    54 80 F 092                           N     8,MASK2           STRIP OFF DIGIT PART OF LAST BYTE
000126    59 80 F 0A2                           C     8,SIGN            COMPARE WITH CODING FOR PLUS SIGN
00012A    47 80 F 02E                           BC    8,OK2             BRANCH IF OK
00012E    0A 00                                 SVC   0                 NO SIGN
000130    1B 88                         OK2     SR    8,8               CLEAR REGISTER 8
000132    18 98                                 LR    9,8               CLEAR REGISTER 9 BY LOACING FROM 8
000134    18 A8                                 LR    10,8              CLEAR REGISTER 10 BY LOADING FROM 8
000136    8B 90 0 001                   LOOP2   SLA   9,1               SHIFT REGISTER 9 LEFT 1 BIT
00013A    43 8A F 07D                           IC    8,COMB(10)        INSERT ONE BYTE IN REGISTER 8 -- INDEXED
00013E    54 80 F 096                           N     8,MASK3           STRIP OFF DIGIT PART
000142    59 80 F 09A                           C     8,PLUS            COMPARE WITH CODING FOR PLUS
000146    47 80 F 052                           BC    8,YES             BRANCH IF PLUS
00014A    59 80 F 09E                           C     8,MINUS           COMPARE WITH CODING FOR MINUS
00014E    47 80 F 056                           BC    8,NO              BRANCH IF MINUS
000152    0A 00                                 SVC   0                 NEITHER PLUS NOR MINUS
000154    5A 90 F 0AA                   YES     A     9,ONE             ADD 1 TO CONTENTS OF REGISER 9 IF PLUS
000158    5A A0 F 0AA                   NO      A     10,ONE            ADD 1 TO REGISTER 10 FOR LOOP TEST
00015C    59 A0 F 0AE                           C     10,TEST           COMPARE
000160    47 70 F 034                           BC    7,LOOP2           BRANCH BACK IF NOT FINISHED
000164    42 90 F 085                           STC   9,CODES           STORE LAST BYTE OF REG 9
000168    94 0F F 084                           NI    COMB+7,15         STRIP OFF OLD ZONE
00016C    96 C0 F 084                           OI    COMB+7,192        ATTACH  ZONED PLUS SIGN
00017C    F2 47 F 086 F 07D                     PACK  NUMERC,COMB       CONVERT TO PACKED FORMAT
000176    0A 00                                 SVC   0                 PROGRAM TERMINATION
000178                                  NUMBER  DS    CL7
00017F                                  COMB    DS    CL8
000187                                  CODES   DS    CL1
000188                                  NUMERC  DS    CL5
000190                                          DS    0F
000190    0000C00F                      MASK1   DC    X'0000000F'
000194    000000E0                      MASK2   DC    X'000000E0'
000198    000000F0                      MASK3   DC    X'000000F0'
00019C    000000C0                      PLUS    DC    X'000000C0'
0001AC    000000D0                      MINUS   DC    X'000000D0'
0001A4    000000C0                      SIGN    DC    X'000000C0'
0001A8    0000000A                      TEN     DC    F'10'
0001AC    00000001                      ONE     DC    F'1'
0001B0    00000008                      TEST    DC    F'8'
                                                END   BEGIN
```

Figure 73. A program to check a decimal field named NUMBER for validity, and to convert a combined field named COMB to a binary number and a packed decimal number

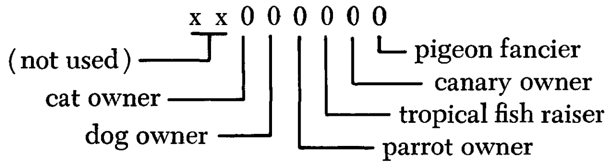
Summary

This chapter has illustrated some of the various techniques for logical operations in the System/360. The emphasis in this study has ranged from logic in the flowcharting sense, to bit-by-bit operations in making decisions, to two extended examples that involved bit operations on sequences of characters. In the course of this study we have seen how the great variety of actions described under the heading of "logical operations" can be effectively approached with the programming features of the System/360.

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 immediately before the end of the operand is reached, when 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 8: Edit, Translate, and Execute Instructions

The design of the System/360 includes a large number of features that make data processing faster and more efficient and which simplify the work of programming once the concepts have been mastered. The three instructions that are the main subject of this chapter are in this category. They can be characterized as extremely powerful, with a broad range of applicability. The Edit instruction provides a way of preparing output for printing in an easily readable form. The operation of the instruction is highly flexible, so that many kinds of editing actions can be performed, all in one pass through the data field. The Translate and Execute instructions have a broad range of application, limited only by the imagination of the programmer.

This chapter provides many examples of the use of these instructions. Illustrations of the Edit instruction are followed by four complete programs. The first shows how the Translate instruction can be used for code conversion, in this case to change a collating sequence. The second employs Translate and Test to search for sentinels in a list of names and addresses. The Execute instruction, together with Translate and Test, is employed in a third program to break apart the subfields of an operand field in an assembler instruction. A final example involves variable-length blocked records.

The Edit Instruction

The Edit instruction is one of the most powerful in the repertoire of the System/360. With proper planning it is possible, as we shall see, to suppress nonsignificant zeros, insert commas and decimal points, insert minus sign or credit symbol, 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 and fast instructions. A variation of the instruction, Edit and Mark, makes possible the rapid insertion of floating currency symbols.

We shall study the operation and application of this powerful 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.

The data to be edited is named as the second operand of the Edit. The first operand must name 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 has been destroyed by the editing process.) The pattern is in zoned format, as is the result; the Edit instruction involves a conversion from packed to zoned format.

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. The other seven characters will contain a special coding, 20_{16}, called a "digit select" character, 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

The 40 is the hexadecimal code for a blank. The 20s are the hexadecimal codes for the digit-select character. Suppose now that at DATA there is

00 01 00 0+

The edited result would be

bbb1000

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 left as they were. This is the desired action. Figure 74 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 in 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
```

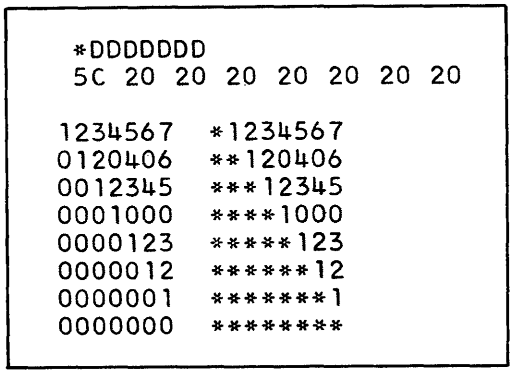
Figure 74. Examples of the application of the Edit instruction. The first line gives the editing pattern used, first in a symbolic form and then in hexadecimal. In the symbolic form, B stands for blank and D for digit select.

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 value for an asterisk is 5C; hence the new pattern is

5C 20 20 20 20 20 20 20

Figure 75 shows the edited results for the same DATA values.



```
*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 75. Examples of the application of the Edit instruction with the fill character as an asterisk

Any characters in the pattern other than the digit select (and two other control characters that we shall study later) 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:

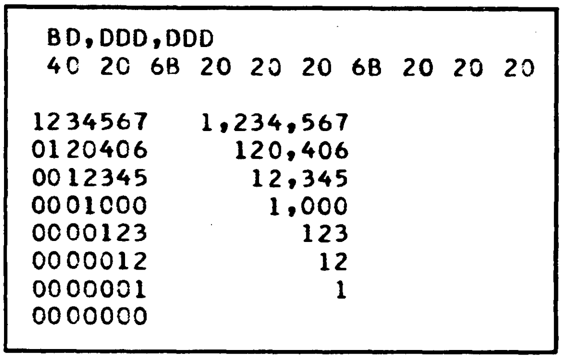
40 20 6B 20 20 20 6B 20 20 20

The 6B is hexadecimal coding for a comma. 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 76 shows the results for the same data values.

The 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:

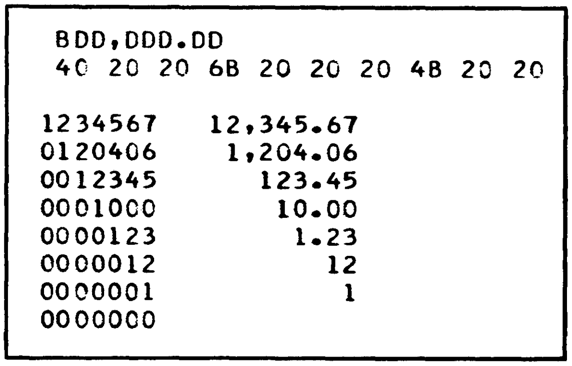
40 20 20 6B 20 20 20 4B 20 20

The edited results this time are in Figure 77.



```
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 76. Examples of the application of the Edit instruction with blank fill and the insertion of commas



```
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 77. Examples of the application of the Edit instruction with blank fill and the insertion of comma and decimal point

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-start character in the pattern. This character, which has the hexadecimal code 21, is either replaced by a digit from the data or replaced with the fill character, just as a digit-select character is. The difference is that the operation proceeds *as though* a significant digit had been found in the position occupied by the significance-start character. In other words, succeeding characters to the right will not be suppressed. (An exception to this generalization will be explored later.)

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 have been found by the time we reach the 21 significance-start character in a left-to-right scan, a "trigger" is turned on anyway. This trigger, called the S trigger, will make succeeding characters 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 78.

```
 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 78. Examples of the application of the Edit instruction with blank fill, comma and decimal point insertion, and significance start. In the symbolic pattern, S stands for significance start.

We can begin to get a little idea of how the machine does its work on this instruction by noting that the S trigger is initially set to zero before the scan begins. It stays at zero until a nonzero data digit is found, or until the significance-start character is encountered, at which time it is set to 1. It is the status of the S trigger that determines whether a digit select character in the pattern will be replaced by a digit or by the fill character. An S trigger setting of zero means that no significant (nonzero) digits have been found yet, so the fill character is used; an S trigger value of 1 means either that a significant digit has been found at some previous character position or that the significance-start character has been found; in either case the digit from the data is inserted even if it is a zero.

We have so far ignored the sign portion of the source data. The four rightmost bits of the source field are examined and used to set the S trigger according to an entirely different rule: zero if plus, 1 if minus. This is done every time the Edit instruction is executed, but if it happens after the completion of the scan of the pattern, as in the examples so far, it has no effect. 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 S trigger in controlling the instruction continues just as before, although the setting of the S trigger was accomplished in a rather different way. There are, of course, no more digits to move. Hence we will not want to place digit-select characters in the pattern in this position, 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: leave them unchanged if the S trigger has the value 1, but replace them with the fill character if the S trigger has the value zero.

What we do, then, is to place in the pattern the characters we want to print if the quantity is negative. If the data is indeed negative, our sign will be left, but if the data is positive, the sign will be replaced by the fill character.

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

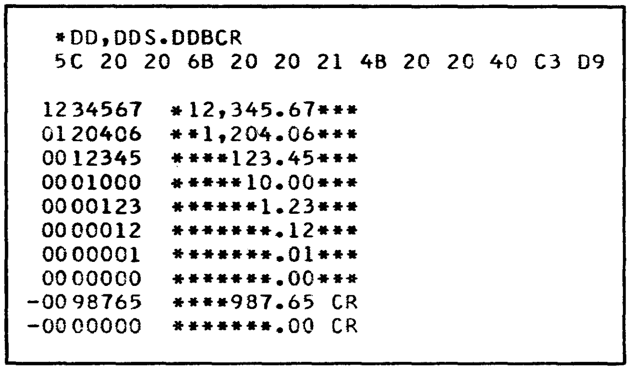
Figure 79 shows the results for sample data values as before, together with two negative values.

```
 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 79. Examples of the application of the Edit instruction with blank fill, comma and decimal point insertion, significance start, and CR symbol for negative numbers. In the symbolic pattern, C and R are themselves.

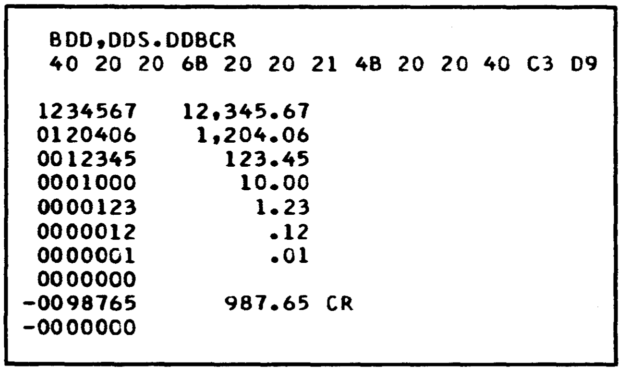
If we use an asterisk now as the fill character, positive quantities will have three asterisks following the cents, as shown in Figure 80. This might or might not be desired. There are other ways to handle the signs, as we shall see next.

We have seen above that an amount of zero prints in the general form .00 when significance start is used. It may in some cases be desirable to make such an amount print as all blanks or all asterisks. This is very



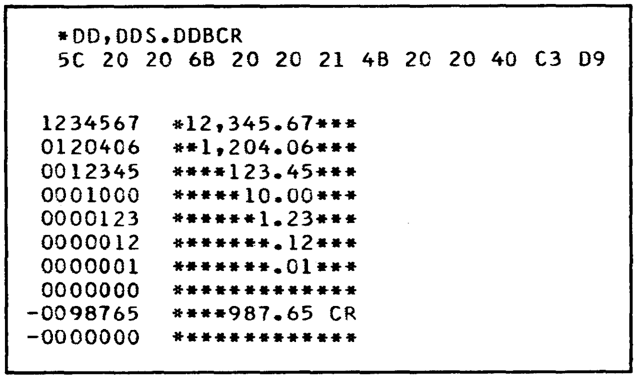
```
*DD,DDS.DDBCR
5C 20 20 6B 20 20 21 4B 20 20 40 C3 D9

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 80. Examples of the application of the Edit instruction (same as in Figure 79 except asterisk fill instead of blank fill)



```
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 81. Examples of the application of the Edit instruction, 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 82. Examples of the application of the Edit instruction with asterisk fill and zero fields filled with asterisks instead of being blanked

easily done by making use of the way the condition code is set by 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
```

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., in effect propagating the leftmost character through the field. This is precisely what we want if the fill character is the one to be substituted. (If some other character is desired, a suitable Move Characters instruction can, of course, be written.)

Figure 81 shows our familiar data values with zero fields blanked, and Figure 82 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 81 and 82; the Edit, the Branch on Condition, and the Move Characters are the same in both cases.

The condition code can be used also 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 special character, the field separator, 22_{16}. This character is replaced in the pattern by the fill character, and causes the S trigger to be set to zero. 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 an example. 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-start character.

The second field is to be printed with three digits to the right of the decimal point, with significance-start to force numbers 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 as just 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 select, f for field separator, s for significance start and other characters for themselves:

```
bd,ddd,dddfsd.ddd-fbbd,ddd,dds.dd
```

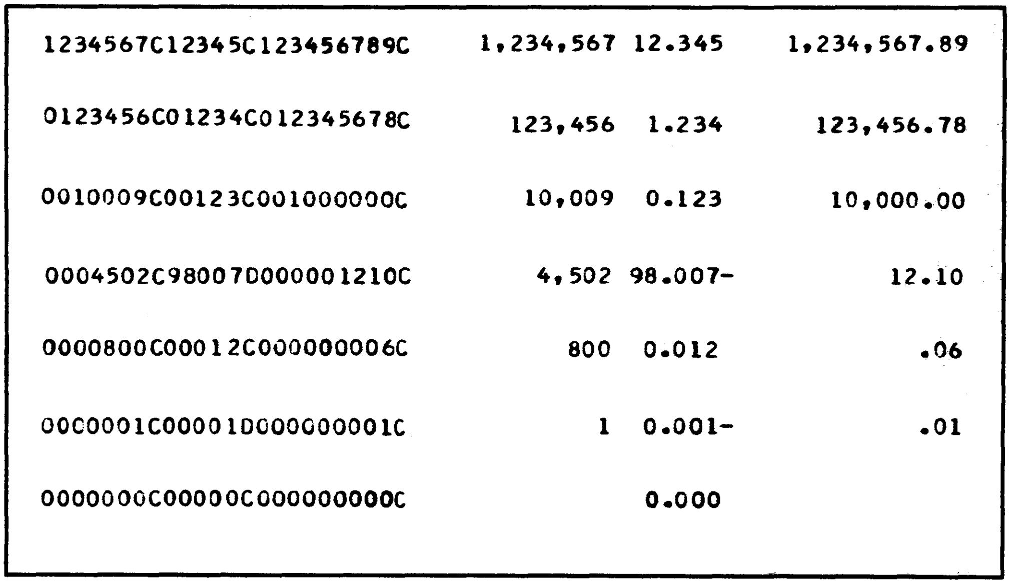
The required blank between the first and second edited result will be placed there by the replacement of the field separator with the fill character. The significance-start character in the part of the pattern corresponding to the second field will give the required handling of quantities less than 1. 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.)

Instructions to do the required actions are as follows:

```
     MVC   WORK,PATTRN
     ED    WORK,DATA
     BC    7,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 83 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

00C0001C00001D000000001C            1  0.001-             .01

0000000C00000C000000000C               0.000
```

Figure 83. 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-start character 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-start character 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. The pattern should, accordingly, be (in shorthand form)

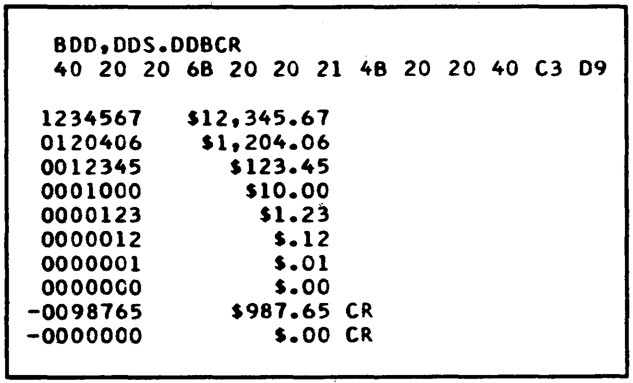
bd,dds.ddbCR

The significance-start character here is five 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+6
EDMK    WORK,DATA
BCTR    1,0
MVC     0(1,1),DOLLAR
```

The Load Address instruction as written, places in register 1 the address of the position one beyond the significance-start character. If significance is forced, this address remains in register 1, but otherwise the address of the first significant digit goes 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 Characters instruction we write an explicit displacement of zero, an explicit length of 1, and an explicit base register number of 1. The net effect is to move a one-character field from DOLLAR to the address specified by the base in register 1. This is the desired action.

Figure 84 shows the effect on sample data values. Zero fields could be blanked by methods we have seen above.



```
  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
 00000C0         $.00
-0098765      $987.65 CR
-0000000         $.00 CR
```

Figure 84. Examples of the application of the Edit and Mark instruction to get a floating currency symbol

The Translate Instruction

Another powerful programming feature of the 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 preestablished conversion table, we can convert a string of characters from one form to another at speeds that compare favorably with that of decimal addition.

In the example program for this topic we shall use the Translate instruction to permit a reversal of letters and digits in the collating sequence. When we use the decimal Compare to 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 using the Compare Logical Character instruction, this is the correct ordering 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 all this can be done fairly simply, using the Translate instruction.

Looking at the instruction itself, Translate (TR) is an SS format instruction with two operands. The first operand address names the leftmost byte of a group to be translated. The second operand address names the start of a list, or table, that is used to make the conversion. The bytes referenced by the first operand address are called *argument* bytes; the bytes in the table referenced by the second operand address are called *function* bytes.

The operation is as follows. An argument byte (first operand) is obtained from storage. The eight-bit byte, interpreted as a binary number, is added to the second operand address, thereby giving a new address somewhere in the conversion table. The byte at this newly computed address is obtained from storage and is placed in the position occupied by the original argument byte. This action is repeated for all argument bytes until the first operand is exhausted.

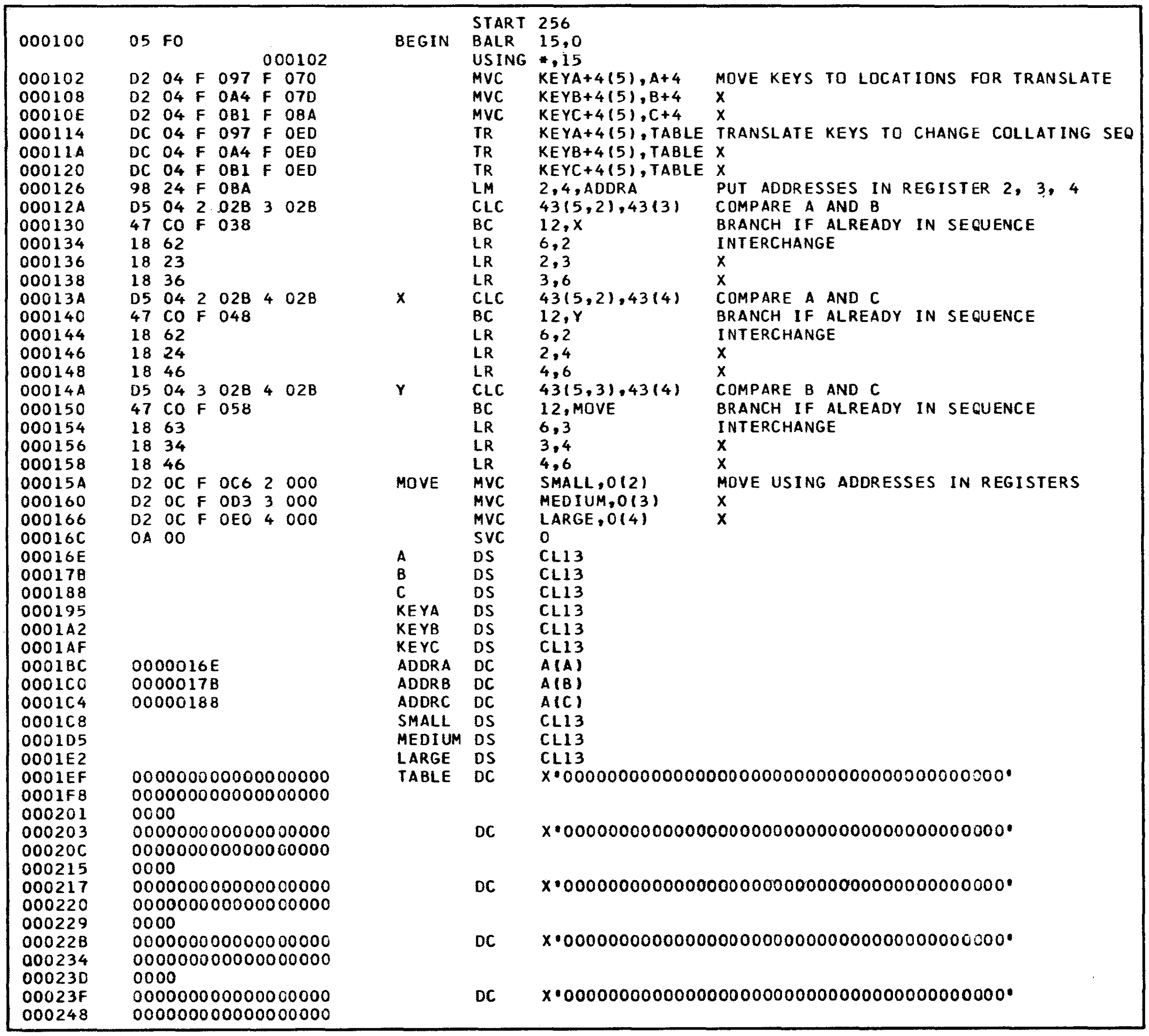
The programming task is in designing the translate table properly. Perhaps the situation can be made clearer by looking at it another way.

We view the byte to be translated as an eight-bit binary number used as an index. Adding this to the starting address of the table gives an address that is unique for each particular eight-bit combination; that is, there is a unique table address corresponding to each possible character to be translated. Our job is to arrange the translate table so that we will find at each such address the byte which should replace the input byte.

Take an example. Suppose we have a translate table starting at 5000. We want an input (argument) character of A to be translated into a Q. What do we do? We look up the coding for A and find that it is 11000001 in binary, which is C1 hexadecimal and 193 decimal. The coding for Q is 11011000 = D8 = 216. We set up the table so that 193 bytes after the start of the table, namely at 5193, there is a byte consisting of 1101100 = D8 = 216. An A appearing in the argument stream will lead to the address 5193, where the translated coding will be found. The A is replaced with Q. If all possible eight-bit combinations can appear in the input stream, there must be 256 translate-table entries.

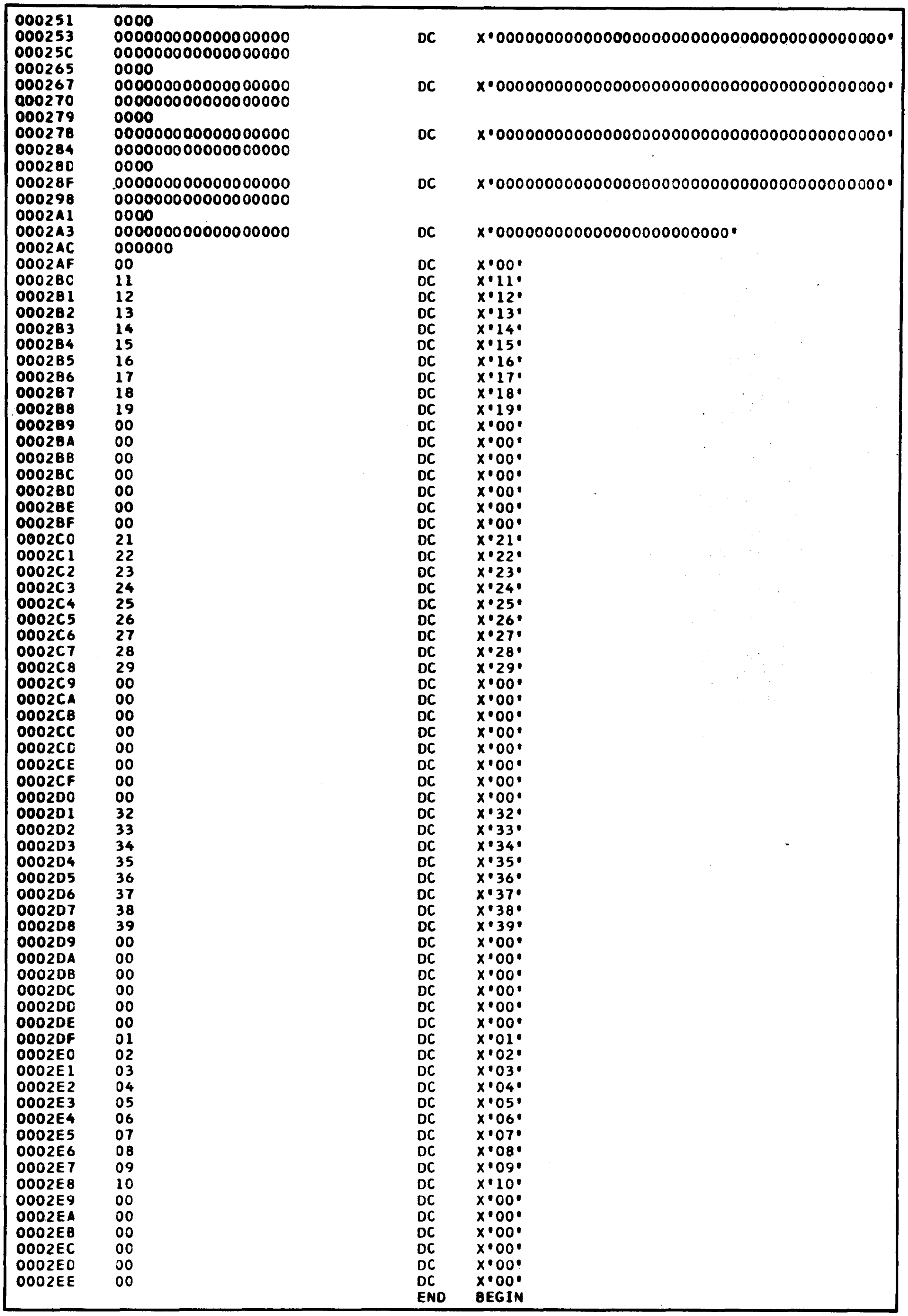
Let us turn to the problem just sketched of rearranging the letters and digits so that digits sort ahead of letters, which is the opposite of the normal collating sequence of the machine. The translated characters will be used *only* for the sorting operation; we are not required to translate the characters 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 85 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.)



```
                                           START 256
000100    05 FO                     BEGIN  BALR  15,0
                        000102             USING *,15
000102    D2 04 F 097 F 070                MVC   KEYA+4(5),A+4   MOVE KEYS TO LOCATIONS FOR TRANSLATE
000108    D2 04 F 0A4 F 07D                MVC   KEYB+4(5),B+4   X
00010E    D2 04 F 0B1 F 08A                MVC   KEYC+4(5),C+4   X
000114    DC 04 F 097 F 0ED                TR    KEYA+4(5),TABLE TRANSLATE KEYS TO CHANGE COLLATING SEQ
00011A    DC 04 F 0A4 F 0ED                TR    KEYB+4(5),TABLE X
000120    DC 04 F 0B1 F 0ED                TR    KEYC+4(5),TABLE X
000126    98 24 F 0BA                      LM    2,4,ADDRA       PUT ADDRESSES IN REGISTER 2, 3, 4
00012A    D5 04 2 02B 3 02B                CLC   43(5,2),43(3)   COMPARE A AND B
000130    47 CO F 038                      BC    12,X            BRANCH IF ALREADY IN SEQUENCE
000134    18 62                            LR    6,2             INTERCHANGE
000136    18 23                            LR    2,3             X
000138    18 36                            LR    3,6             X
00013A    D5 04 2 02B 4 02B         X      CLC   43(5,2),43(4)   COMPARE A AND C
000140    47 CO F 048                      BC    12,Y            BRANCH IF ALREADY IN SEQUENCE
000144    18 62                            LR    6,2             INTERCHANGE
000146    18 24                            LR    2,4             X
000148    18 46                            LR    4,6             X
00014A    D5 04 3 02B 4 02B         Y      CLC   43(5,3),43(4)   COMPARE B AND C
000150    47 CO F 058                      BC    12,MOVE         BRANCH IF ALREADY IN SEQUENCE
000154    18 63                            LR    6,3             INTERCHANGE
000156    18 34                            LR    3,4             X
000158    18 46                            LR    4,6             X
00015A    D2 0C F 0C6 2 000         MOVE   MVC   SMALL,0(2)      MOVE USING ADDRESSES IN REGISTERS
000160    D2 0C F 0D3 3 000                MVC   MEDIUM,0(3)     X
000166    D2 0C F 0E0 4 000                MVC   LARGE,0(4)      X
00016C    0A 00                            SVC   0
00016E                              A      DS    CL13
00017B                              B      DS    CL13
000188                              C      DS    CL13
000195                              KEYA   DS    CL13
0001A2                              KEYB   DS    CL13
0001AF                              KEYC   DS    CL13
0001BC    0000016E                  ADDRA  DC    A(A)
0001C0    0000017B                  ADDRB  DC    A(B)
0001C4    00000188                  ADDRC  DC    A(C)
0001C8                              SMALL  DS    CL13
0001D5                              MEDIUM DS    CL13
0001E2                              LARGE  DS    CL13
0001EF    000000000000000000        TABLE  DC    X'0000000000000000000000000000000000000000'
0001F8    000000000000000000
000201    0000
000203    000000000000000000               DC    X'0000000000000000000000000000000000000000'
00020C    000000000000000000
000215    0000
000217    000000000000000000               DC    X'0000000000000000000000000000000000000000'
000220    000000000000000000
000229    0000
00022B    000000000000000000               DC    X'0000000000000000000000000000000000000000'
000234    000000000000000000
00023D    0000
00023F    000000000000000000               DC    X'0000000000000000000000000000000000000000'
000248    000000000000000000
```

Figure 85. A program to sort three fields named A, B, and C into ascending sequence on five-character keys contained in them, and to place the sorted items in SMALL, MEDIUM, and LARGE. The collating sequence of letters and digits is reversed by the use of a Translate instruction.



```
000251     0000
000253     000000000000000000          DC     X'0000000000000000000000000000000000000000'
00025C     000000000000000000
000265     0000
000267     000000000000000000          DC     X'0000000000000000000000000000000000000000'
000270     000000000000000000
000279     0000
00027B     000000000000000000          DC     X'0000000000000000000000000000000000000000'
000284     000000000000000000
00028D     0000
00028F     000000000000000000          DC     X'0000000000000000000000000000000000000000'
000298     000000000000000000
0002A1     0000
0002A3     000000000000000000          DC     X'000000000000000000000000'
0002AC     000000
0002AF     00                          DC     X'00'
0002BC     11                          DC     X'11'
0002B1     12                          DC     X'12'
0002B2     13                          DC     X'13'
0002B3     14                          DC     X'14'
0002B4     15                          DC     X'15'
0002B5     16                          DC     X'16'
0002B6     17                          DC     X'17'
0002B7     18                          DC     X'18'
0002B8     19                          DC     X'19'
0002B9     00                          DC     X'00'
0002BA     00                          DC     X'00'
0002BB     00                          DC     X'00'
0002BC     00                          DC     X'00'
0002BD     00                          DC     X'00'
0002BE     00                          DC     X'00'
0002BF     00                          DC     X'00'
0002C0     21                          DC     X'21'
0002C1     22                          DC     X'22'
0002C2     23                          DC     X'23'
0002C3     24                          DC     X'24'
0002C4     25                          DC     X'25'
0002C5     26                          DC     X'26'
0002C6     27                          DC     X'27'
0002C7     28                          DC     X'28'
0002C8     29                          DC     X'29'
0002C9     00                          DC     X'00'
0002CA     00                          DC     X'00'
0002CB     00                          DC     X'00'
0002CC     00                          DC     X'00'
0002CD     00                          DC     X'00'
0002CE     00                          DC     X'00'
0002CF     00                          DC     X'00'
0002D0     00                          DC     X'00'
0002D1     32                          DC     X'32'
0002D2     33                          DC     X'33'
0002D3     34                          DC     X'34'
0002D4     35                          DC     X'35'
0002D5     36                          DC     X'36'
0002D6     37                          DC     X'37'
0002D7     38                          DC     X'38'
0002D8     39                          DC     X'39'
0002D9     00                          DC     X'00'
0002DA     00                          DC     X'00'
0002DB     00                          DC     X'00'
0002DC     00                          DC     X'00'
0002DD     00                          DC     X'00'
0002DE     00                          DC     X'00'
0002DF     01                          DC     X'01'
0002E0     02                          DC     X'02'
0002E1     03                          DC     X'03'
0002E2     04                          DC     X'04'
0002E3     05                          DC     X'05'
0002E4     06                          DC     X'06'
0002E5     07                          DC     X'07'
0002E6     08                          DC     X'08'
0002E7     09                          DC     X'09'
0002E8     10                          DC     X'10'
0002E9     00                          DC     X'00'
0002EA     00                          DC     X'00'
0002EB     00                          DC     X'00'
0002EC     00                          DC     X'00'
0002ED     00                          DC     X'00'
0002EE     00                          DC     X'00'
                                       END    BEGIN
```

Figure 85 (continued)

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 zeros, which is not quite representative of what we might do in practice. In an actual application, if we really believed that nothing else could appear, we would use relative addressing to reference the table, and would not store the zeros before and after the table. If, as is more likely, we are concerned about the possibility of erroneous data, we might use the table to do some kind of checking on the data.

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 85 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; it is the 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 Condition 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 was 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 had to be the same distance beyond A as the translated key of B in KEYB was beyond B, and similarly with KEYC and C. This is because the same displacement of 43 had to 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.

We now carry out the same actions using the addresses in registers 2 and 4, thus comparing the keys of A and C. The two addresses are interchanged if necessary, to make the address in 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 86 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 items, in order, were:

1111SMITH1111
2222ADAMS2222
3333567893333

In other words, the original items were in reverse order, according to the sequencing pattern we want.

	Register 2	Register 3	Register 4
Before	0000016E	0000017B	00000188
X	0000017B	0000016E	00000188
Y	00000188	0000016E	0000017B
MOVE	00000188	0000017B	0000016E

Figure 86. The contents of registers 2, 3, and 4 during the execution of the program in Figure 85. The original items were in reverse sequence according to their keys.

The Translate and Test Instruction and The Execute Instruction

The Translate and Test instruction (TRT) adds great power to the processing capability of the System/360. It is related to the Translate instruction, but is different enough and powerful enough to merit our close attention. We shall, in fact, study it in three programs in this and the next two sections.

As with Translate, we work with a table that is accessed exactly as in Translate. That is, a first operand argument byte addresses a particular entry in the table by an address computation. There is no change in the argument bytes as a result, however, despite the name "translate". What happens, instead, is that the argument byte is inspected. If the addressed function byte is zero, the next argument byte is inspected, etc. If all argument bytes reference function bytes that are zero, the condition code is set accordingly and the execution is complete. However, if ever a function byte is nonzero, that function byte is placed in register 2, and the address of the *argument* byte is placed in register 1. The operation is then terminated.

This means that we can, with a single instruction, 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 possibilities for the application of this instruction are almost limitless.

To illustrate one way to use the instruction we take the following problem.

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 (1) after each "line" of information there is a dollar sign, (2) after the last line of an address there are two dollar signs, and (3) 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 will consist of 254 zeros, with entries only in positions 91 and 92 corresponding to dollar sign and asterisk respectively. For the dollar sign we have chosen to enter an 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. (It is assumed, of course, that dollar sign and asterisk would never be legitimate characters in a name or address. If this were not a good assumption, it would not be difficult to assign sentinels for which there is no graphic equivalent.)

The program in Figure 87 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.

```
                                                START 256
000100    05 F0                          BEGIN  BALR  15,0
                         000102                 USING *,15
000102    41 30 F 233                           LA    3,NAME                 STARTING ADDRESS OF RECORD
000106    41 90 0 004                    AGAIN  LA    9,4                    FOR ERROR CHECKING
00010A    41 A0 F 053                           LA    10,LINE1               INITIALIZE TO START OF FIRST LINE
00010E    D2 F0 F 053 F 052                     MVC   LINE 1(241),BLANK      BLANK LINES 1,2 AND 1ST POS 3RD LINE
000114    D2 EE F 144 F 143                     MVC   LINE3+1(239),LINE3     BLANK OUT LAST TWO LINES
00011A    DD 77 3 000 F 306              LOOP   TRT   0(120,3),TABLE         SEARCH FOR DELIMITER
000120    47 80 F 050                           BC    8,ERROR                BRANCH IF NO DELIMITER IN 120 CHARS
000124    18 41                                 LR    4,1                    GET LENGTH OF LINE
000126    1B 43                                 SR    4,3                    X
000128    5B 40 F 2FE                           S     4,ONE                  X
00012C    47 40 F 040                           BC    4,OUT                  BRANCH IF TWO DELIMITERS IN SEQUENCE
000130    44 40 F 2F6                           EX    4,MVCINS               MOVE LINE TO PRINTING POSITION
000134    41 30 1 001                           LA    3,1(0,1)               SET UP NEXT TRT
000138    41 AA 0 078                           LA    10,120(10)             TO GET NEXT LINE
00013C    46 90 F 018                           BCT   9,LOOP                 BRANCH UNLESS FIFTH LINE
000140    0A 00                                 SVC   0
000142    0A 01                          OUT    SVC   1                      THE PRINTING IS DONE HERE
000144    41 30 1 001                           LA    3,1(0,1)               SET UP FOR NEXT ADDRESS
000148    59 20 F 302                           C     2,ENDCON               SEE IF DELIMITER WAS *
00014C    47 70 F 004                           BC    7,AGAIN                BRANCH IF NOT *
000150    0A 00                                 SVC   0                      ALL FINISHED IF HERE
000152    0A 00                          ERROR  SVC   0                      ERROR STOP
000154    40                             BLANK  DC    CL1' '
000155                                   LINE1  DS    CL120
0001CD                                   LINE2  DS    CL120
000245                                   LINE3  DS    CL120
0002BD                                   LINE4  DS    CL120
000335    E2D4C9E3C85BC4C5E3             NAME   DC    C'SMITH$DETROIT$$J. C. JACKSON$1234 MAIN STREET$CHI'
00033E    D9D6C9E35B5BD14B40
000347    C34B40D1C1C3D2E2D6
000350    D55BF1F2F3F440D4C1
000359    C9D540E2E3D9C5C5E3
000362    5BC3C8C9
000366    C3C1C7D66B40C9D3D3                    DC    C'CAGO, ILLINOIS$$F. C. R. ANDERSON$553 MAPLE PLACE APAR'
00036F    C9D5D6C9E25B5BC64B
000378    40C34B40D94B40C1D5
000381    C4C5D9E2D6D55BF5F5
00038A    F340D4C1D7D3C540D7
000393    D3C1C3C540C1D7C1D9
00039C    E3D4C5D5E340F5C35B                    DC    C'TMENT 5C$WHITE PLAINS, NEW YCRK$$D. D. ADAMS AND FAMILY'
0003A5    E6C8C9E3C540D7D3C1
0003AE    C9D5E26B40D5C5E640
0003B7    E8D6D9D25B5BC44B40
0003C0    C44B40C1C4C1D4E240
0003C9    C1D5C440C6C1D4C9D3
0003C2    E8
0003C3    5BF5F0F540C7D9C1E3                    DC    C'$505 GRATHSON$APT. 31$READING,PENN.$*'
0003CC    C8E2D6D55BC1D7E34B
0003E5    40F3F15BD9C5C1C4C9
0003EE    D5C76BD7C5D5D54B5B
0003F7    5C
0003F8    D2 00 A 000 3 000              MVCINS MVC   0(0,10),0(3)
000400    00000001                       ONE    DC    F'1'
000404    00000002                       ENDCON DC    F'2'
000408    000000000000000000             TABLE  DC    X'0000000000000000000000'
000411    0000
000413    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
00041C    000000000000000000
000425    0000
000427    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
000430    000000000000000000
000439    0000
00043B    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
000444    000000000000000000
00044D    0000
00044F    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
000458    000000000000000000
000461    0000
000463    01                                    DC    X'01'
000464    02                                    DC    X'02'
000465    000000                                DC    X'000000'
000468    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
000471    000000000000000000
00047A    0000
00047C    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
000485    000000000000000000
00048E    0000
000490    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
000499    000000000000000000
0004A2    0000
0004A4    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
0004AD    000000000000000000
0004B6    0000
0004B8    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
0004C1    000000000000000000
0004CA    0000
0004CC    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
0004D5    000000000000000000
0004DE    0000
0004E0    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
0004E9    000000000000000000
0004F2    0000
0004F4    000000000000000000                    DC    X'0000000000000000000000000000000000000000'
0004FD    000000000000000000
000506    0000
                                                END   BEGIN
```

Figure 87. 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; each line is of variable length

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 some kind of error in the preparation of the input stream, which would in practice be done by some other program, 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 Condition, accordingly, takes us to an error exit.

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 and address of the first character of the line. The difference is the length of the line, in bytes. We are about 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 one from the difference residing in register 4.

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 Condition to OUT where we 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 4 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  4,MVCINS
```

This means to execute the instruction at the second operand address named (MVCINS), after or-ing together the last eight bits of register 4 and the length code portion of the instruction named. 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 4. One line of the complete name and address is thus moved to a printing position.

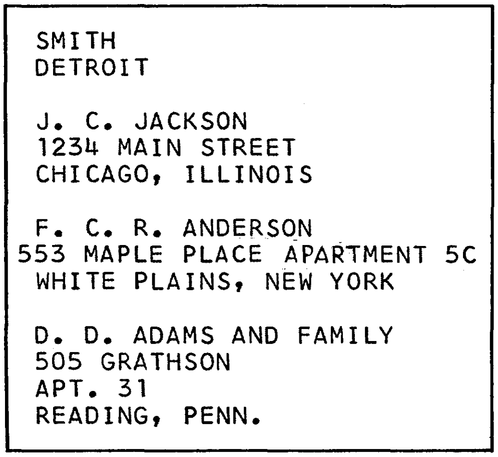
We are now about ready to go back for another look at the input stream. To do that, register 3 must contain the address of the next valid data character in the stream. Register 1 contains almost what we need; it has the address of the dollar sign just prior to the next valid character. We accordingly 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 1, 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 have the program segment to print the output. Here we see the dump request to the supervisor. In a complete program this would, of course, be replaced by something else, such as an IOCS WRITE.

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 Condition back to AGAIN to repeat the whole process. Otherwise we reach the normal exit from the program.

Figure 88 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 88. Four names and addresses produced by the program in Figure 87

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 operand that we shall process will always consist of:

1. A one- or two-digit register number, in decimal
2. A comma
3. A symbol, consisting of from one to six letters
4. Either a plus sign or a minus sign
5. An integer of from one to four digits specifying an increment or decrement
6. A blank

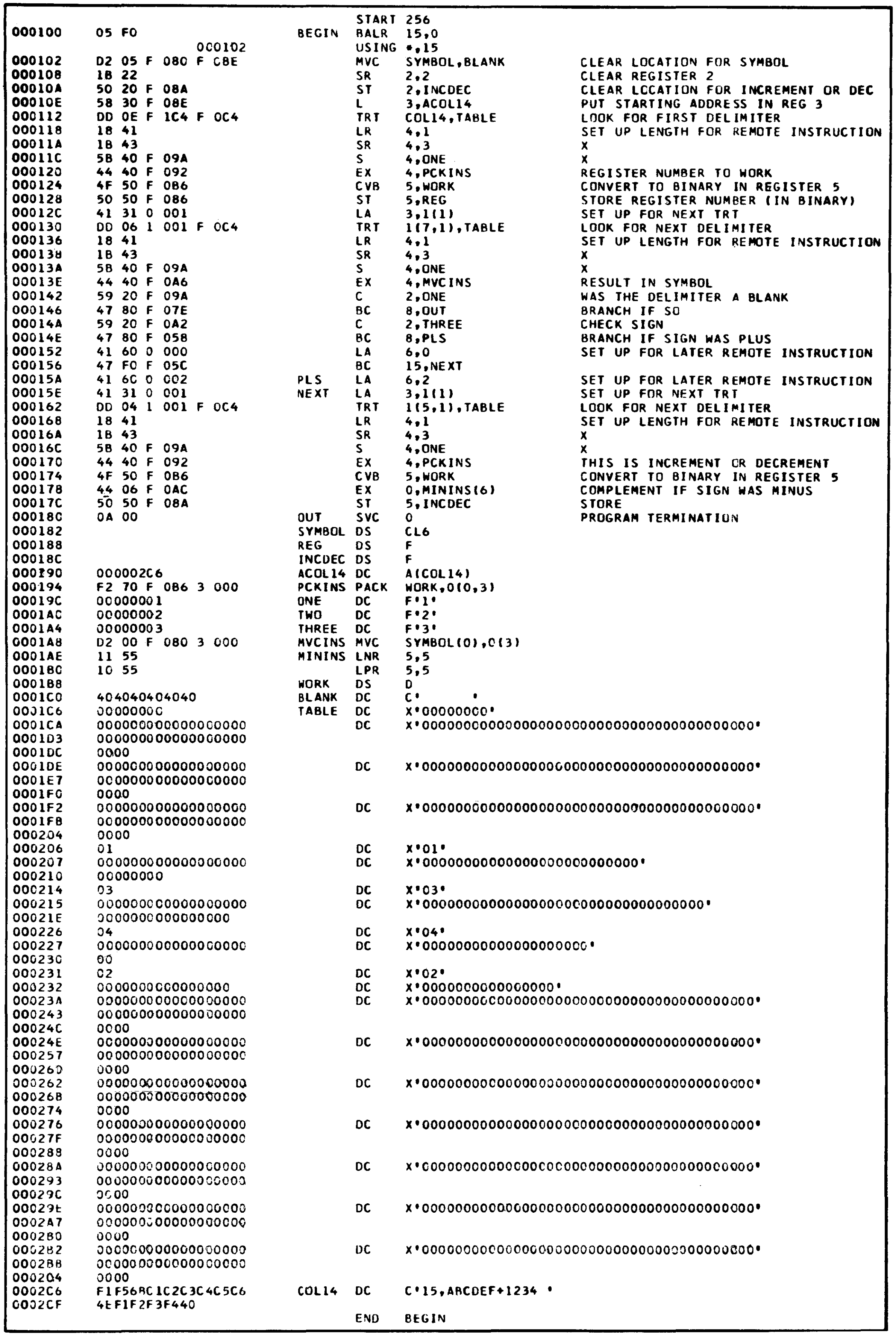
We are required to place the register number in REG as a binary number, to place the symbol in SYMBOL, and to place in INCDEC the increment or decrement as a properly signed binary number.

We are, of course, defining away a great deal of the actual work of an assembler where it would not be known, for example, that all instructions will have the relative addressing after the symbol, or that there are no errors. The complete task uses techniques of the sort we shall use here, but is much more complex.

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 latter. We set up a translate table having entries in the positions corresponding to these four delimiters.

The input stream begins at symbolic location COL14, 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 89 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.



```
                                               START 256
000100     05 F0                        BEGIN  BALR  15,0
                          000102               USING *,15
000102     D2 05 F 080 F 0BE                   MVC   SYMBOL,BLANK           CLEAR LOCATION FOR SYMBOL
000108     1B 22                               SR    2,2                    CLEAR REGISTER 2
00010A     50 20 F 08A                         ST    2,INCDEC               CLEAR LOCATION FOR INCREMENT OR DEC
00010E     58 30 F 08E                         L     3,ACOL14               PUT STARTING ADDRESS IN REG 3
000112     DD 0E F 1C4 F 0C4                   TRT   COL14,TABLE            LOOK FOR FIRST DELIMITER
000118     18 41                               LR    4,1                    SET UP LENGTH FOR REMOTE INSTRUCTION
00011A     1B 43                               SR    4,3                    X
00011C     5B 40 F 09A                         S     4,ONE                  X
000120     44 40 F 092                         EX    4,PCKINS               REGISTER NUMBER TO WORK
000124     4F 50 F 0B6                         CVB   5,WORK                 CONVERT TO BINARY IN REGISTER 5
000128     50 50 F 086                         ST    5,REG                  STORE REGISTER NUMBER (IN BINARY)
00012C     41 31 0 001                         LA    3,1(1)                 SET UP FOR NEXT TRT
000130     DD 06 1 001 F 0C4                   TRT   1(7,1),TABLE           LOOK FOR NEXT DELIMITER
000136     18 41                               LR    4,1                    SET UP LENGTH FOR REMOTE INSTRUCTION
000138     1B 43                               SR    4,3                    X
00013A     5B 40 F 09A                         S     4,ONE                  X
00013E     44 40 F 0A6                         EX    4,MVCINS               RESULT IN SYMBOL
000142     59 20 F 09A                         C     2,ONE                  WAS THE DELIMITER A BLANK
000146     47 80 F 07E                         BC    8,OUT                  BRANCH IF SO
00014A     59 20 F 0A2                         C     2,THREE                CHECK SIGN
00014E     47 80 F 058                         BC    8,PLS                  BRANCH IF SIGN WAS PLUS
000152     41 60 0 000                         LA    6,0                    SET UP FOR LATER REMOTE INSTRUCTION
000156     47 F0 F 05C                         BC    15,NEXT
00015A     41 60 0 002                  PLS    LA    6,2                    SET UP FOR LATER REMOTE INSTRUCTION
00015E     41 31 0 001                  NEXT   LA    3,1(1)                 SET UP FOR NEXT TRT
000162     DD 04 1 001 F 0C4                   TRT   1(5,1),TABLE           LOOK FOR NEXT DELIMITER
000168     18 41                               LR    4,1                    SET UP LENGTH FOR REMOTE INSTRUCTION
00016A     1B 43                               SR    4,3                    X
00016C     5B 40 F 09A                         S     4,ONE                  X
000170     44 40 F 092                         EX    4,PCKINS               THIS IS INCREMENT OR DECREMENT
000174     4F 50 F 0B6                         CVB   5,WORK                 CONVERT TO BINARY IN REGISTER 5
000178     44 06 F 0AC                         EX    0,MININS(6)            COMPLEMENT IF SIGN WAS MINUS
00017C     50 50 F 08A                         ST    5,INCDEC               STORE
000180     0A 00                        OUT    SVC   0                      PROGRAM TERMINATION
000182                                  SYMBOL DS    CL6
000188                                  REG    DS    F
00018C                                  INCDEC DS    F
000190     000002C6                     ACOL14 DC    A(COL14)
000194     F2 70 F 0B6 3 000            PCKINS PACK  WORK,0(0,3)
00019C     00000001                     ONE    DC    F'1'
0001AC     00000002                     TWO    DC    F'2'
0001A4     00000003                     THREE  DC    F'3'
0001A8     D2 00 F 080 3 000            MVCINS MVC   SYMBOL(0),0(3)
0001AE     11 55                        MININS LNR   5,5
0001BC     10 55                               LPR   5,5
0001B8                                  WORK   DS    D
0001C0     404040404040                 BLANK  DC    C'      '
0001C6     00000000                     TABLE  DC    X'00000000'
0001CA     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
0001D3     000000000000000000
0001DC     0000
0001DE     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
0001E7     000000000000000000
0001F0     0000
0001F2     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
0001FB     000000000000000000
000204     0000
000206     01                                  DC    X'01'
000207     000000000000000000                  DC    X'00000000000000000000000000'
000210     00000000
000214     03                                  DC    X'03'
000215     000000000000000000                  DC    X'0000000000000000000000000000000000'
00021E     0000000000000000
000226     04                                  DC    X'04'
000227     000000000000000000                  DC    X'00000000000000000000'
000230     00
000231     02                                  DC    X'02'
000232     0000000000000000                    DC    X'0000000000000000'
00023A     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
000243     000000000000000000
00024C     0000
00024E     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
000257     000000000000000000
000260     0000
000262     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
00026B     000000000000000000
000274     0000
000276     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
00027F     000000000000000000
000288     0000
00028A     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
000293     000000000000000000
00029C     0000
00029E     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
0002A7     000000000000000000
0002B0     0000
0002B2     000000000000000000                  DC    X'0000000000000000000000000000000000000000'
0002BB     000000000000000000
0002C4     0000
0002C6     F1F56BC1C2C3C4C5C6           COL14  DC    C'15,ABCDEF+1234 '
0002CF     4EF1F2F3F440
                                               END   BEGIN
```

Figure 89. A program to break an assembler language operand into its constituent parts. All the subfields of the operand are of variable length. The program uses Translate and Test instructions to detect the delimiters of comma, plus or minus, and blank.

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 first operand from the address given in register 3, its length from register 4, and places the result in WORK. The latter had been 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 with 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 can be 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 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 (separating the various parts of the operand) would facilitate these further operations.

Loops with Variable-Length Blocked Records

The following illustrative program applies techniques that are highly useful in certain commercial applications, and which the features of the System/360 make particularly easy to accomplish. The task is the processing of blocked tape records — 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 design, furthermore, which 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. Immediately following the description, however long it might be, will be 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 an 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 which 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.

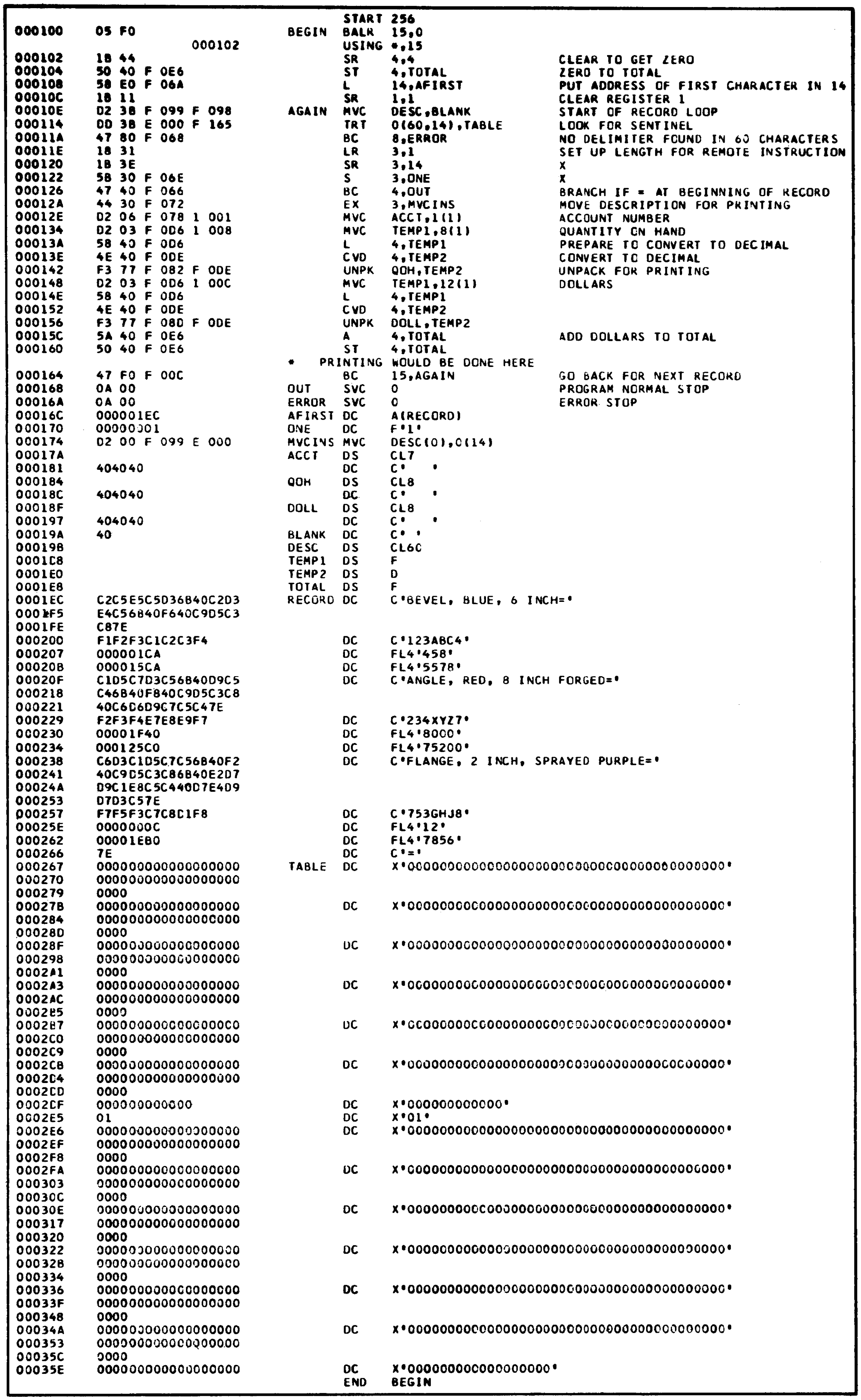
The program is shown in Figure 90. After the usual preliminaries we clear register 4 and store the resulting zero in TOTAL in order to be sure that the accumulator for total sales is zeroed. Register 14 is next loaded with the address of the first character of the block; register 14 will always contain the address of the first character of the *next* record as the loop is repeated.

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 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 14 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 Condition 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 definitely is 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 regis-



```
                                             START 256
000100    05 F0                       BEGIN  BALR  15,0
                    000102                   USING *,15
000102    1B 44                              SR    4,4                      CLEAR TO GET ZERO
000104    50 40 F 0E6                        ST    4,TOTAL                  ZERO TO TOTAL
000108    58 E0 F 06A                        L     14,AFIRST                PUT ADDRESS OF FIRST CHARACTER IN 14
00010C    1B 11                              SR    1,1                      CLEAR REGISTER 1
00010E    D2 3B F 099 F 098           AGAIN  MVC   DESC,BLANK               START OF RECORD LOOP
000114    DD 3B E 000 F 165                  TRT   0(60,14),TABLE           LOOK FOR SENTINEL
00011A    47 80 F 068                        BC    8,ERROR                  NO DELIMITER FOUND IN 60 CHARACTERS
00011E    18 31                              LR    3,1                      SET UP LENGTH FOR REMOTE INSTRUCTION
000120    1B 3E                              SR    3,14                     X
000122    5B 30 F 06E                        S     3,ONE                    X
000126    47 40 F 066                        BC    4,OUT                    BRANCH IF = AT BEGINNING OF RECORD
00012A    44 30 F 072                        EX    3,MVCINS                 MOVE DESCRIPTION FOR PRINTING
00012E    D2 06 F 078 1 001                  MVC   ACCT,1(1)                ACCOUNT NUMBER
000134    D2 03 F 0D6 1 008                  MVC   TEMP1,8(1)               QUANTITY ON HAND
00013A    58 40 F 0D6                        L     4,TEMP1                  PREPARE TO CONVERT TO DECIMAL
00013E    4E 40 F 0DE                        CVD   4,TEMP2                  CONVERT TO DECIMAL
000142    F3 77 F 082 F 0DE                  UNPK  QOH,TEMP2                UNPACK FOR PRINTING
000148    D2 03 F 0D6 1 00C                  MVC   TEMP1,12(1)              DOLLARS
00014E    58 40 F 0D6                        L     4,TEMP1
000152    4E 40 F 0DE                        CVD   4,TEMP2
000156    F3 77 F 08D F 0DE                  UNPK  DOLL,TEMP2
00015C    5A 40 F 0E6                        A     4,TOTAL                  ADD DOLLARS TO TOTAL
000160    50 40 F 0E6                        ST    4,TOTAL
                                      *   PRINTING WOULD BE DONE HERE
000164    47 F0 F 00C                        BC    15,AGAIN                 GO BACK FOR NEXT RECORD
000168    0A 00                       OUT    SVC   0                        PROGRAM NORMAL STOP
00016A    0A 00                       ERROR  SVC   0                        ERROR STOP
00016C    000001EC                    AFIRST DC    A(RECORD)
000170    00000001                    ONE    DC    F'1'
000174    D2 00 F 099 E 000           MVCINS MVC   DESC(0),0(14)
00017A                                ACCT   DS    CL7
000181    404040                             DC    C'   '
000184                                QOH    DS    CL8
00018C    404040                             DC    C'   '
00018F                                DOLL   DS    CL8
000197    404040                             DC    C'   '
00019A    40                          BLANK  DC    C' '
00019B                                DESC   DS    CL60
0001C8                                TEMP1  DS    F
0001E0                                TEMP2  DS    D
0001E8                                TOTAL  DS    F
0001EC    C2C5E5C5D36B40C2D3          RECORD DC    C'BEVEL, BLUE, 6 INCH='
0001F5    E4C56B40F640C9D5C3
0001FE    C87E
000200    F1F2F3C1C2C3F4                     DC    C'123ABC4'
000207    000001CA                           DC    FL4'458'
00020B    000015CA                           DC    FL4'5578'
00020F    C1D5C7D3C56B40D9C5                 DC    C'ANGLE, RED, 8 INCH FORGED='
000218    C46B40F840C9D5C3C8
000221    40C6D6D9C7C5C47E
000229    F2F3F4E7E8E9F7                     DC    C'234XYZ7'
000230    00001F40                           DC    FL4'8000'
000234    000125C0                           DC    FL4'75200'
000238    C6D3C1D5C7C56B40F2                 DC    C'FLANGE, 2 INCH, SPRAYED PURPLE='
000241    40C9D5C3C86B40E2D7
00024A    D9C1E8C5C440D7E4D9
000253    D7D3C57E
000257    F7F5F3C7C8D1F8                     DC    C'753GHJ8'
00025E    0000000C                           DC    FL4'12'
000262    00001EB0                           DC    FL4'7856'
000266    7E                                 DC    C'='
000267    000000000000000000          TABLE  DC    X'0000000000000000000000000000000000000000'
000270    000000000000000000
000279    0000
00027B    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
000284    000000000000000000
00028D    0000
00028F    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
000298    000000000000000000
0002A1    0000
0002A3    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
0002AC    000000000000000000
0002B5    0000
0002B7    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
0002C0    000000000000000000
0002C9    0000
0002CB    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
0002D4    000000000000000000
0002DD    0000
0002DF    000000000000                       DC    X'000000000000'
0002E5    01                                 DC    X'01'
0002E6    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
0002EF    000000000000000000
0002F8    0000
0002FA    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
000303    000000000000000000
00030C    0000
00030E    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
000317    000000000000000000
000320    0000
000322    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
00032B    000000000000000000
000334    0000
000336    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
00033F    000000000000000000
000348    0000
00034A    000000000000000000                 DC    X'0000000000000000000000000000000000000000'
000353    000000000000000000
00035C    0000
00035E    000000000000000000                 DC    X'000000000000000000'
                                             END   BEGIN
```

Figure 90. A program to process variable-length blocked records. The block, read from tape by IOCS instructions not shown, contains a variable number of records. Each record is of variable length, with fixed-length fields at the end of each record. The program uses a Translate and Test instruction in a loop to detect sentinels.

]

ter 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.

This completes the actions needed to set up the line for printing. A Supervisor Call to the dump routine prints the line. There may still be another record in the block, so we branch back to AGAIN to see whether there is, and, if so, process.

At OUT we set up the total for printing and print it.

The sample block that appears at BLOCK 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. Such a block would have been set up by a previous program in real life. 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 modifier of 4 is, of course, the same as we would get with a fullword; the whole purpose is to prevent boundary alignment.

Summary

This chapter has presented three of the most powerful instructions in the System/360 repertoire, Edit, Translate and Execute, with their variations. If the wide applicability of these instructions is properly understood, it is possible to write programs that are executed quickly and take full advantage of the data processing capability of the system.

The extended examples of the chapter were presented to indicate the wide range of application of the instructions.

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 meaning:

Character	*Meaning*	*Hexadecimal Equivalent*
B	Blank	40
S	Significance-start character	21
D	Digit-select character	20
,	Comma	6B
.	Decimal	4B
C	C	C3
R	R	D9
*	*	5C
F	Field-separator character	22

```
1. WORK    BDDDDDD
   SOURCE  001540+
2. WORK    BDDDDDDDCR
   SOURCE  005721+
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 0092500.01– 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–. Name 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
e. ACBD

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    X'0A'
WORK    DC    CL16'1234567899123456'
AREA    DS    CL20
        LR    2,CON1
        MVI   AREA,C'0'
        MVC   AREA+1(19),AREA
        EX    2,MOVE
        BC    15,ROU2
MOVE    MVC   AREA(1),WORK
```

What will AREA contain after the instruction BC 15,ROU2 is executed?

13. What will AREA contain if the EX instruction were EX 0,MOVE?

Chapter 9: Subroutines and Subprograms

Subroutines and subprograms are frequently encountered in programming. They are used in some cases to conserve storage space when a routine at one location in memory can be called into operation from many other points. They are also used to conserve programming time and effort when an existing subroutine can be incorporated into a new program.

A subroutine is a segment of a complete program that is assembled at one time. A subprogram, on the other hand, is in object program form; it is called into operation by a main program that is not assembled with it. In fact, more than one subprogram may be loaded into storage with one main calling routine, which may actually be little but a sequence of calls to subprograms. This can become the organizing principle of the overall program. It also raises a question of how to handle subprograms that may originally have been assembled to load into the same core storage locations; this leads to the need for *relocation* of subprograms.

In this chapter we shall be concerned primarily with the general question of communication between a main calling routine and the subroutines and subprograms that it calls. This investigation will involve such questions as:

How does the subroutine know where to return to when it is finished?

How does the main program give the subroutine information about the location of data and results?

How is a program modified to make it operate correctly when it is relocated for execution from some place in storage other than where it was originally assembled?

How do we inform the assembler that certain symbolic addresses will be defined only when the assembled program is loaded with other programs?

Questions such as these, which are important in programming any computer, are answered in a powerful and flexible manner by the design of the System /360, particularly the base registers. In fact, the easy relocatability of programs provided by suitable use of base registers is one major advantage of the system.

The basic idea of a subroutine is to put it in storage at one place, then call it into action whenever its function is needed. If we are using a square root subroutine, for instance, we put it in one group of storage locations, available for use as needed. Then, at any point in the rest of the 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 routine from which the subroutine was called.

This raises two questions: How does the subroutine know where to go back to when its work is finished? How does the main routine provide the subroutine with information on the location of the number and its square root?

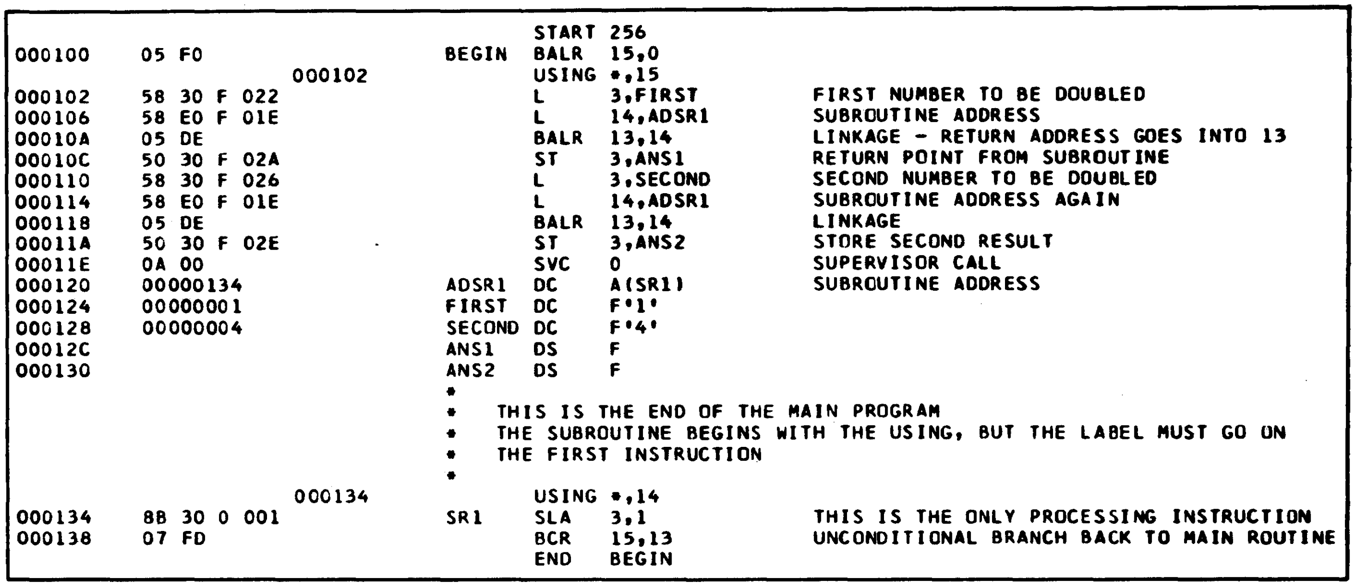
The question of where to return to is answered by a *linkage* that deposits in a register the address of the next instruction after the one that branches to the subroutine. In the System/360 we do this with the Branch and Link Register (BALR) instruction that we have seen so frequently for loading a base register; but now we specify some second operand other than zero, so that it really is a branch. The technique is to place in a register, say 14, the address of the first instruction of the subroutine. Then, if we have chosen register 13 to hold the link, we execute the instruction

```
    BALR    13,14
```

This says to place in register 13 the address of the next byte after the BALR, and branch to the address given in register 14. At the end of the subroutine it is merely necessary to execute an unconditional branch to the address in register 13. This is done with a Branch on Condition Register (BCR) instruction in which the mask is 15 to specify an unconditional branch.

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 choose an unrealistically simple job for the subroutine to do: double a number by shifting it left one place. We solve the problem of communicating data and result locations by an agreement between the main routine and the subroutine that the number to be doubled is to be placed in register 3 before branching to the subroutine, and that the doubled result is to be left in register 3 on the return to the main program.

Figure 91 is a listing of one program consisting of a main, or calling, routine and the subroutine. (We recall, from the Introduction, the distinction in terminology: a subroutine is assembled with its calling routine. If separate assemblies are done and the resulting programs used together, we have a sub*program.*)



```
                                               START 256
000100    05 F0                          BEGIN BALR  15,0
                      000102                   USING *,15
000102    58 30 F 022                          L     3,FIRST          FIRST NUMBER TO BE DOUBLED
000106    58 E0 F 01E                          L     14,ADSR1         SUBROUTINE ADDRESS
00010A    05 DE                                BALR  13,14            LINKAGE - RETURN ADDRESS GOES INTO 13
00010C    50 30 F 02A                          ST    3,ANS1           RETURN POINT FROM SUBROUTINE
000110    58 30 F 026                          L     3,SECOND         SECOND NUMBER TO BE DOUBLED
000114    58 E0 F 01E                          L     14,ADSR1         SUBROUTINE ADDRESS AGAIN
000118    05 DE                                BALR  13,14            LINKAGE
00011A    50 30 F 02E                          ST    3,ANS2           STORE SECOND RESULT
00011E    0A 00                                SVC   0                SUPERVISOR CALL
000120    00000134                       ADSR1 DC    A(SR1)           SUBROUTINE ADDRESS
000124    00000001                       FIRST DC    F'1'
000128    00000004                       SECOND DC   F'4'
00012C                                   ANS1  DS    F
000130                                   ANS2  DS    F
                                         *
                                         *  THIS IS THE END OF THE MAIN PROGRAM
                                         *  THE SUBROUTINE BEGINS WITH THE USING, BUT THE LABEL MUST GO ON
                                         *  THE FIRST INSTRUCTION
                                         *
                      000134                   USING *,14
000134    8B 30 0 001                    SR1   SLA   3,1              THIS IS THE ONLY PROCESSING INSTRUCTION
000138    07 FD                                BCR   15,13            UNCONDITIONAL BRANCH BACK TO MAIN ROUTINE
                                               END   BEGIN
```

Figure 91. A program containing a subroutine, showing a subroutine linkage

The first three instructions in Figure 91 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 to be doubled by the subroutine. Register 14 is now loaded with the address of the subroutine, using an address constant, in preparation for branching to the subroutine with the BALR. The BALR as written here takes its branch address from register 14 and places in register 13 the address of the next instruction, the Store.

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 in the main routine following the BALR. This address is precisely what is in register 13 now, so an unconditional branch to the address specified in 13 is the correct return. The BCR instruction is unconditional because of the 15 in the R_1 field.

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 14 with the address of the subroutine and linking to it. Although it is true that register 14 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. In realistic programs, it is all too easy to cause trouble by trying to save a few microseconds.

Figure 92 shows the values of FIRST, SECOND, ANSI, and ANS2, in that order, after the execution of the program. The dump that gave these results showed that at the completion of the program register 14 contained 134 (the address of the subroutine) and register 13 contained 11A (the address of the next instruction after the second BALR). In short, everything worked as we expected.

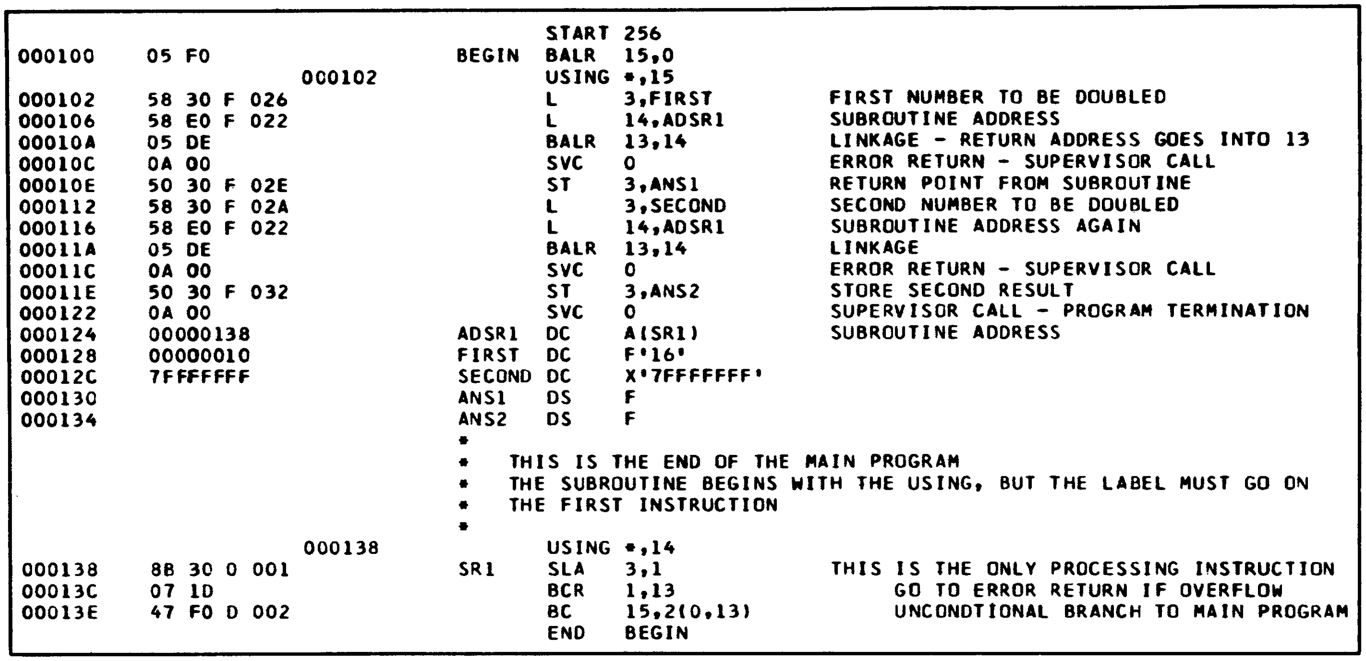
```
00000001 00000004 00000002 00000008
```

Figure 92. Values of FIRST, SECOND, ANS1, and ANS2, respectively, after execution of program in Figure 91

Let us now add a feature to the program. Shifting a number left can, of course, result in loss of a bit from large numbers. Let us suppose that such a loss would be an unexpected event, one that should be signaled back to the main routine as an error. The method of signaling is as follows. If such a loss of information occurs, the subroutine returns to the instruction after the BALR; if there is no loss of information, the subroutine returns to the instruction that is two bytes beyond the one after the BALR. In other words, the two-byte instruction after the BALR will be executed only in the error condition; this is called the *error return*. The *normal return* will skip past this.

We shall insert in the program, following each BALR to the subroutine, an SVC 0 instruction to discontinue the program if the error arises. (In practical applications, of course, there might be corrective action that could be taken, rather than giving up completely. The other actions might require space for a four-byte instruction at the error-return point; this could easily be arranged in the subroutine, as we shall see.)

The choice of whether to go back to the error return or the normal return is, of course, made by the subroutine. Figure 93 shows the modifications required. After shifting left, we execute a Branch on Condition Register instruction that, according to the setting of the condition code by the Shift Left Single instruction, determines whether a bit is lost. If so, the branch is taken, and the error return is reached. If not, we should like to go back to the normal return, which is two bytes beyond the address now standing in register

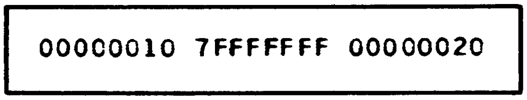


```
                                           START 256
000100  05 F0                    BEGIN     BALR  15,0
                        000102             USING *,15
000102  58 30 F 026                        L     3,FIRST        FIRST NUMBER TO BE DOUBLED
000106  58 E0 F 022                        L     14,ADSR1       SUBROUTINE ADDRESS
00010A  05 DE                              BALR  13,14          LINKAGE - RETURN ADDRESS GOES INTO 13
00010C  0A 00                              SVC   0              ERROR RETURN - SUPERVISOR CALL
00010E  50 30 F 02E                        ST    3,ANS1         RETURN POINT FROM SUBROUTINE
000112  58 30 F 02A                        L     3,SECOND       SECOND NUMBER TO BE DOUBLED
000116  58 E0 F 022                        L     14,ADSR1       SUBROUTINE ADDRESS AGAIN
00011A  05 DE                              BALR  13,14          LINKAGE
00011C  0A 00                              SVC   0              ERROR RETURN - SUPERVISOR CALL
00011E  50 30 F 032                        ST    3,ANS2         STORE SECOND RESULT
000122  0A 00                              SVC   0              SUPERVISOR CALL - PROGRAM TERMINATION
000124  00000138                 ADSR1     DC    A(SR1)         SUBROUTINE ADDRESS
000128  00000010                 FIRST     DC    F'16'
00012C  7FFFFFFF                 SECOND    DC    X'7FFFFFFF'
000130                           ANS1      DS    F
000134                           ANS2      DS    F
                                 *
                                 *   THIS IS THE END OF THE MAIN PROGRAM
                                 *   THE SUBROUTINE BEGINS WITH THE USING, BUT THE LABEL MUST GO ON
                                 *   THE FIRST INSTRUCTION
                                 *
                        000138             USING *,14
000138  8B 30 0 001              SR1       SLA   3,1            THIS IS THE ONLY PROCESSING INSTRUCTION
00013C  07 1D                              BCR   1,13              GO TO ERROR RETURN IF OVERFLOW
00013E  47 F0 D 002                        BC    15,2(0,13)        UNCONDTIONAL BRANCH TO MAIN PROGRAM
                                           END   BEGIN
```

Figure 93. A program containing a subroutine, showing a subroutine linkage with an error return

13. This is easily done with a Branch on Condition instruction that uses register 13 for a base register and has a displacement of 2.

Figure 94 shows the information dumped at the end of execution of the program, with the new values for FIRST and SECOND noted. A doubled value for SECOND has not been stored, since the error return was taken and the second Store instruction was never reached.



```
00000010 7FFFFFFF 00000020
```

Figure 94. Values of FIRST, SECOND, and ANS1, respectively, after execution of the program in Figure 93. ANS2 was not computed because the error return was taken for SECOND.

So far we have seen the linkage mechanism in action, with a variation that allows a choice between two return points. But there was a significant simplification in that the communication of data and results was handled by agreement on a register to be used for the purpose. This is probably not typical, and it is certainly not always acceptable. We need, rather, to be able to specify data and data addresses in some much more flexible manner. The most common such technique is to write the data and/or data addresses in the instruction stream immediately following the BALR, from which the subroutine can readily obtain them. Let us see, by means of an example, how this might be done. The example this time is representative of something that might actually be done with a subroutine. We shall be able to use this example through the rest of this publication, including the sections on subprograms.

We have in storage a group of fullwords in consecutive fullword locations. The list is described by its starting address and the number, *n*, of entries. These two parameters are to be communicated to the subroutine, along with the address at which it should store, after computing it, the average of the numbers.

There are several possible ways to give the necessary information to the subroutine. We choose one that is representative. Immediately following the BALR that branches to the subroutine, there will be a *calling sequence* giving, in order, the address of the first word of the list, the value of *n* as a fullword, and the address where the average should be stored. A typical calling sequence might be:

```
BALR     13,14
DC       A(LIST1)
DC       F'4'
DC       A(AVER1)
```

The subroutine will be required to pick up the information it needs from this calling sequence, which it can do since it has in register 13 the address of the first word after the BALR. The return from the subroutine will, of course, have to be to the instruction after the calling sequence, or twelve bytes beyond what is in register 13.

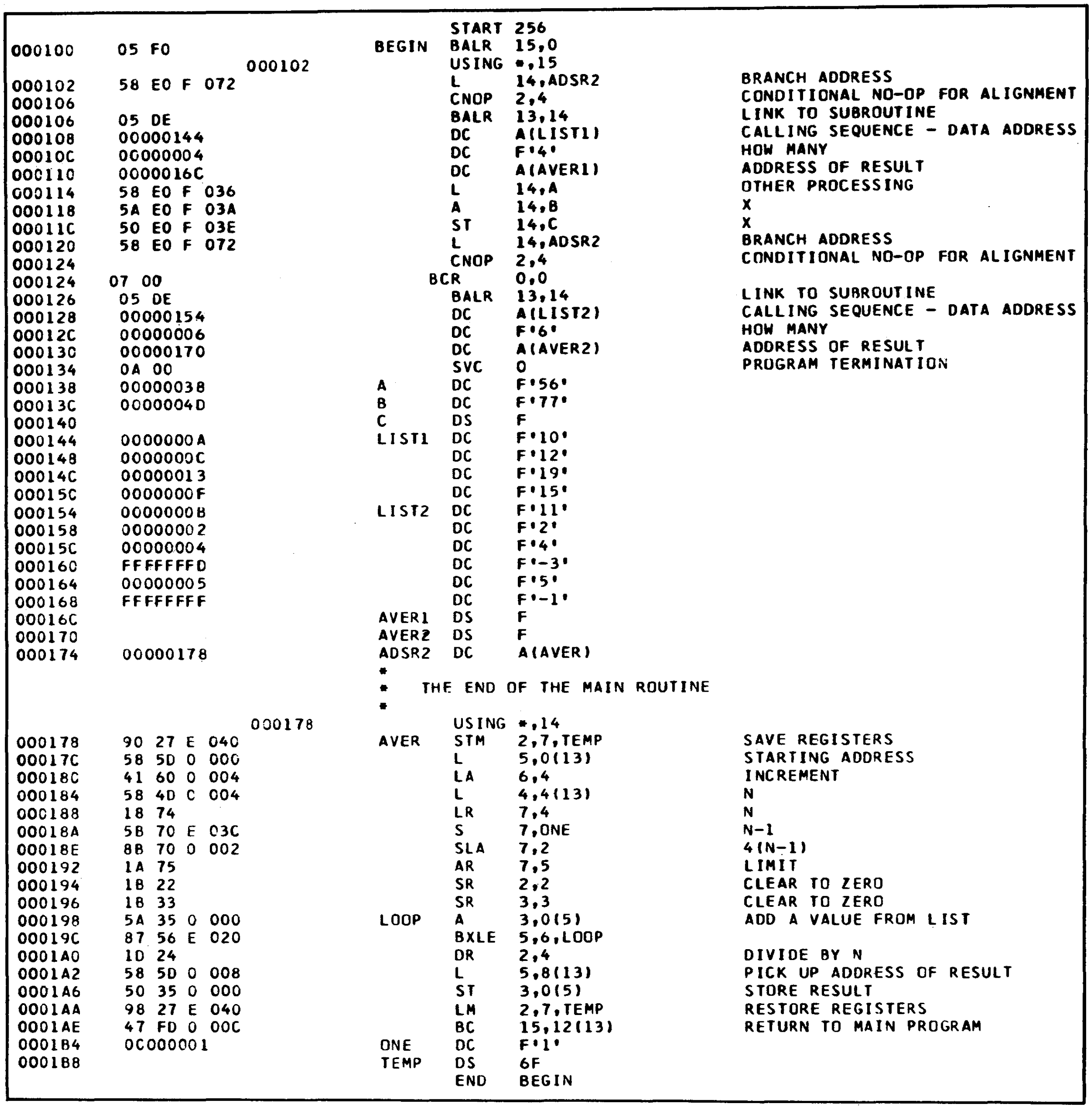
But what if the instruction before the BALR ended on a fullword boundary? Then the BALR (a two-byte instruction) would occupy the first two bytes of the next word. The assembler, since it automatically aligns on a fullword boundary an A-type constant for which no length is specified, would skip two bytes before locating the A-type constant. When the BALR is executed, register 13 would contain the address of the byte following the BALR instruction, but this address would not be the address of the first byte of the A-type constant. This would cause a problem because the subroutine counts on register 13 containing the address of that constant.

Solving this problem is the function of the assembler instruction Conditional No-Operation (CNOP). Just before the BALR we shall write the instruction

```
    CNOP    2,4
```

If, when the assembler reaches the BALR, the location counter is already set to a value that is two (2) greater than a fullword (4) boundary, the CNOP is ignored. If, on the other hand, this is not true, the assembler inserts a Branch on Condition (BCR) instruction with a mask of zero, which never causes a branch regardless of the condition code. This BCR instruction, then, is equivalent to a no-operation. Its presence will put the BALR where required to cause the calling sequence to be located immediately following the BALR.

Let us turn to the program in Figure 95 to see all this in context.



```
                                                      START 256
000100    05 F0                             BEGIN     BALR   15,0
                              000102                  USING *,15
000102    58 E0 F 072                                 L      14,ADSR2                 BRANCH ADDRESS
000106                                                CNOP   2,4                      CONDITIONAL NO-OP FOR ALIGNMENT
000106    05 DE                                       BALR   13,14                    LINK TO SUBROUTINE
000108    00000144                                    DC     A(LIST1)                 CALLING SEQUENCE - DATA ADDRESS
00010C    00000004                                    DC     F'4'                     HOW MANY
000110    0000016C                                    DC     A(AVER1)                 ADDRESS OF RESULT
000114    58 E0 F 036                                 L      14,A                     OTHER PROCESSING
000118    5A E0 F 03A                                 A      14,B                     X
00011C    50 E0 F 03E                                 ST     14,C                     X
000120    58 E0 F 072                                 L      14,ADSR2                 BRANCH ADDRESS
000124                                                CNOP   2,4                      CONDITIONAL NO-OP FOR ALIGNMENT
000124   07 00                                      BCR      0,0
000126    05 DE                                       BALR   13,14                    LINK TO SUBROUTINE
000128    00000154                                    DC     A(LIST2)                 CALLING SEQUENCE - DATA ADDRESS
00012C    00000006                                    DC     F'6'                     HOW MANY
000130    00000170                                    DC     A(AVER2)                 ADDRESS OF RESULT
000134    0A 00                                       SVC    0                        PROGRAM TERMINATION
000138    00000038                          A         DC     F'56'
00013C    0000004D                          B         DC     F'77'
000140                                      C         DS     F
000144    0000000A                          LIST1     DC     F'10'
000148    0000000C                                    DC     F'12'
00014C    00000013                                    DC     F'19'
00015C    0000000F                                    DC     F'15'
000154    0000000B                          LIST2     DC     F'11'
000158    00000002                                    DC     F'2'
00015C    00000004                                    DC     F'4'
000160    FFFFFFFD                                    DC     F'-3'
000164    00000005                                    DC     F'5'
000168    FFFFFFFF                                    DC     F'-1'
00016C                                      AVER1     DS     F
000170                                      AVER2     DS     F
000174    00000178                          ADSR2     DC     A(AVER)
                                            *
                                            *   THE END OF THE MAIN ROUTINE
                                            *
                              000178                  USING *,14
000178    90 27 E 040                       AVER      STM    2,7,TEMP                 SAVE REGISTERS
00017C    58 5D 0 000                                 L      5,0(13)                  STARTING ADDRESS
00018C    41 60 0 004                                 LA     6,4                      INCREMENT
000184    58 4D C 004                                 L      4,4(13)                  N
000188    18 74                                       LR     7,4                      N
00018A    5B 70 E 03C                                 S      7,ONE                    N-1
00018E    8B 70 0 002                                 SLA    7,2                      4(N-1)
000192    1A 75                                       AR     7,5                      LIMIT
000194    1B 22                                       SR     2,2                      CLEAR TO ZERO
000196    1B 33                                       SR     3,3                      CLEAR TO ZERO
000198    5A 35 0 000                       LOOP      A      3,0(5)                   ADD A VALUE FROM LIST
00019C    87 56 E 020                                 BXLE   5,6,LOOP
0001A0    1D 24                                       DR     2,4                      DIVIDE BY N
0001A2    58 5D 0 008                                 L      5,8(13)                  PICK UP ADDRESS OF RESULT
0001A6    50 35 0 000                                 ST     3,0(5)                   STORE RESULT
0001AA    98 27 E 040                                 LM     2,7,TEMP                 RESTORE REGISTERS
0001AE    47 FD 0 00C                                 BC     15,12(13)                RETURN TO MAIN PROGRAM
0001B4    0C000001                          ONE       DC     F'1'
0001B8                                      TEMP      DS     6F
                                                      END    BEGIN
```

Figure 95. A program containing a subroutine to compute the average of a list of numbers

After the usual preliminaries we load register 14 with the subroutine address. The Conditional No Op in this case has no effect; we see that the BALR is already located as described by the CNOP (that is, it starts on the second byte of a four-byte fullword), and the DC is thus on a fullword boundary without skipping any space between the BALR and the address. The starting address is given as LIST1, there are said to be four entries in the list, and the average should be stored at AVER1. At execution time, the BALR branches to the subroutine without, of course, trying to execute the calling sequence as instructions, which they are not.

In the subroutine we begin with a Store Multiple instruction that saves the contents of the registers that will be used by the subroutine. This is normal practice; it is almost never the case that any registers are assumed to be available to the subroutine without first saving their contents.

The thing the subroutine must do is to get the address of the first word of the list of numbers to be averaged. This is easily done by using a Load instruction in which the effective address is simply the contents of register 13. After execution of this Load, register 5 contains the address of the first word of the list. 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 4, the increment between loop repetitions. With register 6 containing the increment, register 7 must contain the final address. This is: the starting address, plus four times one less than the number of entries. We load register 4 with the contents of the fullword that is four bytes beyond the address in register 13; this is the second word of the calling sequence, giving the number of entries. It is to be left in register 4 for computing the average later. For the loop purposes we move it to register 7, subtract 1, shift left two places (in effect multiplying by four), and add the starting address. After clearing registers 2 and 3 we are ready to go into the loop.

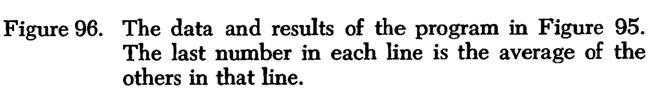
The Add instruction at LOOP uses as its address the contents of register 5, which is the index of the loop. Between loop repetitions, register 5 is incremented by the contents of register 6, which we set to be 4. The looping stops when all entries in the list have been added to register 3.

Now we are ready to compute the average, which is a simple 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) can be used to advantage. The quotient is the average, which is left 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 eight bytes beyond the contents of register 13; 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. We restore the registers that had been saved and branch back to the main routine. We re-enter the main routine just after the calling sequence because of the displacement of 12 in the Branch on Condition instruction.

Back in the main routine, we do some simple processing using register 14 to dramatize the fact that registers used for base registers and linkages have no special characteristics. Then we wish to average another list, which, as shown, requires a different calling sequence. This time, we note that the CNOP resulted in the creation of a No Operation instruction. If this had not been done, the BALR would have been placed at 124, thereby leaving a two-byte gap between the BALR and the first fullword of the calling sequence because the assembler aligns A-type DC's on fullword boundaries. The subroutine would know nothing but what the BALR told it by putting the address of the next byte after the BALR in register 13. The attempt to call for a fullword from 126 would cause a specification exception, leading to interruption of the program.

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.

Figure 96 shows the four numbers of the first list followed by the average as computed by this program, together with the six numbers of the second list and the corresponding average.



```
10   12   19   15         14
11   2    4    -3   5     -1         3
```

Figure 96. The data and results of the program in Figure 95. The last number in each line is the average of the others in that line.

Program Relocation

We now approach the important question of program relocation: how do we arrange to execute an assembled object program from some set of storage locations other than those assigned in assembly? This matter is important for a number of reasons. For one, it is sometimes necessary to change storage locations of assembled programs because of changes in storage requirements of other programs that are in core at the same time. Reassembly is a possible solution, of course, but we prefer a less time-consuming method, if one is available. For another reason, it is often a major convenience to be able to borrow a completed program written by some other programmer and use it in combination with other program segments. In many such instances there will be conflicts in core storage assignments that must be resolved by relocating some or all of the segments. Finally, and to many users most important, a machine run under control of a monitor program can have a constantly varying set of storage requirements, making necessary a job-by-job reallocation of storage locations.

For whatever reason, it is not uncommon to want to execute a program from locations other than the ones assumed in assembly without reassembling. The question to be explored in this section is: how do we devise a program-loading procedure that makes program relocation simple? In the following section we shall move on to the equally interesting question of communication among several programs, any or all of which may have been relocated in loading: how do they now know each others' locations? For now, however, we consider only the actions necessary to permit *one* program to operate correctly after it has been relocated.

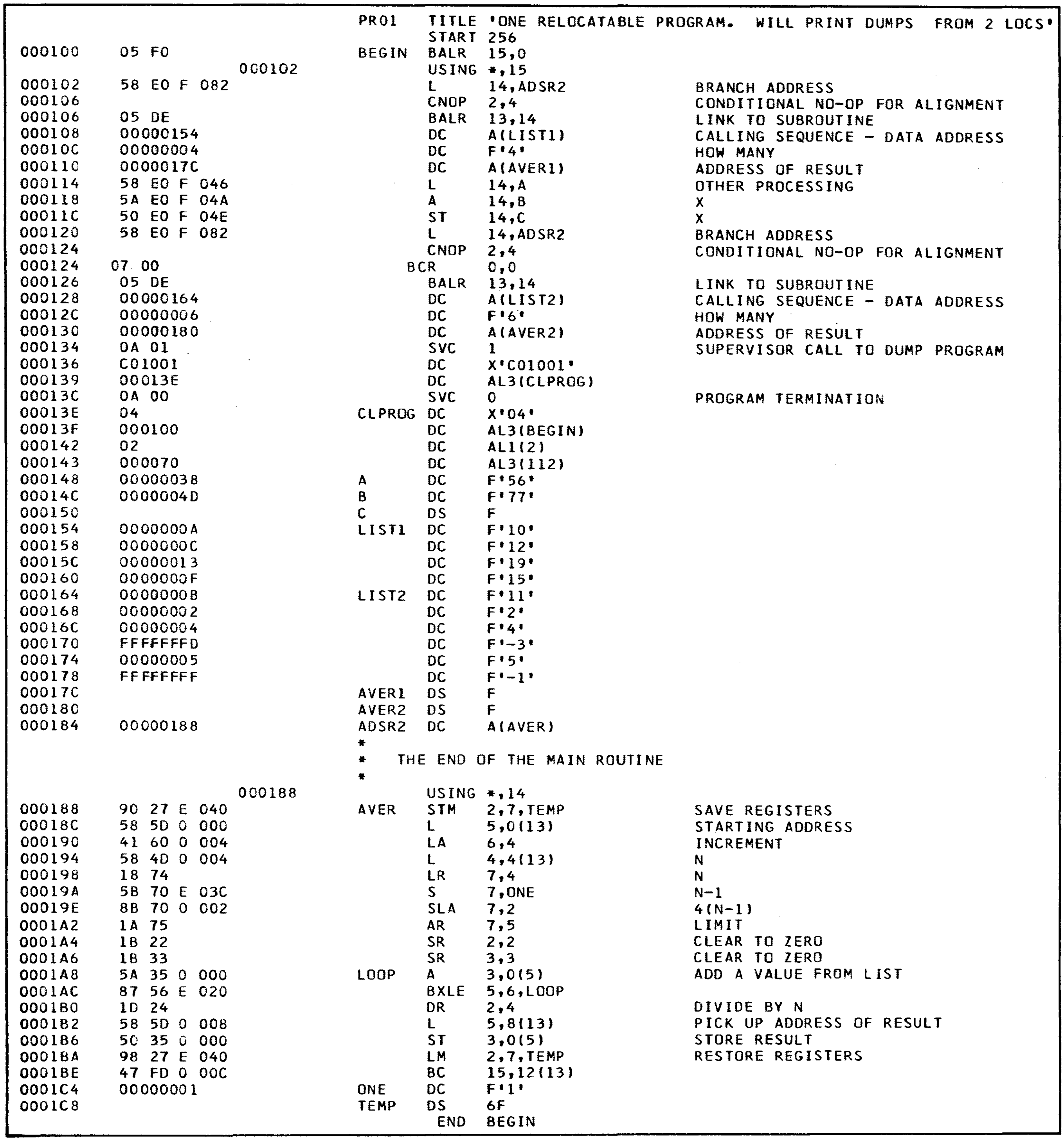
It is perhaps worthwhile to emphasize that throughout we are discussing an *assembled object program.* With a few important but quite special exceptions, the object program with which we are dealing is no longer in symbolic form. All addresses are in absolute form; that is, they are formed from actual displacements, base register contents, and index register contents. In order to emphasize this aspect of the problem, let us see how a program actually appears in core storage after it has been assembled and subsequently loaded.

Figure 97 is a slightly modified version of the program in Figure 95. The modification is the inclusion of a "dump call" after the conclusion of the main program. The Supervisor Call (SVC) with an operand of 1 causes activation of a program that is usually in core storage thereby enabling us to print out (dump) core storage information. This is not the place to give a complete description of the dump program, but we may at least describe briefly what the parts of this call do. (Note the similarity to the calling sequences above.)

The SVC with an operand of 1 starts the operation. The DC that follows is part of a calling sequence, in which we specify, in this case, that we wish to see the contents of the general registers but not the floating point registers, and that there is one "control list" elsewhere. In the DC that comes next, we give the address of the control list. Note the length modifier on the address-type DC; this permits us to pack more information into a fullword.

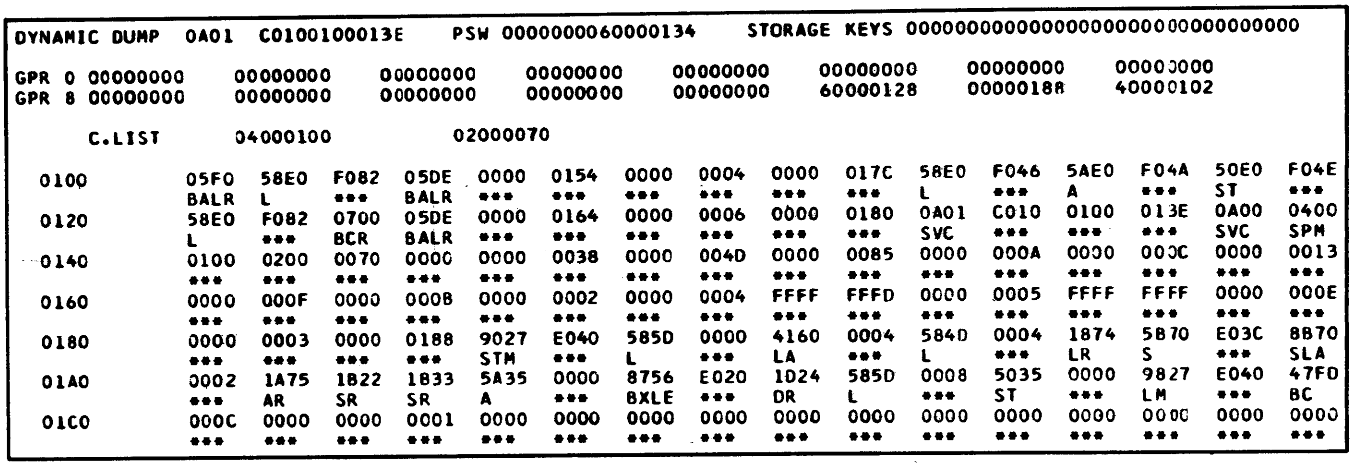
The control list appears in this case at the beginning of the constants, immediately after the SVC 0 that will terminate the program when the dump is finished. This control list says: print halfwords in hexadecimal, with mnemonic operation codes below any halfword that can be interpreted as an operation code. The first halfword to be printed is located at BEGIN, each halfword is two bytes long (this is not really needed in this case), and 112 halfwords are to be printed. This last number was chosen to cause printing of the entire program, including main program, constants, and subroutines.

Figure 98 is the dump produced when the object program was loaded and executed.



```
                                 PR01   TITLE 'ONE RELOCATABLE PROGRAM.  WILL PRINT DUMPS  FROM 2 LOCS'
                                        START 256
000100    05 F0                  BEGIN  BALR  15,0
                      000102            USING *,15
000102    58 E0 F 082                   L     14,ADSR2             BRANCH ADDRESS
000106                                  CNOP  2,4                  CONDITIONAL NO-OP FOR ALIGNMENT
000106    05 DE                         BALR  13,14                LINK TO SUBROUTINE
000108    00000154                      DC    A(LIST1)             CALLING SEQUENCE - DATA ADDRESS
00010C    00000004                      DC    F'4'                 HOW MANY
000110    0000017C                      DC    A(AVER1)             ADDRESS OF RESULT
000114    58 E0 F 046                   L     14,A                 OTHER PROCESSING
000118    5A E0 F 04A                   A     14,B                 X
00011C    50 E0 F 04E                   ST    14,C                 X
000120    58 E0 F 082                   L     14,ADSR2             BRANCH ADDRESS
000124                                  CNOP  2,4                  CONDITIONAL NO-OP FOR ALIGNMENT
000124   07 00                        BCR     0,0
000126    05 DE                         BALR  13,14                LINK TO SUBROUTINE
000128    00000164                      DC    A(LIST2)             CALLING SEQUENCE - DATA ADDRESS
00012C    00000006                      DC    F'6'                 HOW MANY
000130    00000180                      DC    A(AVER2)             ADDRESS OF RESULT
000134    0A 01                         SVC   1                    SUPERVISOR CALL TO DUMP PROGRAM
000136    C01001                        DC    X'C01001'
000139    00013E                        DC    AL3(CLPROG)
00013C    0A 00                         SVC   0                    PROGRAM TERMINATION
00013E    04                     CLPROG DC    X'04'
00013F    000100                        DC    AL3(BEGIN)
000142    02                            DC    AL1(2)
000143    000070                        DC    AL3(112)
000148    00000038               A      DC    F'56'
00014C    0000004D               B      DC    F'77'
000150                           C      DS    F
000154    0000000A               LIST1  DC    F'10'
000158    0000000C                      DC    F'12'
00015C    00000013                      DC    F'19'
000160    0000000F                      DC    F'15'
000164    0000000B               LIST2  DC    F'11'
000168    00000002                      DC    F'2'
00016C    00000004                      DC    F'4'
000170    FFFFFFFD                      DC    F'-3'
000174    00000005                      DC    F'5'
000178    FFFFFFFF                      DC    F'-1'
00017C                           AVER1  DS    F
000180                           AVER2  DS    F
000184    00000188               ADSR2  DC    A(AVER)
                                 *
                                 *   THE END OF THE MAIN ROUTINE
                                 *
                      000188            USING *,14
000188    90 27 E 040            AVER   STM   2,7,TEMP             SAVE REGISTERS
00018C    58 5D 0 000                   L     5,0(13)              STARTING ADDRESS
000190    41 60 0 004                   LA    6,4                  INCREMENT
000194    58 4D 0 004                   L     4,4(13)              N
000198    18 74                         LR    7,4                  N
00019A    5B 70 E 03C                   S     7,ONE                N-1
00019E    8B 70 0 002                   SLA   7,2                  4(N-1)
0001A2    1A 75                         AR    7,5                  LIMIT
0001A4    1B 22                         SR    2,2                  CLEAR TO ZERO
0001A6    1B 33                         SR    3,3                  CLEAR TO ZERO
0001A8    5A 35 0 000            LOOP   A     3,0(5)               ADD A VALUE FROM LIST
0001AC    87 56 E 020                   BXLE  5,6,LOOP
0001B0    1D 24                         DR    2,4                  DIVIDE BY N
0001B2    58 5D 0 008                   L     5,8(13)              PICK UP ADDRESS OF RESULT
0001B6    50 35 0 000                   ST    3,0(5)               STORE RESULT
0001BA    98 27 E 040                   LM    2,7,TEMP             RESTORE REGISTERS
0001BE    47 FD 0 00C                   BC    15,12(13)
0001C4    00000001               ONE    DC    F'1'
0001C8                           TEMP   DS    6F
                                         END  BEGIN
```

Figure 97. A modified version of the program in Figure 95, changed to include a calling sequence to dump the program after execution



```
DYNAMIC DUMP  0A01  C0100100013E    PSW 0000000060000134    STORAGE KEYS 00000000000000000000000000000000

GPR 0 00000000    00000000    00000000    00000000    00000000    00000000    00000000    00000000
GPR 8 00000000    00000000    00000000    00000000    00000000    60000128    0000018B    40000102

     C.LIST       04000100          02000070

 0100        05F0  58E0  F082  05DE  0000  0154  0000  0004  0000  017C  58E0  F046  5AE0  F04A  50E0  F04E
             BALR  L     ***   BALR  ***   ***   ***   ***   ***   ***   L     ***   A     ***   ST    ***
 0120        58E0  F082  0700  05DE  0000  0164  0000  0006  0000  0180  0A01  C010  0100  013E  0A00  0400
             L     ***   BCR   BALR  ***   ***   ***   ***   ***   ***   SVC   ***   ***   ***   SVC   SPM
 0140        0100  0200  0070  0000  0000  0038  0000  004D  0000  0085  0000  000A  0000  000C  0000  0013
             ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
 0160        0000  000F  0000  000B  0000  0002  0000  0004  FFFF  FFFD  0000  0005  FFFF  FFFF  0000  000E
             ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
 0180        0000  0003  0000  0188  9027  E040  585D  0000  4160  0004  584D  0004  1874  5B70  E03C  8B70
             ***   ***   ***   ***   STM   ***   L     ***   LA    ***   L     ***   LR    S     ***   SLA
 01A0        0002  1A75  1B22  1B33  5A35  0000  8756  E020  1D24  585D  0008  5035  0000  9827  E040  47FD
             ***   AR    SR    SR    A     ***   BXLE  ***   DR    L     ***   ST    ***   LM    ***   BC
 01C0        000C  0000  0000  0001  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000
             ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
```

Figure 98. The dump produced when the object program in Figure 97 was executed

The line that begins DYNAMIC DUMP gives certain information about the machine at the time the dump was requested, after which we have the 16 general register contents printed in hexadecimal. C. LIST means control list; this is a condensed form of the information we used to describe the dump we wanted.

Next comes the storage dump proper. The number at the left of each line gives the address of the first halfword on that line. We see that at 100 the halfword was 05F0; under that we have BALR, since 05 is the actual machine operation code for Branch and Link Register, which is indeed the instruction at 100. Next comes a halfword consisting of 58E0. The mnemonic operation code is L for Load. Next is a halfword F082. There is no such operation code as F0 in the System/360, so the dump program has printed asterisks.

Now we are finally ready to attack the question posed at the beginning of this section: What would we have to do to execute this program from some starting location other than 100?

Suppose it were desired to load it starting at 1100_{16}. The simplest part of the job is telling the loading program the new starting location, which is done with a control card that specifies the new loading address. In the system used in preparing this program, the control card was called a BASE card; this may vary in different installations.

Now, what does the loader have to do to ensure that the program will operate correctly? Remarkably little, it turns out, Consider what happens. The loader places the program into consecutive locations starting at 1100, carries out another action that we shall discuss in detail shortly, and then transfers control to the instruction named on our END card. That, of course, is the BALR. We recall that the BALR places in the named register ($F_{16}=15_{10}$) the address of the next byte after the BALR. The two-byte instruction BALR itself was loaded at 1100, so the address of the next byte after the BALR is 1102; this goes into register 15 as the base for the program.

Now the Load asks for a storage reference. The effective address is formed from the contents of register 15 (1102) and the displacement: $1102 + 08A = 118C$. We have correctly reached the constant desired, which, in the relocated program, is indeed at 118C. It thus appears that addresses that use base registers will all be correct in the new locations, since the base register contents have been modified to reflect the new location of the program. This is indeed true and is one of the major advantages of the base register idea in the System/360.

However, there is a little more work to be done. Not all addresses in our program *are* formed with base registers. Addresses in address-type DC instructions, of which we have a number, are not. As a matter of fact, the constant obtained by the Load instruction we were just discussing was an address constant giving the address of the subroutine. The constant we loaded said that the subroutine was located in 188; this is no longer true. And in the calling sequence to the subroutine we said that the first word of the list to be averaged was in 154, which is also wrong now.

The solution to this problem is simple. What we need to do is to add to each such address constant the difference between the new loading address and the one assumed in the assembly. This number, which is called the *relocation constant* for the program, would be just 1000 in this example: 1100 minus 100. All we need is a list of the addresses of address constants that should be modified. With such a list available to the loader, the relocation constant can be added to each address constant. This is done after the object program has been loaded.

Now that we know what the loader of the object program has to be able to do, let us investigate the makeup of the object program itself to see how the information needed by the loader is supplied.

The sequence of cards in a complete object program for this example is as follows:

1. A Starting-Location card, which we supply, giving the desired starting address of the loaded program.

2. An External Symbol Dictionary card (ESD), type 1. This gives the name of the program, if there is one, and the starting address that was assumed by the assembler. The address will be needed in computing the relocation constant. The ESD card is produced automatically by the assembler, on the basis of the value given in the START card. If there is no START card, which is permissible, zero is used.

3. A series of Text (TXT) cards containing the object program. Each card states the starting address of the sequence of bytes contained on that card, and the number of bytes.

4. The Relocation List Dictionary card (RLD). This gives the following information for each address constant in the program:

a. Its address in the program as originally assembled

b. Its length in bytes (address constants are most commonly four bytes but may be shorter)

c. A flag if the value is in complement form, indicating that the relocation constant should be subtracted

With this much information, together with the new and old starting addresses of the program, the loader can correct all address constants.

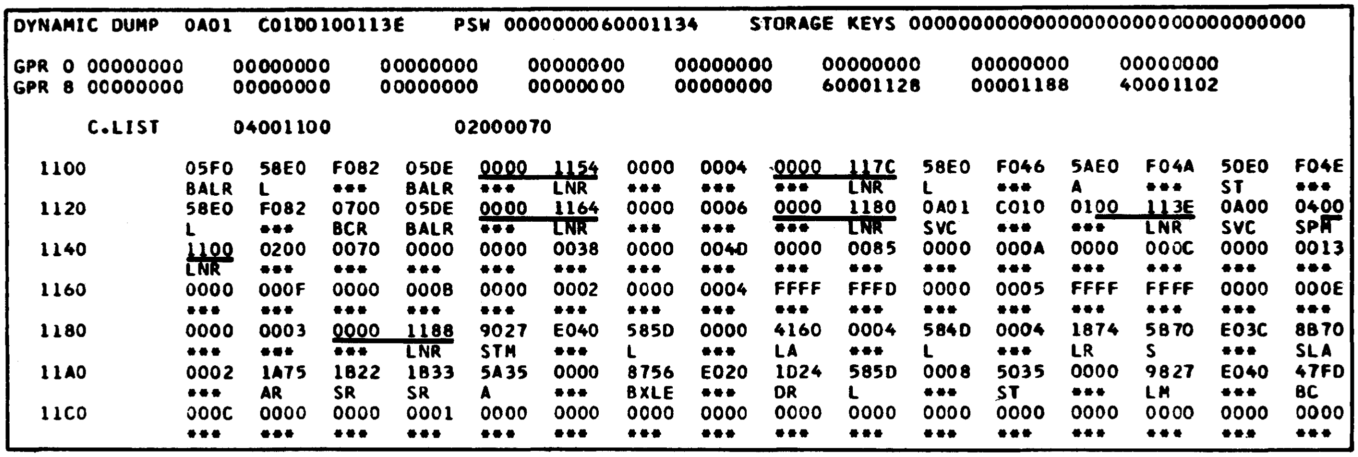
5. A Load END card. This is not the same as the END assembler instruction card in the source program, of course, although it is automatically produced from it by the assembler program. If the END instruction in the source program stated an entry point, the address of that entry point is contained in the loader end card, and control is later transferred to this location. If the assembler END had no operand address, the first byte of the program is taken to be the starting point.

6. A Load Terminate card (LDT) is supplied by us to indicate that the complete object program deck has been loaded and that program execution should begin.

The exact formats of these various cards are given in appropriate reference manuals. They are moderately complex, and many programmers will not find it necessary to be able to read these cards, so we pass over the subject.

We may now load the object program deck described above, execute it, and see what happens. Figure 99 is a printout of the dump after execution. We have underlined the values that are different from the corresponding position in the dump of Figure 98; all the differences are in address constants. A comparison will show that all the underlined address constants are precisely 1000 greater than those in Figure 98. All other values, including the computed results, are the same.

Thus have we solved the relocation problem, for a single program. In the remainder of this publication we shall have to study what happens when *several* programs are relocated, all by different amounts in general, and they have to be able to communicate with each other.



```
DYNAMIC DUMP  0A01  C0100100113E   PSW 0000000060001134    STORAGE KEYS 00000000000000000000000000000000

GPR 0 00000000   00000000   00000000   00000000   00000000   00000000   00000000   00000000
GPR 8 00000000   00000000   00000000   00000000   00000000   60001128   00001188   40001102

     C.LIST        04001100           02000070

 1100          05F0  58E0  F082  05DE  0000  1154  0000  0004  0000  117C  58E0  F046  5AE0  F04A  50E0  F04E
               BALR  L     ***   BALR  ***   LNR   ***   ***   ***   LNR   L     ***   A     ***   ST    ***
 1120          58E0  F082  0700  05DE  0000  1164  0000  0006  0000  1180  0A01  C010  0100  113E  0A00  0400
               L     ***   BCR   BALR  ***   LNR   ***   ***   ***   LNR   SVC   ***   ***   LNR   SVC   SPM
 1140          1100  0200  0070  0000  0000  0038  0000  0040  0000  0085  0000  000A  0000  000C  0000  0013
               LNR   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
 1160          0000  000F  0000  000B  0000  0002  0000  0004  FFFF  FFFD  0000  0005  FFFF  FFFF  0000  000E
               ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
 1180          0000  0003  0000  1188  9027  E040  585D  0000  4160  0004  584D  0004  1874  5870  E03C  8B70
               ***   ***   ***   LNR   STM   ***   L     ***   LA    ***   L     ***   LR    S     ***   SLA
 11A0          0002  1A75  1B22  1B33  5A35  0000  8756  E020  1D24  585D  0008  5035  0000  9827  E040  47FD
               ***   AR    SR    SR    A     ***   BXLE  ***   DR    L     ***   ST    ***   LM    ***   BC
 11C0          000C  0000  0000  0001  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000
               ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
```

Figure 99. The dump produced when the object program in Figure 97 was relocated and executed. The program in this example is loaded beginning at 1100. The underlined values are the only differences between the values in this dump and those in Figure 98.

Communication Between Relocated Segments

The preceding example has 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 keep track of addresses within each other as they are both relocated, in general by different amounts?

Let us investigate this question in terms of the illustrative program of the preceding sections. This time, however, we shall assemble the main calling program and the subroutine separately. Out of the two assemblies we shall get two object program decks, 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.

Let us look first at what problems we have created in assembling the main program. The biggest problem is that we seem to have created an undefined symbol: AVER is used as an address constant in the main program, but it is, of course, not defined in the main program since AVER is the name of the subroutine. If we simply took the main program part of Figure 97 and assembled it, the assembly would not be completed because of the undefined symbol.

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 a zero and mark the location as one that will be supplied during the loading of the object program."

This is precisely what the assembler instruction EXTRN does. Immediately after the START instruction we place the EXTRN, naming AVER in the operand field and leaving 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 100 is the assembly listing of the main program. An assembler TITLE instruction has been used in order to get an identification (in this case, MAIN) into the object deck in columns 73-76, thus distinguishing this object deck from other object decks. Just after the START is the EXTRN. Nothing is printed on the listing to describe the action of the EXTRN; this action really takes place only when the named symbol is encountered.

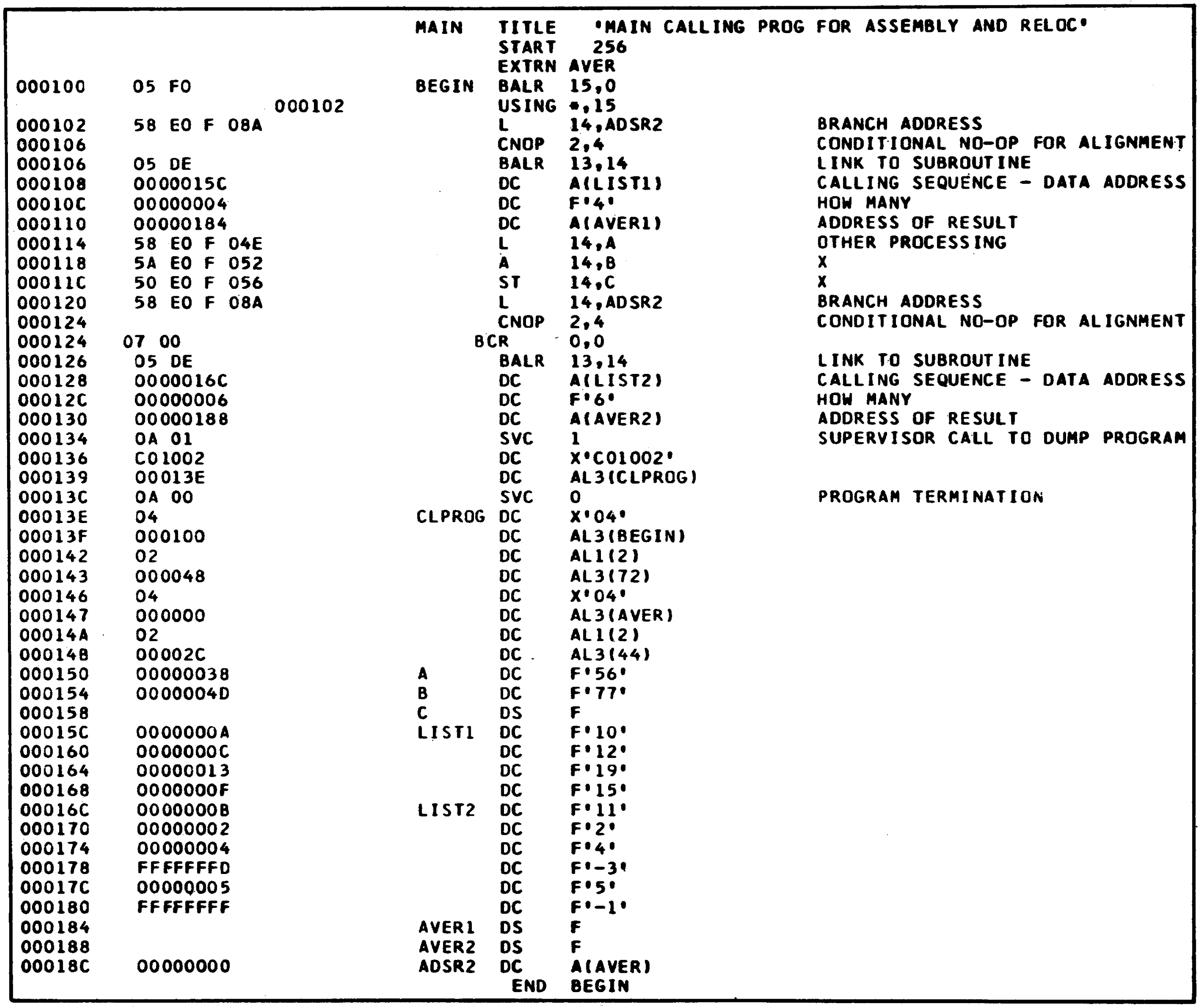
Scanning down the listing we see that things are just about the same as in the combined program of Figure 97, with the exception of the two address constants for AVER, where zeros have been assembled.

There is one rather minor change in the program. Now that the main program and the subroutine will be located in different parts of storage, we will want to be able to dump them separately. There are, accordingly, two control lists in the Supervisor Call reference. One asks for the main program to be printed, starting at the symbol BEGIN and continuing for a total of 72 halfwords. The other starts dumping at AVER and prints a total of 44 halfwords. This creates another address constant reference to AVER. This address constant, incidentally, is three bytes long rather than the more common four; we saw before that the RLD card will contain a note of this fact so that the relocation can be done correctly.

An inspection of the object program, which we shall not describe in detail, shows only one major change: another ESD card has been produced. This one gives the name of the symbol, AVER, and an identification number corresponding to it. Then, in the RLD card, the entries for AVER refer to the identification number. By the time the loader relocates these address constants, it will have found a numerical equivalent for AVER.

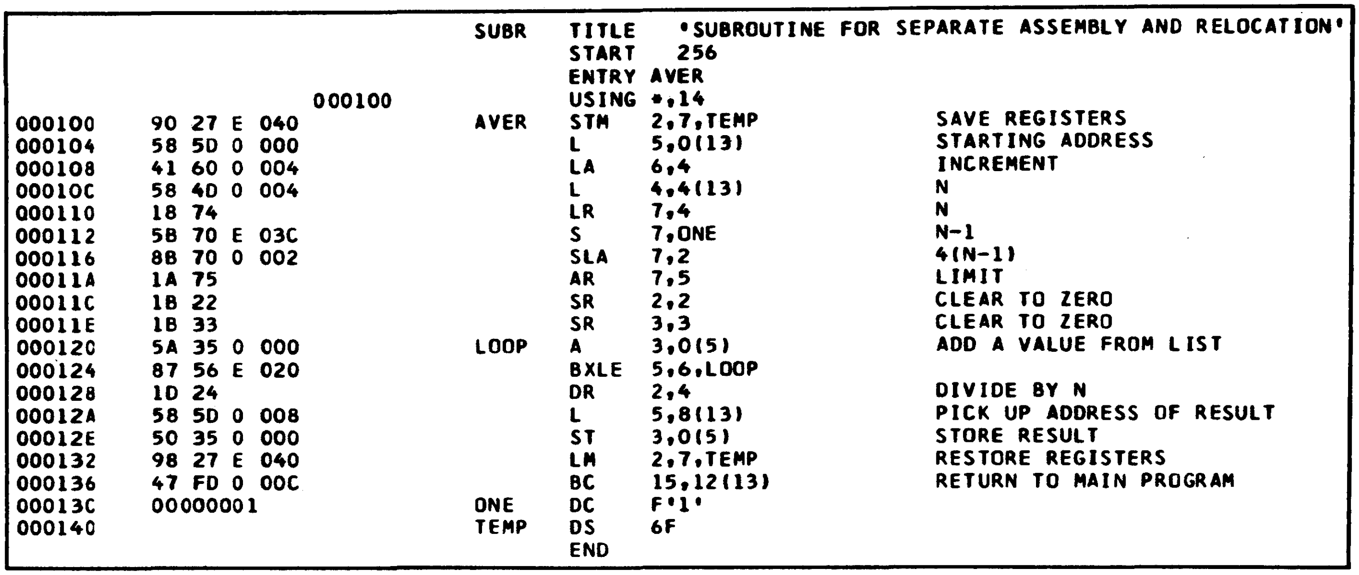
So much for the main program, in which there are references to symbols defined elsewhere. What about programs in which there are symbols that are defined here but used elsewhere, such as the symbol AVER in the subroutine? In a separate assembly of the subroutine there are no references to AVER; furthermore, there are no address constants: If the subroutine were assembled just as it is, there would be nothing to indicate to the assembler (and later to the loader) that there was anything special about AVER. Actually, of course, there *is* something special: this symbol is used in the loading process to supply the missing information in the main program. The assembler, however, cannot know this without explicit notification, because we are not assembling the two programs at the same time.

The answer is the assembler instruction ENTRY, which says that the symbol named in the operand field is used by some other program segment in refer-



```
                                            MAIN    TITLE   'MAIN CALLING PROG FOR ASSEMBLY AND RELOC'
                                                    START   256
                                                    EXTRN   AVER
000100    05 F0                             BEGIN   BALR    15,0
                          000102                    USING   *,15
000102    58 E0 F 08A                               L       14,ADSR2                 BRANCH ADDRESS
000106                                              CNOP    2,4                      CONDITIONAL NO-OP FOR ALIGNMENT
000106    05 DE                                     BALR    13,14                    LINK TO SUBROUTINE
000108    0000015C                                  DC      A(LIST1)                 CALLING SEQUENCE - DATA ADDRESS
00010C    00000004                                  DC      F'4'                     HOW MANY
000110    00000184                                  DC      A(AVER1)                 ADDRESS OF RESULT
000114    58 E0 F 04E                               L       14,A                     OTHER PROCESSING
000118    5A E0 F 052                               A       14,B                     X
00011C    50 E0 F 056                               ST      14,C                     X
000120    58 E0 F 08A                               L       14,ADSR2                 BRANCH ADDRESS
000124                                              CNOP    2,4                      CONDITIONAL NO-OP FOR ALIGNMENT
000124   07 00                                    BCR     0,0
000126    05 DE                                     BALR    13,14                    LINK TO SUBROUTINE
000128    0000016C                                  DC      A(LIST2)                 CALLING SEQUENCE - DATA ADDRESS
00012C    00000006                                  DC      F'6'                     HOW MANY
000130    00000188                                  DC      A(AVER2)                 ADDRESS OF RESULT
000134    0A 01                                     SVC     1                        SUPERVISOR CALL TO DUMP PROGRAM
000136    C01002                                    DC      X'C01002'
000139    00013E                                    DC      AL3(CLPROG)
00013C    0A 00                                     SVC     0                        PROGRAM TERMINATION
00013E    04                                CLPROG  DC      X'04'
00013F    000100                                    DC      AL3(BEGIN)
000142    02                                        DC      AL1(2)
000143    000048                                    DC      AL3(72)
000146    04                                        DC      X'04'
000147    000000                                    DC      AL3(AVER)
00014A    02                                        DC      AL1(2)
00014B    00002C                                    DC      AL3(44)
000150    00000038                          A       DC      F'56'
000154    0000004D                          B       DC      F'77'
000158                                      C       DS      F
00015C    0000000A                          LIST1   DC      F'10'
000160    0000000C                                  DC      F'12'
000164    00000013                                  DC      F'19'
000168    0000000F                                  DC      F'15'
00016C    0000000B                          LIST2   DC      F'11'
000170    00000002                                  DC      F'2'
000174    00000004                                  DC      F'4'
000178    FFFFFFFD                                  DC      F'-3'
00017C    00000005                                  DC      F'5'
000180    FFFFFFFF                                  DC      F'-1'
000184                                      AVER1   DS      F
000188                                      AVER2   DS      F
00018C    00000000                          ADSR2   DC      A(AVER)
                                                     END    BEGIN
```

Figure 100. The main program portion of the program in Figure 97, separately assembled



```
                                          SUBR    TITLE  'SUBROUTINE FOR SEPARATE ASSEMBLY AND RELOCATION'
                                                  START   256
                                                  ENTRY  AVER
                              000100              USING  *,14
000100     90 27 E 040                    AVER    STM    2,7,TEMP                SAVE REGISTERS
000104     58 5D 0 000                            L      5,0(13)                 STARTING ADDRESS
000108     41 60 0 004                            LA     6,4                     INCREMENT
00010C     58 4D 0 004                            L      4,4(13)                 N
000110     18 74                                  LR     7,4                     N
000112     5B 70 E 03C                            S      7,ONE                   N-1
000116     8B 70 0 002                            SLA    7,2                     4(N-1)
00011A     1A 75                                  AR     7,5                     LIMIT
00011C     1B 22                                  SR     2,2                     CLEAR TO ZERO
00011E     1B 33                                  SR     3,3                     CLEAR TO ZERO
000120     5A 35 0 000                    LOOP    A      3,0(5)                  ADD A VALUE FROM LIST
000124     87 56 E 020                            BXLE   5,6,LOOP
000128     1D 24                                  DR     2,4                     DIVIDE BY N
00012A     58 5D 0 008                            L      5,8(13)                 PICK UP ADDRESS OF RESULT
00012E     50 35 0 000                            ST     3,0(5)                  STORE RESULT
000132     98 27 E 040                            LM     2,7,TEMP                RESTORE REGISTERS
000136     47 FD 0 00C                            BC     15,12(13)               RETURN TO MAIN PROGRAM
00013C     00000001                       ONE     DC     F'1'
000140                                    TEMP    DS     6F
                                                  END
```

Figure 101. The subroutine portion of the program in Figure 97, separately assembled

ring to this one. The typical reference is as a branch address, hence the name ENTRY, but the usage is actually not so restricted.

Figure 101 is the assembly listing for the separately assembled subroutine. There is nothing exceptional about it. The object deck contains a third type of ESD card to convey to the loader the name of the symbol so that an actual numerical equivalent can be computed at load time and thus supplied to the program segments in which EXTRN instructions said it would be needed. There is no RLD card in this deck, because there are no address constants.

Now we may load the two segments and execute them. A loading location of 2000_{16} was supplied for the subroutine and 3000_{16} for the main program. When the program consisting of the two relocated segments was run, the two listings in Figures 102 and 103 were produced.

We see that where the main program referred to AVER, at 3047 and 308C, the loader supplied 2000_{16}, which is the correct value for AVER since the subroutine was, in fact, loaded at 2000_{16}. The various address constants that make references within the main program have all been handled correctly.

This program has illustrated important concepts. The most fundamental idea is that with a proper combination of notifications to the assembler, through the ENTRY and EXTRN statements, we can arrange for programs to communicate correctly with each other even though any or all of the segments may be relocated by various amounts.

There is no restriction in this scheme to just two program segments, nor to one symbol per program, or to branch addresses. A program can refer to many external symbols and/or have many entry points from other programs.

In our example, we arranged the object programs so that the main program was loaded after the subroutine. This meant that the symbol AVER had a value by the time the main program was loaded. With some loading programs this is mandatory, but the more sophisticated loaders can "stack" requests for values of symbols that have not yet been defined, and relocate such references later.

The symbol AVER was named in an ENTRY statement. Strictly speaking, this was not necessary because AVER, as it happens, was the name of a program. It was thus already named on the first type of ESD card. The duplication is no disadvantage, however, and may even have some value in helping the programmer to keep straight the interrelationships among program segments.

```
3000     05F0  58E0  F08A  05DE  0000  305C  0000  0004  0000  3084  58E0  F04E  5AE0  F052  50E0  F056
         BALR  L     ***   BALR  ***   LPER  ***   ***   ***   LPER  L     ***   A     ***   ST    ***
3020     58E0  F08A  0700  05DE  0000  306C  0000  00Q6  0000  3088  0A01  C010  0200  303E  0A00  0400
         L     ***   BCR   BALR  ***   LPER  ***   ***   ***   LPER  SVC   ***   ***   LPER  SVC   SPM
3040     3000  0200  0048  0400  2000  0200  002C  Q000  0000  0038  0000  004D  0000  0085  0000  000A
         LPER  ***   ***   SPM   LPDR  ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
3060     0000  000C  0000  0013  0000  000F  0000  000B  0000  0002  0000  0004  FFFF  FFFD  0000  0005
         ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
3080     FFFF  FFFF  0000  000E  0000  0003  0000  2000
         ***   ***   ***   ***   ***   ***   ***   LPDR
```

Figure 102. The dump of the main program in Figure 100, after the main program was relocated at 3000_{16} and the subroutine at 2000_{16}

```
2000     9027  E040  585D  0000  4160  0004  584D  0004  1874  5B70  E03C  8B70  0002  1A75  1B22  1B33
         STM   ***   L     ***   LA    ***   L     ***   LR    S     ***   SLA   ***   AR    SR    SR
2020     5A35  0000  8756  E020  1D24  585D  0008  5035  0000  9827  E040  47FD  000C  0000  0000  0001
         A     ***   BXLE  ***   DR    L     ***   ST    ***   LM    ***   BC    ***   ***   ***   ***
2040     0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000
         ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***   ***
```

Figure 103. The dump of the subroutine in Figure 101, after the main program was relocated at 3000_{16} and the subroutine at 2000_{16}

Summary

Subroutines which can be called into action from many places in a calling program are common programming techniques. We have shown one general form of calling sequence for making the transfer to the subroutine, providing the return link, and identifying necessary values and addresses.

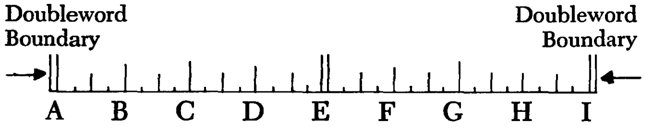
When a subroutine is not assembled as a part of another program, it is called a subprogram. We have explored briefly the basic issue of how two program segments can communicate even though, at the time each segment is assembled, the location of neither of them is known.

This capability of relocating programs with little effort either in original programming or in operator actions becomes a major feature of computer application. It is one of the important advantages of the System/360 that the base registers concept provides for such simplicity.

It must be said, however, that this treatment leaves some challenges for the more advanced student. In particular, when there are a great many subroutine and/or subprogram calls, typically with many programmers working on different parts of the same large task, it becomes mandatory to establish rather rigid conventions on register usage. In fact, the programming systems for System/360 specify conventions for the use of certain general registers (see the appropriate SRL publication for details of these conventions).

Questions and Exercises

1. The second operand of a CNOP instruction *may* specify that the instruction that follows is to be aligned within a doubleword (8) or a fullword (4) area. The first operand of the CNOP specifies that the location counter is to be set to 0 or 2 bytes past a fullword boundary, or 0, 2, 4, or 6 bytes past a doubleword boundary. Thus, CNOP 0,8 would specify that the next instruction is to be aligned at a doubleword boundary. If



is a representation of two doubleword areas, of the names we shall use to refer to various points within the areas, and the location counter stands at a point equal to point C:

   a. With what point would CNOP 2,8 align the beginning of the next instruction?
   b. CNOP 6,8
   c. CNOP 0,4
   d. CNOP 0,8
   e. CNOP 2,4

2. What kind of an operation does BCR 0,0 cause?

3. Assume subprograms CALLER and CALLED. CALLER is to enter CALLED at an instruction in CALLED that is named ROUT1. Write the necessary statements in both subprograms to properly define the name (ROUT1) to both subprograms.

4. Add the necessary statements in CALLER and CALLED to branch to CALLED (via register 13) leaving in register 14 the address of the next location in CALLER.

5. Consider the following program. Note that the locations, effective addresses, and object instruction base register specification and displacement are written in hexadecimal. Assume that this program is relocated and loaded starting with location 4000_{16} instead of 1000_{16}. In the spaces provided fill in:

   a. The location into which the instructions and data will actually be loaded.

   b. The relocation constant.

   c. The constants that will be assembled at BASE1 and BASE2 at assembly time.

6. What will be the contents of BASE1 and BASE2 at the end of the loading process (after the addition of the relocation constant)?

7. Fill in the values loaded into registers 13, 14, and 15 at execution time.

8. Fill in the effective addresses developed at execution time for each encircled operand.

Relocation Constant

	Value Assumed by Assembler	Value Loaded at Execution Time	
		Program loaded at 1000_{16}	Program loaded at 4000_{16}
Reg 15	1002	1002	
Reg 14	2002	2002	
Reg 13	3002	3002	

Location (relocated) at 4000_{16}	Location	Object Instruction		Execution Time Effective Address				
		Base Register	Displace-ment	Program Loaded at 1000_{16}	Program Loaded at 4000_{16}			
							START	4096
	1000					BEGIN	BALR	15,0
							USING	FIRST, 15
	1002					FIRST	BC	15, SKIP
	1006					DATA	DC	F'3472'
	⋮						⋮	
	1024					BASE1	DC	A(FIRST+4096)
	1028					BASE2	DC	A(FIRST+8192)
	⋮						⋮	
	1104	F	022	1024		SKIP	L	14,BASE1
							USING	FIRST+4096, 14
	1108	F	026	1028			L	13,BASE2
	⋮						USING	FIRST+8192, 13
	⋮						⋮	
	2504	D	902	3904			BC	15,CK8
	⋮						⋮	
	2898	F	004	1006		LOOP	A	4,DATA
	⋮						⋮	
	3204					LOOPB	S	5,DATA
	⋮						⋮	
	3508	E	896	2898			BC	8,LOOP
	⋮						⋮	
	3904	D	202	3204		CK8	BC	8,LOOPB
							END	BEGIN

Symbol Table

Symbol	Location
BASE 1	1024
BASE 2	1028
BEGIN	1000
CK8	3904
DATA	1006
FIRST	1002
LOOP	2898
LOOP B	3204
SKIP	1104

Chapter 10: Floating Point and Advanced Loops in Scientific Applications

For certain types of problems, typically those in the scientific and engineering areas, it is helpful or even essential to let the computer assume the task of keeping track of decimal points. Without this facility, that is, using only the "fixed-point" binary or decimal instructions of the System/360, it is necessary to know a great deal about the sizes and quantities appearing in the calculation. It is necessary to know the maximum possible sizes of all data, intermediate results, and final results; it is often also necessary to know the minimum sizes as well. This knowledge is necessary to avoid the possibility of exceeding the capacity of a register or of a storage location, and to avoid such things as divide exception. This exceptionally thorough knowledge about problem data is often difficult, and sometimes impossible to develop.

Furthermore, working in fixed point requires a considerable effort to align decimal or binary points correctly throughout the often complex process of the shifting and rearranging needed to maintain significance while avoiding capacity overflow.

For these reasons it is a great convenience to let the computer take over the clerical details of a complete accounting for number sizes and decimal point alignment. The saving in programming time is important, and floating point makes possible the solution of problems that would otherwise be almost impossible.

The basic idea of floating-point numbers is that each quantity is represented as a combination of two items: a numerical fraction and a power of 16 by which the fraction is multiplied to get the number represented. The power of 16 is called the characteristic by analogy with logarithms. (The 16 applies specifically to the System/360; other floating-point systems use binary or decimal multipliers.) For instance, in the System/360 the number 1 is represented as $\frac{1}{16} \cdot 16^1$. The $\frac{1}{16}$ is stored as a hexadecimal fraction: $0.1_{16} = \frac{1}{16}$. The characteristic is written in "excess 64" notation, which means that every characteristic is 64_{10} greater than the power actually represented. Thus, instead of $+1$ we write $+1 + 64 = 65_{10} = 41_{16}$. The characteristic -1 becomes $-1 + 64 = 63_{10} = 3F_{16}$. The excess 64 method is used to avoid the need of a sign for the characteristic.

System/360 permits two types of floating-point numbers, called *short* and *long*. In each, the excess 64 exponent is contained in the first byte, along with the sign of the number. In a short floating-point number, the fractional part consists of six hexadecimal digits contained in the next three bytes. In a long floating-point number the fractional part consists of 14 hexadecimal digits contained in the next seven bytes. A short number therefore occupies a fullword and a long number a doubleword.

The assembler accepts a DC instruction with a type specification of E for a short floating-point number and D for a long number. The number to be entered is written in quotes as usual, in decimal, with or without a decimal point as desired. Figure 104 shows DC entries for a series of floating-point numbers that will hopefully clarify the scheme of representation.

We see that the integers from 1 to 15 are represented by the corresponding hexadecimal digits, with a characteristic of 41. The form of 9 should be read as $16^1 \cdot \frac{9}{16}$. For 16 itself we have 42.100000, which we read as $16^2 \cdot \frac{1}{16}$. Decimal 32 becomes 42.200000, or $16^2 \cdot \frac{2}{16}$

These hexadecimal forms, considered as floating-point numbers, are arranged by the assembler to make easy reading. It might be well to display a few of these in pure hexadecimal form, as a demonstration that the plus signs and decimal points presented by the assembler are, of course, not in storage with the floating-point numbers. (See Figure 105, where the comment field gives the "pure" hexadecimal form.)

Figure 106 displays the same numbers as in Figure 104, except that they are negative. In Figure 107 we have these in pure hexadecimal. Note that the sign of the entire number is contained in the leftmost bit. In the representation of 1, for instance, the exponent of C1 would be 1100 0001 in binary; the leftmost 1 is the minus sign, with the other seven bits being the exponent in excess 64 form.

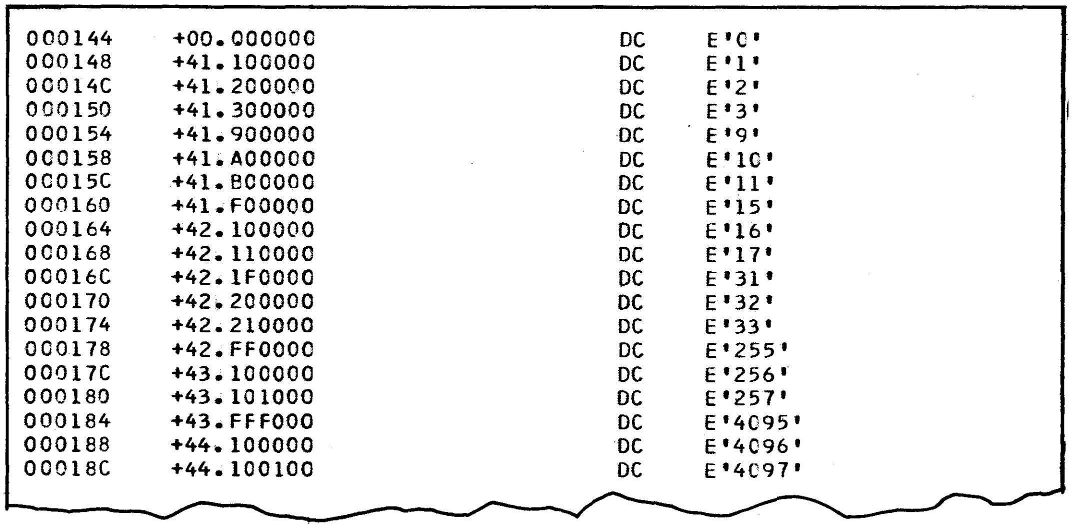
In Figure 108 we have some numbers that are not integers. The decimal number 0.5, for instance, becomes 40.800000, which we should read as $16^0 \cdot \frac{8}{16}$. The decimal number 1.5 becomes 41.180000, or $16^1 \cdot \frac{24}{16}$ in decimal. It is interesting to note that the simple decimal number 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 floating hexadecimal numbers, as 0.00390625 = 3F.100000.

Figure 109 shows a few long floating-point numbers. The scheme is the same, the only difference being the presence of eight additional hexadecimal digits. This permits a more accurate representation of numbers that do not have an exact hexadecimal representation, and naturally permits retention of much greater precision when arithmetic is performed. We shall explore the latter aspect in studying the action of the floating-point instructions.

The assembler allows a DC entry of type E or D to include a decimal exponent, for convenience in writing very large or very small numbers. This is done simply by following the number by the letter E and a plus or minus integer. Figure 110 shows some samples, both short and long. In E'12.78E+8', for instance, we intend the decimal number $12.78 \cdot 10^8$, which we see becomes +48.4C2CBC in hexadecimal.

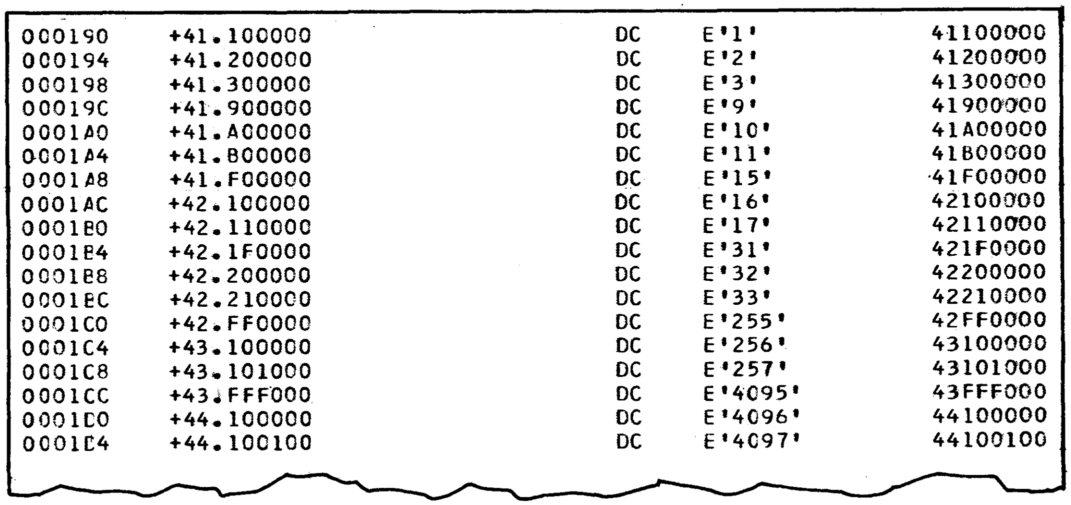
When a floating-point number has a nonzero first hexadecimal digit, it is said to be *normalized.* It should be realized that a normalized floating hexadecimal number may have as many as three leading binary zeros. DC entries in assembler language are always converted to normalized form, with the exception of a floating-point entry written with a *scale factor*. This facility is used in the rare occasions when it is desired to have leading zeroes in the hexadecimal fraction.

Floating-point representation can express decimal values ranging from about $5.4 \cdot 10^{-79}$ to about $7.2 \cdot 10^{75}$.



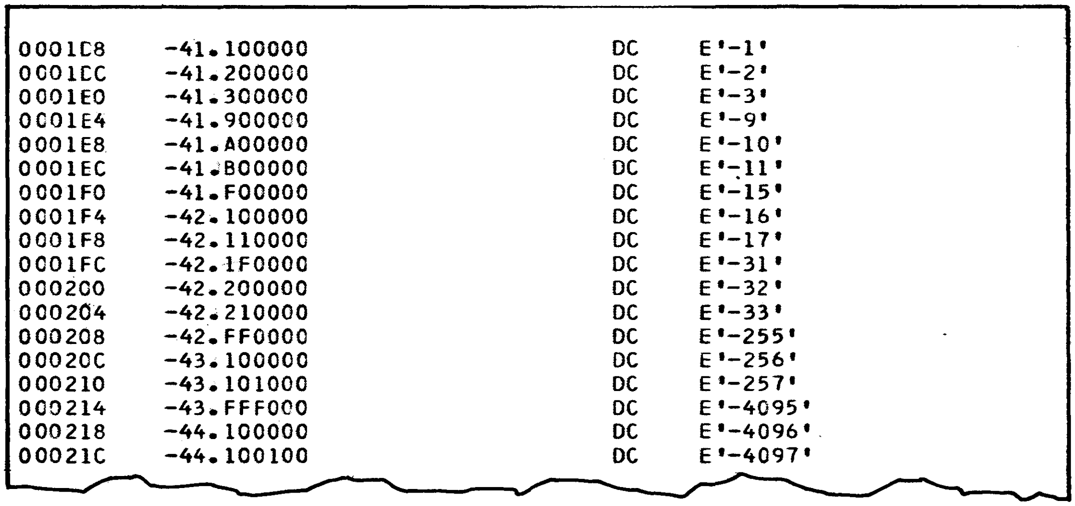
```
000144    +00.000000                      DC    E'0'
000148    +41.100000                      DC    E'1'
00014C    +41.200000                      DC    E'2'
000150    +41.300000                      DC    E'3'
000154    +41.900000                      DC    E'9'
000158    +41.A00000                      DC    E'10'
00015C    +41.B00000                      DC    E'11'
000160    +41.F00000                      DC    E'15'
000164    +42.100000                      DC    E'16'
000168    +42.110000                      DC    E'17'
00016C    +42.1F0000                      DC    E'31'
000170    +42.200000                      DC    E'32'
000174    +42.210000                      DC    E'33'
000178    +42.FF0000                      DC    E'255'
00017C    +43.100000                      DC    E'256'
000180    +43.101000                      DC    E'257'
000184    +43.FFF000                      DC    E'4095'
000188    +44.100000                      DC    E'4096'
00018C    +44.100100                      DC    E'4097'
```

Figure 104. Assembly listing illustrations of single-length floating-point DC entries



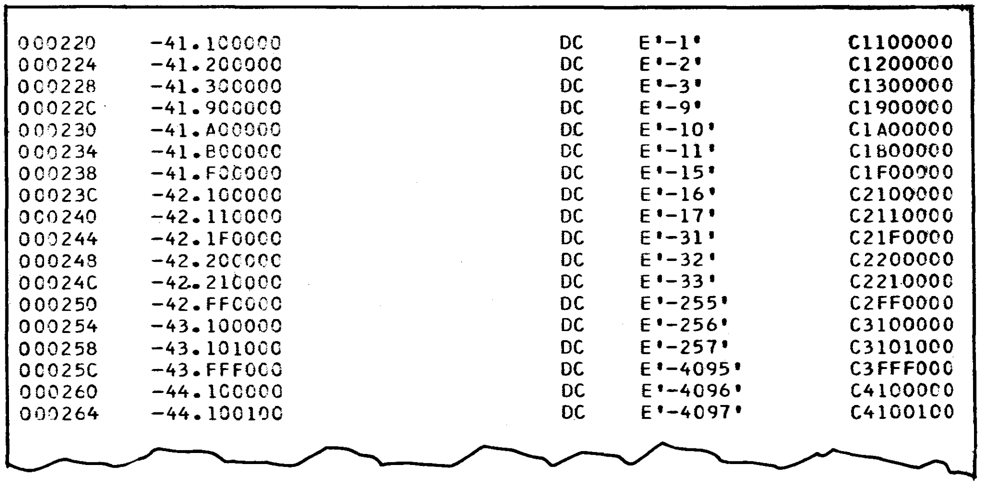
```
000190    +41.100000                      DC    E'1'          41100000
000194    +41.200000                      DC    E'2'          41200000
000198    +41.300000                      DC    E'3'          41300000
00019C    +41.900000                      DC    E'9'          41900000
0001A0    +41.A00000                      DC    E'10'         41A00000
0001A4    +41.B00000                      DC    E'11'         41B00000
0001A8    +41.F00000                      DC    E'15'         41F00000
0001AC    +42.100000                      DC    E'16'         42100000
0001B0    +42.110000                      DC    E'17'         42110000
0001B4    +42.1F0000                      DC    E'31'         421F0000
0001B8    +42.200000                      DC    E'32'         42200000
0001BC    +42.210000                      DC    E'33'         42210000
0001C0    +42.FF0000                      DC    E'255'        42FF0000
0001C4    +43.100000                      DC    E'256'        43100000
0001C8    +43.101000                      DC    E'257'        43101000
0001CC    +43.FFF000                      DC    E'4095'       43FFF000
0001D0    +44.100000                      DC    E'4096'       44100000
0001D4    +44.100100                      DC    E'4097'       44100100
```

Figure 105. The examples of Figure 104, with entries in the comment field showing the pure hexadecimal form of the constants



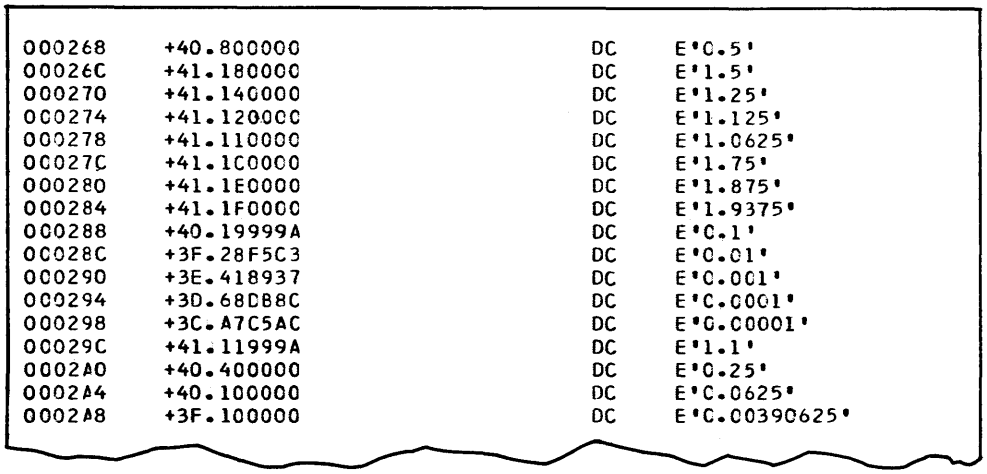
```
0001D8    -41.100000                      DC    E'-1'
0001DC    -41.200000                      DC    E'-2'
0001E0    -41.300000                      DC    E'-3'
0001E4    -41.900000                      DC    E'-9'
0001E8    -41.A00000                      DC    E'-10'
0001EC    -41.B00000                      DC    E'-11'
0001F0    -41.F00000                      DC    E'-15'
0001F4    -42.100000                      DC    E'-16'
0001F8    -42.110000                      DC    E'-17'
0001FC    -42.1F0000                      DC    E'-31'
000200    -42.200000                      DC    E'-32'
000204    -42.210000                      DC    E'-33'
000208    -42.FF0000                      DC    E'-255'
00020C    -43.100000                      DC    E'-256'
000210    -43.101000                      DC    E'-257'
000214    -43.FFF000                      DC    E'-4095'
000218    -44.100000                      DC    E'-4096'
00021C    -44.100100                      DC    E'-4097'
```

Figure 106. The examples of Figure 104 as negative numbers



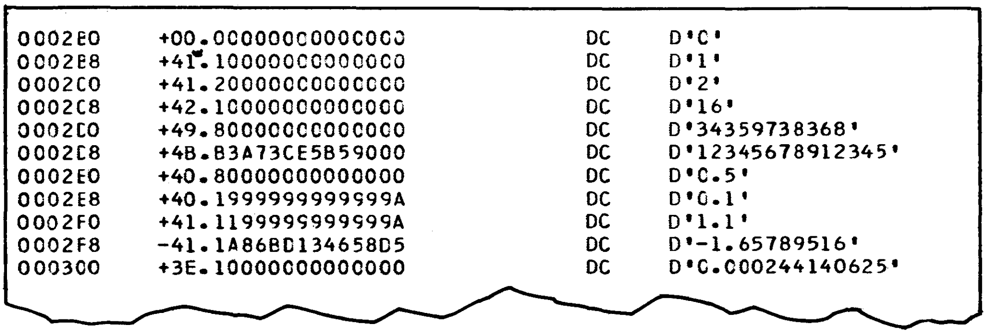
```
000220    -41.100000                          DC    E'-1'          C1100000
000224    -41.200000                          DC    E'-2'          C1200000
000228    -41.300000                          DC    E'-3'          C1300000
00022C    -41.900000                          DC    E'-9'          C1900000
000230    -41.A00000                          DC    E'-10'         C1A00000
000234    -41.B00000                          DC    E'-11'         C1B00000
000238    -41.F00000                          DC    E'-15'         C1F00000
00023C    -42.100000                          DC    E'-16'         C2100000
000240    -42.110000                          DC    E'-17'         C2110000
000244    -42.1F0000                          DC    E'-31'         C21F0000
000248    -42.200000                          DC    E'-32'         C2200000
00024C    -42.210000                          DC    E'-33'         C2210000
000250    -42.FF0000                          DC    E'-255'        C2FF0000
000254    -43.100000                          DC    E'-256'        C3100000
000258    -43.101000                          DC    E'-257'        C3101000
00025C    -43.FFF000                          DC    E'-4095'       C3FFF000
000260    -44.100000                          DC    E'-4096'       C4100000
000264    -44.100100                          DC    E'-4097'       C4100100
```

Figure 107. The pure hexadecimal forms of the constants in Figure 106



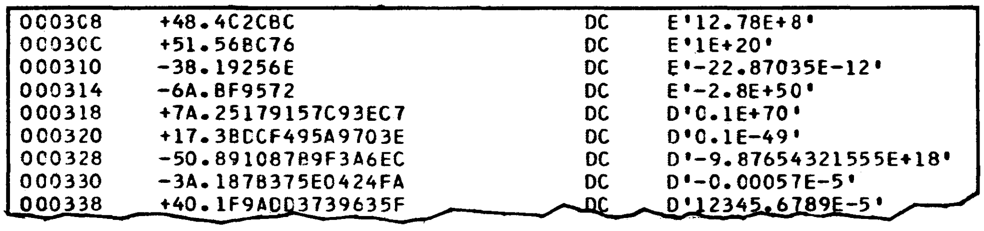
```
000268    +40.800000                          DC    E'0.5'
00026C    +41.180000                          DC    E'1.5'
000270    +41.140000                          DC    E'1.25'
000274    +41.120000                          DC    E'1.125'
000278    +41.110000                          DC    E'1.0625'
00027C    +41.1C0000                          DC    E'1.75'
000280    +41.1E0000                          DC    E'1.875'
000284    +41.1F0000                          DC    E'1.9375'
000288    +40.19999A                          DC    E'0.1'
00028C    +3F.28F5C3                          DC    E'0.01'
000290    +3E.418937                          DC    E'0.001'
000294    +3D.68DB8C                          DC    E'0.0001'
000298    +3C.A7C5AC                          DC    E'0.00001'
00029C    +41.11999A                          DC    E'1.1'
0002A0    +40.400000                          DC    E'0.25'
0002A4    +40.100000                          DC    E'0.0625'
0002A8    +3F.100000                          DC    E'0.00390625'
```

Figure 108. Assembly listing illustrations of single-length floating-point DC entries for numbers that are not integers



```
0002B0    +00.00000000000000                  DC    D'0'
0002B8    +41.10000000000000                  DC    D'1'
0002C0    +41.20000000000000                  DC    D'2'
0002C8    +42.10000000000000                  DC    D'16'
0002D0    +49.80000000000000                  DC    D'34359738368'
0002D8    +4B.B3A73CE5B59000                  DC    D'12345678912345'
0002E0    +40.80000000000000                  DC    D'0.5'
0002E8    +40.1999999999999A                  DC    D'0.1'
0002F0    +41.1199999999999A                  DC    D'1.1'
0002F8    -41.1A86BD134658D5                  DC    D'-1.65789516'
000300    +3E.10000000000000                  DC    D'0.000244140625'
```

Figure 109. Assembly listing of illustrations of double-length floating-point DC entries



```
0003C8    +48.4C2CBC                          DC    E'12.78E+8'
00030C    +51.56BC76                          DC    E'1E+20'
000310    -38.19256E                          DC    E'-22.87035E-12'
000314    -6A.BF9572                          DC    E'-2.8E+50'
000318    +7A.25179157C93EC7                  DC    D'0.1E+70'
000320    +17.3BDCF495A9703E                  DC    D'0.1E-49'
000328    -50.891087B9F3A6EC                  DC    D'-9.87654321555E+18'
000330    -3A.187B375E0424FA                  DC    D'-0.00057E-5'
000338    +40.1F9ADD3739635F                  DC    D'12345.6789E-5'
```

Figure 110. Assembly listing of illustrations of single- and double-length floating-point DC entries with decimal multipliers

There are four floating-point registers, numbered 0, 2, 4, and 6. All floating-point arithmetic instructions take one of the operands from a register and leave the result in the same register. There is one set of instructions for short operands; these are distinguished by the presence of the letter E in the instruction mnemonics for short operands. An entirely separate set operates on long operands; these are distinguished by a D in the mnemonics. A second classification applies to add and subtract, which may be done with or without normalization. The various actions are also available in RX and RR varieties.

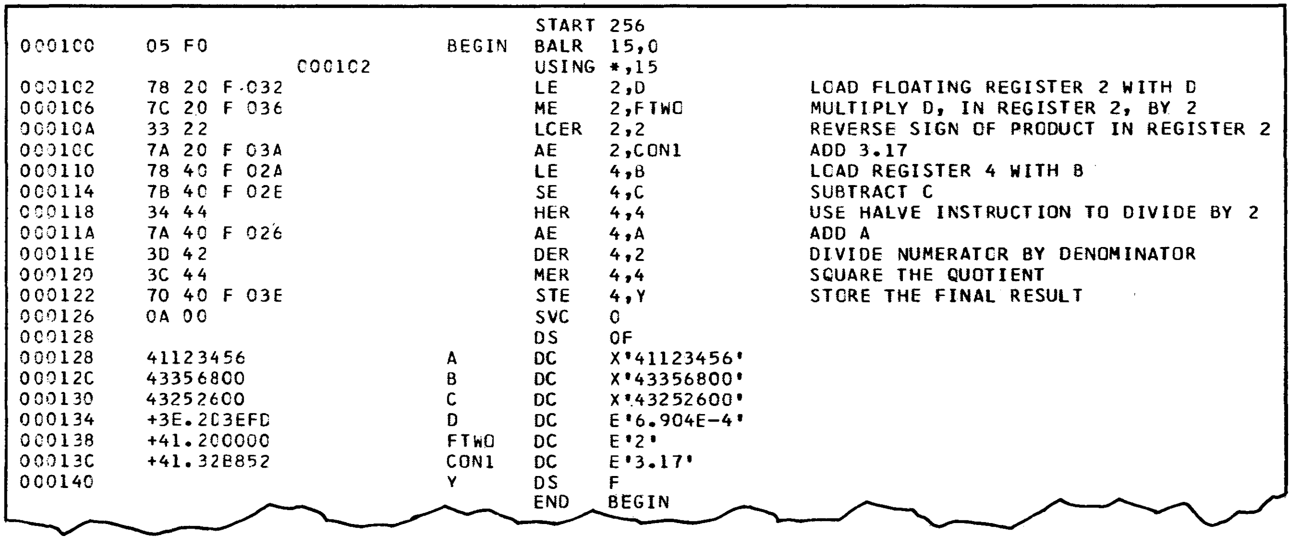
The many variations of the basic actions lead to quite a number of floating-point instructions, but there are not actually a great many basic types. For instance, there are eight separate floating-point add instructions and eight for subtract. Thus, although there are 44 separate floating-point instructions, the various functions they perform are almost obvious from a listing of the names.

Perhaps the best way to get an idea of the functioning of the registers and instructions is to study an example. Figure 111 is a listing of a program to evaluate the formula:

$$y = \left(\frac{a + \frac{b - c}{2}}{3.17 - 2d} \right)^2$$

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 implied when we write LE. Floating-point register 2, along with the other three, is a double-length register. This short operation will load the left half of the double-length register, leaving the lower half unchanged. The previous value in the lower half, if any, 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. Both operands are *prenormalized*, if necessary, which means to shift the fraction left until the leftmost hexadecimal digit is nonzero and reduce the characteristic by the number of shifts required. 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. The characteristic of the result is reduced by one if postnormalization is performed. The



```
                                         START 256
000100    05 F0                   BEGIN  BALR  15,0
                        000102           USING *,15
000102    78 20 F 032                    LE    2,D          LOAD FLOATING REGISTER 2 WITH D
000106    7C 20 F 036                    ME    2,FTWO       MULTIPLY D, IN REGISTER 2, BY 2
00010A    33 22                          LCER  2,2          REVERSE SIGN OF PRODUCT IN REGISTER 2
00010C    7A 20 F 03A                    AE    2,CON1       ADD 3.17
000110    78 40 F 02A                    LE    4,B          LCAD REGISTER 4 WITH B
000114    7B 40 F 02E                    SE    4,C          SUBTRACT C
000118    34 44                          HER   4,4          USE HALVE INSTRUCTION TO DIVIDE BY 2
00011A    7A 40 F 026                    AE    4,A          ADD A
00011E    3D 42                          DER   4,2          DIVIDE NUMERATCR BY DENOMINATOR
000120    3C 44                          MER   4,4          SQUARE THE QUOTIENT
000122    70 40 F 03E                    STE   4,Y          STCRE THE FINAL RESULT
000126    0A 00                          SVC   0
000128                                   DS    0F
000128    41123456                A      DC    X'41123456'
00012C    43356800                B      DC    X'43356800'
000130    43252600                C      DC    X'43252600'
000134    +3E.203EFD              D      DC    E'6.904E-4'
000138    +41.200000              FTWO   DC    E'2'
00013C    +41.32B852              CON1   DC    E'3.17'
000140                            Y      DS    F
                                         END   BEGIN
```

Figure 111. Assembly listing of a program illustrating single-length floating-point operations

characteristic of the fraction is computed by adding together the characteristics of the operands, and subtracting 40_{16}. The reason for the latter is that both operands are in excess 40 form; adding together two such gets the extra 40 in twice. The basic idea of adding the exponents is based on the familiar rules of algebra, which we might symbolize as:

$$(A \cdot 16^{a}) \cdot (B \cdot 16^{b}) = A \cdot B \cdot 16^{a+b}$$

The Load Complement Short Register 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 Short instruction (AE). 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. When this has been done, the decimal points (hexadecimal points, really) have been "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 to account for overflow or underflow.

If the addition caused overflow, 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. The latter is called *floating-point underflow.*

The extreme of floating-point underflow is for the result to be zero because the operands were equal but of opposite signs. The loss of significance is then 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 is designed to provide a warning in the form of a *significance exception.* If floating-point addition or subtraction results in complete loss of significance, and if the *significance mask bit* in the PSW is 1, a program interruption occurs. What is done as a result depends on the interrupt program.

With the sample numbers that have been entered in this program, there will be no loss of significance. We leave the result in floating-point register 2 for later use and turn 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, in order to illustrate 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. There is no adjustment of the exponent, and the result is *not* postnormalized. This means that if the first hexadecimal was exactly 1 (as it will be about once every 15 times, since there are 15 nonzero hexadecimal digits), the result will not be a normalized number. If what follows could be disabled by the presence of the unnormalized number, some other course should have been taken. In our situation, however, no damage can be done. An unnormalized number is a correct representation of a quantity; the only issue is the form of representation and the possibility of loss of significance. In our case, there is no problem.

The Add Short that follows is as before, except that it may find one of its operands to be an unnormalized number. If this operand is the one with the smaller characteristic, there is no net effect of the lack of normalization: the fraction will be shifted right and some right-end digits lost anyway. However, if the unnormalized number also has the larger charactertistic, the other number may lose digits in shifting it to the right to make the characteristics the same.

Here, however, we come to a feature of the System/360 designed to protect against loss of significance: the sum formed in short addition and subtraction is actually *seven* hexadecimal digits, rather than six. Then, if postnormalization is required because one or more leading digits are zero, the extra seventh digit is shifted back, thus preserving some extra significance. The extra digit is called the guard digit. (There is nothing analogous for long operands.)

For example, suppose that the values of A, B, and C are as follows:

VARIABLE	HEXADECIMAL VALUE	FLOATING HEXADECIMAL REPRESENTATION
A	1.23456	41123456
B	356.8	43356800
C	252.6	43252600

Subtracting C from B, we get 104.2, or 43104200. Writing this out in binary, we have:

0100 0011 0001 0000 0100 0010 0000 0000

The halve operation means to shift just the fraction part to the right one place, giving:

0100 0011 0000 1000 0010 0001 0000 0000

Converted back to floating hexadecimal, this is 43082100, which is the correct result of dividing by 2: $104_{16} = 260_{10}$, and $82_{16} = 130_{10}$.

This result, however, is unnormalized, and in the halve operation it stays that way. Now we Add Short, naming a second operand that in floating hexadecimal form is 41123456. This second operand happens to be the one with the smaller exponent, so it is shifted right and its exponent increased until the exponent is the same as the larger. This leads to a shifted number 430012345; the fraction here is seven digits, the extra one being the guard digit. Now we add fractions, as the fractions are now lined up.

```
 0012345
+082100
 0833345
```

This result is not normalized, so a postnormalization takes place: the fraction is shifted left and the exponent decreased, until there is a leading nonzero hexadecimal digit in the fraction. This takes just one shift, of course, with the guard digit included:

42833345

This is the result that is left in the register.

Without the guard digit, the result would have been:

42833340

That is, we would have lost a digit at the right because of the unnormalized operand. We have of course lost the 6 that originally was a part of the variable named A, but that is unavoidable, since after lining up decimal points the sum has too many digits to be contained in a short floating-point word. The loss of significant digits as in this case is a serious concern in many scientific applications. The long form floating-point is often the answer to the problem.

The guard digit was naturally not provided just to cover such contrived situations. It will be valuable any time both a prenormalization and a postnormalization are required in short floating addition or subtraction. This will of course be a small fraction of all additions and subtractions, but nevertheless a significant number.

At this point we have the numerator in floating-point register 4 and the denominator in 2; an RR format floating division places the quotient in register 4. Floating-point division works as follows. Both operands are prenormalized. The fractions are then divided to get the quotient fraction. The characteristic of the denominator is subtracted from that of the numerator, and then 40_{16} is added to get the characteristic again into excess 40 form. The subtraction is based on the rule from algebra:

$$\frac{A \cdot 16^{a}}{B \cdot 16^{b}} = A/B \cdot 16^{a-b}$$

Division of two normalized six-digit fractions will always give either six or seven digits, never more or less. The quotient may need to be shifted right by one position and the characteristic correspondingly increased by 1.

An attempted division in which the fraction of the divisor is zero leads to suppression of the operation, without the dividend having been affected. A program interruption occurs.

When the dividend fraction is zero the quotient is made a true zero, which means that the exponent and fraction are all zeroes and the sign is plus. In terms of bits, a floating-point true zero is all binary zeroes.

The final operation is to square the result of the division, which is standing in register 4. An RR format multiply in which both operands are register 4 does the job. Finally a Store Short (STE) puts the result away as specified.

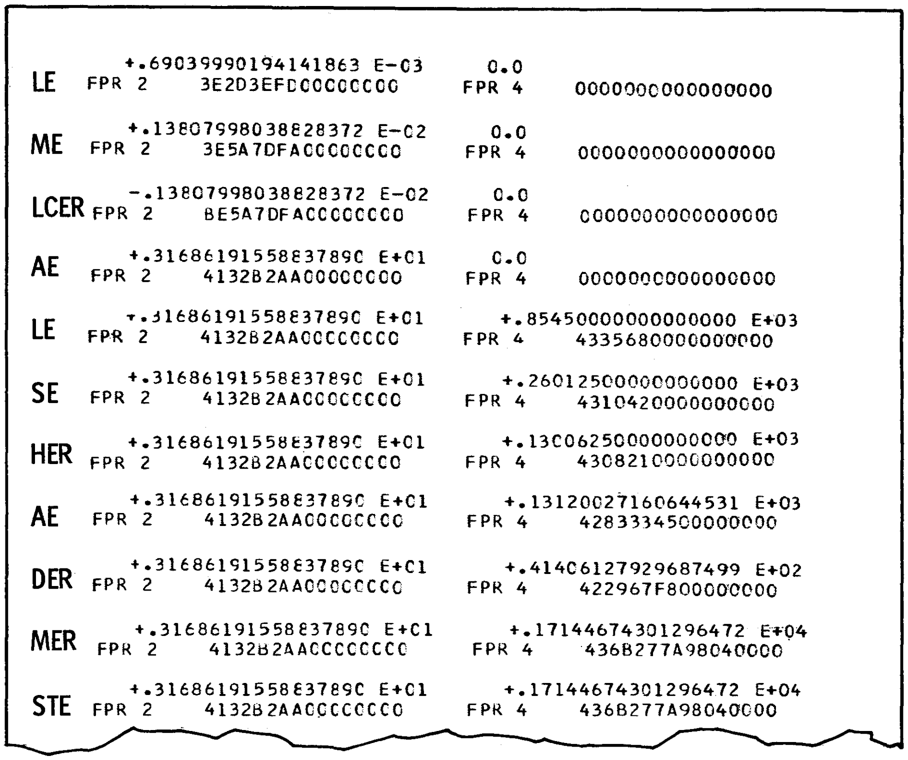
Figure 112 is provided for those who may wish to follow the details of the arithmetic in this example. This output is a composite from the dump program. Each line gives the mnemonic for the instruction and floating-point registers 2 and 4 in floating hexadecimal and in floating decimal. The hexadecimal gives the full-length number, exactly as it appears in the machine. The floating decimal is of course a converted number, and usually not an exact equivalent, since exact equivalents usually do not exist. Furthermore, the exponent in the decimal numbers is a true exponent, for our convenience, rather than the internal excess 64. The floating-point form here is the way floating-point numbers are usually presented on output.

Figure 113 is a listing of the same program rewritten to do all operations in double length. The basic ideas are pretty much the same, but there are differences in detailed considerations.

The registers named here are the same registers as before; but this time the full length is used, where before the low-order half was generally ignored.

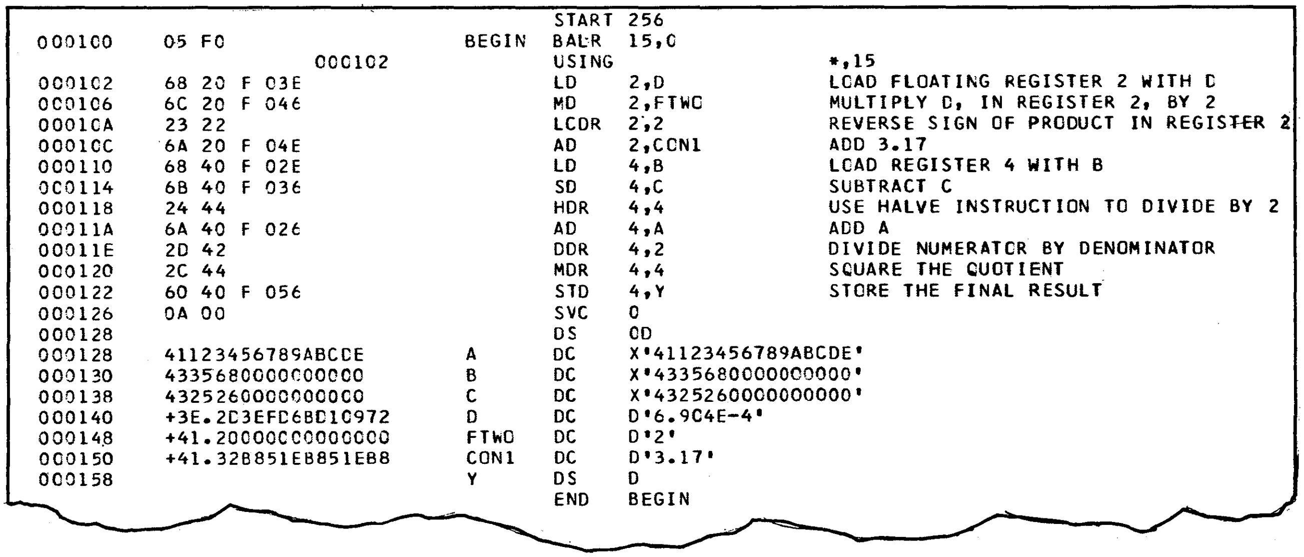
Perhaps the most important difference between short and long operations is that there is no guard digit for long operations. The reason is that, in contrast with short operations, there is no simple way to obtain it (from the standpoint of the machine designer), and anyway it is not as badly needed. Long operations have 14 hexadecimal digits, equivalent to about 16 decimal digits, so in a certain sense they already contain their own guard digits.

Figure 114 presents a trace listing of the results of the program, again in floating hexadecimal and in floating decimal.



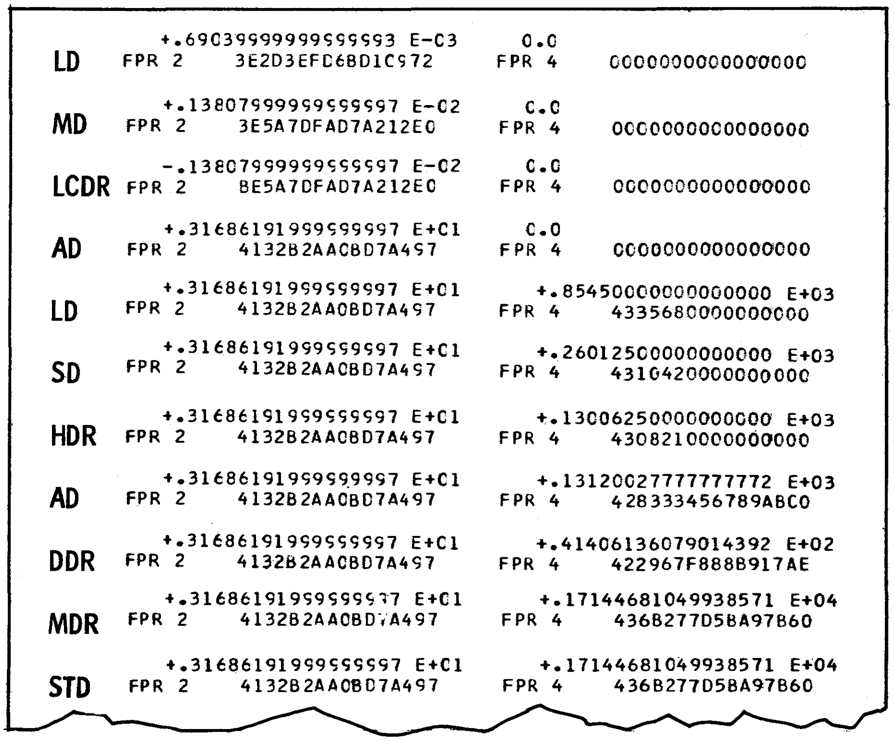
```
       +.6903999019414186З E-03     0.0
LE   FPR 2      3E2D3EFD00000000    FPR 4     0000000000000000

       +.13807998038828372 E-02     0.0
ME   FPR 2      3E5A7DFA00000000    FPR 4     0000000000000000

       -.13807998038828372 E-02     0.0
LCER FPR 2      BE5A7DFA00000000    FPR 4     0000000000000000

       +.31686191558837890 E+01     0.0
AE   FPR 2      4132B2AA00000000    FPR 4     0000000000000000

       +.31686191558837890 E+01      +.85450000000000000 E+03
LE   FPR 2      4132B2AA00000000    FPR 4     4335680000000000

       +.31686191558837890 E+01      +.26012500000000000 E+03
SE   FPR 2      4132B2AA00000000    FPR 4     4310420000000000

       +.31686191558837890 E+01      +.13006250000000000 E+03
HER  FPR 2      4132B2AA00000000    FPR 4     4308210000000000

       +.31686191558837890 E+01      +.13120027160644531 E+03
AE   FPR 2      4132B2AA00000000    FPR 4     4283334500000000

       +.31686191558837890 E+01      +.41406127929687499 E+02
DER  FPR 2      4132B2AA00000000    FPR 4     422967F800000000

       +.31686191558837890 E+01      +.17144674301296472 E+04
MER  FPR 2      4132B2AA00000000    FPR 4     436B277A98040000

       +.31686191558837890 E+01      +.17144674301296472 E+04
STE  FPR 2      4132B2AA00000000    FPR 4     436B277A98040000
```

Figure 112. The contents of floating-point registers 2 and 4 after the execution of each instruction of the program of Figure 111. The top line is floating decimal, the bottom line pure hexadecimal, in each case.



```
                                                START 256
000100    05 F0                          BEGIN  BALR  15,0
                          000102                USING                     *,15
000102    68 20 F 03E                           LD    2,D                 LOAD FLOATING REGISTER 2 WITH D
000106    6C 20 F 046                           MD    2,FTWO              MULTIPLY D, IN REGISTER 2, BY 2
00010A    23 22                                 LCDR  2,2                 REVERSE SIGN OF PRODUCT IN REGISTER 2
00010C    6A 20 F 04E                           AD    2,CON1              ADD 3.17
000110    68 40 F 02E                           LD    4,B                 LOAD REGISTER 4 WITH B
000114    6B 40 F 036                           SD    4,C                 SUBTRACT C
000118    24 44                                 HDR   4,4                 USE HALVE INSTRUCTION TO DIVIDE BY 2
00011A    6A 40 F 026                           AD    4,A                 ADD A
00011E    2D 42                                 DDR   4,2                 DIVIDE NUMERATOR BY DENOMINATOR
000120    2C 44                                 MDR   4,4                 SQUARE THE QUOTIENT
000122    60 40 F 056                           STD   4,Y                 STORE THE FINAL RESULT
000126    0A 00                                 SVC   0
000128                                          DS    0D
000128    41123456789ABCDE               A      DC    X'41123456789ABCDE'
000130    4335680000000000               B      DC    X'4335680000000000'
000138    4325260000000000               C      DC    X'4325260000000000'
000140    +3E.2D3EFD6BD10972             D      DC    D'6.904E-4'
000148    +41.20000000000000             FTWO   DC    D'2'
000150    +41.32B851EB851EB8             CON1   DC    D'3.17'
000158                                   Y      DS    D
                                                END   BEGIN
```

Figure 113. Assembly listing of the program of Figure 111, modified to do all operations in double-length floating point



```
        +.69039999999999993 E-03    0.0
LD   FPR 2     3E2D3EFD6BD1C972     FPR 4     0000000000000000

        +.13807999999999997 E-02    0.0
MD   FPR 2     3E5A7DFAD7A212E0     FPR 4     0000000000000000

        -.13807999999999997 E-02    0.0
LCDR FPR 2     BE5A7DFAD7A212E0     FPR 4     0000000000000000

        +.31686191999999997 E+01    0.0
AD   FPR 2     4132B2AA0BD7A497     FPR 4     0000000000000000

        +.31686191999999997 E+01      +.85450000000000000 E+03
LD   FPR 2     4132B2AA0BD7A497     FPR 4     4335680000000000

        +.31686191999999997 E+01      +.26012500000000000 E+03
SD   FPR 2     4132B2AA0BD7A497     FPR 4     4310420000000000

        +.31686191999999997 E+01      +.13006250000000000 E+03
HDR  FPR 2     4132B2AA0BD7A497     FPR 4     4308210000000000

        +.31686191999999997 E+01      +.13120027777777772 E+03
AD   FPR 2     4132B2AA0BD7A497     FPR 4     4283333456789ABC0

        +.31686191999999997 E+01      +.41406136079014392 E+02
DDR  FPR 2     4132B2AA0BD7A497     FPR 4     422967F888B917AE

        +.31686191999999997 E+01      +.17144681049938571 E+04
MDR  FPR 2     4132B2AA0BD7A497     FPR 4     436B277D5BA97B60

        +.31686191999999997 E+01      +.17144681049938571 E+04
STD  FPR 2     4132B2AA0BD7A497     FPR 4     436B277D5BA97B60
```

Figure 114. The contents of floating-point registers 2 and 4 after the execution of each instruction of the program of Figure 113

Loop Control in a Floating-Point Matrix Multiply Subroutine

Matrix multiplication is a familiar application of computers, since it involves a great deal of highly repetitive arithmetic. Furthermore, it is a good sample problem because it provides a demonstration of what a computer can do in an area of heavily used floating-point loops, where time considerations become especially important.

For readers interested in the loop control and subroutine ideas here but not familiar with matrix multiplication, the problem is easily stated.

We are given two arrays, called *matrices*, named A and B. Matrix A has M rows and N columns; matrix B has N rows and R columns. The numbers M, N, and R are fixed for any *particular* execution of the subroutine we shall write, but the subroutine must be able to operate for any values. The task is to compute the elements of another matrix C which has M rows and R columns. Each element of C is the sum of the product of one row of A and one column of B. In subscript notation, the elements of C are given by:

$$c_{ik} = \sum_{j=1}^{N} a_{ij}\, b_{jk} \qquad \begin{array}{l} i = 1, 2, \ldots, M \\ k = 1, 2, \ldots, R \end{array}$$

This process can actually be described much more quickly, in terms of an example. Figure 115 shows three sample matrices, arranged as in A × B = C. Matrix A here has four rows and three columns; B has three rows and two columns. The element shown as 22 in C is the sum of the products in multiplying the first row of A (1 2 5) by the first column of B (1 8 1):

$1 \cdot 1 + 2 \cdot 8 + 5 \cdot 1 = 22$

$$\overset{\textstyle A}{\begin{pmatrix} 1 & 2 & 5 \\ 3 & 1 & 6 \\ -1 & 1 & 0 \\ 4 & 2 & -5 \end{pmatrix}} \times \overset{\textstyle B}{\begin{pmatrix} 1 & 8 \\ 8 & 2 \\ 1 & -4 \end{pmatrix}} = \overset{\textstyle C}{\begin{pmatrix} 22 & -8 \\ 17 & 2 \\ 7 & -6 \\ 15 & 56 \end{pmatrix}}$$

Figure 115. Illustrations of matrices. The first two are multiplied together to give the third, as described in the text

We then multiply the first row of A (again) by the *second* column of B:

$1 \cdot 8 + 2 \cdot 2 + 5 \cdot (-4) = -8$

If there were further columns in B, they would be multiplied by the first row of A also, with the results going to make up the first row of C.

Having multiplied the first row of A by all the columns of B, we multiply the *second* row of A by all the columns of B in turn, to get the *second* row of C:

$3 \cdot 1 + 1 \cdot 8 + 6 \cdot 1 = 17$
$3 \cdot 8 + 1 \cdot 2 + 6 \cdot (-4) = 2$

Now we multiply the third row of A by all the columns of B in turn, to get the third row of C:

$-1 \cdot 1 + 1 \cdot 8 + 0 \cdot 1 = 7$
$-1 \cdot 8 + 1 \cdot 2 + 0 \cdot (-4) = -6$

We finally multiply the fourth and last row of A by the columns of C to get the fourth and last row of C:

$4 \cdot 1 + 2 \cdot 8 + -5 \cdot 1 = 15$
$4 \cdot 8 + 2 \cdot 2 -5 \cdot (-4) = 56$

This process can be described in a flowchart, if we agree on some notation. Let us denote by a_{ij} the element in the ith row and the jth column of A, by b_{jk} the element in the jth row and kth column of B, and by c_{ik} the element in the ith row and kth column of C. With this much agreement on notation, we can describe the flowchart of Figure 116. The flowchart is of course drawn to reflect the coding methods that we shall use in the program to be described shortly.

The first box has to do with the addressing system that will be used to run through the elements of the result matrix C. The i = 1 box says to start with the first row of A. This action is never repeated. The k =1 means to pick up the first column of B; this is repeated. The action j = 1 means to start with the first product in the series of products; it refers to the first element in the ith row of A and to the first element in the kth column of B. Then before starting the multiplication loop we clear the element storage in C.

The inner loop gets the sum of the products. The values of i, j, and k, which will be in base and index registers in the program, pick out the elements of A and B. We perform the multiplication and add the product to the partial sum of products previously accumulated. We then increment j to move on to the next element in the row of A and the next element in the column of B, and ask whether the computation of the element of C has been completed. If it has not, we return to form another product and add it to the partial product.

If we have completed the computation of an element of C, we are ready to move on to the next column of B and repeat the whole process in order to compute the next element of C. The return is to the box that resets the j index, so that we begin with the first element of the ith row of A and the first element of the kth column of B.

If we have multiplied one row of A by all the columns of B, we are ready to move to another row of A and repeat the entire middle loop.

Translating this flowchart into a program requires first of all a decision on how the elements of the three matrices are to be stored, that is, in row order or column order. For many purposes it would be best for all three to be in the same order, usually along the rows. We could certainly write a matrix multiplication program based on this assumption. For our purposes here, however, where we choose to emphasize the possibility of rapid loops, we shall set up the storage so as to save time and so as to make possible efficient computing loops.

The storage scheme will be as follows. Matrix A is stored in row order, which, to be explicit, means that the elements are stored in consecutive fullword locations starting with the first element in the first row. The second element in the list is the second element in the first row, etc., across the first row. The next element then is the first in the second row, etc., etc. Matrix B is stored in column order: down the columns, instead of across the rows. Matrix C is stored in row order.

Now let us turn to the question of setting up this program as a subroutine. We are to be able to call the subroutine with a description of the three matrices, and have it operate correctly for any two matrices A and B. This will require an agreement on a calling sequence, as follows:

```
L       13,AMMPY
CNOP    2,4
BALR    14,13
DC      A(A)
DC      A(B)
DC      A(C)
DC      F'M'
DC      F'N'
DC      F'R'
```

The symbol AMMPY stands for an address constant that contains the address of the subroutine. A,

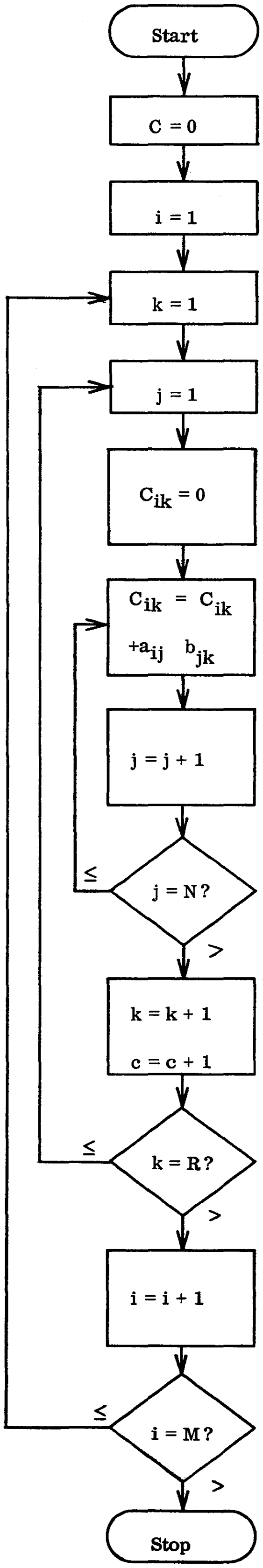


Figure 116. Flowchart of a method of matrix multiplication

B, and C are used to stand for the addresses of the first elements in the three matrices. Where we have written M, N, and R in the sample calling sequence, it is necessary to substitute actual numerical values.

Figure 117 is the assembly listing of a main program that calls our matrix multiplication subroutine twice. In the first call, matrix A is to be multiplied by matrix B and the result called C. A is 4 by 3; B is 3 by 2; and C is therefore 4 by 2. In the second call, we have asked for the 3 by 3 matrix named D to be multiplied by the 3 by 1 matrix E and the result called F. The values for the elements of these matrices are entered with DC's.

Figure 118 is the assembly listing of the subroutine. The basic idea of the loop control and element addressing will be as follows. Register 2 will be used as an index to address the successive elements of matrix C. Incrementing this index will be done by a Load Address instruction. This index need never be tested. Register 3 will select a row of A, and must therefore start out with the address given in the calling sequence. This base will be incremented by the number of bytes in a row, 4N. This base will furthermore be incremented and tested with a BXLE instruction, so we place 4N in register 4 for an increment and use 5 for the limit test. Register 5 needs to contain the starting address plus 4N times one less than the number of rows, or 4(M–1)N.

Register 3 as a base picks a row of A; an element within that row is selected by register 7 used as an index. This starts at zero, is incremented by 4 (register 8) and has a limit test value of 4(N–1) in register 9.

Register 11 picks one column of B. The starting address is the first address in B; the increment is 4N (register 12); the limit test value is 4N(R–1) (register 13). Register 7 as an index picks an element in a particular column of B, as well as picking an element in a particular row of A.

In the subroutine program we begin by saving all registers. The instructions that follow set up the various registers as we have described above. Register 6 happens to be available, so a zero is placed in it in advance to use in clearing storage locations in C.

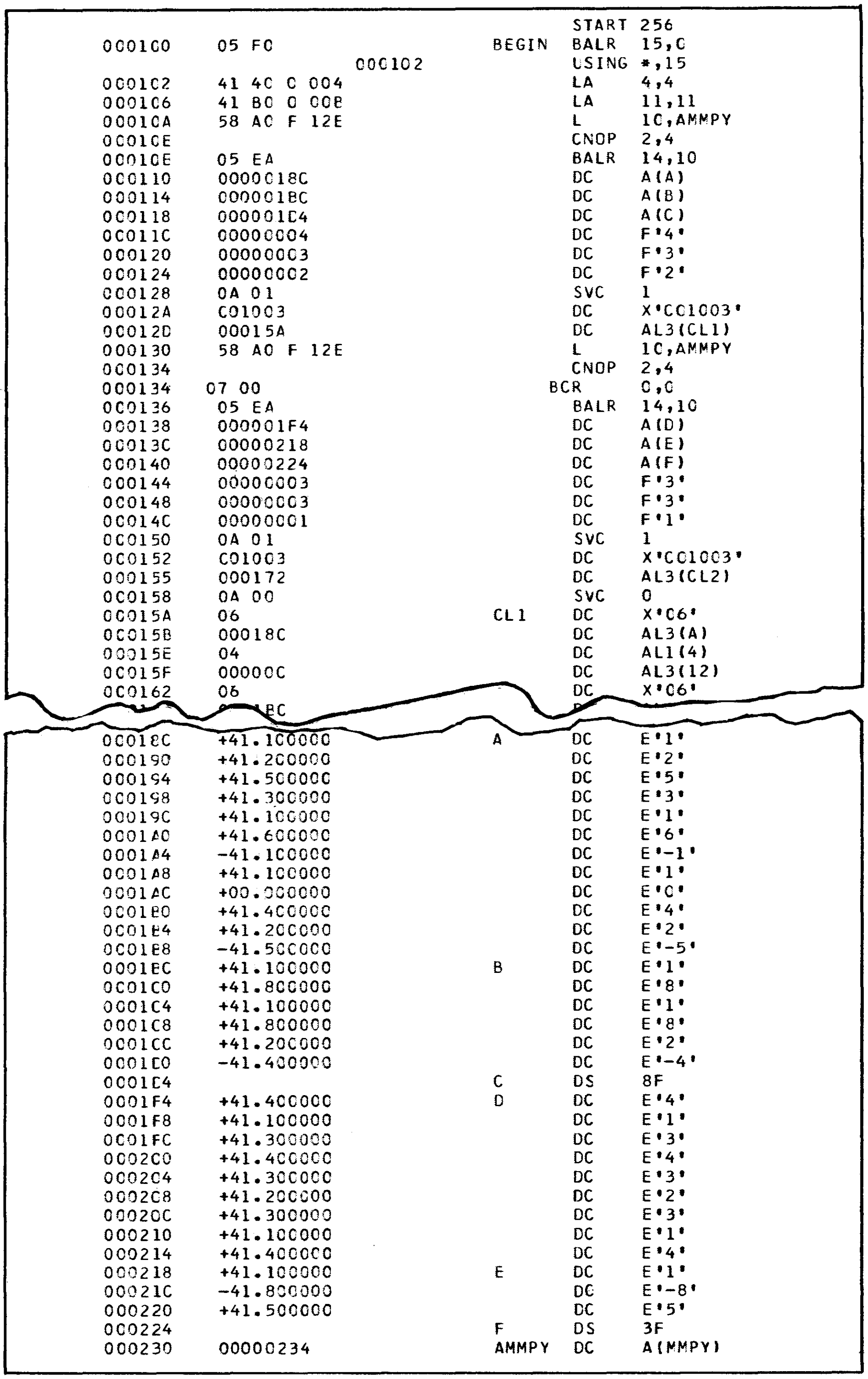
The outer loop begins at RET3, where we set register 11 (column selector for B) to its starting value, which is easily picked up from the calling sequence. We shall return to this point each time we move to a new row of A, at which time we start over on the columns of B.

At RET2 we set index register 7 back to zero, which means to go back to the beginning of the A row, and to start at the top of what will be a new column of B. This happens only as we begin the multiplication and summing loop that computes another element of C, so we clear the C element location at this time as well.

At RET1 we enter the multiplication and summing loop itself. This inner loop is just five instructions long, including the BXLE.

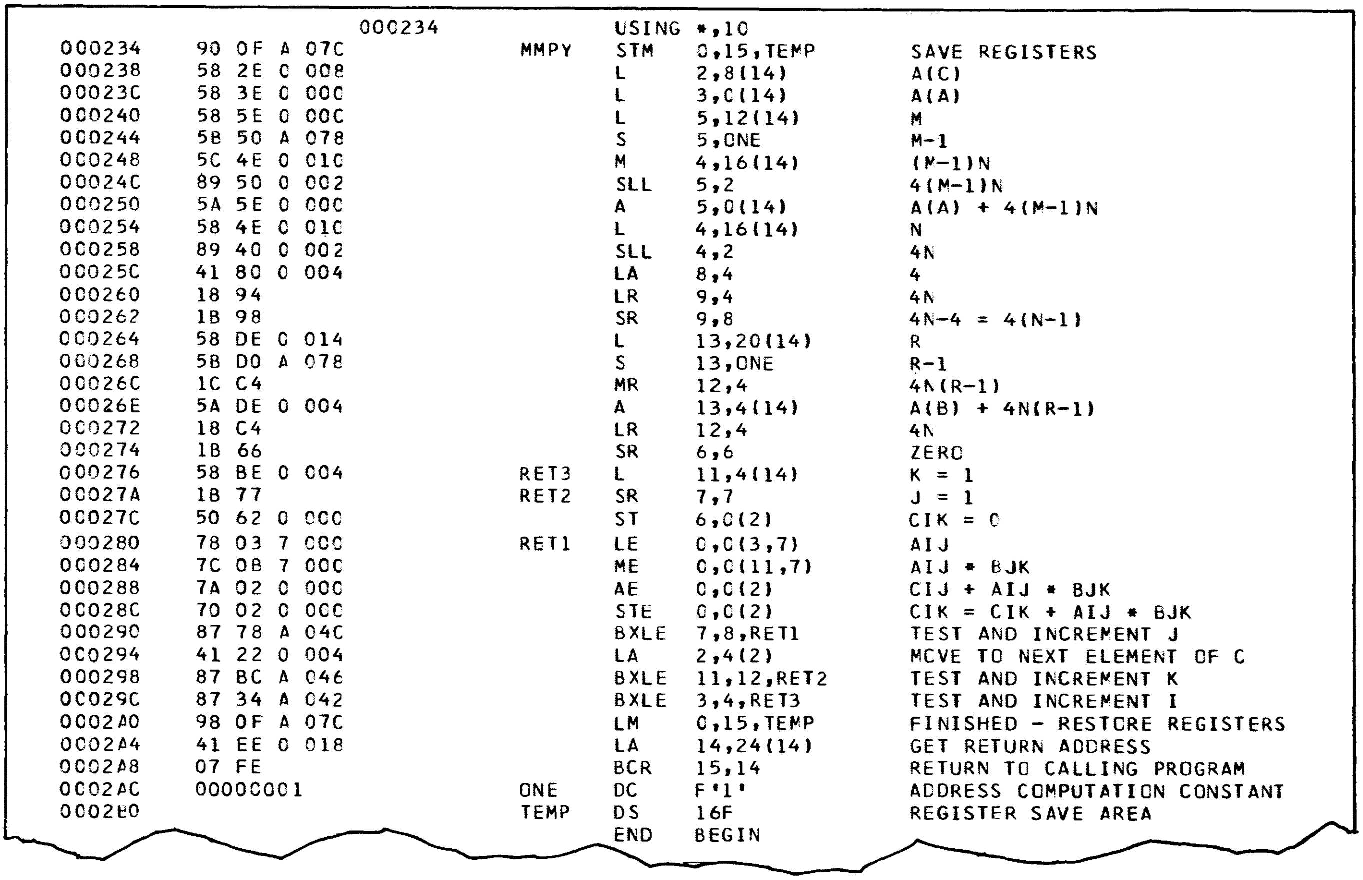
When the BXLE does not branch, it means that we have completed the computation of one element. We add 4 to the contents of register 2, in order to move to another element location in C, then execute a BXLE on 11, the base that picks a column of B. When this finally does not branch, we reach a BXLE that operates on register 3, which picks a row of A. When this BXLE finally does not branch, the task is finished. We reset the registers, add 24 to link register 14, and branch back to the instruction just beyond the calling sequence.

Figure 119 shows the computed values of matrices C and F.



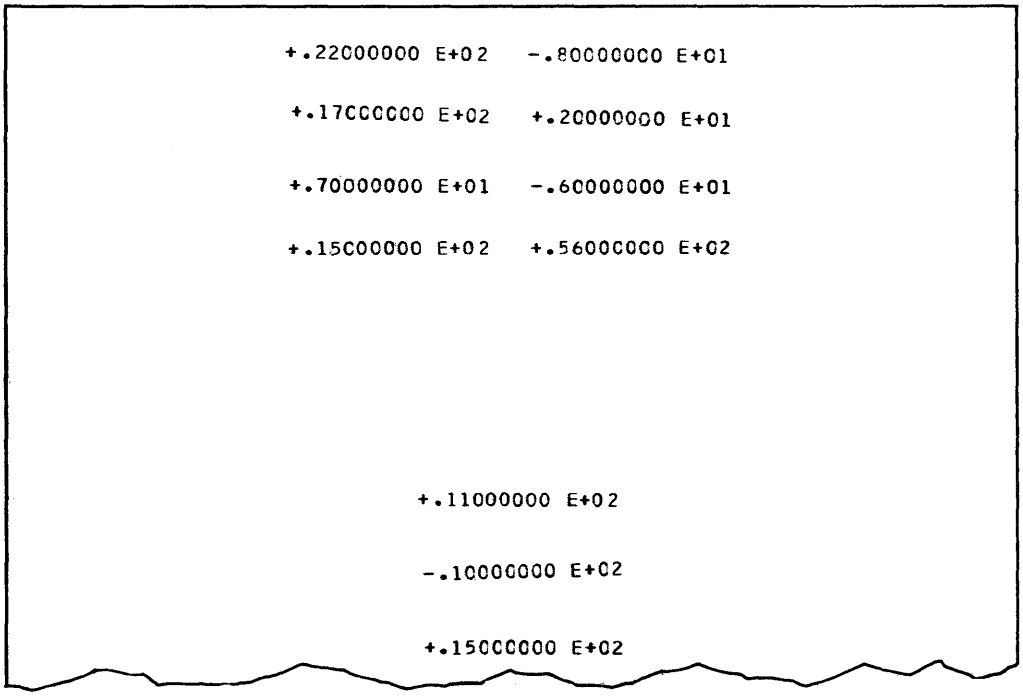
```
                                                  START 256
000100    05 F0                           BEGIN   BALR  15,0
                            000102                USING *,15
000102    41 40 0 004                             LA    4,4
000106    41 B0 0 00B                             LA    11,11
00010A    58 A0 F 12E                             L     10,AMMPY
00010E                                            CNOP  2,4
00010E    05 EA                                   BALR  14,10
000110    0000018C                                DC    A(A)
000114    000001BC                                DC    A(B)
000118    000001D4                                DC    A(C)
00011C    00000004                                DC    F'4'
000120    00000003                                DC    F'3'
000124    00000002                                DC    F'2'
000128    0A 01                                   SVC   1
00012A    C01003                                  DC    X'C01003'
00012D    00015A                                  DC    AL3(CL1)
000130    58 A0 F 12E                             L     10,AMMPY
000134                                            CNOP  2,4
000134   07 00                                  BCR     0,0
000136    05 EA                                   BALR  14,10
000138    000001F4                                DC    A(D)
00013C    00000218                                DC    A(E)
000140    00000224                                DC    A(F)
000144    00000003                                DC    F'3'
000148    00000003                                DC    F'3'
00014C    00000001                                DC    F'1'
000150    0A 01                                   SVC   1
000152    C01003                                  DC    X'C01003'
000155    000172                                  DC    AL3(CL2)
000158    0A 00                                   SVC   0
00015A    06                              CL1     DC    X'06'
00015B    00018C                                  DC    AL3(A)
00015E    04                                      DC    AL1(4)
00015F    00000C                                  DC    AL3(12)
000162    06                                      DC    X'06'
            BC
00018C    +41.100000                      A       DC    E'1'
000190    +41.200000                              DC    E'2'
000194    +41.500000                              DC    E'5'
000198    +41.300000                              DC    E'3'
00019C    +41.100000                              DC    E'1'
0001A0    +41.600000                              DC    E'6'
0001A4    -41.100000                              DC    E'-1'
0001A8    +41.100000                              DC    E'1'
0001AC    +00.000000                              DC    E'0'
0001B0    +41.400000                              DC    E'4'
0001B4    +41.200000                              DC    E'2'
0001B8    -41.500000                              DC    E'-5'
0001BC    +41.100000                      B       DC    E'1'
0001C0    +41.800000                              DC    E'8'
0001C4    +41.100000                              DC    E'1'
0001C8    +41.800000                              DC    E'8'
0001CC    +41.200000                              DC    E'2'
0001D0    -41.400000                              DC    E'-4'
0001D4                                    C       DS    8F
0001F4    +41.400000                      D       DC    E'4'
0001F8    +41.100000                              DC    E'1'
0001FC    +41.300000                              DC    E'3'
000200    +41.400000                              DC    E'4'
000204    +41.300000                              DC    E'3'
000208    +41.200000                              DC    E'2'
00020C    +41.300000                              DC    E'3'
000210    +41.100000                              DC    E'1'
000214    +41.400000                              DC    E'4'
000218    +41.100000                      E       DC    E'1'
00021C    -41.800000                              DC    E'-8'
000220    +41.500000                              DC    E'5'
000224                                    F       DS    3F
000230    00000234                        AMMPY   DC    A(MMPY)
```

Figure 117. Assembly listing of a main program that calls a matrix multiplication subroutine. The four sample matrices are entered with DC's.



```
                         000234               USING *,10
000234      90 0F A 07C              MMPY     STM   0,15,TEMP          SAVE REGISTERS
000238      58 2E 0 008                       L     2,8(14)            A(C)
00023C      58 3E 0 000                       L     3,0(14)            A(A)
000240      58 5E 0 00C                       L     5,12(14)           M
000244      5B 50 A 078                       S     5,ONE              M-1
000248      5C 4E 0 010                       M     4,16(14)           (M-1)N
00024C      89 50 0 002                       SLL   5,2                4(M-1)N
000250      5A 5E 0 000                       A     5,0(14)            A(A) + 4(M-1)N
000254      58 4E 0 010                       L     4,16(14)           N
000258      89 40 0 002                       SLL   4,2                4N
00025C      41 80 0 004                       LA    8,4                4
000260      18 94                             LR    9,4                4N
000262      1B 98                             SR    9,8                4N-4 = 4(N-1)
000264      58 DE 0 014                       L     13,20(14)          R
000268      5B D0 A 078                       S     13,ONE             R-1
00026C      1C C4                             MR    12,4               4N(R-1)
00026E      5A DE 0 004                       A     13,4(14)           A(B) + 4N(R-1)
000272      18 C4                             LR    12,4               4N
000274      1B 66                             SR    6,6                ZERO
000276      58 BE 0 004              RET3     L     11,4(14)           K = 1
00027A      1B 77                    RET2     SR    7,7                J = 1
00027C      50 62 0 000                       ST    6,0(2)             CIK = 0
000280      78 03 7 000              RET1     LE    0,0(3,7)           AIJ
000284      7C 0B 7 000                       ME    0,0(11,7)          AIJ * BJK
000288      7A 02 0 000                       AE    0,0(2)             CIJ + AIJ * BJK
00028C      70 02 0 000                       STE   0,0(2)             CIK = CIK + AIJ * BJK
000290      87 78 A 04C                       BXLE  7,8,RET1           TEST AND INCREMENT J
000294      41 22 0 004                       LA    2,4(2)             MOVE TO NEXT ELEMENT OF C
000298      87 BC A 046                       BXLE  11,12,RET2         TEST AND INCREMENT K
00029C      87 34 A 042                       BXLE  3,4,RET3           TEST AND INCREMENT I
0002A0      98 0F A 07C                       LM    0,15,TEMP          FINISHED - RESTORE REGISTERS
0002A4      41 EE 0 018                       LA    14,24(14)          GET RETURN ADDRESS
0002A8      07 FE                             BCR   15,14              RETURN TO CALLING PROGRAM
0002AC      00000001                 ONE      DC    F'1'               ADDRESS COMPUTATION CONSTANT
0002B0                               TEMP     DS    16F                REGISTER SAVE AREA
                                              END   BEGIN
```

Figure 118. Assembly listing of a matrix multiplication subroutine



```
          +.22000000 E+02   -.80000000 E+01

          +.17000000 E+02   +.20000000 E+01

          +.70000000 E+01   -.60000000 E+01

          +.15000000 E+02   +.56000000 E+02








                   +.11000000 E+02

                   -.10000000 E+02

                   +.15000000 E+02
```

Figure 119. Printouts of the two product matrices produced when the programs of Figures 117 and 118 were run

The BXLE and BXH Instructions in Solving Laplace's Equation

In this section we shall explore further loop control ideas through the medium of a common problem in scientific computing. Those who have worked with the problem will perhaps find it especially indicative of the powers of the System/360, but the presentation assumes no knowledge of the mathematics involved. Furthermore, the full program will be developed in several stages, beginning with a very much simplified special case.

Let us suppose that we are given an array of 20 single-length floating-point numbers as indicated in Figure 120. These 20 numbers are stored in consecutive fullword locations beginning at symbol MATRIX, in row order, as suggested in Figure 121. Looking at either of these figures, we shall refer to the "outside" numbers as *boundary* values, and the other six as *interior* points. Our task is to find a solution to Laplace's equation for the boundary values given. We shall say that we have arrived at a "solution" when each interior point is the average of its four neighbors; the exact solution is shown in Figure 122, in which we see, for instance, that the interior value 6 is the average of 8, 9, 4, and 3; 4 is the average of 6, 6, 2, and 2.

At the outset, of course, we do not know the values of the interior points that solve the problem, or at best we have only some initial guesses. We shall find the solution by continually improving a set of approximations to the solution values, as follows.

Beginning with the given initial values, that is, with zeroes for the interior points, we compute a new value for the interior point at MATRIX+24. This is done by averaging the four points at MATRIX+4, MATRIX+20, MATRIX+28, and MATRIX+44. The average is 4.25, which we immediately store in MATRIX+24. We now move to the point at MATRIX+28, and compute a new value for it by averaging its four neighbors, using the new value at MATRIX+24. The average of 4.25, 6, 0, and 0 is about 2.56, which is immediately stored in MATRIX+28. A similar process places a value of 1.39 in MATRIX+32. Moving down to the next row, we find a new value for the interior point at MATRIX+44, by averaging the values at MATRIX+24, MATRIX+40, MATRIX+48, and MATRIX+64; this value is 2.06. In computing a new value at MATRIX+48, we shall be using two recently computed approximations, the ones at MATRIX+44 and MATRIX+28; the average is 1.15. At MATRIX+52 we get a value of 0.63. The *new* values are shown in Figure 123.

12	9	6	3	0
8	0	0	0	0
4	0	0	0	0
0	0	0	0	0

Figure 120. Array of numbers used to illustrate solution of Laplace's equation

MATRIX	MATRIX+4	MATRIX+8	MATRIX+12	MATRIX+16
MATRIX+20	MATRIX+24	MATRIX+28	MATRIX+32	MATRIX+36
MATRIX+40	MATRIX+44	MATRIX+48	MATRIX+52	MATRIX+56
MATRIX+60	MATRIX+64	MATRIX+68	MATRIX+72	MATRIX+76

Figure 121. Addressing system used in illustration of solution of Laplace's equation

12	9	6	3	0
8	6	4	2	0
4	3	2	1	0
0	0	0	0	0

Figure 122. Exact solution of the example of Laplace's equation

12.00	9.00	6.00	3.00	0.00
8.00	4.25	2.56	1.39	0.00
4.00	2.06	1.15	0.63	0.00
0.00	0.00	0.00	0.00	0.00

Figure 123. After the first sweep of the method of solving the example of Laplace's equation

This completes the computations in one *sweep* of the array. Are we finished? Clearly not, as we can see by looking at the solution in Figure 122. But naturally in a real application we would not be doing the problem if we knew the solution. One acceptable way to determine whether the process has converged to a solution is to form the sum of absolute values of the *residues* at the interior points. The absolute value of the residue at a point is just the difference between the new and the old value, with any minus signs changed to plus. In the example at hand, the old values were all zero, so the sum of the residues is just the sum of the new values, or 12.04.

Now we proceed to another complete sweep of the array, in all cases using the most recently computed values for the interior points. For MATRIX+24 this time we average the numbers 9.00, 8.00, 2.56, and 2.06, to get 5.40. Before actually storing this result back in MATRIX+24, we subtract the old value to get the residue of 1.15. The new value at MATRIX+28 is 3.49, with a residue of 0.93; this residue is added to the 1.15. This process is continued until we have computed new values for all six interior points, thus completing the second sweep. The values are as shown in Figure 124; the sum of the residues is 3.83. This sum of residues is a good bit less than on the first sweep, but still not very good. We sweep the array again, to get the values shown in Figure 125 and a sum of residues of 1.35. Another sweep leads to Figure 126 and a sum of residues of 0.48.

12.00	9.00	6.00	3.00	0.00
8.00	5.40	3.49	1.78	0.00
4.00	2.64	1.69	0.87	0.00
0.00	0.00	0.00	0.00	0.00

Figure 124. After the second sweep of the example of Laplace's equation

12.00	9.00	6.00	3.00	0.00
8.00	5.78	3.81	1.92	0.00
4.00	2.87	1.89	0.95	0.00
0.00	0.00	0.00	0.00	0.00

Figure 125. After the third sweep

12.00	9.00	6.00	3.00	0.00
8.00	5.92	3.93	1.97	0.00
4.00	2.95	1.95	0.98	0.00
0.00	0.00	0.00	0.00	0.00

Figure 126. After the fourth sweep

The decision when to stop must be made by the problem formulator. If we were to require the sum of the residues to be less than 0.01, it would take another four iterations, giving the values shown in Figure 127.

12.0000	9.0000	6.0000	3.0000	0.0000
8.0000	5.9986	3.9988	1.9995	0.0000
4.0000	2.9992	1.9993	0.9997	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000

Figure 127. Final solution of Laplace's equation, with a tolerance of 0.01

So much for the most basic ideas of the method of solution. Let us now write a program to do this, taking just the same problem.

This is of course a very small example. In practice, the arrays of interest are a great deal larger, and dozens or even hundreds of sweeps may be required for convergence. We therefore quickly become very interested in making the program as efficient as possible. To this end, we would in particular like to minimize the amount of loop-control red tape.

Suppose that we use general purpose register 1 as a base register, loading it initially with the address of the first element of the array, MATRIX. We then use general purpose register 5 as an index register, loading it initially with zero. Now observe that the following five instructions refer to the five array locations we need in computing a new value for MATRIX +24:

```
LE    2,4(1,5)
AE    2,20(1,5)
AE    2,28(1,5)
AE    2,44(1,5)
AE    4,24(1,5)
```

In the first instruction, for instance, the effective address is the sum of: the absolute equivalent of MATRIX, from the base; zero, from the index register; and 4, from the displacement. The other addresses are the same, except for the various other displacements.

Now observe what happens if we increment the index by 4: every effective address is increased by 4, leading to the five addresses needed to compute a new value for MATRIX+28. Incrementing the index by 4 again leads to the five addresses needed for MATRIX+32. We are now at the end of the row. We should like to terminate this loop and branch out to the larger loop that increments the base register. We know, of course, that incrementing and testing a register can be done with one instruction, Branch on Index Low or Equal (BXLE).

Upon reaching the end of the row, we should like to add 20 to the base register, set the index back to zero, and go to work again: with these values for the base and index, the effective addresses will be those needed for MATRIX+44. Then incrementing the index by 4, with the base register contents unchanged, leads to MATRIX+48. One more increment leads to the last interior point.

To summarize: With the base and index register contents described above, we can access all the data values we need by using fixed displacements and varying the base and index. The importance of all this, from a program execution time standpoint, is that the incrementing and testing of the registers can be done with one instruction each, if we set up the proper increment and final values for the BXLE instruction.

The increment for the index is clearly 4. The limit is found to be 8, remembering that the test is made *after* adding the increment and that the branch is made if the sum is low *or equal*. The increment for the base is 20 and the limit is 20 more than MATRIX.

We are almost ready to display a program segment to do all this in less space than it takes to describe it. The only additional feature needed is a way to limit the execution of the program in case for any reason convergence is much slower than anticipated. To this end we set up a fixed-point value LIMIT which is loaded into register 4. This value is counted down after each sweep; if it ever reaches zero, we come to a Supervisor Call to terminate program execution.

The program is shown in Figure 128. The first five processing instructions are for initializing the various control registers. At AGAIN we have the beginning of a new sweep; we set the sum of the absolute values of the residues to zero. Then the base register is loaded with the initial address of the array, and at LOOPA the index is set to zero. At LOOPB begin the actual arithmetic instructions. The first four of these form the sum of the four surrounding values, which is divided by 4. Next we pick up the old value in floating-point register 4 and subtract the newly computed value from it. The difference is loaded positively into register 6, to destroy any possible minus sign. This absolute value is added to the previous sum of the residues in floating register zero. Finally, the new value is stored back in the array.

We now reach the BXLE instruction to increment and test the index register. If the branch is taken, we go back to compute another new value in the same row. If the branch is not taken, we reach the BXLE that increments and tests the base register. If this branch is taken, we go back one extra instruction in order to reset the index to zero. If the branch is not taken, a sweep has been completed. A floating-point comparison of the sum of the residues against the preestablished tolerance determines whether the process has converged. If not, we use a Branch on Count instruction to go back for another sweep – unless the limit of sweeps has been reached.

The printout of Figure 129 shows the computed values with a tolerance of 0.001. Eleven sweeps are required.

Now that we have a clear idea of the basic addressing scheme, let us generalize the program. It is, after all, not very helpful to have a program that can only be used with four-by-five arrays. The next stage in the program development will be to make M, the number of rows, and N, the number of columns, parameters of the assembly. Then we can assemble an object program to handle any size array, simply by changing a couple of cards.

This approach will require use of the Equate Symbol (EQU) assembler instruction. We shall write:

```
M     EQU    4
N     EQU    10
```

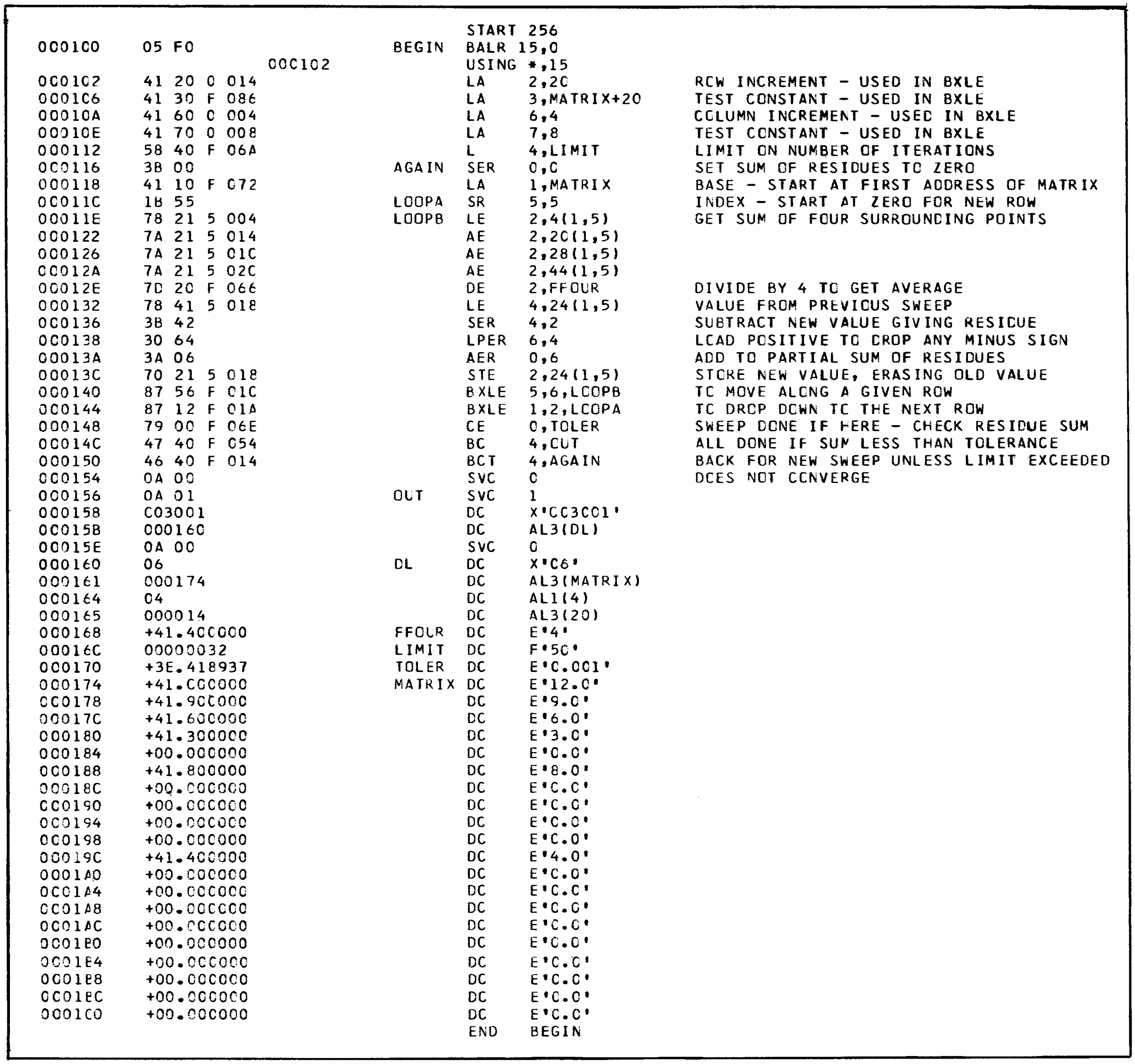
or whatever we wish for a particular array. Then, wherever M or N appears in the source program, the value written will be substituted – by the assembler program. We can therefore write the displacements and the instructions for setting and incrementing the registers in terms of M and N. The program as thus modified is shown in Figure 130. We see that the EQU technique led to the same displacements as before.

In order to appreciate the need for speed in larger problems, let us go to a larger array. We substitute cards giving:

```
M     EQU    11
N     EQU    16
```

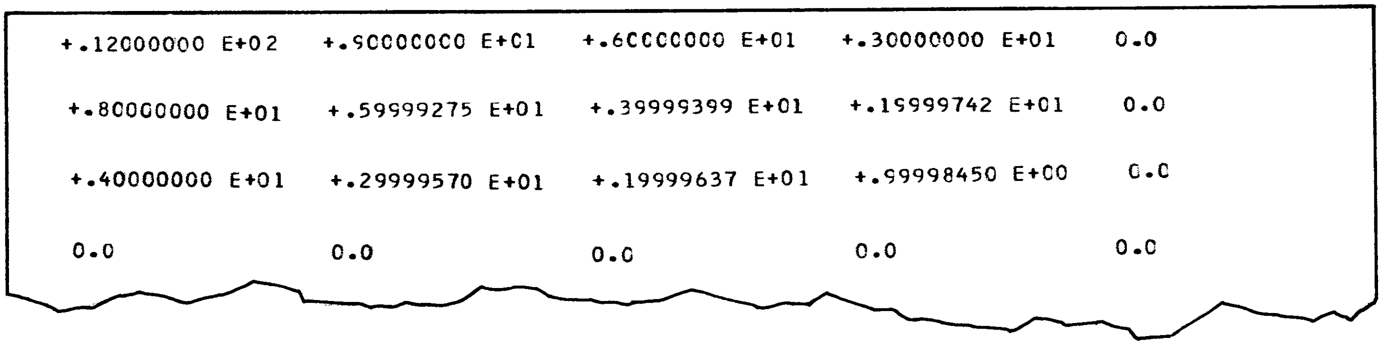
The entry for MATRIX is modified to provide 176 fullwords instead of 20, and input values for the boundary are established. The "shape" of the boundary is very similar to that in the previous example. In the "upper left" there is a value of 30; the values along the top decrease by two's, and along the side by three's. The values on the bottom and right are all zero. All interior points start at zero, as before. The new assembly listing is shown in Figure 131.

This object program was run, using the 0.01 tolerance shown. Convergence required 116 iterations, involving the execution of some 165,000 instructions. Now, this number of instructions can be executed in less than one second on the medium and large models of System/360, so for this problem speed is not a major consideration. But it must be realized that problems of this sort are seldom as small as this one, where



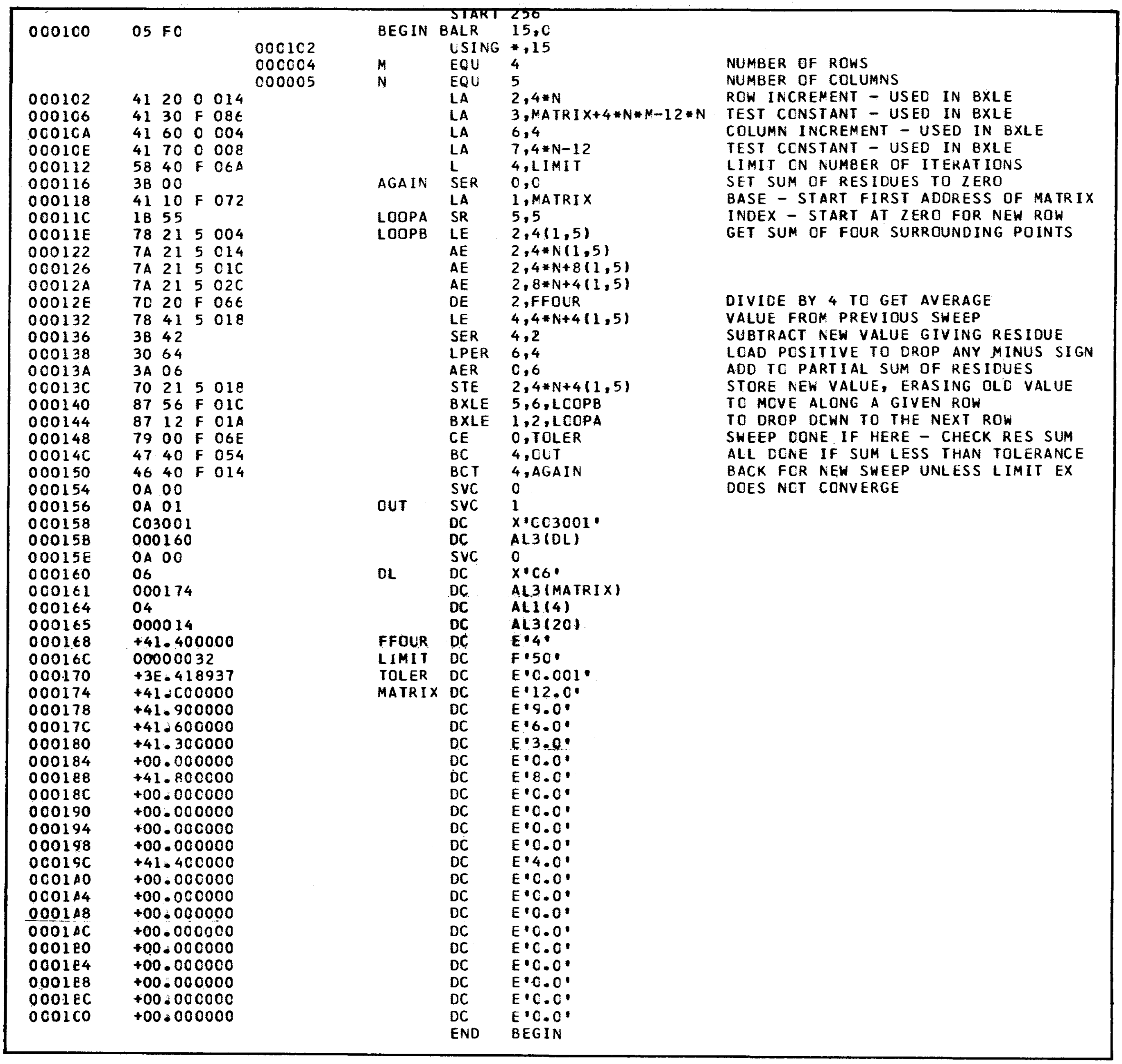
```
                                                      START 256
000100     05 F0                             BEGIN    BALR 15,0
                            000102                    USING *,15
000102     41 20 0 014                                LA    2,20            ROW INCREMENT - USED IN BXLE
000106     41 30 F 086                                LA    3,MATRIX+20     TEST CONSTANT - USED IN BXLE
00010A     41 60 0 004                                LA    6,4             COLUMN INCREMENT - USED IN BXLE
00010E     41 70 0 008                                LA    7,8             TEST CONSTANT - USED IN BXLE
000112     58 40 F 06A                                L     4,LIMIT         LIMIT ON NUMBER OF ITERATIONS
000116     3B 00                             AGAIN    SER   0,0             SET SUM OF RESIDUES TO ZERO
000118     41 10 F 072                                LA    1,MATRIX        BASE - START AT FIRST ADDRESS OF MATRIX
00011C     1B 55                             LOOPA    SR    5,5             INDEX - START AT ZERO FOR NEW ROW
00011E     78 21 5 004                       LOOPB    LE    2,4(1,5)        GET SUM OF FOUR SURROUNDING POINTS
000122     7A 21 5 014                                AE    2,20(1,5)
000126     7A 21 5 01C                                AE    2,28(1,5)
00012A     7A 21 5 02C                                AE    2,44(1,5)
00012E     7D 20 F 066                                DE    2,FFOUR         DIVIDE BY 4 TO GET AVERAGE
000132     78 41 5 018                                LE    4,24(1,5)       VALUE FROM PREVIOUS SWEEP
000136     3B 42                                      SER   4,2             SUBTRACT NEW VALUE GIVING RESIDUE
000138     30 64                                      LPER  6,4             LOAD POSITIVE TO DROP ANY MINUS SIGN
00013A     3A 06                                      AER   0,6             ADD TO PARTIAL SUM OF RESIDUES
00013C     70 21 5 018                                STE   2,24(1,5)       STORE NEW VALUE, ERASING OLD VALUE
000140     87 56 F 01C                                BXLE  5,6,LOOPB       TO MOVE ALONG A GIVEN ROW
000144     87 12 F 01A                                BXLE  1,2,LOOPA       TO DROP DOWN TO THE NEXT ROW
000148     79 00 F 06E                                CE    0,TOLER         SWEEP DONE IF HERE - CHECK RESIDUE SUM
00014C     47 40 F 054                                BC    4,OUT           ALL DONE IF SUM LESS THAN TOLERANCE
000150     46 40 F 014                                BCT   4,AGAIN         BACK FOR NEW SWEEP UNLESS LIMIT EXCEEDED
000154     0A 00                                      SVC   0               DOES NOT CONVERGE
000156     0A 01                             OUT      SVC   1
000158     C03001                                     DC    X'C03001'
00015B     000160                                     DC    AL3(DL)
00015E     0A 00                                      SVC   0
000160     06                                DL       DC    X'06'
000161     000174                                     DC    AL3(MATRIX)
000164     04                                         DC    AL1(4)
000165     000014                                     DC    AL3(20)
000168     +41.400000                        FFOUR    DC    E'4'
00016C     00000032                          LIMIT    DC    F'50'
000170     +3E.418937                        TOLER    DC    E'0.001'
000174     +41.C00000                        MATRIX   DC    E'12.0'
000178     +41.900000                                 DC    E'9.0'
00017C     +41.600000                                 DC    E'6.0'
000180     +41.300000                                 DC    E'3.0'
000184     +00.000000                                 DC    E'0.0'
000188     +41.800000                                 DC    E'8.0'
00018C     +00.000000                                 DC    E'0.0'
000190     +00.000000                                 DC    E'0.0'
000194     +00.000000                                 DC    E'0.0'
000198     +00.000000                                 DC    E'0.0'
00019C     +41.400000                                 DC    E'4.0'
0001A0     +00.000000                                 DC    E'0.0'
0001A4     +00.000000                                 DC    E'0.0'
0001A8     +00.000000                                 DC    E'0.0'
0001AC     +00.000000                                 DC    E'0.0'
0001B0     +00.000000                                 DC    E'0.0'
0001B4     +00.000000                                 DC    E'0.0'
0001B8     +00.000000                                 DC    E'0.0'
0001BC     +00.000000                                 DC    E'0.0'
0001C0     +00.000000                                 DC    E'0.0'
                                                      END   BEGIN
```

Figure 128. Assembly listing of a program for solving Laplace's equation for the boundary values shown in Figure 120



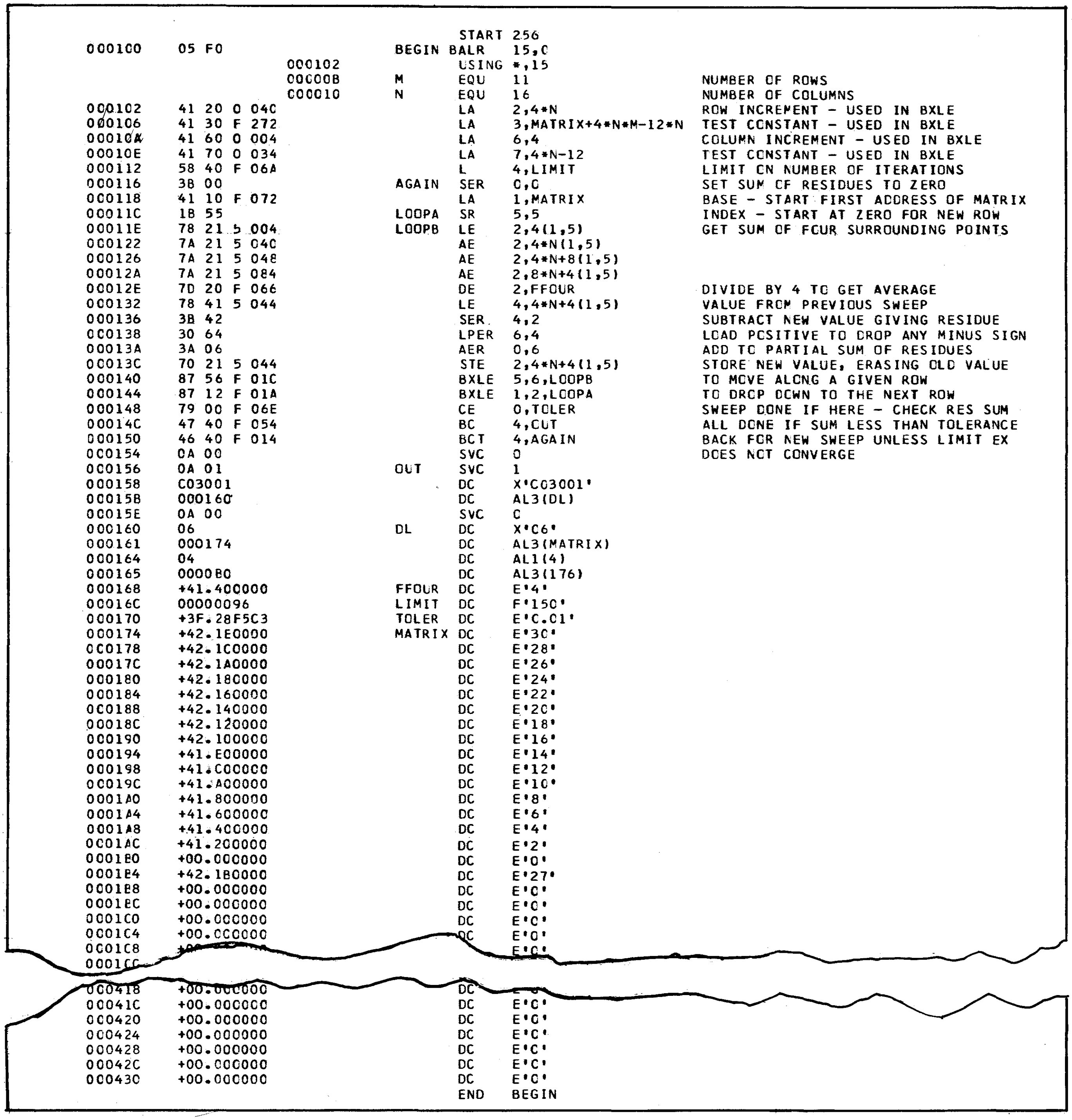
```
+.12000000 E+02    +.90000000 E+01    +.60000000 E+01    +.30000000 E+01    0.0

+.80000000 E+01    +.59999275 E+01    +.39999399 E+01    +.19999742 E+01    0.0

+.40000000 E+01    +.29999570 E+01    +.19999637 E+01    +.99998450 E+00    0.0

0.0                0.0                0.0                0.0                0.0
```

Figure 129. Final solution of Laplace's equation, with a tolerance of 0.001



```
                                                START 256
000100     05 F0                         BEGIN BALR   15,C
                          000102               USING *,15
                          000004         M     EQU    4                       NUMBER OF ROWS
                          000005         N     EQU    5                       NUMBER OF COLUMNS
000102     41 20 0 014                         LA     2,4*N                   ROW INCREMENT - USED IN BXLE
000106     41 30 F 086                         LA     3,MATRIX+4*N*M-12*N     TEST CCNSTANT - USED IN BXLE
0001CA     41 60 0 004                         LA     6,4                     COLUMN INCREMENT - USED IN BXLE
00010E     41 70 0 008                         LA     7,4*N-12                TEST CCNSTANT - USED IN BXLE
000112     58 40 F 06A                         L      4,LIMIT                 LIMIT CN NUMBER OF ITERATIONS
000116     3B 00                         AGAIN SER    0,C                     SET SUM OF RESIDUES TO ZERO
000118     41 10 F 072                         LA     1,MATRIX                BASE - START FIRST ADDRESS OF MATRIX
00011C     1B 55                         LOOPA SR     5,5                     INDEX - START AT ZERO FOR NEW ROW
00011E     78 21 5 004                   LOOPB LE     2,4(1,5)                GET SUM OF FOUR SURROUNDING POINTS
000122     7A 21 5 014                         AE     2,4*N(1,5)
000126     7A 21 5 01C                         AE     2,4*N+8(1,5)
00012A     7A 21 5 02C                         AE     2,8*N+4(1,5)
00012E     7D 20 F 066                         DE     2,FFOUR                 DIVIDE BY 4 TO GET AVERAGE
000132     78 41 5 018                         LE     4,4*N+4(1,5)            VALUE FROM PREVIOUS SWEEP
000136     3B 42                               SER    4,2                     SUBTRACT NEW VALUE GIVING RESIDUE
000138     30 64                               LPER   6,4                     LOAD PCSITIVE TO DROP ANY MINUS SIGN
00013A     3A 06                               AER    C,6                     ADD TC PARTIAL SUM OF RESIDUES
00013C     70 21 5 018                         STE    2,4*N+4(1,5)            STORE NEW VALUE, ERASING OLD VALUE
000140     87 56 F 01C                         BXLE   5,6,LCOPB               TC MCVE ALONG A GIVEN ROW
000144     87 12 F 01A                         BXLE   1,2,LCOPA               TO DROP DCWN TO THE NEXT ROW
000148     79 00 F 06E                         CE     0,TOLER                 SWEEP DONE IF HERE - CHECK RES SUM
00014C     47 40 F 054                         BC     4,CUT                   ALL DCNE IF SUM LESS THAN TOLERANCE
000150     46 40 F 014                         BCT    4,AGAIN                 BACK FCR NEW SWEEP UNLESS LIMIT EX
000154     0A 00                               SVC    0                       DOES NCT CONVERGE
000156     0A 01                         OUT   SVC    1
000158     C03001                              DC     X'CC3001'
00015B     000160                              DC     AL3(DL)
00015E     0A 00                               SVC    0
000160     06                            DL    DC     X'C6'
000161     000174                              DC     AL3(MATRIX)
000164     04                                  DC     AL1(4)
000165     000014                              DC     AL3(20)
000168     +41.400000                    FFOUR DC     E'4'
00016C     00000032                      LIMIT DC     F'50'
000170     +3E.418937                    TOLER DC     E'0.001'
000174     +41.C00000                   MATRIX DC     E'12.0'
000178     +41.900000                          DC     E'9.0'
00017C     +41.600000                          DC     E'6.0'
000180     +41.300000                          DC     E'3.0'
000184     +00.000000                          DC     E'0.0'
000188     +41.800000                          DC     E'8.0'
00018C     +00.000000                          DC     E'0.0'
000190     +00.000000                          DC     E'0.0'
000194     +00.000000                          DC     E'0.0'
000198     +00.000000                          DC     E'0.0'
00019C     +41.400000                          DC     E'4.0'
0001A0     +00.000000                          DC     E'0.0'
0001A4     +00.000000                          DC     E'0.0'
0001A8     +00.000000                          DC     E'0.0'
0001AC     +00.000000                          DC     E'0.0'
0001B0     +00.000000                          DC     E'0.0'
0001B4     +00.000000                          DC     E'0.0'
0001B8     +00.000000                          DC     E'0.0'
0001BC     +00.000000                          DC     E'0.0'
0001C0     +00.000000                          DC     E'0.0'
                                               END    BEGIN
```

Figure 130. Assembly listing of the program of Figure 128, modified to use EQU entries to define the size of the matrix



```
                                           START 256
000100     05 F0                     BEGIN BALR  15,0
                          000102           USING *,15
                          00000B     M     EQU   11                 NUMBER OF ROWS
                          000010     N     EQU   16                 NUMBER OF COLUMNS
000102     41 20 0 040                     LA    2,4*N              ROW INCREMENT - USED IN BXLE
000106     41 30 F 272                     LA    3,MATRIX+4*N*M-12*N TEST CONSTANT - USED IN BXLE
00010A     41 60 0 004                     LA    6,4                COLUMN INCREMENT - USED IN BXLE
00010E     41 70 0 034                     LA    7,4*N-12           TEST CONSTANT - USED IN BXLE
000112     58 40 F 06A                     L     4,LIMIT            LIMIT ON NUMBER OF ITERATIONS
000116     3B 00                     AGAIN SER   0,0                SET SUM OF RESIDUES TO ZERO
000118     41 10 F 072                     LA    1,MATRIX           BASE - START FIRST ADDRESS OF MATRIX
00011C     1B 55                     LOOPA SR    5,5                INDEX - START AT ZERO FOR NEW ROW
00011E     78 21 5 004               LOOPB LE    2,4(1,5)           GET SUM OF FOUR SURROUNDING POINTS
000122     7A 21 5 040                     AE    2,4*N(1,5)
000126     7A 21 5 048                     AE    2,4*N+8(1,5)
00012A     7A 21 5 084                     AE    2,8*N+4(1,5)
00012E     7D 20 F 066                     DE    2,FFOUR            DIVIDE BY 4 TO GET AVERAGE
000132     78 41 5 044                     LE    4,4*N+4(1,5)       VALUE FROM PREVIOUS SWEEP
000136     3B 42                           SER   4,2                SUBTRACT NEW VALUE GIVING RESIDUE
000138     30 64                           LPER  6,4                LOAD POSITIVE TO DROP ANY MINUS SIGN
00013A     3A 06                           AER   0,6                ADD TO PARTIAL SUM OF RESIDUES
00013C     70 21 5 044                     STE   2,4*N+4(1,5)       STORE NEW VALUE, ERASING OLD VALUE
000140     87 56 F 01C                     BXLE  5,6,LOOPB          TO MOVE ALONG A GIVEN ROW
000144     87 12 F 01A                     BXLE  1,2,LOOPA          TO DROP DOWN TO THE NEXT ROW
000148     79 00 F 06E                     CE    0,TOLER            SWEEP DONE IF HERE - CHECK RES SUM
00014C     47 40 F 054                     BC    4,CUT              ALL DONE IF SUM LESS THAN TOLERANCE
000150     46 40 F 014                     BCT   4,AGAIN            BACK FOR NEW SWEEP UNLESS LIMIT EX
000154     0A 00                           SVC   0                  DOES NOT CONVERGE
000156     0A 01                     OUT   SVC   1
000158     C03001                          DC    X'C03001'
00015B     000160                          DC    AL3(DL)
00015E     0A 00                           SVC   0
000160     06                        DL    DC    X'C6'
000161     000174                          DC    AL3(MATRIX)
000164     04                              DC    AL1(4)
000165     0000B0                          DC    AL3(176)
000168     +41.400000                FFOUR DC    E'4'
00016C     00000096                  LIMIT DC    F'150'
000170     +3F.28F5C3                TOLER DC    E'0.01'
000174     +42.1E0000                MATRIX DC   E'30'
000178     +42.1C0000                      DC    E'28'
00017C     +42.1A0000                      DC    E'26'
000180     +42.180000                      DC    E'24'
000184     +42.160000                      DC    E'22'
000188     +42.140000                      DC    E'20'
00018C     +42.120000                      DC    E'18'
000190     +42.100000                      DC    E'16'
000194     +41.E00000                      DC    E'14'
000198     +41.C00000                      DC    E'12'
00019C     +41.A00000                      DC    E'10'
0001A0     +41.800000                      DC    E'8'
0001A4     +41.600000                      DC    E'6'
0001A8     +41.400000                      DC    E'4'
0001AC     +41.200000                      DC    E'2'
0001B0     +00.000000                      DC    E'0'
0001B4     +42.1B0000                      DC    E'27'
0001B8     +00.000000                      DC    E'0'
0001BC     +00.000000                      DC    E'0'
0001C0     +00.000000                      DC    E'0'
0001C4     +00.000000                      DC    E'0'
0001C8     +00                                   E'0'
0001CC
000418     +00.000000                      DC
00041C     +00.000000                      DC    E'0'
000420     +00.000000                      DC    E'0'
000424     +00.000000                      DC    E'0'
000428     +00.000000                      DC    E'0'
00042C     +00.000000                      DC    E'0'
000430     +00.000000                      DC    E'0'
                                           END   BEGIN
```

Figure 131. Assembly listing of the program of Figure 130, modified to accept an 11 by 16 matrix, and with suitably modified boundary values

we have only 11 rows and 16 columns. Problems with more than 100 rows and columns are not exceptional, and it is frequently necessary to solve each such problem with many variations in the boundary values. It is not really difficult to devise tasks of this general sort that can use dozens of hours of time on the fastest computers available. We have thus come to a situation where anything reasonable that can reduce object program running time is worth considerable effort in numerical analysis and programming.

Let us first turn to a possibility for improvement that will nicely illustrate the powers of the instruction repertoire and register operation of System/360.

Suppose that the boundary values had been arranged a little differently, as shown in the four-by-five case below; the zeros are at the left and top instead of the right and bottom.

0	0	0	0	0
0	0	0	0	4
0	0	0	0	8
0	3	6	9	12

This means that as we sweep along the first row, all the new approximations computed in the first sweep will be zero except the last one in the row. In the second sweep, all but the last two values in the first row will still be zeros, etc. The new approximations are working "upstream," so to speak. Might not convergence be faster if the sweeps worked from right to left and bottom to top, rather than the way they do? To prove that there is some possible value to the scheme, we can run the program as it stands, but with the rearranged boundary values. Convergence this time requires 122 iterations as against 116. This is not a truly huge difference, but rewriting the program will be sufficiently educational to be well worth the effort.

The task before us is to rework the sweep logic. The basic tool this time will be the Branch on Index High (BXH) instruction, rather than the BXLE. This change will require rather different starting values for the index and base registers, and for their decrement and limit registers. We shall finally see an example where the decrement and limit can usefully be the same value, in the same register.

This time we want the base register, which we remember picks the row, to start at the second from the last row. The address of the first element in this row is MATRIX+4N(M–3); a Load Address instruction can put the value there. Register 8 is used for the base this time. Register 12 is the decrement register; it needs to contain –4N this time. This is best done by loading 4N into the register with a Load Address and then loading the register negatively from itself. The limiting value must be considered carefully. We recall that BXH first adds the increment (which means a subtraction as we have arranged things), then branches if the sum is greater than the limit (*not* greater than *or equal*). Since we do want the loop executed with the base register containing the address of the first row, the limit will have to be the address that the row before the first row would have, if there were one. We accordingly load register 13 with MATRIX–4N.

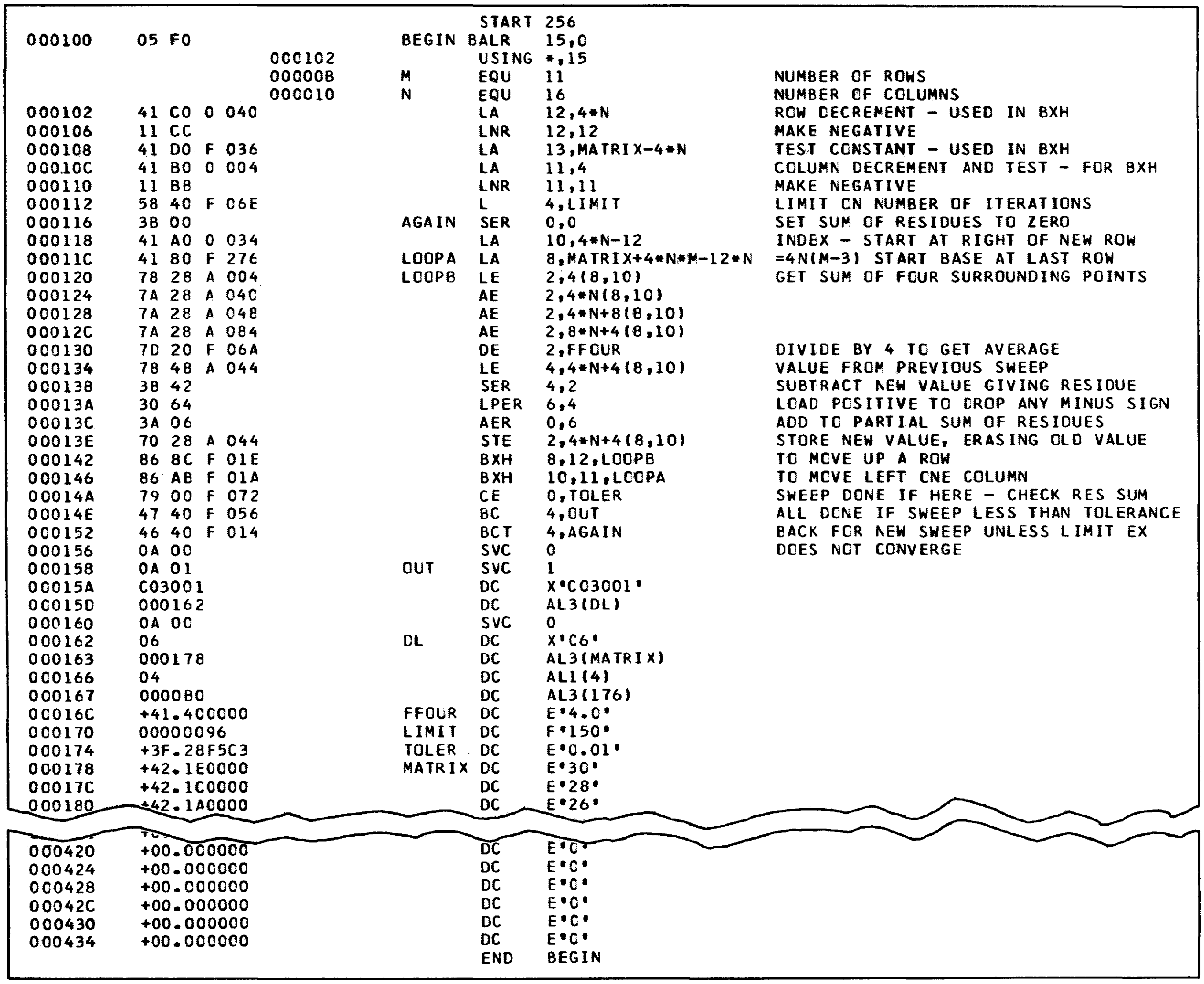
The index register this time should initially pick out the second from the last column. The element in the second from the last column in a row is 4(N–3) bytes beyond the first element in that row; this is accordingly the value we place in register 10, the index register. The decrement value is –4. The limit value, as it happens, is also –4. We want the index loop to be executed with the index containing zero, which requires a limit 4 less than zero, the way the BXH instruction works. Since the decrement and limit are both –4, there is no reason not to put them in the same register, 11 in this case. This is based on the phrase in the description of the BXH: "The limit register is odd and is either the same as the increment or one greater". That is, if the increment register is odd, then the limit is in the same register.

These changes in the program are readily made, leading to Figure 132. The BXH instructions are sequenced so that the base register varies most rapidly. This means that we first go up the rightmost interior column, then go up the next to the rightmost interior column, etc. If it were thought preferable, it would be a simple matter to have the index vary most rapidly, so that we would be going across the rows from bottom to top, instead of going up the columns from right to left.

This program was run requiring the same number of iterations to converge that the previous program did on the original boundary values.

As noted, this is not a very large percentage increase, however instructive it may be. We therefore turn to another method for speeding up convergence that will definitely be important.

Suppose that each time a new approximation is computed, we inspect the difference between the new value and the old value. We can assume pretty safely that yet further change will be required for convergence; why not extrapolate a bit? In other words, we shall add to the old value some constant times the difference between the new and old. This constant we my call ω. Writing u for the values of the matrix elements and using "old" and "new" subscripts to denote the previous and newly computed values, the formula



```
                                         START  256
000100  05 F0                     BEGIN  BALR   15,0
                        000102           USING  *,15
                        00000B    M      EQU    11                   NUMBER OF ROWS
                        000010    N      EQU    16                   NUMBER OF COLUMNS
000102  41 C0 0 040                      LA     12,4*N               ROW DECREMENT - USED IN BXH
000106  11 CC                            LNR    12,12                MAKE NEGATIVE
000108  41 D0 F 036                      LA     13,MATRIX-4*N        TEST CONSTANT - USED IN BXH
00010C  41 B0 0 004                      LA     11,4                 COLUMN DECREMENT AND TEST - FOR BXH
000110  11 BB                            LNR    11,11                MAKE NEGATIVE
000112  58 40 F 06E                      L      4,LIMIT              LIMIT CN NUMBER OF ITERATIONS
000116  3B 00                     AGAIN  SER    0,0                  SET SUM OF RESIDUES TO ZERO
000118  41 A0 0 034                      LA     10,4*N-12            INDEX - START AT RIGHT OF NEW ROW
00011C  41 80 F 276               LOOPA  LA     8,MATRIX+4*N*M-12*N  =4N(M-3) START BASE AT LAST ROW
000120  78 28 A 004               LOOPB  LE     2,4(8,10)            GET SUM OF FCUR SURROUNDING POINTS
000124  7A 28 A 040                      AE     2,4*N(8,10)
000128  7A 28 A 048                      AE     2,4*N+8(8,10)
00012C  7A 28 A 084                      AE     2,8*N+4(8,10)
000130  7D 20 F 06A                      DE     2,FFOUR              DIVIDE BY 4 TC GET AVERAGE
000134  78 48 A 044                      LE     4,4*N+4(8,10)        VALUE FROM PREVIOUS SWEEP
000138  3B 42                            SER    4,2                  SUBTRACT NEW VALUE GIVING RESIDUE
00013A  30 64                            LPER   6,4                  LCAD PCSITIVE TO DROP ANY MINUS SIGN
00013C  3A 06                            AER    0,6                  ADD TC PARTIAL SUM OF RESIDUES
00013E  70 28 A 044                      STE    2,4*N+4(8,10)        STORE NEW VALUE, ERASING OLD VALUE
000142  86 8C F 01E                      BXH    8,12,LOOPB           TC MCVE UP A ROW
000146  86 AB F 01A                      BXH    10,11,LCOPA          TC MCVE LEFT CNE COLUMN
00014A  79 00 F 072                      CE     0,TOLER              SWEEP DONE IF HERE - CHECK RES SUM
00014E  47 40 F 056                      BC     4,OUT                ALL DCNE IF SWEEP LESS THAN TOLERANCE
000152  46 40 F 014                      BCT    4,AGAIN              BACK FCR NEW SWEEP UNLESS LIMIT EX
000156  0A 00                            SVC    0                    DCES NCT CONVERGE
000158  0A 01                     OUT    SVC    1
00015A  C03001                           DC     X'C03001'
00015D  000162                           DC     AL3(DL)
000160  0A 00                            SVC    0
000162  06                        DL     DC     X'C6'
000163  000178                           DC     AL3(MATRIX)
000166  04                               DC     AL1(4)
000167  0000B0                           DC     AL3(176)
00016C  +41.400000                FFOUR  DC     E'4.0'
000170  00000096                  LIMIT  DC     F'150'
000174  +3F.28F5C3                TOLER  DC     E'0.01'
000178  +42.1E0000                MATRIX DC     E'30'
00017C  +42.1C0000                       DC     E'28'
000180  +42.1A0000                       DC     E'26'

000420  +00.000000                       DC     E'C'
000424  +00.000000                       DC     E'C'
000428  +00.000000                       DC     E'C'
00042C  +00.000000                       DC     E'C'
000430  +00.000000                       DC     E'C'
000434  +00.000000                       DC     E'C'
                                         END    BEGIN
```

Figure 132. The program of Figure 131 modified through the use of BXH's instead of BXLE's to sweep in a different manner

becomes

$$u_{new} = u_{old} + \omega\,(\text{average} - u_{old})$$

If the new value were all that was needed, we would write the program to implement this formula. But we also need the difference in explicit form itself, in order to accumulate the sum of the absolute values of the residues. The formula for the difference is:

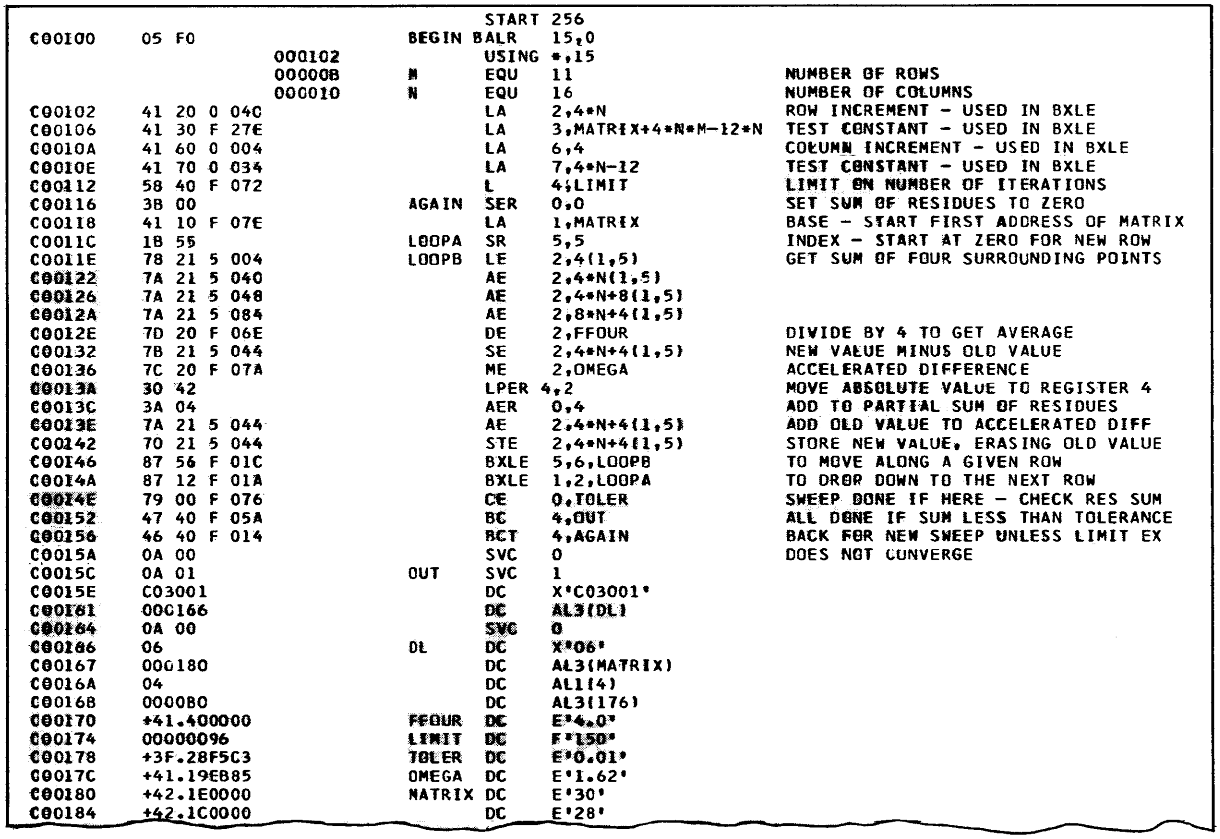
$$\begin{aligned}\text{residue} &= u_{new} - u_{old}\\ &= u_{old} + \omega(\text{average} - u_{old}) - u_{old}\\ &= \omega(\text{average} - u_{old})\end{aligned}$$

In other words, after computing the average as be-before, we can subtract the old value, then multiply by ω to get the residue. The absolute value is taken as before. We finally add the old value to the residue (*with* its minus sign, if any) and store the extrapolated new value. Figure 133 shows the program of Figure 131 modified to compute the extrapolated residue and new value in the manner just described. Note that ω is called OMEGA in the program.

The optimum value of ω is a function of the shape of the region and of the boundary values. For a value of ω =1, we have the original unaccelerated method; values of 2 and greater prevent convergence ever. The program of Figure 133 was run with a number of values for ω, to find the most effective value of ω. This turned out to be 1.62, for which only 22 iterations were required. Clearly, the time saving is highly significant, approaching a factor of six over the unaccelerated version.

Summary

In this chapter we have considered two topics of prime importance to the scientific and engineering user of computers: floating-point operations and efficient loops. System/360 offers powerful capabilities in both areas. In floating operations System/360 provides the guard digit and fully automatic double-length operations, features not found in most other computing systems. For heavily used loops, System/360 provides the power of extremely efficient and flexible loop testing and modification instructions, together with the ability to modify an address with a base register and an index register, both of which may vary.



```
                                               START 256
C00100    05 F0                          BEGIN BALR   15,0
                         000102                USING *,15
                         00000B          M     EQU    11                  NUMBER OF ROWS
                         000010          N     EQU    16                  NUMBER OF COLUMNS
C00102    41 20 0 040                          LA     2,4*N               ROW INCREMENT - USED IN BXLE
C00106    41 30 F 27E                          LA     3,MATRIX+4*N*M-12*N TEST CONSTANT - USED IN BXLE
C0010A    41 60 0 004                          LA     6,4                 COLUMN INCREMENT - USED IN BXLE
C0010E    41 70 0 034                          LA     7,4*N-12            TEST CONSTANT - USED IN BXLE
C00112    58 40 F 072                          L      4;LIMIT             LIMIT ON NUMBER OF ITERATIONS
C00116    3B 00                          AGAIN SER    0,0                 SET SUM OF RESIDUES TO ZERO
C00118    41 10 F 07E                          LA     1,MATRIX            BASE - START FIRST ADDRESS OF MATRIX
C0011C    1B 55                          LOOPA SR     5,5                 INDEX - START AT ZERO FOR NEW ROW
C0011E    78 21 5 004                    LOOPB LE     2,4(1,5)            GET SUM OF FOUR SURROUNDING POINTS
C00122    7A 21 5 040                          AE     2,4*N(1,5)
C00126    7A 21 5 048                          AE     2,4*N+8(1,5)
C0012A    7A 21 5 084                          AE     2,8*N+4(1,5)
C0012E    7D 20 F 06E                          DE     2,FFOUR             DIVIDE BY 4 TO GET AVERAGE
C00132    7B 21 5 044                          SE     2,4*N+4(1,5)        NEW VALUE MINUS OLD VALUE
C00136    7C 20 F 07A                          ME     2,OMEGA             ACCELERATED DIFFERENCE
C0013A    30 42                                LPER 4,2                   MOVE ABSOLUTE VALUE TO REGISTER 4
C0013C    3A 04                                AER    0,4                 ADD TO PARTIAL SUM OF RESIDUES
C0013E    7A 21 5 044                          AE     2,4*N+4(1,5)        ADD OLD VALUE TO ACCELERATED DIFF
C00142    70 21 5 044                          STE    2,4*N+4(1,5)        STORE NEW VALUE, ERASING OLD VALUE
C00146    87 56 F 01C                          BXLE   5,6,LOOPB           TO MOVE ALONG A GIVEN ROW
C0014A    87 12 F 01A                          BXLE   1,2,LOOPA           TO DROR DOWN TO THE NEXT ROW
C0014E    79 00 F 076                          CE     0,TOLER             SWEEP DONE IF HERE - CHECK RES SUM
C00152    47 40 F 05A                          BC     4,OUT               ALL DONE IF SUM LESS THAN TOLERANCE
C00156    46 40 F 014                          BCT    4,AGAIN             BACK FOR NEW SWEEP UNLESS LIMIT EX
C0015A    0A 00                                SVC    0                   DOES NOT CONVERGE
C0015C    0A 01                          OUT   SVC    1
C0015E    C03001                               DC     X'C03001'
C00161    000166                               DC     AL3(DL)
C00164    0A 00                                SVC    0
C00166    06                             DL    DC     X'06'
C00167    000180                               DC     AL3(MATRIX)
C0016A    04                                   DC     AL1(4)
C0016B    0000B0                               DC     AL3(176)
C00170    +41.400000                     FFOUR DC     E'4.0'
C00174    00000096                       LIMIT DC     F'150'
C00178    +3F.28F5C3                     TOLER DC     E'0.01'
C0017C    +41.19EB85                     OMEGA DC     E'1.62'
C00180    +42.1E0000                     NATRIX DC    E'30'
C00184    +42.1C0000                           DC     E'28'
```

Figure 133. The program of Figure 131 modified to develop the residue and the new value by extrapolation

Questions and Exercises.

1. Write the DC instructions for the following short floating-point numbers:

   3.14159265
   −2.78
   38754 x 10^6
   .00000278
   −.000236 x 10^{-7}

2. Write the DC instructions for the following long floating-point numbers:

   3.141592653589793
   −2.78
   −0.003 x 10^{-3}
   3.8 x 10^{30}
   0.000000008

3. Show the "pure" hexadecimal form that the following DC entries will generate in storage (Note that 16777216 equals 16^6 and that .59604644 × 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'
DC  X'C7100000'
```

4. If BIN assumes the value of a positive binary integer, what will general register 14 contain at the end of the Add command?

```
           L    14,BIN
           A    14,CHAR
           .    .
           .    .
           .    .
BIN        DS   F
CHAR       DC   X'46000000'
```

5. What will be in the affected registers after execution of each of the following sets of instructions?

```
a.          LE    2,A
            AE    2,B
            HER   4,2
            .     .
            .     .
            .     .
        A   DC    X'41789ABC'
        B   DC    X'41876544'
b.          LE    6,A
            SR    6,6
            .     .
            .     .
            .     .
        A   DC    E'15'
c.          L     3,A
            A     3,B
            .     .
            .     .
            .     .
        A   DC    E'1.'
        B   DC    X'01000000'
```

6. Assume A, B, and C are floating-point numbers. Write a program segment to evaluate:

$$\frac{A - B \times C}{A + B \times C}$$

7. Assume a_i values are floating-point numbers, N is a binary number and cannot exceed 100. Write a program segment to evaluate:

$$\sum_{i=1}^{N} \frac{a_i}{N}$$

   a. Where the a_i values are in the short form.
   b. Where the a_i values are in the long form.

Chapter 11: Automatic Interrupts

Since System/360 is designed to operate with a minimum of manual intervention, it must be capable of redirecting its activity when prescribed or unusual situations arise. Such situations, which require an interruption of the main program, may be the result of conditions external to the system, in input/output (I/O) units, or in the CPU itself. An I/O operation or the entire job may be finished, and the machine must be told what to do next. Unacceptable input data, a program error, or a machine error may require a corrective routine or, if the error is irrecoverable, a program dump. All these and many more situations necessitate use of the automatic interrupt system. Finally, the programmer himself frequently finds it necessary to switch from the problem state to the supervisory state, by calling the Supervisor via the automatic interrupt system.

In the following pages we shall explore some of the features and implications of the System/360 interrupt system of interest to the prospective programmer. We shall find out what the system does by itself and what must be done by the problem programmer. In the process we shall also gain some familiarity with the control program (called the "Interrupt Supervisor") that handles all System/360 interrupts. No specific level of programming support is assumed in the discussion of control program and problem program functions. (We shall not be overly concerned with the details of the control program, since these are the responsibility of the IBM programmer and are described in the appropriate SRL publication.)

Introductory Concepts

Programming vs Machine Interrupts

Interruptions and changes of state in the CPU necessitated by program or machine errors, completion of input/output operations, and other conditions used to be handled by rather complex programming instead of automatic hardware-controlled interrupts. Where the equipment provided one or more independent channels to permit overlapping I/O operations with processing, the *program* had to interrogate the channels repeatedly to determine whether they were free to start an I/O operation, such as reading input data or printing out results. At the end of each operation, the interrogation would start all over again, before the next I/O operation could begin.

In contrast, in the System/360 the channels themselves signal the processing unit when they become free upon completion of an I/O operation. The channel signals are recognized by *electronic circuitry,* which *automatically* calls in the interrupt supervisory program to initiate and handle the I/O interrupt. This automatic interrupt procedure results in far more efficient utilization of the processor than is possible with a program of repeated interrogation.

Similarly, in processing arithmetic operations, it used to be the concern of the problem *programmer* to tell the machine what to do in case of an overflow. He had the choice of either letting the computer grind to a halt, for manual intervention and correction, or inserting a Branch on Condition to a fixup routine after each instruction that might conceivably result in an overflow. With the automatic interrupt system, however, the *machine itself* recognizes the overflow condition and automatically fetches the address of a corrective routine from a fixed location in storage. The CPU then branches to this routine, while temporarily interrupting the program in progress. Again, the program interrupt is accomplished automatically by circuitry, while the branch to the overflow fixup routine is handled by the interrupt supervisory program; the overflow fixup routine itself is, of course, the problem programmer's concern, as before.

At this point we begin to see that there are really *two* types of *independently prepared programs* stored in any System/360 machine: one is the IBM-prepared supervisory program ("Interrupt Supervisor") that handles the various types of machine interrupts and automatic branching; the other consists of the user's application program and various fixup and other corrective routines, to which the machine is directed to branch after the interrupt. Only the application program and corrective branch routines are prepared by the "problem" programmer at the installation; the IBM-prepared supervisory program is not normally his concern. Later on we shall explore how the System/360 architecture distinguishes between the two types of independently prepared programs in storage.

Return to Problem Program

After an interrupt has occurred and has been taken care of, how does the system "remember" the point of return from which to resume processing the problem program? In the Branch on Condition type of situation discussed earlier, the programmer himself provided the linkage from the branch back to the original program. With an automatic interrupt, however, the programmer does not know the return point and, therefore, *the machine itself must save the return address* in a fixed storage location. The return address alone, however, is insufficient information to automatically control the interrupt action and return. Along with the address, the complete status of the system at the time of the interrupt must be recorded, including the cause of the interrupt and related information. This will make it possible, at the conclusion of the interrupt routine, for the machine to be restored to the state existing before the interrupt.

The Program Status Word (PSW)

The necessary status information of the system at the time of interruption is contained in a doubleword (eight bytes) known as the *program status word* (PSW). We shall look at the format and precise information contained in the PSW in more detail later on, but suffice it to say for the moment that the controlling or "current" PSW contains the location of the next instruction, the program state of the CPU (whether interruptible or not, stopped or operating, running or waiting, and in a problem or supervisory state), the length of the last instruction, and the outcome of arithmetic or logical operations (called *condition code*). The current PSW is essentially equivalent to the "control register" of earlier computers.

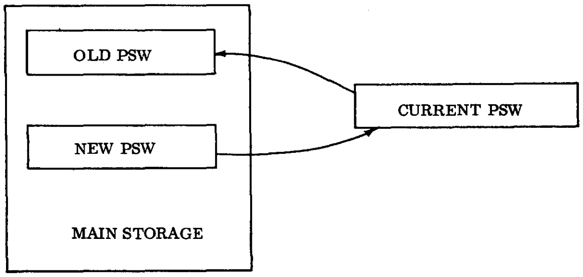


Figure 134. PSW switching during interrupt

Since the relevant status information of the system is available at any time in the current PSW, all the machine need do upon the occurrence of an interrupt is to record in some way the cause of the interrupt and then file away the current PSW at a fixed doubleword location in main storage; from there it may be retrieved after the interrupt has been dealt with. Upon encounter of an interrupt, the current PSW thus becomes an "old" PSW, which is stored at a fixed location in core (see Figure 134). A doubleword location with a different address is reserved for each major class of interrupt. Simultaneously, to service the interrupt and branch to the appropriate fixup (or other) routine, a "new" PSW must be fetched from a different fixed location in storage. Again, a new PSW doubleword location with a different address is reserved in storage for each class of interrupt. Thus there are *two* fixed locations in main storage for each class of interrupt: one to receive the old PSW upon occurrence of an interrupt, and the other to furnish the new PSW that services that class of interrupt (see Figure 134).

In brief, an interrupt of a particular class (there are five major classes) simply *replaces the entire current PSW,* by placing it in the old PSW location in main storage for that class, and then fetching a new PSW for the required class from its location in main storage. The new PSW contains the information necessary (location of fixup routine, etc.) for handling the interruption. After the interrupt has been serviced, a single instruction recalls the old PSW and processing continues from the point of interruption.

Through previous studies in this book, you may recall that a simple branch instruction replaces only the *instruction address portion* of the current PSW; this should be contrasted with the interrupt action, which replaces the *entire* current PSW. In essence, however, an interrupt is simply an *automatic branch* to a new sequence of instructions.

Interrupt Action

The general pattern of interrupt action now begins to emerge; it is also graphically illustrated in Figure 135. When an interrupt occurs (perhaps because of an arithmetic overflow, completion of an I/O operation, or other condition), the *machine* first determines the general class of interrupts involved, and then proceeds to:

1. Store the current PSW (which controls the problem program) in the appropriate old PSW location. The old PSW preserves the general status of the processor, gives the reason for the interrupt, and also contains the address of the "next" instruction of the problem program (that is, the point at which we left it). All this is done automatically.

2. Fetch a new PSW from the appropriate location in storage (for the particular class of interrupt) and load it as the current PSW. This new PSW "points" to the first instruction of the interrupt-handling routine, which is part of the supervisory control program (the "Interrupt Supervisor"). Again, this is done by electronic circuitry, though the "pointer" to the interrupt routine had to be initialized by a programmer.

The interrupt routine now proceeds to analyze the cause of the interrupt, stored in the old PSW location. and then takes the appropriate action required for the type of interrupt involved. Depending on the cause of interrupt, this may consist of initiating an I/O operation, branching to a "fixup routine" to correct the problem (see Figure 135), calling for a dump program, etc.

Evidently, the interrupt-handling routines, with the associated corrective routines, dump programs, etc., are prepared by a programmer, but not usually the problem programmer at the installation. As we shall see later on in more detail, the interrupt-handling routines, or Interrupt Supervisors, for various application programs are part of the control program package that

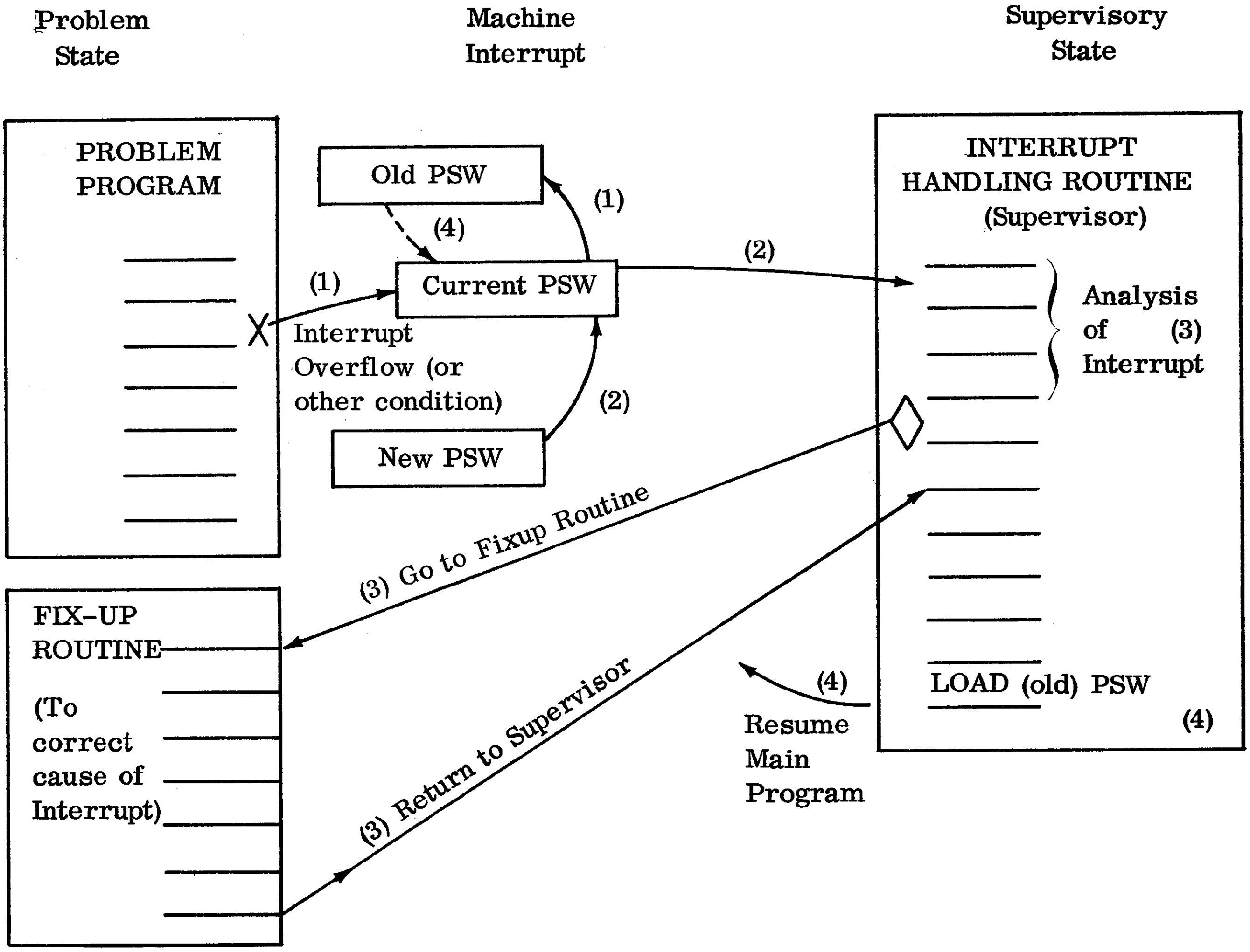


Figure 135. Action occurring during interrupt

each installation receives. The Interrupt Supervisor portion of this control program resides permanently in storage. However, when desired, special fixup routines for particular applications may, of course, be prepared by the installation's programmers.

After the interrupt has been taken care of, the instruction sequence of the original problem program may be resumed from the point of interruption, if desired. This is accomplished by the last instruction ("Load PSW") of the interrupt-handling routine. The Load PSW instruction recalls the old PSW from its location in storage and makes it again the current PSW; thus we are back in the problem program.

Why must the Load PSW instruction address the old PSW? Why can't the machine recall the cause of the last interrupt and automatically fetch the old PSW associated with that cause? It could. However, by addressing a specific old PSW, the supervisory program gains a bit of added flexibility over the automatic approach. As we shall see later on, the occurrence of several simultaneous interrupts may require the supervisory (IBM) programmer to stack up a series of old PSW's in order of priority. Upon servicing a particular interrupt, he may wish to return to a different status or task from the one last interrupted. He then simply addresses the old PSW of his choice. This situation may occur quite frequently in multiprogramming, where switching to another task instead of returning to the original program would be desired. In all these cases, the nonautomatic addressing of the desired old PSW by the Load PSW instruction provides the supervisory program with the requisite flexibility.

The general interrupt action sketched above leaves out many details and refinements with which we shall have to concern ourselves later on. We have not described, for example, how the machine is able to identify (1) what caused the interrupt, (2) how a fixup routine would go about finding the last-completed instruction, rather than the "next" instruction address contained in the old PSW, (3) what the machine would do in the event of several interrupts occurring at roughly the same time, or (4) how certain types of interrupt causes might be ignored altogether. Leaving these important questions for later, let us look at a typical example of a program interrupt occurring during the processing of a few simple arithmetical operations.

An Example

To illustrate the difference between a programmed branch and an automatic machine interrupt, we have chosen a simple example, involving fixed-point addition of a few numbers whose sum might overflow the register.

Figure 136 shows the partial assembly listing of the fixed-point add routine, in which the *programmer* has inserted instructions to check for overflow after each addition. If an overflow occurred, the program is to branch to an overflow fixup routine to correct the condition (if possible), before returning to the mainline program. Since we are interested only in the branching process, we shall not concern ourselves with the overflow fixup routine at this time.

The first three lines of the listing (Figure 136) – START, BALR (Branch and Link Register), and USING – are the standard preliminary instructions with which we are familiar. In brief, they establish that the assembly will start at location 256, and that register 15 will serve as base register and contain the location of the first byte after the USING instruction.

The next two instructions (SR 4,4 and SPM 4) are still preliminary, but have some relevance to later processing.

The Subtract instruction

SR 4,4

specifies that the contents of register 4 are to be subtracted from register 4, which is equivalent, of course, to clearing the register to zero. Whenever the first and second operand locations of the SR instruction specify the same register, the register is simply to be cleared or "initialized". We are clearing register 4 at this time in anticipation of a possible overflow from the add operations performed in register 5 later on. Register 4 will receive any high-order overflow bits.

The Set Program Mask instruction

SPM 4

simply places the zero bits in register 4 into the condition code (CC) and program mask bit positions (36-39) of the current PSW. As we shall learn in more detail later on, this "mask" prevents all types of program interrupts, including overflow. Since the *program* is going to do the branching in case of overflow (in this initial illustration), we do not want an automatic program interrupt. In effect the SPM instruction simulates a system *without* automatic interrupts.

Processing proper begins with the Load

L 5,A

instruction, which simply places the data at effective storage location A (to be defined by a DC) into register 5. For simplicity, we shall say that the value or number A is loaded into register 5. By means of the Add instruction

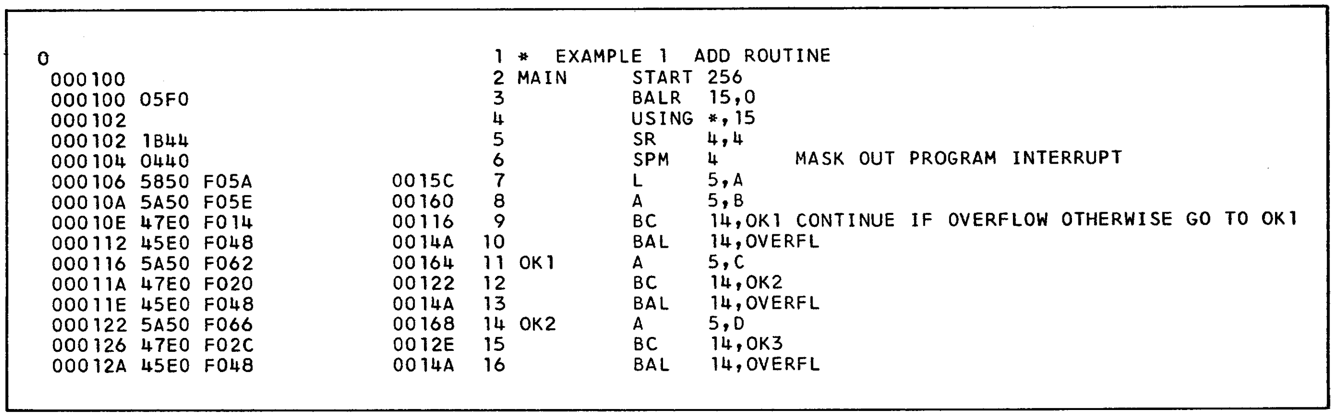
A 5,B

we then add the number B (defined by another DC) to A in register 5. This is a standard operation.

Since we don't know the magnitudes of A and B, an overflow may already have occurred, and, consequently, the next instruction

BC 14,OK1

is designed to check for overflow – or, rather, its absence.



```
0                                        1 *  EXAMPLE 1  ADD ROUTINE
 000100                                  2 MAIN     START 256
 000100 05F0                             3          BALR  15,0
 000102                                  4          USING *,15
 000102 1B44                             5          SR    4,4
 000104 0440                             6          SPM   4      MASK OUT PROGRAM INTERRUPT
 000106 5850 F05A             0015C      7          L     5,A
 00010A 5A50 F05E             00160      8          A     5,B
 00010E 47E0 F014             00116      9          BC    14,OK1 CONTINUE IF OVERFLOW OTHERWISE GO TO OK1
 000112 45E0 F048             0014A     10          BAL   14,OVERFL
 000116 5A50 F062             00164     11 OK1      A     5,C
 00011A 47E0 F020             00122     12          BC    14,OK2
 00011E 45E0 F048             0014A     13          BAL   14,OVERFL
 000122 5A50 F066             00168     14 OK2      A     5,D
 000126 47E0 F02C             0012E     15          BC    14,OK3
 00012A 45E0 F048             0014A     16          BAL   14,OVERFL
```

Figure 136. Partial assembly listing of add routine with branching in case of overflow

You will recall from earlier work in fixed-point operations and branching, that the Branch on Condition (BC) instruction tests the *condition code* in the program status word (PSW). The condition code (0, 1, 2, or 3) is set by any of a large number of instructions and indicates the result of executing that instruction. Thus, for the example of the Add instruction above (A 5,B), the condition code settings 0, 1, 2, and 3 indicate, respectively, whether the result of the add was zero, less than zero (i.e., negative), more than zero (positive), or caused an overflow. To test the condition code, the BC instruction has a four-bit mask, with each bit checking one of the condition code settings. The decimal mask value of 14 in the BC 14,OK1 instruction, above, sets the mask bits to 1110, which in turn tests condition code setting 0, 1, or 2. If the Add instruction resulted in setting up any one of these condition codes (that is, the result was equal to, less, or more than zero but did *not* overflow), one of the mask bits in the BC instruction will match it and the *Branch to OK1 will be taken.* As we shall see presently, the OK1 branch is simply a continuation of the adding routine.

If, however, an overflow occurred during the first add, condition code 3 (rather than 0, 1, or 2) will be set up, the branch to OK1 will not be taken, and the program will go on to the next line of coding, which is the Branch and Link instruction

BAL 14,OVERFL

This places the address of the next instruction in sequence into register 14, to provide a return link, and then branches to the OVERFL (overflow) routine, which is not shown in the assembly listing (Figure 136). The overflow fixup routine is designed to correct a part of the overflow condition (the sign bit), so that continued addition can take place. At the end of this routine it is necessary only to execute an unconditional branch (not shown) to the address in register 14 by means of the BCR instruction, in order to continue the main program.

OK1 continues the adding routine with the

A 5,C

instruction, which adds C to the contents of register 5. Again we test for overflow by means of the Branch on Condition instruction

BC 14,OK2

If there was *no* overflow (condition code equals a bit in decimal mask value 14), we branch to OK2, the continuation of the adding routine; otherwise, we go on to the next instruction

BAL 14,OVERFL

which is the Branch and Link to the overflow fixup routine, described earlier.

Routine OK2 is entered either directly, through the branch address in the BC 14,OK2 instruction, or at the end of the overflow fixup routine through an unconditional branch instruction. The routine adds D to the sum contained in register 5, followed again by the BC and BAL instructions, as described earlier. This group of three instructions presents nothing new.

The final part of the program is entered at OK3, either through the branch address in the previous BC 14,OK3 instruction or at the end of the overflow routine (in the event of an overflow). The OK3 routine, which is not shown in the assembly listing, would consist of checking whether or not an overflow occurred anywhere in the program, and if so, it would fix up the portion that could not be corrected in the overflow routine.

Consider now the same example programmed to take advantage of the automatic interrupt mechanism, as shown in the partial assembly listing (Figure 137). Note that only eight lines of coding were needed to get to the point in the program (completion of adding) that required 15 lines of coding in the branching routine (Figure 136).

The first four instructions are preliminary and are identical to those shown in Figure 136. The Set Program Mask (SPM 4) instruction is omitted, since we want an automatic program interrupt in case of overflow during this run. The actual processing is coded very compactly in the remaining four (Load and Add) instructions, which consist of loading A into register 5 and then adding B, C, and D in sequence. No Branch on Condition (BC) and Branch and Link (BAL) instructions were needed, since in the event of overflow after any add, the program will be interrupted automatically and control will be passed to the "Interrupt Supervisor".

So that you not be deceived, however, by the seeming simplicity of this program, consider that the problem programmer still has to take care of the overflow by writing the appropriate fixup routine. Upon occurrence of the interrupt, the Supervisor will immediately branch to this user-prepared routine, and, if this routine determines that recovery is not possible, it will request the Supervisor program to dump and abort the program.

```
0                                    1   * EXAMPLE 2  AN AUTOMATIC INTERRUPT ROUTINE
 000100                              2   PROGA     START 256
 000100 05F0                         3             BALR  15,0
 000102                              4             USING *,15
 000102 1B44                         5             SR    4,4
 000104 5850 F02E          00130     6             L     5,A
 000108 5A50 F032          00134     7             A     5,B        IF OVERFLOW OCCURS,CONTROL WILL BE PASSED
                                     8   *                          AUTOMATICALLY TO AN INTERRUPT ROUTINE
 00010C 5A50 F036          00138     9             A     5,C
 000110 5A50 F03A          0013C    10             A     5,D
```

Figure 137. Partial assembly listing of add routine shown in Figure 136, with automatic program interrupt in case of overflow

Classes of Interrupts

We have studied one type of interrupt, a fixed-point overflow, which is one of 15 possible interrupt conditions in the general class of program interrupts. Let us briefly review the five classes of interrupts: program, input/output, machine check, external, and supervisor call. Detailed information on each of these appears in IBM *System/360 Principles of Operation (A22-6821).*

Each of the five classes of interrupts has two distinct storage locations, for the old and new PSW's respectively, as listed below (in decimal notation):

INTERRUPT CLASS	OLD PSW LOCATION	NEW PSW LOCATION
External	24	88
Supervisor	32	96
Program	40	104
Machine	48	112
Input/output	56	120

Note that the storage locations are all divisible by eight, since each contains a doubleword (eight bytes). The new PSW location is always 64 bytes higher than the old. These storage locations are, of course, permanently assigned.

Program Interrupts

Program interrupts are caused by various kinds of programming errors and other unusual conditions, such as incorrect operands or operand specifications, and exceptional results. Eight of the 15 possible interrupt conditions involve arithmetic overflow, improper divides, lost significance, and exponent underflow. (The last two can occur only in floating-point arithmetic.) The remaining seven conditions are concerned with improper addressing and specifications, attempted execution of invalid instructions, violation of storage protection, and similar conditions. (The 15 program interrupt causes are explained in *A22-6821.*)

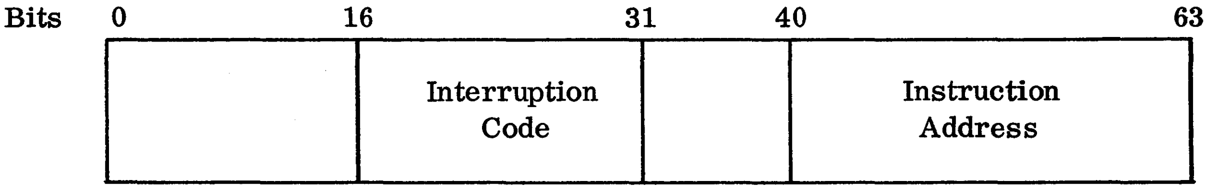


Figure 138. Interruption code and instruction address portions of PSW

How can the Interrupt Supervisor program determine what caused the interruption? There are really two ways. The *general* class of interrupts (one of five) is apparent from the fixed storage locations of the old and new PSW's. Thus, the fact that the machine fetched the new PSW from location 104 and stored the old PSW in location 40 tells the Supervisor program that the interrupt was caused by a *program* check. Identifying what, *specifically*, caused the program check is the function of the *interrupt code* in the PSW. The interrupt code takes up bit positions 16–31 in the 64-bit PSW (see Figure 138). These bits identify the *specific* cause of the interrupt. When an interrupt occurs, the current PSW is stored in one of the permanent locations reserved for old PSW's – in this case, location 40 for a program check. At the same time the interrupt code is automatically set and recorded in this location. Thus the Interrupt Supervisor need only examine the interrupt code specified by bits 16–31 of the old PSW to determine whether the cause of the program check was faulty addressing, improper specification, fixed-point overflow, or some other exceptional condition.

For reference, the codes for the 15 conditions that cause program interrupt are given in the following chart (only bits 24–31 are listed, since bits 16–23 are all zeros):

NO.	INTERRUPT CODE (BITS 24–31)	PROGRAM INTERRUPT 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

Input/Output Interrupts

When an input/output device ends an operation, a signal goes to the CPU that the channel is free and ready to accept a new operation. This signal causes an I/O interrupt. You will recall that in System/360 the I/O operations on one or more channels are overlapped with processing; that is, they take place independently from and at the same time as processing. The channels must, therefore, let the processor (or more specifically, the supervisor program) know whenever an I/O operation has been completed; the I/O interrupt circuitry does just that in the most efficient manner. In addition, error or other conditions (bad tape, end of tape, etc.) existing on channels or I/O devices alert the system by causing an I/O interrupt. The interruption code of the old PSW (stored in location 56) will tell the Interrupt Supervisor which channel and I/O unit caused the interrupt, so that appropriate action can be taken. Additional information about the status of the I/O device and channel is preserved in the *channel status word* (CSW), which is also stored in a fixed location (64) as part of the machine's interrupt procedure.

You will learn considerably more about I/O interrupts if you take up I/O programming. For the present, note that more than one request for an I/O interrupt may occur at the same time, but one only can be processed at a time. The remaining requests are stored in the I/O channel, control unit, or device until they can be accepted by the CPU, one at a time and in order of priority. For reference, the following chart lists the I/O interruption code (stored in bits 21–31 of the old PSW) and identifies the channel and device causing the interrupt.

I/O INTERRUPT CODE (PSW BITS 16–31)	CHANNEL IDENTIFICATION
00000000 aaaaaaaa	Multiplexor Channel
00000001 aaaaaaaa	Selector Channel 1
00000010 aaaaaaaa	Selector Channel 2
00000011 aaaaaaaa	Selector Channel 3
00000100 aaaaaaaa	Selector Channel 4
00000101 aaaaaaaa	Selector Channel 5
00000110 aaaaaaaa	Selector Channel 6

NOTE: a = I/O device address

Machine Check Interrupts

Machine check interrupts are caused by various types of machine errors and hardware malfunctions, as detected by the machine-checking circuits. Invalid instructions or data cannot result in a machine check, with one exception: after a power interruption or system reset, incorrect parity may exist in storage or registers, which will cause a machine check. Once the registers have been cleared and new information has been loaded, this can no longer happen.

Upon occurrence of a machine check, the current instruction is abruptly terminated (that is, the results may not be stored) and a hardware procedure to record the status of the system is initiated. Part of this procedure consists of a "scan-out" of the CPU hardware status into the storage area, starting with location 128 and extending through as many words as are required by the CPU and I/O channels of the particular system used. The old PSW is then stored in location 48 with an interrupt code (bits 16-31) of all zeros, and a new PSW is brought out from location 112. The storage locations and interrupt code of zero will inform the Interrupt Supervisor that a machine check interrupt has occurred. The supervisor then takes certain actions depending on the programming system used (Basic Programming Support, Basic Operating System, or Operating System/360). Some supervisors place the system in a wait state, allowing the operator to load and execute a stand-alone program (obtainable from the Customer Engineer) that will preserve in the form of printed output (for later analysis by the Customer Engineer), the contents of the scan-out area and such other information as the old machine check PSW. Other supervisors allow the automatic recording of the scan-out area on disk or drum without operator intervention.

We shall see later that a machine check interrupt can be prevented (or *masked*), if desired. When this is done, no interrupt or recording procedure occurs, and the machine attempts to continue the instruction sequence in the presence of a machine check.

External Interrupts

Three types of conditions can cause an external interrupt of the CPU:

1. The operator presses the Interrupt key on the console.

2. The value in the built-in interval timer turns from positive to negative.

3. An external signal (part of the Direct Control feature) occurs.

Requests for an external interrupt may occur at any time and, possibly, from several different sources simultaneously. When an external interrupt occurs, the old PSW is stored at location 24 and a new PSW is fetched from location 88. The source of the interrupt is recorded by the interrupt code bits 24–31 of the old PSW; bits 16–23 are made zero. As is evident from the following listing, each external interruption source sets a corresponding bit of the interrupt code to 1.

INTERRUPT CODE BIT	INTERRUPT CODE (PSW BITS 24–31)	EXTERNAL INTERRUPT CAUSE
24	1nnnnnnn	Timer
25	n1nnnnnn	Interrupt key
26	nn1nnnnn	External signal 2
27	nnn1nnnn	External signal 3
28	nnnn1nnn	External signal 4
29	nnnnn1nn	External signal 5
30	nnnnnn1n	External signal 6
31	nnnnnnn1	External signal 7

n = Other external-interruption conditions

Supervisor Call Interrupts

A Supervisor Call is a special *instruction used by the program* to force an interrupt. A supervisor call interrupt, therefore, differs from the other classes of interrupts in that the *program makes use of the automatic interrupt linkage,* rather than the machine itself recognizing some exceptional condition. The most frequent use of the supervisor call interrupt is to switch from the *problem state* to the *supervisory state;* that is, to turn over control to the supervisor (operating system). Certain instructions – for example, to carry out an I/O operation – are executable only in the supervisory state of the machine. We shall consider this and other forms of status switching through use of the Supervisor Call instruction in greater detail, in the section entitled "The Interrupt Supervisor".

When a Supervisor Call instruction causes a supervisor call interrupt, the old PSW is automatically stored at location 32 and a new PSW is fetched from location 96. Simultaneously, eight bits from the Supervisor Call instruction (bits 8–15 of the R_1 and R_2 fields) are placed into bit positions 24–31 of the interrupt code in the old PSW, to permit identification of the interrupt. These eight bits are used, in effect, to convey messages from the calling, or problem program to the supervisory program. The message may consist, for example, of a request to begin an I/O operation for the problem program, or of notification that the problem program is finished and a request to the supervisor to read in a new program. After the request has been honored, the supervisor program returns control to the problem program via the Load PSW instruction. In the examples above, the Supervisor Call is the *last* instruction of the problem program. Depending upon the service requested, however, a supervisor call interrupt may occur at any point of the problem program. More about the SVC instruction later on.

How Interrupts Can Be Prevented (Masking)

Automatic interrupts are convenient, but sometimes it is desirable, or even necessary, to prevent interrupts or, at least, postpone recognizing them. Preventing an interrupt, or keeping it pending till later, is called *masking*. External, I/O and machine check interrupts can be kept pending until a later time. Four of the 15 programming exceptions can be ignored (masked) completely (prevented from causing interrupts). However, the remaining eleven program interrupt conditions and the supervisor call interrupt cannot be masked.

One situation where masking is absolutely essential is characterized by multiple interrupts of the same type. Without masking, this condition would cause the CPU to be caught in a program loop, from which it could not return to the problem program. Consider, for example, a condition of simultaneous I/O operations occurring on two or more channels. Since, with more than one channel present, simultaneous I/O operations are going on all the time, this situation could easily occur. As shown in Figure 139, the first I/O interrupt – occurring sometime during the execution of the problem program – will cause the current PSW to be stored in the old PSW location 56. The old PSW provides for the eventual return to the problem program and also tells the Supervisor program (via the interrupt code) which channel and I/O device caused the interrupt. A new PSW will then be fetched from location 120 and the instruction address in this PSW will immediately direct the Supervisor program to the appropriate I/O interrupt routine. Note (in Figure 139) that the last instruction of this routine provides for loading the old PSW from location 56, in order to resume the problem program.

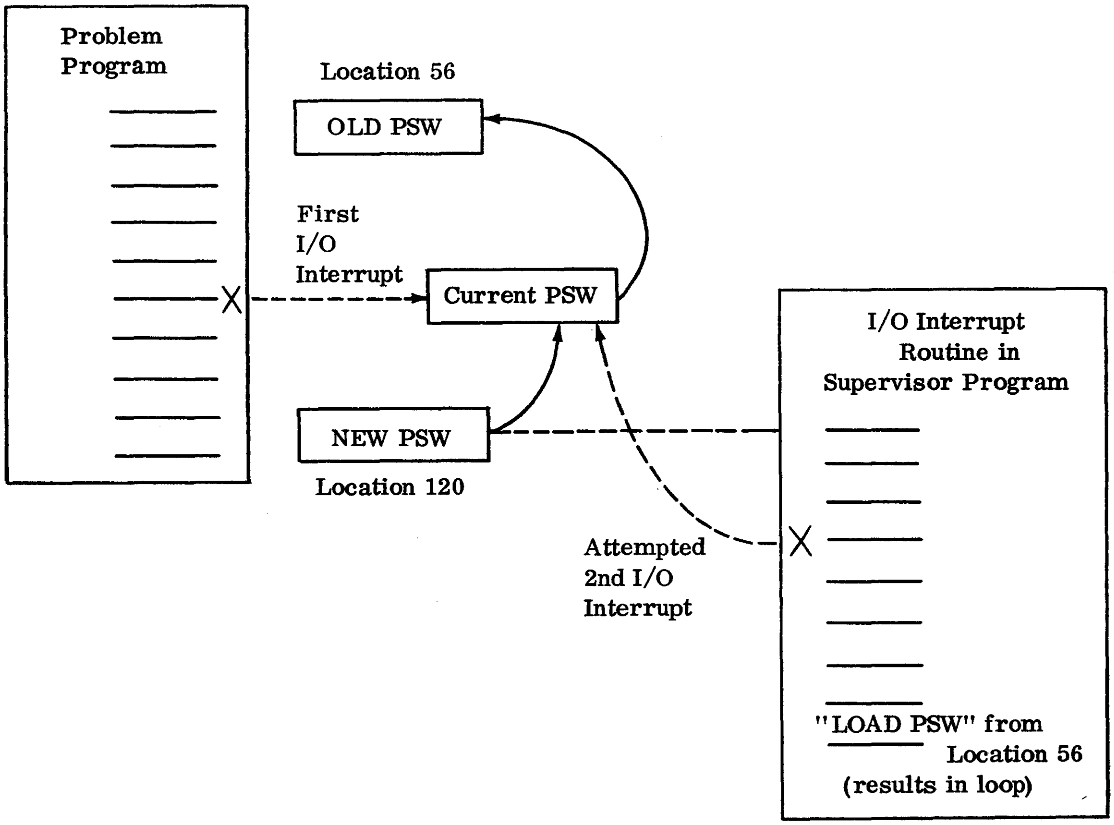


Figure 139. Need for masking. If an I/O interrupt were allowed to occur during execution of a previous I/O interrupt-handling routine, the "old" PSW from the problem program would be destroyed and the CPU would be caught in an I/O interrupt loop, making return to the problem program impossible.

Assume now that *during the execution* of the I/O interrupt-handling routine (that is, before the first I/O interrupt has been completely serviced), a second I/O interrupt occurred. This new interrupt would cause the current PSW from the (first) I/O interrupt routine to be stored in location 56, thus *destroying the old PSW* from the original problem program. A new PSW would then be fetched to handle the second I/O interrupt, at the completion of which the old PSW would bring us back to the first I/O interrupt routine, and so on. There would be no way to return to the problem program.

Clearly, in this or any similar case, the same type of interrupt must be prevented from occurring during the execution of an interrupt routine. As we shall see presently, this is done by masking one's own class of interrupts in the new PSW. However, the problem programmer need never worry about this case, since the masking is performed by the control (Supervisor) program.

Regardless of who does it (sometimes the Supervisor, sometimes the programmer), how is masking accomplished? Like almost everything else, masking — the prevention of interrupts — is taken care of in the PSW. Specifically for the purpose of masking, the PSW provides certain fields, known as *system mask, machine check mask,* and *program mask* (see Figure 140). The system mask is concerned with the prevention of I/O and external interrupts, while the machine check and program masks apply to the corresponding classes of interrupts. As is evident in Figure 140:

Bits 0–7 of the PSW form the system mask.
Bit 13 of the PSW is the machine check mask.
Bits 36–39 of the PSW constitute the program mask.

Whenever the mask bits for particular types of interrupts are made *zero,* these types are prevented from occurring, or *masked.* When the mask bits are set to *one,* on the other hand, the CPU is *interruptible* for the corresponding types of interrupts. Let us look a little more closely at each of the masks.

System Mask

As mentioned, the system mask (consisting of PSW bits 0–7) can be used to mask all types of external and I/O interrupts. Masking here means that the affected types of interrupts *can be held pending* for later execution, but *not* completely ignored. The following chart lists the system mask bits that must be made *zero* to individually or collectively mask the corresponding I/O and external interrupts:

SYSTEM MASK BIT	INTERRUPTION SOURCE	CLASS
0	Multiplexor Channell	I/O
1	Selector Channel 1	I/O
2	Selector Channel 2	I/O
3	Selector Channel 3	I/O
4	Selector Channel 4	I/O
5	Selector Channel 5	I/O
6	Selector Channel 6	I/O
7	External Signal, Timer, Interrupt Key	External

During an I/O or external interrupt the Supervisor program will mask the identical interruption source, by setting the corresponding system mask bit in the new PSW to zero, at the beginning of the interrupt. In addition, the programmer has the choice of keeping interrupts from certain channels or external sources pending, by setting the appropriate mask bits to zero. For example, a system mask of 10000001 would hold I/O interrupts from all selector channels pending, while permitting interrupts only from the multiplexor channel and external sources. Similarly, a system mask of 01110000 would permit I/O interrupts only from selector channels 1–3, and mask interrupts from selector channels 4–6, the multiplexor channel, and external sources.

System masks may be set in two different ways. The first consists of introducing an entirely new PSW with the desired mask bits, either through the Load PSW instruction or a supervisor call interrupt. The second way is through use of the Set System Mask (SSM) privileged instruction, which replaces only the system mask bits of the current PSW for the duration of the routine, until execution of the next SSM or Load PSW instruction.

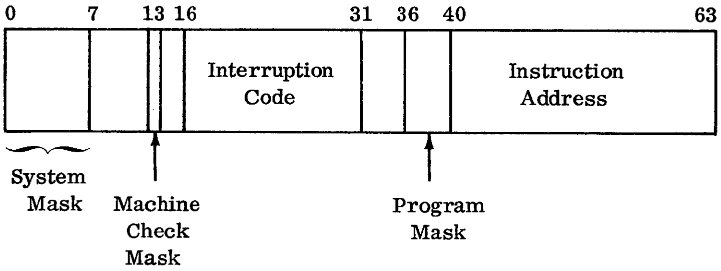


Figure 140. Location of system, machine check, and program masks in PSW

Machine Check Mask

A machine check can cause an interrupt only when PSW bit 13, the machine check mask, is set to 1. As described earlier, the machine check will start a recording routine to locate the fault, begin a scanout of the status of the CPU from location 128, store the old PSW, and finally cause a machine interrupt. (The scanout, sometimes called logout, automatically places information concerning the state of the internal circuitry into storage.)

The conditions would be rare, indeed, where one would want to keep pending a machine check and the resulting recording procedure, by masking the interrupt. The IBM supervisor programmer can do so, however, by making PSW bit 13 a zero. Then if a machine check occurs, the associated recording procedure and interrupt do not take place and the machine will attempt to continue the instruction sequence. Of course, you always have the additional option of turning on the Error Stop switch on the CE section of the System Control Panel. A machine check will then cause an error stop of the system, regardless of the setting of the machine check mask bit.

Program Mask

Bits 36–39 of the PSW constitute the program mask (Figure 140). These bits are reserved for certain program exceptions which, on occasion, are not to be treated as program checks. To prevent interruptions caused by these exceptions, the corresponding program mask bits must be made zero, as follows:

PROGRAM MASK BIT (IN PSW)	PROGRAM EXCEPTION
36	Fixed-point overflow
37	Decimal overflow
38	Exponent underflow
39	Significance

The first two of these exceptions pertain to fixed-point and decimal overflow conditions. A programmer may want to mask either or both of these exceptions – for example, when he is using a general register as a counter in a program. He may want to test the counter for overflow without incurring a program check. Thus, he would set PSW bit 36 and/or 37 to zero, to mask the corresponding overflow exception. In our earlier example of a fixed-point overflow during an add operation, we used the Set Program Mask instruction to make PSW bits 36 and 37 zero, thus preventing an automatic program interrupt due to overflow; we then branched to a "fixup" routine to correct the overflow condition resulting from continued addition. Exponent underflow and "significance" (PSW bits 38 and 39, respectively) are concerned with floating-point operations.

Interrupt Mechanics

Now that we have a better appreciation of the PSW, masking, and other features, we are in a position to recap the interrupt action in more detail and become acquainted with some refinements (see Figure 141, which gives a more complete picture of the mechanics of interrupts).

We have discussed the five classes of interrupts (I/O, program, external, machine check, and supervisor call) and the numerous possible sources of interrupts within each class. When one of these many possible exceptions takes place, an interrupt *may* occur, provided certain conditions are fulfilled – and then only after certain preliminaries are out of the way. The necessary condition for an interrupt to occur is that the CPU be interruptible for the particular interruption source. As we have seen in the last section, this means that the system mask, program mask, or machine check mask (depending upon the class of interrupt) in the PSW must be checked to verify whether the appropriate mask bits are zero or one. If the mask bits for the source of interruption are one, the program will be interrupted; if they are zero, the CPU is not interruptible for this source. (Some interruptions cannot be masked, of course.) Before an interrupt can take place, however, even with masking absent, the CPU has to decide what to do with the instruction presently being processed – that is *when* to interrupt.

Timing of Interrupt

In general, interrupts can take place only *after* the current instruction has been finished and before the next instruction is started; you cannot have an interrupt in the middle of an instruction. However, the manner in which the instruction preceding an interrupt is finished depends to some extent on the cause and type of interrupt. The preceding instruction may be *completed, terminated, or suppressed.*

In the case of input/output, external, and supervisor call interrupts, the current instruction is *completed* in a normal manner before the interrupt is taken. This means that the result of the instruction is stored (though it may be wrong, depending upon the exception) and the *condition code* is set as for any normal instruction. (We shall take a look at the condition code in the PSW presently.)

In the case of program or machine interrupts – indicating programming and hardware errors, respectively – the interrupt still occurs at the end of the current instruction. However, upon detection of a machine malfunction (machine check), the execution of the instruction is *terminated* during *E* (execution) time, rather than being completed. When an instruction is terminated, all, part, or none of the result may be stored; hence, it should not be used for further computation. The setting of the condition code also is unpredictable after an instruction termination.

In the case of a program check interrupt, depending upon the type of programming error and just when in the machine cycle the error is detected, the current instruction may be completed, terminated, or *suppressed.* The instruction is completed after detection of most computation errors, such as fixed-point and decimal overflows, or floating-point operations. A fixed-point divide error detected during instruction fetch time, however, causes the execution of the current instruction to be *suppressed;* that is, the called-for operation is not carried out at all, the results are not stored, and the condition code is not changed. If the program check is caused by a violation of storage protection or improper addressing, the current instruction is either suppressed or terminated, depending upon whether the error was detected during instruction fetch time or execution (E) time, respectively. ("Summary Chart of Interrupt Action" at the end of this chapter shows the mode of execution of instructions preceding all types of interrupts.)

The Instruction Length Code

After the current instruction has been finished, the interrupt actually takes place (provided, of course, the CPU is interruptible for the interruption source). At this point, the machine is ready to record the specific cause of the interrupt by setting the interrupt code in the current PSW and then filing it away in the appropriate old PSW location for the class of interrupts involved. After the interrupt has been serviced, the machine will want to come back to this old PSW to resume the instruction sequence of the problem program. This should not be difficult. With the instruction preceding the interrupt having been executed, the Instruction Address portion of the old PSW (bits 40–63) will automatically have been updated to point to the address of the *next* instruction in the sequence. In the case of external and input/output interrupts, where the

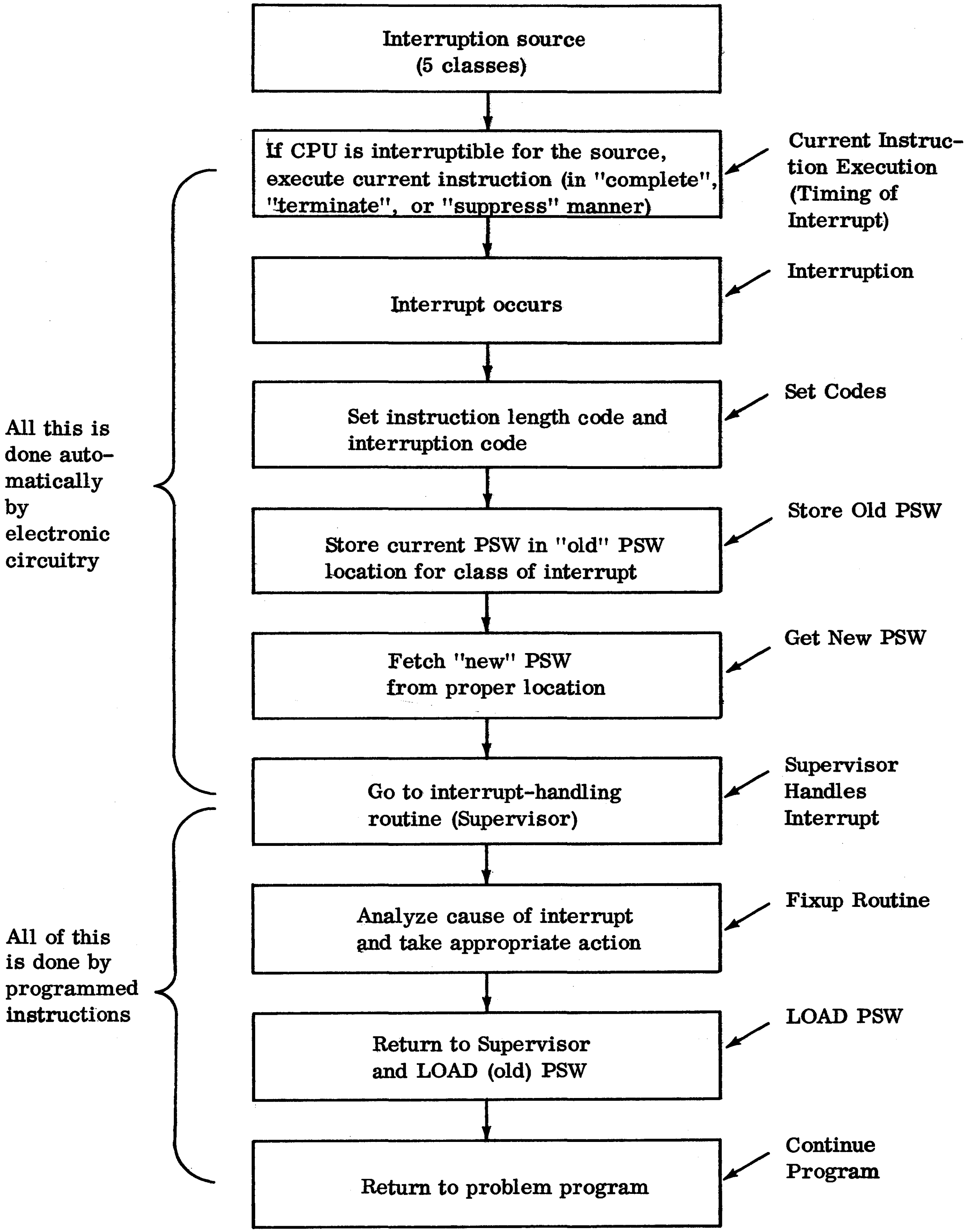

Interruption source (5 classes)
If CPU is interruptible for the source, execute current instruction (in "complete", "terminate", or "suppress" manner)
Current Instruction Execution (Timing of Interrupt)
Interrupt occurs
Interruption
Set instruction length code and interruption code
Set Codes
Store current PSW in "old" PSW location for class of interrupt
Store Old PSW
Fetch "new" PSW from proper location
Get New PSW
Go to interrupt-handling routine (Supervisor)
Supervisor Handles Interrupt
Analyze cause of interrupt and take appropriate action
Fixup Routine
Return to Supervisor and LOAD (old) PSW
LOAD PSW
Return to problem program
Continue Program
All this is done automatically by electronic circuitry
All of this is done by programmed instructions

Figure 141. Mechanics of interrupt

problem program had nothing to do with the cause of interrupt, this is exactly what we want. After the interrupt has been serviced, we shall want the machine to continue with the next instruction of the problem program, as if nothing had happened.

Consider now, however, the case of program or supervisor call interrupts, where an instruction in the *problem* program actually caused the interrupt. We may want to get back to this *last* instruction preceding the interrupt, either to analyze it for the cause of trouble or to execute it correctly after the cause of the interrupt has been taken care of. The instruction address of the PSW tells us the location of the *next* instruction that would have been executed if the interrupt had not taken place. To find the location of the *last* instruction executed, we must know its *length* (in bytes) and then subtract this value from the "next" instruction address in the PSW. This is the function of the *instruction length code* (ILC) in the PSW (see Figure 142). The ILC, consisting of PSW bits 32 and 33, is set at the time of an interrupt to 1, 2, or 3 depending upon the length of the last instruction, as follows:

ILC SETTING	PSW BITS 32–33	INSTRUCTION LENGTH	INSTRUCTION FORMAT
1	01	1 halfword (2 bytes)	RR
2	10	2 halfwords (4 bytes)	RX, RS, or SI
3	11	3 halfwords (6 bytes)	SS

Thus, if the instruction address in the old PSW is 3000 and the ILC setting is 3 (binary 11), the op code of the *last* instruction executed before the interrupt is located at address 2994 (that is, six bytes less than the "next" instruction address).

A Supervisor Call instruction is one halfword in length and, therefore, the ILC for a supervisor call interrupt is always 1. For program interrupts, the ILC setting may be 1, 2, or 3, depending upon the format of the instruction preceding the interrupt. If the length of the instruction preceding a program interrupt is not available, an instruction length code of 0 is entered in the PSW (bits 32–33 are 00). For machine check interruptions, the malfunction may affect the setting of the ILC, so that the code is not predictable. Finally, for external and I/O interrupts, which were *not* caused by the last-interpreted instruction, the ILC is not needed and is not predictable. ("Summary Chart of Interrupt Action" at the end of this chapter summarizes all ILC settings.)

The Condition Code

Figure 142 also shows the location of the *condition code* (CC), consisting of bits 34 and 35, in the PSW. In brief, the setting of the condition code reflects the status of the CPU at the time of the interrupt. After an arithmetic operation, as we have seen in our example, the setting of the condition code (0, 1, 2, or 3) indicates, respectively, whether the result is zero, negative, positive, or has caused an overflow condition. The condition codes for all operations and instructions are summarized in the applicable SRL publication (see *A22-6821,* the section on branching). The setting of the condition code in the current PSW is updated after each instruction and (as illustrated in Figure 136) it can be examined by means of the Branch on Condition (BC) instruction. The important information contained in the condition code is automatically stored with the old PSW at the time of interrupt.

The Interrupt-Handling Routine (Supervisor)

After the current PSW has been stored in the appropriate doubleword location at the time of interrupt, a new PSW is fetched from one of the doubleword locations reserved for the five major classes of interrupts. The instruction address portion of this new PSW "points" to the first instruction of the interrupt-handling routine, or Interrupt Supervisor, as we have seen earlier. Note that everything so far has been complete-

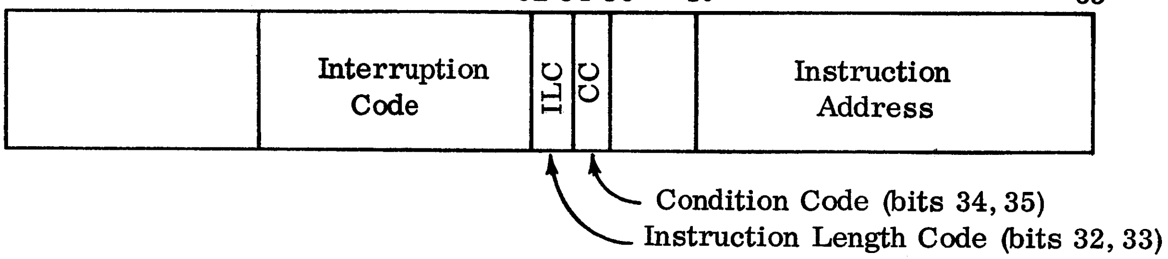

16
32 34 36
40
63
Interruption Code
ILC
CC
Instruction Address
Condition Code (bits 34, 35)
Instruction Length Code (bits 32, 33)

Figure 142. Location of instruction length code and condition code in PSW

ly automatic and that nothing has required the use of programmed instructions; it has all been done by electronic circuitry.

As indicated in Figure 141, the Interrupt Handler or Supervisor is a *program of instructions* that tells the CPU what to do for each possible type of interrupt. The program is not usually prepared by the installation's problem programmers, but is part of the control program package. We shall learn in the next section how the system is able to differentiate between the problem program and the supervisory interrupt-handling routines. For the moment, however, it is sufficient to note that the interrupt-handling routine (Supervisor) analyzes the cause of the interrupt, as indicated in the interrupt code of the old PSW, and then takes the appropriate action. Depending on the cause, this may require correcting the error via a user-prepared fix-up routine and later resumption of the mainline program (see Figure 135) or it may require calling for a dump program resulting in termination of the problem program. Different situations will require various corrective actions and routines, and it is sometimes impossible to recover and resume the mainline program.

Return to Problem Program

The last instruction of the interrupt-handling routine returns control to the problem program in those cases where it is possible and desirable to recover the error and resume the mainline instruction sequence from the point of interrupt. This is the Load PSW instruction, which has an SI format, and is generally used by the Supervisor when it wants to change the current PSW and go on to something else. Its main use is, of course, the resumption of the problem program after an I/O, supervisor call, or external interrupt has been serviced. In this case, the Supervisor programmer will simply write the effective address of the old PSW that was in charge before the interrupt into the B1 and D1 fields of the Load PSW instruction. The instruction will then fetch the old PSW from the designated doubleword location and load it again as the current PSW. The CPU now resumes the problem instruction sequence from the point of interruption. (Note, by the way, that the current PSW in charge during the interrupt routine is not automatically stored anywhere and therefore is lost.)

The Interrupt Supervisor

The basic design philosophy of the operating systems available for System/360, that of nonstop operation, assumes that a Supervisor program be in control of the system at all times, and that manual intervention be kept to a minimum. This design philosophy requires not only the automatic interrupt mechanism, but also a comprehensive control program that will free the installation's programmer from handling such common control functions as program loading, input/output operations, error detection and recovery, and communication between the program and operator. Thus, the presence of an independently prepared *control program* for handling these common functions found in all programs, permits the installation programmer to concentrate on the problem-solving aspects of his application or *problem programs.*

Only a portion of the control program known as the *Supervisor* is kept permanently in main storage. The Supervisor calls in other sections of the control program when necessary. Since the Supervisor and the problem program constitute two distinct programs in core storage, processing in the CPU alternates between the two, and the CPU is said to be operating in either the *problem* or *supervisor* state, depending upon which program is currently being executed. As we have seen, the Supervisor receives control of the CPU either through an *automatic,* machine-caused interrupt or through a *programmed* (supervisor call) interrupt. We shall learn a little later just how the System/360 CPU is able to distinguish between the separate Supervisor and problem programs in storage.

The Supervisor program performs the following major functions:

1. Interrupt handling
2. Input/output control and channel scheduling
3. Device error recovery
4. Program loading and retrieval
5. Operator communication
6. End-of-job handling
7. Checkpoint and restart
8. Label processing

In this section we shall be concerned only with the portion of the Supervisor that handles interrupts – the *Interrupt Supervisor.* Let us see just what it does upon occurrence of each major class of interrupts. This is a generalized discussion of concepts applicable to all operating systems for System/360. For a discussion of the functions of the Supervisor in a specific operating system, see the appropriate SRL publication.

How the Interrupt Supervisor Handles Interrupts

The Interrupt Supervisor must perform certain tasks common to all types of interrupts. It gains control of the CPU, by an interrupt. This is accomplished by the automatic exchange of PSW's, as we have seen. The status of the interrupted program is saved by placing the current PSW in the appropriate old PSW location, in accordance with the class of interrupt. A new PSW for this class of interrupt is then fetched from its reserved storage location. The instruction address portion of the new PSW "points" to the beginning of the Interrupt Supervisor routine, thus transferring control to it.

Having gained control of the CPU, the Interrupt Supervisor must perform three major tasks (see Figure 143).

1. It must *analyze* the type of reason and the specific reason for the interrupt. As was described earlier, the type or class of interrupt is revealed by the permanent storage locations of the old and new PSW's. The specific cause of the interrupt is contained in the interruption code (bits 16–31) of the old PSW.

2. It must take the appropriate *action* – that is, it must transfer control to the appropriate routine for handling the particular type of interrupt. As will be explained presently, some interrupt-handling routines are prepared by the user, others by IBM – again depending on the type.

3. Upon completion of the handling routine, it must, whenever possible, *restore* the mainline program via the old PSW. This is accomplished by the Supervisor program issuing a Load PSW instruction, which normally returns control to the interrupted program at the point of interruption recorded in the old PSW.

Input/Output Interrupts

Upon detection of an input/output interrupt, caused either by completion of an I/O operation or by the need for an I/O device to receive attention, control is immediately transferred to an element that might be termed the Input/Output Supervisor portion of the Supervisor program (see Figure 143). In general, the Input/Output Supervisor consists of a series of routines that handle all I/O operations, including I/O interrupts. At the normal end of an I/O operation, the Input/Output Supervisor starts any pending new I/O operations and then returns control to the problem

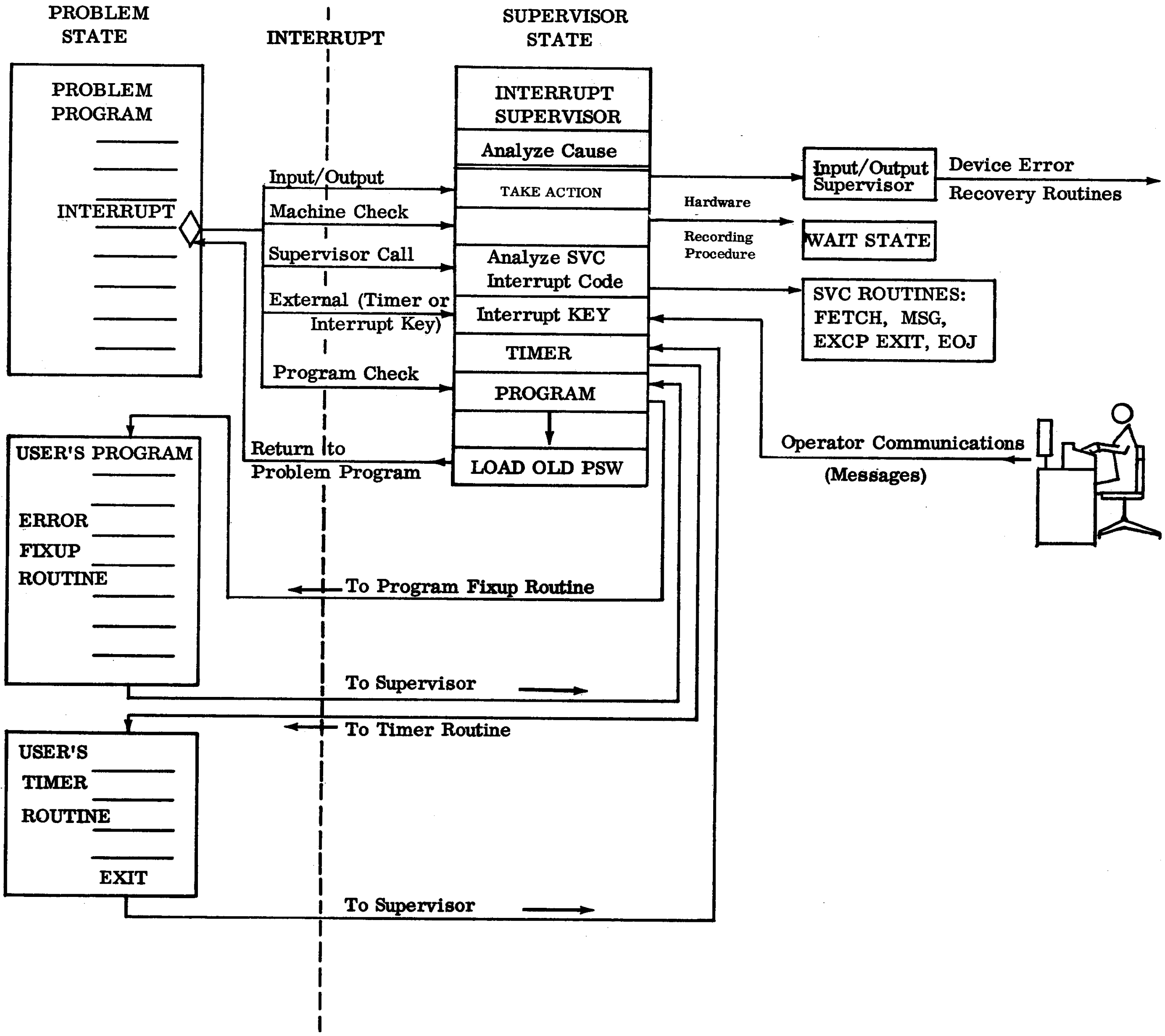


Figure 143. Flow of control between Supervisor and problem program during interrupt

program. The Input/Output Supervisor also detects and handles such specific conditions as a parity error, end of file (EOF), or a wrong-length record (WLR). Upon detection of an input parity error, for example, all interrupts are masked, and control is passed to the appropriate error recovery routine for the particular I/O device involved. If recovery is not possible, the user may (1) have the error ignored, (2) bypass the erroneous record, (3) transfer control to a user-prepared routine, or (4) terminate the job.

Machine Check Interrupts

Detection of a machine check interrupt caused by some malfunction causes the type of action previously described in the discussion of machine checks in the section on "Classes of Interrupts".

Supervisor Call Interrupts

This interrupt is caused by a request from the problem program via the Supervisor Call (SVC) instruction. The SVC interrupt routine in the Supervisor examines the interrupt code supplied with the SVC instruction and transfers control to the proper routine to handle the request. Some of the Supervisor routines that may be requested via the SVC instruction are listed below:

TYPE OF ROUTINE	FUNCTION
FETCH	To load a program from the core image library into storage for execution
MSG	To provide communication between the operator and the problem program
EXCP	Execute Channel Program – to request an I/O operation by the Input/Output Supervisor
EXIT	To return to the main problem program after a user-prepared timer routine or operator inquiry
EOJ	To terminate the program and prepare for the next job to be run

External Interrupts

External interrupts may be caused by external signals (available with the Direct Control feature), by the timer going from a positive to a negative value, or by the operator pressing the interrupt key at the console. External device *signal* interrupts are ignored by the general purpose version of the Supervisor; control is passed directly back to the problem program at the point of interrupt. Interrupts caused by pressing the interrupt key on the console are handled by Operator Communication Routines. These routines handle messages directed to the Supervisor program by the operator and issued to the operator by the Supervisor program. In some operating systems, the operator indicates his desire to send a message to the Supervisor by the depression of a "request" key on the system console device (a 1052, for example). In this situation, operator communication routines to handle the message would be entered because of an input/output interrupt. Finally, timer interrupts are turned over to a user-supplied timer routine for handling. The last instruction of this routine returns control to the Supervisor via the EXIT Supervisor Call. If there is no user's timer routine, the interrupt will be ignored.

Program Interrupts

In the event of a program interrupt, the Supervisor will transfer control to the address of a user-supplied program error fixup routine, provided one exists. Thus, in our earlier example of a fixed-point overflow during adding, the installation's programmer would have to supply an overflow fixup routine to which the Supervisor could branch. The user's subroutine itself would determine where the interrupt occurred by examining the address in the program check old PSW. Upon completion of this routine, control can be returned to the Supervisor and then to the interrupted problem program. If no fixup routine is supplied, however, the program is terminated, either immediately or after printing the contents of the problem program area in core storage (that is, program dump).

How the System/360 CPU Recognizes Supervisor and Problem Programs

We have indicated that two independently prepared programs are always in main storage: the Supervisor and the problem program. It is evident that these two programs should be kept separate at all times and that, moreover, the Supervisor must be protected from accidental interference or destruction by the problem program. Thus, two questions arise: How does the machine know which of the two programs is in control at any time? and How is the Supervisor program portected from interference? To answer these interrelated questions, we shall need to explore briefly the following aspects of the System/360 architecture:

Processor program states
Instruction classification
Storage protection

These topics are more fully covered in *A22-6821* and will be reviewed only briefly in the following paragraphs.

Processor Program States

The overall status of the processor is determined by four types of program states, with which you have already become somewhat familiar. They are the *stopped or operating, wait or running, masked or interruptible,* and *problem or Supervisor program* states. Each of these program state alternatives is identified by the setting of a bit in the current PSW, but in this context we are interested only in bit 15, which is the problem state bit (see Figure 144). When bit 15 of the current PSW is zero, the processor is in the Supervisor state; when bit 15 is one, the processor is in the problem state. Thus, the machine assumes that the problem program is being executed when current PSW bit 15 is one, and that the Supervisor program is being executed when bit 15 is zero.

Switching between the problem and Supervisor states is accomplished by changing bit 15 of the current PSW, and this can be done only by *introducing a new PSW*. As we know already, a new PSW can be introduced as the current PSW either through an interrupt or through the Load PSW instruction. Thus, an interrupt – whether executed automatically by circuitry or through the Supervisor Call instruction – will switch from the problem to the Supervisor state by in-

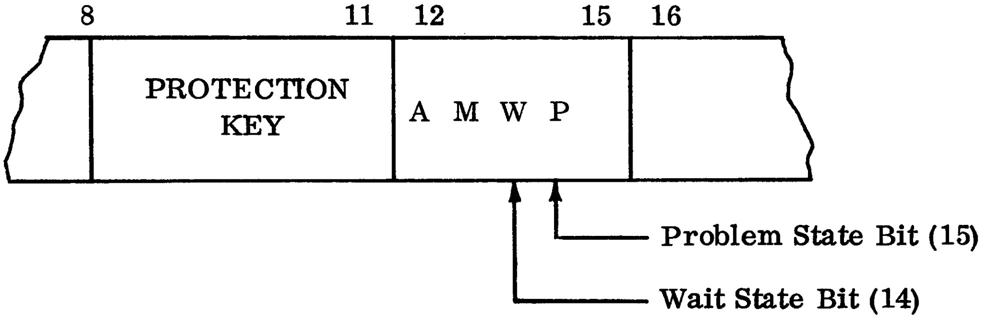


Figure 144. Protection key, problem state, and wait state indications in PSW

troducing a new PSW with a *zero* in bit position 15. At the same time, an old PSW will be stored with a *one* in bit position 15, to indicate where we left the problem program.

Instruction Classification

A second way the CPU distinguishes between problem and Supervisor program states is by the type of instruction: either *processing* or *privileged.* Processing instructions are concerned only with the problem-solving aspects of the application program and have no control over the supervisory portions, such as I/O operations and program loading. Privileged instructions, on the other hand, are concerned with the devices and operations traditionally controlled by an Input/Output Control System (IOCS) or other supervisor. Thus, all instructions relating to I/O operations, storage protection, direct-control devices, and other parts of the Supervisor program are considered *privileged.* This means that they cannot be called for and are not valid, when the processor is in the problem state. A privileged instruction encountered in the problem state results in a program exception that causes a program interruption. In the *Supervisor state,* however, *all instructions* – whether processing or privileged – *are valid.*

Let us look at some privileged instructions. We would expect the Supervisor rather than the problem programmer to have control over the entire current PSW and CPU status. Consequently, the Load PSW instruction, which replaces the entire current PSW, is obviously *privileged.* The Load PSW instruction may be used by the Supervisor to switch back to the problem program, but not vice versa. In contrast, the Supervisor Call instruction is *not* privileged and, hence, can be used by the problem program to switch to the Supervisor via the automatic interrupt linkage. Similarly, the Set System Mask instruction is privileged, since it affects I/O interrupts, which are the concern of the Supervisor. The Set Program Mask (SPM) instruction is not privileged, since we would want the problem programmer to be able to mask his program exceptions. Other privileged instructions are listed in the applicable SRL publication.

Storage Protection

The storage protection feature is concerned with the protection of a number of programs and data in main storage, not only the Supervisor program. In this context, we will only touch on the principle underlying this feature.

You are familiar with the *file protect rings* used in earlier computers to protect the data on a reel of tape from accidental destruction. In System/360 this feature has been extended to all-electronic protection of large areas of storage. In brief, each block of 2048 bytes in main storage is associated with a four-bit *storage key.* (Four bits provide for protection of up to 15 programs.) The storage key is not part of addressable storage, but is controlled by special instructions. The storage key is set or changed by the *privileged* instruction Set Storage Key (SSK) and is inspected by the privileged instruction Insert Storage Key (ISK).

When data is to be stored in a protected storage block, the storage key is compared with a four-bit *protection key,* occupying bits 8–11 of the current PSW (see Figure 144). Storage takes place only if the protection key and storage key match, or when the protection key is zero. Any other condition results in a program interrupt.

Simultaneous Interrupts

What happens when several interrupt requests occur at roughly the same time – that is during the same instruction cycle? We know that interrupt routines are normally handled in the order in which they occur, but only one request at a time can be serviced. When multiple interrupt requests occur at the same time as a result of a variety of causes, which will be taken first, and which last?

To answer these questions, consider first that we have some control over the order and occurrence of interrupts by means of the masking process. You will recall that during an I/O or external interrupt the supervisory programmer had to "mask out" the same type of interrupt source in order not to get caught in an endless program loop. He did this by setting the appropriate system mask bits in the new PSW to zero through use of the Set System Mask (SSM) privileged instruction. Thus, if an I/O or external interrupt request occurred on top of an interrupt from the same source, the new interrupt request would be kept pending until the first request had been serviced. Similarly, we have seen that the problem programmer has the choice of ignoring four of the 15 possible program exceptions that cause a program check interrupt, by setting (to zero) the appropriate program mask bits (36–39) in the PSW, through use of the Set Program Mask (SPM) instruction. The remaining eleven program exceptions and the supervisor call interrupt cannot be masked, however.

When simultaneous interrupt requests occur that are not or cannot be masked, they are "stacked up" in a certain pre-determined order, just like airplanes in a landing pattern. The order in which simultaneous, unmasked interrupt requests are stacked (recognized or honored) is as follows (see Figure 145):

1. Machine check
2. Program or supervisor call
3. External
4. Input/output

The program and supervisor call interrupts are shown on the same line, since they are mutually exclusive; both cannot occur at the same time. (A supervisor call is a valid instruction and cannot result in a program error.)

The fact that interrupt requests are stacked in the order shown above does not mean, however, that they are *executed* in this order; almost the reverse is true. The order of *priority* for executing interrupt subroutines is:

1. Machine check
2. Input/output
3. External
4. Program or supervisor call

The machine check interrupt has the highest priority. When it occurs, the current operation is terminated and no other interrupts are normally taken until the recording procedure is completed.

When no machine check occurs, but other simulta-

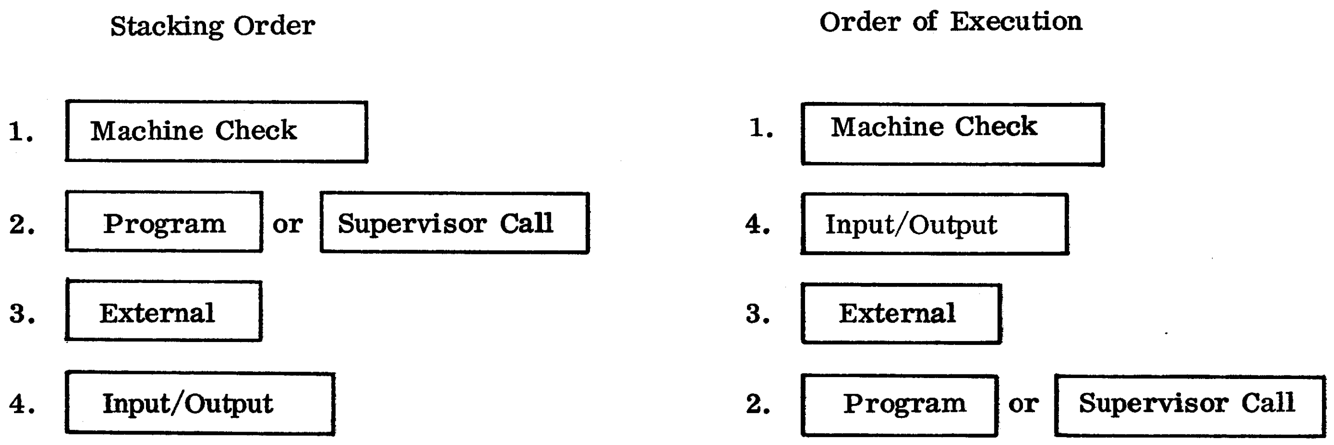


Figure 145. Order of stacking and executing simultaneous interrupt requests

neous interrupt requests are present, the program or supervisor call interrupt is stacked first, the external interrupt is next, and the I/O interrupt is stacked last. The execution of the problem program is delayed, while the old PSW's are stored and the new ones are fetched, for each interrupt in turn. No instructions are actually executed during the stacking of the PSW's, and the process of storing and fetching continues until no more interrupts are to be recognized. When the last interrupt request has been serviced, the execution of instructions is resumed, starting with the PSW *last* fetched. Thus, with the exception of the machine check interrupt, the order of execution of interrupt subroutines (the priority order) is the *reverse* of the order in which PSW's are fetched (the stacking order). The most important interrupts — input/output — are actually serviced first; the external interrupts are next and then come program or supervisor call interrupts.

A business analogy illustrates why interrupt requests are stacked in reverse order of priority. A secretary has three phones on her desk. The most important, a red phone, is connected to the manager's office. A white phone — slightly less important — is for calls from important clients, and a black phone is for calls from vendors and salesmen, considered least important. If all three phones ring simultaneously and she wants to take care of all callers, she will answer the phones in reverse order of their importance (taking a chance, however, on antagonizing her manager). She first answers the black phone, notes down the vendor's name and tells him to hold. She then answers the white phone, takes note of the client's name and request, and puts the phone on hold. Finally, she answers the boss's phone and takes care of his request completely. Now she comes back first to the client and handles his request, and then to the vendor and takes care of him. In this way, she has serviced all requests in order of priority without ignoring any calls.

Consider now the mechanics of the action involved if, for example, an I/O, a program check, and an external interrupt all occur during the execution of a single instruction (see Figure 146). The program interrupt request is recognized first. Accordingly, the current PSW is stored in location 40, the old PSW location for

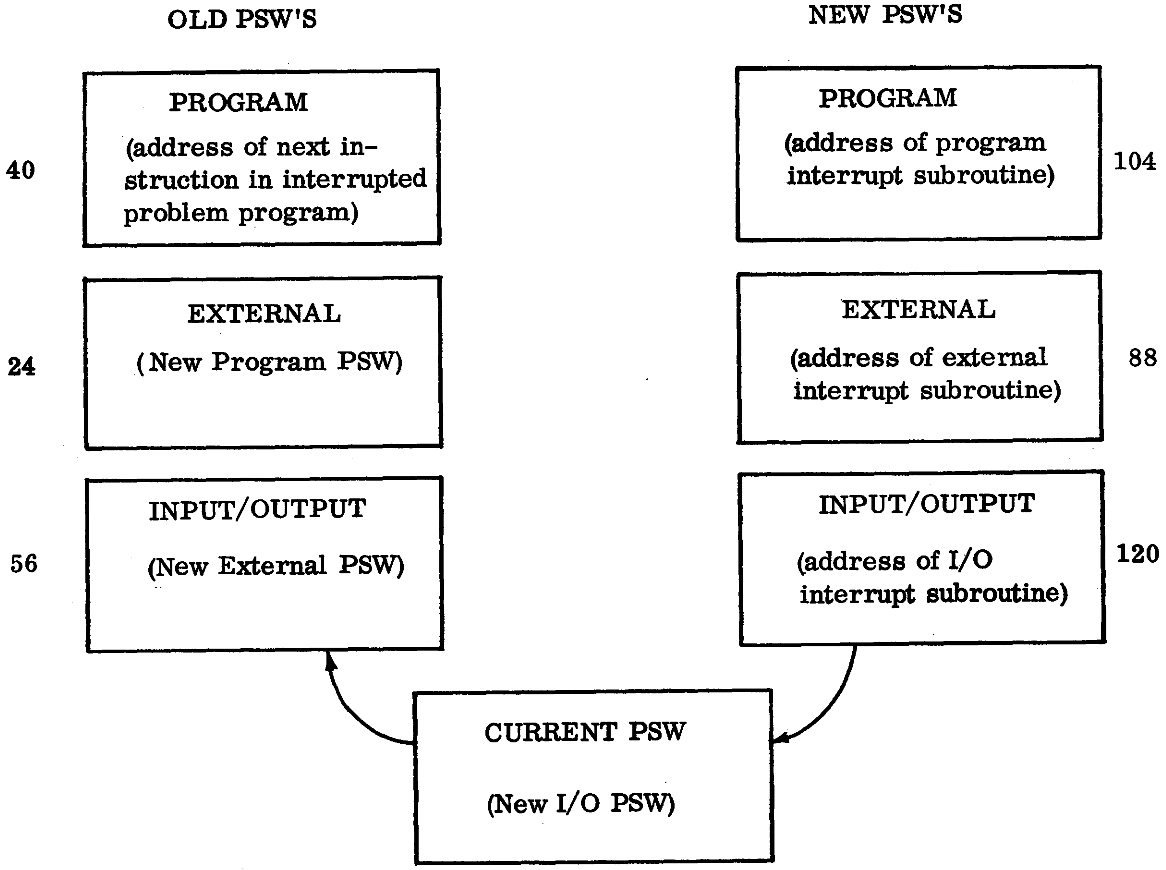

OLD PSW'S
NEW PSW'S
40
PROGRAM
(address of next instruction in interrupted problem program)
PROGRAM
(address of program interrupt subroutine)
104
24
EXTERNAL
(New Program PSW)
EXTERNAL
(address of external interrupt subroutine)
88
56
INPUT/OUTPUT
(New External PSW)
INPUT/OUTPUT
(address of I/O interrupt subroutine)
120
CURRENT PSW
(New I/O PSW)

Figure 146. Stacking of PSW's during simultaneous program, external, and I/O interrupt requests

a program interrupt, and a new PSW is fetched from location 104 to serve as the current PSW. The external interrupt request will prevent the first instruction of the program interrupt subroutine from being executed at this time, provided, however, that the current PSW is not masked for the external interrupt cause. If the CPU is interruptible for the external interrupt source, the current PSW is stored in old PSW location 24, and a new external PSW is fetched from location 88 and becomes the current PSW. Again, the existence of an I/O interrupt request prevents the execution of the external interrupt subroutine, provided the current PSW is interruptible for the I/O interruption cause. If so, the latest current PSW is stored in location 56, reserved for old I/O PSW's, and a new I/O PSW is fetched from location 120. This now becomes the current PSW.

With no further interrupts pending, the CPU automatically branches to the address of the I/O subroutine, contained in the current PSW, and executes it. The last instruction of this subroutine (the Load PSW instruction) causes the old I/O PSW (in location 56) to be restored as the current PSW. It, in turn, contains the address of the external interrupt subroutine, which is entered and executed. Again, the last step of this subroutine causes the old external PSW (in location 24) to be restored as the current PSW. Since this PSW contains the address of the program interrupt subroutine, the CPU enters it and executes it, in turn. The old program PSW now becomes the current PSW again, and directs the CPU to the address of the next instruction in the problem program, at the point of the multiple interruption. Instruction sequencing and execution of the original problem program then resume.

The example illustrates the logical order of stacking and executing simultaneous interrupts. All interrupt requests were serviced, with the highest priority being given to the I/O devices. The slight delay encountered in handling pending interrupt requests is of little practical consequence. Most important, all PSW switching was done automatically by the circuitry and, hence, required no attention whatsoever from the problem programmer.

Summary

System/360 is designed to operate continuously with a minimum of manual intervention. The comprehensive automatic interrupt system handles input/output interrupts, external device signals, program exceptions, machine errors, and similar conditions that may affect the sequence of instructions being executed. Since interrupts are handled and serviced by electronic circuitry and an IBM-prepared supervisory program, the problem programmer need not usually concern himself with these conditions.

The interrupt system recognizes five general classes of interrupts: machine check, program, supervisor call, external, and input/output interrupts. Each class of interrupts has both an old and a new PSW in reserved storage locations. In all classes, an interrupt involves storing the current PSW in its old storage position and making the PSW at the new position the current PSW. Machine check interrupts, external interrupts, I/O interrupts, and some program interrupts can be masked (ignored) by setting the appropriate mask bits in the current PSW (usually through the control program). In the event of an interrupt, the old PSW holds the cause of the interruption and also all necessary status information pertaining to the system at the time of interrupt. If the last instruction of the interrupt routine is the Load PSW instruction, the old PSW is restored to current status and the interrupted program continues.

Summary Chart of Interrupt Action

SOURCE IDENTIFICATION	INTERRUPTION CODE PSW BITS 16-31	MASK BITS	ILC SET	EXECUTION
Input/Output (old PSW 56, new PSW 120)				
Channel 0	00000000 aaaaaaaa	0	x	completed
Channel 1	00000001 aaaaaaaa	1	x	completed
Channel 2	00000010 aaaaaaaa	2	x	completed
Channel 3	00000011 aaaaaaaa	3	x	completed
Channel 4	00000100 aaaaaaaa	4	x	completed
Channel 5	00000101 aaaaaaaa	5	x	completed
Channel 6	00000110 aaaaaaaa	6	x	completed
Program (old PSW 40, new PSW 104)				
Operation	00000000 00000001		1,2,3	suppressed
Privileged operation	00000000 00000010		1,2	suppressed
Execute	00000000 00000011		2	suppressed
Protection	00000000 00000100		0,2,3	suppressed or terminated
Addressing	00000000 00000101		0,1,2,3	suppressed or terminated
Specification	00000000 00000110		1,2,3	suppressed
Data	00000000 00000111		2,3	terminated
Fixed-point overflow	00000000 00001000	36	1,2	completed
Fixed-point divide	00000000 00001001		1,2	suppressed or completed
Decimal overflow	00000000 00001010	37	3	completed
Decimal divide	00000000 00001011		3	suppressed
Exponent overflow	00000000 00001100		1,2	terminated
Exponent underflow	00000000 00001101	38	1,2	completed
Significance	00000000 00001110	39	1,2	completed
Floating-point divide	00000000 00001111		1,2	suppressed
Supervisor Call (old PSW 32, new PSW 96)				
Instruction bits	00000000 rrrrrrrr		1	completed
External (old PSW 24, new PSW 88)				
Timer	00000000 1nnnnnnn	7	x	completed
Interrupt key	00000000 n1nnnnnn	7	x	completed
External signal 2	00000000 nn1nnnnn	7	x	completed
External signal 3	00000000 nnn1nnnn	7	x	completed
External signal 4	00000000 nnnn1nnn	7	x	completed
External signal 5	00000000 nnnnn1nn	7	x	completed
External signal 6	00000000 nnnnnn1n	7	x	completed
External signal 7	00000000 nnnnnnn1	7	x	completed
Machine Check (old PSW 48, new PSW 112)				
Machine malfunction	00000000 00000000	13	x	terminated

NOTES

a	Device address bits
n	Other external-interruption conditions
r	Bits of R_1 and R_2 field of SUPERVISOR CALL
x	Unpredictable

Questions and Exercises

Note: Blanks are not necessarily to be filled in with only one word.

1. In noninterruptible systems of the past, exceptional conditions, such as arithmetic overflows or I/O errors, had to be handled by the program, by repeated program testing for the particular condition and branching to the appropriate corrective subroutine. With the automatic interrupt system, the machine itself recognizes the exceptional condition and automatically fetches the appropriate corrective routine. True or False?

2. An automatic branch to the Supervisor program is called an ____________.

3. An interrupt of a particular class replaces the entire ____________ program status word (PSW) by placing it in the ____________ location in main storage for that class and then fetching a ____________ from main storage.

4. The general status of the processor, the reason for the interrupt, and the address of the next instruction in the problem program sequence are all contained in the ____________ PSW.

5. The interrupt-handling routine is initiated by the automatic fetching of the ____________ from storage and loading it as the ____________ PSW.

6. After the interrupt has been serviced, the instruction sequence of the original problem program may be resumed from the point of interruption by means of the ____________ instruction, which is the ____________ instruction of the interrupt-handling routine.

7. In the example of a fixed-point overflow *without* automatic machine interrupt, the programmer uses the ____________ instruction after each arithmetic operation to test the ____________ code in the PSW for overflow.

8. If an overflow occurred in the example (7, above), the ____________ instruction is used to go to the overflow fixup routine and provide a return link.

9. In the example of fixed-point overflow using the automatic interrupt mechanism, neither BC nor BAL instructions are needed, since the program is interrupted automatically in the event of overflow and control is passed to a corrective subroutine. True or False?

10. a. Name the five general classes of interrupts.
 b. Each class has two distinct ____________ word storage locations for the ____________ and ____________ PSW's, respectively.

11. The *specific* reason for an interrupt is given by the ____________ code, contained in bits 16–31 of the ____________ PSW. The interruption code is set automatically after an ____________ occurs.

12. Program interrupts can be caused by ____________ different types of programming errors. Most of these are due to ____________ errors; the remainder are concerned with improper ____________, invalid ____________ or data specification, and violation of ____________.

13. One cause of an input/output interrupt is the ____________ of an I/O operation.

14. Which of the following is/are true?
 a. Machine check interrupts are caused by various machine errors and hardware malfunctions.
 b. Invalid data can never result in a machine check.
 c. A machine check can cause an error stop.
 d. Incorrect parity due to a power interruption will cause a machine check.

15. External interrupts may be caused by (a) external device signals (available with the Direct Control feature), (b) the depression of the Interrupt key at the console, and (c) a condition *within* the CPU. Describe this condition.

16. The supervisor call interrupt differs from other types of interrupts in that the ____________ makes use of the automatic interrupt linkage by means of the ____________ instruction. The most common use of the supervisor call interrupt is to switch from the ____________ state to the ____________ state.

17. Preventing an interrupt or keeping it pending is called ____________.

18. By masking the PSW, the detailed status of the CPU can be preserved for subsequent inspection. True or False?

19. Machine check interrupts, __________, __________, and four of 15 ____________ exceptions can be masked: the remaining __________ exceptions and the __________ interrupt cannot be masked.

20. When an interrupt condition occurs on top of an interrupt of the same type, the second interrupt must be ________ to prevent the CPU from getting caught in a program ________ due to destruction of the ________ PSW from the problem program.

21. Masking is handled by three fields in the PSW: the ________ mask, ________ mask, and the ________ mask.

22. Input/output and external interruptions may be masked by the ________ mask, which contains ________ bits: ________ bits are used for selector channel masking, while one bit each is used for the ________ and for ________.

23. To prevent a particular type of interrupt from occurring, the mask bits for that type must be made ________; when the mask bits are set to ________, the CPU is interruptible for that type of condition.

24. A system mask contains the bit configuration 01100001. Which interrupts are permitted and which are masked?

25. Bits 36–39 of the PSW constitute the ________ mask. These bits permit masking the following program exceptions: (a) ________ (b) ________ (c) ________ (d) ________.

26. Interrupts take place after the current instruction is finished and before the next instruction is started. The execution of the current instruction is not affected by the occurrence of an interrupt. True or False?

27. Depending on the cause and type of interrupt, the preceding instruction may be either ________, ________, or ________.

28. In the event of I/O, external, and supervisor call interrupts, the instruction preceding the interrupt is ________; in the event of a machine check interrupt, the current instruction is ________ and the result may not be stored.

29. In the event of a program check interrupt, the current instruction may be ________, ________, or ________, depending upon the type of programming error and the time of its detection.

30. The length of the last executed instruction in the interrupted program is indicated by the ________ code in the ________ PSW.

31. For the supervisor call interruption, the ILC is ________, while for a program interrupt it may be ________, ________, or ________.

32. The instruction length code must be 1, 2, or 3. True or False?

33. The Interrupt Supervisor is a ________ that analyzes the cause of the interrupt and takes the proper action.

34. The Supervisor is the part of the IBM-prepared ________ program that is permanently kept in main storage. When necessary, the Supervisor calls in other portions of the control program and processing alternates between the ________ and ________ programs.

35. In case of I/O interrupts, the Supervisor passes control immediately to the ________, which consists of a series of routines for handling I/O operations or detecting I/O error conditions.

36. Describe five supervisor functions that may be requested by the problem program via the Supervisor Call (SVC) instruction.

37. Timer and program interrupts are turned over by the Supervisor to the appropriate ________ routines for handling.

38. There are always two independently prepared programs in main storage: the ________ program and the ________ program.

39. The major way the machine recognizes which of the two programs is in control at any time is by differentiating between (a) ________ and ________ program states, and (b) ________ and ________ instructions.

40. When bit 15 of the PSW is ________, the CPU is in the supervisory state: when bit 15 is ________, the CPU is in the problem state.

41. Switching between the problem and supervisor states can be done only by introducing a new PSW via the Load PSW instruction. True or False?

42. Instructions relating to I/O operations, storage protection, direct-control devices, and other parts of the Supervisor program are considered ________ and, hence, are not ________ when the CPU is in the ________ state; in the ________ state all instructions are valid.

43. If a privileged instruction is encountered in the problem state, a ________ results.

44. Which of the following instructions is/are valid when the machine is in the problem state (PSW bit 15 is 1) and which may be issued only by the Supervisor program (PSW bit 15 is 0):

a. Load PSW
b. Supervisor Call
c. Set System Mask
d. Set Program Mask
e. Start I/O
f. Any I/O instruction

Answers to Questions and Exercises

Chapter 1: Architecture

1. Data from a tape unit may be transmitted over a multiplexor channel, in which case the channel operates in burst mode. Like other I/O devices, tape units are attached to a control unit, which, in turn, is attached to a channel.
2. The Load PSW instruction is a privileged instruction, and an attempt to execute this instruction by the problem program will cause a program interrupt.
3. Because the Supervisor Call instruction contains an eight-bit code that is stored in the old supervisor call PSW in the course of interruption, the routine must first examine this code in the old PSW. The code may be regarded as a message conveyed by the instruction to the supervisor.
4. The valid packed decimal digit codes are:
   0000 0001 0010 0011 0100
   0101 0110 0111 1000 1001
which represent the digits 0-9.
5. Hollerith code read by a card reader is transferred from the card reader's control unit as EBCDIC.
6. Fixed-point arithmetic instructions are part of the standard instruction set. Neither the optional decimal nor floating-point instruction set is sufficient in itself to perform processing.
7. The position of the sign depends on the format of the data. A binary quantity is represented internally by a 32-bit binary number. The sign occupies the high-order (leftmost) bit position. The sign of a number in EBCDIC occupies the zone position of the least significant digit. The sign of a packed decimal number occupies the low-order four bits of the field.
8. The effective address is specified by the 24 least significant bits in register 5.
9. Length is not a criterion for the selection of instructions. The programmer knows the location (in storage, registers, or both) of data to be operated upon and the operations to be performed. His selection is made accordingly, and halfword, word, and three-halfword instructions are mixed within a program.

Chapter 2: Number Systems

1. $(121{,}001)_3 = (3213)_5 = (110{,}110{,}001) = (433)_{10}$
   $(100011)_2 = (35)_{10}$; $(0.111111)_2 = (0.984375)_{10}$

2. $(9B4D.3A7)_{16} = (39{,}757.228...)_{10}$

3. $10001010_2 = 8A_{16}$; $1001110100_2 = 274_{16}$;
   $(1110101.001110101)_2 = (75.3A8)_{16}$

4. A72B = 1010 0111 0010 1011;
   39BF4D = 111001 1011 1111 0100 1101;
   ABCDEF = 1010 1011 1100 1101 1110 1111;
   52.A7EF98 = 1010010.1010 0111 1110 1111 1001 1;
   123.ABC = 1 0010 0011.1010 1011 11

5.
$$\begin{array}{r} 1100011 \\ +\ 0111001 \\ \hline 10011100 \end{array} \qquad \begin{array}{r} 111101111 \\ +\ 111101111 \\ \hline 1111011110 \end{array}$$
$$\begin{array}{r} 1000\ 1111\ 1010\ 0001 \\ +\ 0001\ 0011\ 1110\ 0101 \\ \hline 1010\ 0011\ 1000\ 0110 \end{array}$$

6.
$$\begin{array}{r} 12345_{16} \\ +\ 56789 \\ \hline 68ACE \end{array} \qquad \begin{array}{r} 8FB5 \\ +\ CD69 \\ \hline 15DIE \end{array} \qquad \begin{array}{r} 345.789 \\ +\ 832.BDE \\ \hline B78.367 \end{array}$$
$$\begin{array}{r} ABCD.09EF \\ +\ 1234.5698 \\ \hline BE01.6087 \end{array}$$

7. $100000 - 1 = 11111$;

$$\begin{array}{rrr} 111010 & 111111111 & 10001.11001 \\ -\ 100100 & -\ 100000000 & -\ 1101.00110 \\ \hline 10110 & 11111111 & 100.10011 \end{array}$$

8.

$$\begin{array}{rr} \mathtt{F865} & \mathtt{E73F.A983} \\ -\mathtt{9AB7} & -\mathtt{A9CD.87FE} \\ \hline \mathtt{5DAE} & \mathtt{3D72.2185} \end{array}$$

9. $1100 \times 11 = 100100$; $1010 \times 1001 = 1011010$; $10.001 \times 1.01 = 10.10101$

10. 3E7 × 5B9 = 1654EF; D.38 × 6.EF = 5B.A748

11. $(39)_{10} = 100111_2$; $583 = 1001000111$; $7946 = 1111100001010$

12. $(89)_{10} = (59)_{16}$; $438 = \text{1B6}$; $999 = \text{3E7}$; $5793 = \text{16A1}$; $(875{,}472{,}925)_{10} = (\text{34,2EA,81D})_{16}$

13. $110101_2 = 53_{10}$; $1110110001 = 945$; $111111111 = 511$ (i.e., $2^9 - 1$)

14. $(\text{7E5})_{16} = (2021)_{10}$; $\text{F8D} = 3981$; $\text{89F7} = 35{,}319_{10}$

15. $(0.79)_{10} = (0.1100\ 1010\ 0011)_2 = (\text{0.CA3})_{16}$
$(0.6666666)_{10} = (0.1010\ 1010\ 1010\ 1010\ 1010\ldots)_2$
$= (\text{0.AAAAAA}\ldots)_{16}$
$(0.123)_{10} = (0.0001\ 1111\ 011)_2 = (\text{.1F6})_{16}$
$(34.675)_{10} = (100010.1010\ 1100\ 1100\ 1100\ldots)_2 =$
$(\text{22.ACCC}\ldots)_{16}$

Chapter 3: Introduction to Assembler Language Programming

1. Leftmost, leftmost

2. Hexadecimal

3a. 112
 b. 130
 c. (1) 15
 (2) 102
 (3) 02E
 (4) 130

4. The DC will cause data to be placed into storage. The DS will allocate space in storage without placing any data there.

5. 130

6a. 130-133
 b. 138-13B
 c. 140-147

7. 148, 150, 158, 160
The assembler will automatically align a doubleword area specified as shown on a doubleword boundary. Therefore, locations were skipped preceding AREA2, AREA4, and AREA6 so that each starts on a doubleword boundary.

8. DS D
DS D
DS D
DS F
DS F
DS F
Note that this arrangement saves 12 bytes of the 48 bytes assumed by the arrangement in question 7. Thus ¼ of the original 48 bytes were saved by the new sequence of statements.

Chapter 4: Fixed-Point Operations

1. Fullword
2. Receives
3. Sends
   Exception
4. No. The first operand must specify an even-numbered register of 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      15,0
                Using     *,15
                L         2,XANDY
                SRDL      2,12
                SRL       3,20
                ST        2,X
                STH       3,Y
                SVC       0
       XANDY    DS        F
       X        DS        F
       Y        DS        H
                END       BEGIN
```

7. (c) Condition code is 1 or 3.
8. BC 15,NEWONE
9. LM 2,5,X1
10. SR 5,5
11. It will be the sum of the contents of register 15 (the base register), register 11 (the index register), and the displacement.
12. BXLE 5,6,NEWONE

Chapter 5: Programming with Base Registers and the USING Instruction

1a. USING *,15

b. BALR 15,0

	Value Assumed by Assembler	Value Loaded at Execution Time	
		Program loaded at 200_{16}	Program loaded at 1200_{16}
Reg 15	202	202	1202

Location (relocated)	Location	Object Instruction		Execution Time Effective Address				
		Base Register	Displace-ment	Program Loaded at 200_{16}	Program Loaded at 1200_{16}			
							START	512
1200	200					BEGIN	BALR	15,0
							USING	*,15
1202	202	F	102	304	1304		L	2,DATA
1206	206	F	122	324	1324		A	2,TEN
	⋮						⋮	
1234	234	F	106	308	1308		S	2,DATA+4
1238	238	F	126	328	1328		ST	2,RESULT
	⋮						⋮	
1252	252	F	142	344	1344		L	6,BIN1
	⋮						⋮	
1304	304					DATA	DC	F'25'
1308	308						DC	F'15'
	⋮						⋮	
1324	324					TEN	DC	F'10'
1328	328					RESULT	DS	F
	⋮						⋮	
1344	344					BIN1	DC	F'12'
							⋮	
							END	BEGIN

Symbol Table

	Location	Length
BEGIN	200	2
BIN1	344	4
DATA	304	4
RESULT	328	4
TEN	324	4

Note that the object instruction base register specifications and displacements still work perfectly even though the program is relocated. This is because the displacement factor always indicates how far away the location of the symbolic operand is from the base address. If the program is loaded at 200, DATA is 104 bytes past the base address 102. If the program is loaded at 1200, DATA is still 104 bytes past the base address.

	Value Assumed by Assembler	Value Loaded at Execution Time	
		Program loaded at 1000_{16}	
Reg 15	1002	1002	
Reg 14	2002	2002	
Reg 13	3002	3002	

	Location	Object Instruction		Execution Time Effective Address				
		Base Register	Displace-ment	Program Loaded at 1000_{16}				
							START	4096
	1000					BEGIN	BALR	15,0
							USING	FIRST,15
	1002					FIRST	BC	15,SKIP
	1006					DATA	DC	F'3472'
	⋮						⋮	
	1024					BASE1	DC	A(FIRST+4096)
	1028					BASE2	DC	A(FIRST+8192)
	⋮						⋮	
	1104	F	022	1024		SKIP	L	14,BASE1
	⋮						USING	FIRST+4096,14
	1108	F	026	1028			L	13,BASE2
	⋮						USING	FIRST+8192,13
	⋮						⋮	
	2504	D	902	3904			BC	15,CK8
	⋮						⋮	
	2898	F	004	1006		LOOP	A	4,DATA
	⋮						⋮	
	3204					LOOPB	S	5,DATA
	⋮						⋮	
	3508	E	896	2898			BC	8,LOOP
	⋮						⋮	
	3904	D	202	3204		CK8	BC	8,LOOPB
							END	BEGIN

Symbol Table

Symbol	Location
BASE 1	1024
BASE2	1028
BEGIN	1000
CK8	3904
DATA	1006
FIRST	1002
LOOP	2898
LOOP B	3204
SKIP	1104

Chapter 6: Decimal Operations

1a. CON3 DC PL5'3'
b. 000000003F

2. Assembler
Data definitions
Programmer

3. Equal to

4. One less than

5a. A storage area which contains the multiplicand in the low-order positions and zeros in the high-order positions.
Multiplier
b. In the storage area specified by the first operand.

6a. 00 02 48 9+ 10 3+
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. 2
b. 8

9a. 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 45 55 6S
c. SOURCE 66 55 44 33 22 11
DEST 00 00 00 04 43 3S

10. 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.

11a. MVN RESULT+5(1),FACTOR+4
MVO RESULT,FACTOR(4)
b. MVN FACTOR+3(1),FACTOR+4
ZAP RESULT,FACTOR(4)

12a. SI
b. NI HOLD,X'00'
c. NI HOLD+3,X'0F'

13. 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.

14. Packed decimal

15. PACK

16. UNPK (Unpack)

17a. DC F'578'
b. DC ZL3'578'
c. DC PL2'578'

18. 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.

19. IC 6,OLD

20. STC 6,OLD

21a. 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 OF could be inserted immediately before the DC defining MASK.

c. DC F'15'

Chapter 7: 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 +)
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 8: Edit, Translate, and Execute Instructions

1. BBB1540
2. BBB5721BB
3. BBBBBBB.01BCR
4. BBBBBBBBB
5. BB0,000.10BB
6. BBBB101.43CRBBBBBB1.07 BCR

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 9: Subroutines and Subprograms

1a. Point F

b. Point D

c. Point C

d. Point E

e. Point D

2. No operation

```
3.         CALLER              CALLED
        EXTRN ROUT1         ENTRY ROUT1
                               .     .
                               .     .
                               .     .
                     ROUT1 . . . . . . . .
4.         CALLER              CALLED
        EXTRN ROUT1         ENTRY ROUT1
          .     .              .     .
          .     .              .     .
          .     .              .     .
        LR     13,ACON   ROUTI. . . . . . . .
        BALR   14,13
          .     .
          .     .
          .     .
ACON DC  A(ROUT1)
```

5. See next page.
6. BASE1 contains 00005002.
   BASE2 contains 00006002.
7. See next page.
8. See next page.

Relocation Constant

3000

	Value Assumed by Assembler	Value Loaded at Execution Time	
		Program loaded at 1000_{16}	Program loaded at 4000_{16}
Reg 15	1002	1002	4002
Reg 14	2002	2002	5002
Reg 13	3002	3002	6002

Location (relocated) at 4000_{16}	Location	Object Instruction		Execution Time Effective Address				
		Base Register	Displacement	Program Loaded at 1000_{16}	Program Loaded at 4000_{16}			
							START	4096
4000	1000					BEGIN	BALR	15,0
							USING	FIRST, 15
4002	1002					FIRST	BC	15, SKIP
4006	1006					DATA	DC	F'3472'
	⋮						⋮	
4024	1024					BASE1	DC	A(FIRST+4096)
4028	1028					BASE2	DC	A(FIRST+8192)
	⋮						⋮	
4104	1104	F	022	1024	4024	SKIP	L	14,BASE1
							USING	FIRST+4096, 14
4108	1108	F	026	1028	4028		L	13,BASE2
	⋮						USING	FIRST+8192, 13
	⋮						⋮	
5504	2504	D	902	3904	6904		BC	15,CK8
	⋮						⋮	
5898	2898	F	004	1006	4006	LOOP	A	4,DATA
	⋮						⋮	
6204	3204					LOOPB	S	5, DATA
	⋮						⋮	
6508	3508	E	896	2898	5898		BC	8,LOOP
	⋮						⋮	
6904	3904	D	202	3204	6204	CK8	BC	8,LOOPB
							END	BEGIN

Symbol Table

Symbol	Location
BASE 1	1024
BASE 2	1028
BEGIN	1000
CK8	3904
DATA	1006
FIRST	1002
LOOP	2898
LOOP B	3204
SKIP	1104

Chapter 10: Floating Point and Advanced Loops in Scientific Applications

```
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
    C7100000
```

4. 46XXXXXX
The effect of the instructions is to construct a short floating-point unnormalized number from a positive binary integer and a hexadecimal constant.

```
5a. Floating-point register 2 – 42100000
    Floating-point register 4 – 42080000
 b. Floating-point register 6 – 41F00000
    General register 6        – 00000000
 c. A=42100000
    General register 3        – 42100000
```

```
6.          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

             .      .
             .      .
             .      .
     A      DS     F
     B      DS     F
     C      DS     F
```

```
7a.          L      13,N         N to GR13
             LR     4,13         N to GR4
             A      4,CON46      Converted to
                                   floating
             ST     4,FLN          unnormalized
             S      13,ONE       (N–1) to GR13
             SLL    13,2         4(N–1) to GR13
             LA     12,4         4 to GR12
             LA     11,4         4 to GR11
             LE     6,A          A(1) to FPR 6
     NEXT    AE     6,A(11)      Add Ai to FPR6
             BXLE   11,12,NEXT   Test
             DE     6,FLN        Prenormalize N
                                 and divide into Σ Ai
             STE    6,AVER       Store result
     ONE     DC     F'1'
     CON46   DC     X'46000000'
     N       DS     F
     FLN     DS     F
     AVER    DS     F
     A       DS     100F

7b.          L      13,N         N to GR13
             LR     4,13         N to GR4
             A      4,CON46      Converted to
                                   floating
             ST     4,FLN          unnormalized
             S      13,ONE       (N–1) to GR13
             SLL    13,3         8–(N–1) to GR13
             LA     12,8         8 to GR12
             LA     11,8         8 to GR11
             LD     6,A          A(1) to FPR6
     NEXT    AD     6,A(11)      Add Ai to FPR6
             BXLE   11,12,NEXT   Test
             DD     6,FLN        Prenormalize N
                                 and divide into Σ Ai
             STD    6,AVER       Store result
     ONE     DC     F'1'
     N       DS     F
     CON46   DC     X'46000000'
     FLN     DC     D'0'
                    (see note below)
     AVER    DS     D
     A       DS     100D
```

Note: Provide for use of N as a long floating-point number with zeros in the last 8 positions.

Chapter 11: Automatic Interrupts

1. True

2. Interrupt

3. Current, old PSW, new PSW

4. Old

5. New PSW, current

6. Load PSW, last

7. Branch on Condition, condition

8. Branch and Link

9. True

10. a. external, supervisor, program, machine, input/output
    b. double, old, new

11. Interrupt, old, interrupt

12. 15, arithmetic, addressing, instructions, storage protection

13. Completion

14. a, c, and d. b is false because, after a power interruption or system reset, incorrect parity may exist in storage or registers

15. Timer value changes from positive to negative

16. Program, Supervisor Call, problem, supervisor

17. Masking

18. False

19. External, I/O, program, program, Supervisor Call

20. Masked, loop, old

21. System, machine check, program

22. System, eight, six, multiplexor channel, external interrupts

23. Zero, one

24. External interrupts and I/O interrupts from selector channels 1 and 2 are permitted. Multiplexor channel interrupts and selector channels 3–6 are masked, that is, kept pending.

25. Program
    a. fixed-point overflow
    b. decimal overflow
    c. exponent overflow
    d. significance

26. False. The cause of the interrupt may affect the manner in which the current instruction is completed.

27. Completed, terminated, suppressed

28. Completed, terminated

29. Completed, terminated, suppressed

30. Instruction length, old

31. 1, 1, 2, 3

32. False; it could be zero

33. Program of instructions

34. Control, problem, supervisor

35. Input/output supervisor

36. Load a library program
    Provide operator communication
    Request an I/O operation (Execute Channel Program)
    Return to the main program after a timer routine or operator inquiry
    Terminate the program and prepare for the next job

37. User-prepared

38. Supervisor, problem

39. a. Problem, supervisor
    b. Processing, privileged

40. Zero, one

41. False. Switching between states also occurs as a result of any interrupt, including the supervisor call interrupt.

42. Privileged, valid, problem, supervisor

43. Program interruption

44. The problem program may issue b and d.
a, c, e, and f are privileged instructions.
All of the listed instructions may be issued by the Supervisor program.

STANDARD INSTRUCTION SET

NAME	MNEMONIC	TYPE	OPERAND	CODE
Add	AR	RR	R1, R2	1A
Add	A	RX	R1, D2(X2, B2)	5A
Add Halfword	AH	RX	R1, D2(X2, B2)	4A
Add Logical	ALR	RR	R1, R2	1E
Add Logical	AL	RX	R1, D2(X2, B2)	5E
AND	NR	RR	R1, R2	14
AND	N	RX	R1, D2(X2, B2)	54
AND	NI	SI	D1(B1), I2	94
AND	NC	SS	D1(L, B1), D2(B2)	D4
Branch and Link	BALR	RR	R1, R2	05
Branch and Link	BAL	RX	R1, D2(X2, B2)	45
Branch on Condition	BCR	RR	M1, R2	07
Branch on Condition	BC	RX	M1, D2(X2, B2)	47
Branch on Count	BCTR	RR	R1, R2	06
Branch on Count	BCT	RX	R1, D2(X2, B2)	46
Branch on Index High	BXH	RS	R1, R3, D2(B2)	86
Branch on Index Low or Equal	BXLE	RS	R1, R3, D2(B2)	87
Compare	CR	RR	R1, R2	19
Compare	C	RX	R1, D2(X2, B2)	59
Compare Halfword	CH	RX	R1, D2(X2, B2)	49
Compare Logical	CLR	RR	R1, R2	15
Compare Logical	CL	RX	R1, D2(X2, B2)	55
Compare Logical	CLC	SS	D1(L, B1), D2(B2)	D5
Compare Logical	CLI	SI	D1(B1), I2	95
Convert to Binary	CVB	RX	R1, D2(X2, B2)	4F
Convert to Decimal	CVD	RX	R1, D2(X2, B2)	4E
Diagnose		SI		83
Divide	DR	RR	R1, R2	1D
Divide	D	RX	R1, D2(X2, B2)	5D
Exclusive OR	XR	RR	R1, R2	17
Exclusive OR	X	RX	R1, D2(X2, B2)	57
Exclusive OR	XI	SI	D1(B1), I2	97
Exclusive OR	XC	SS	D1(L, B1), D2(B2)	D7
Execute	EX	RX	R1, D2(X2, B2)	44
Halt I/O	HIO	SI	D1(B1)	9E
Insert Character	IC	RX	R1, D2(X2, B2)	43
Load	LR	RR	R1, R2	18
Load	L	RX	R1, D2(X2, B2)	58
Load Address	LA	RX	R1, D2(X2, B2)	41
Load and Test	LTR	RR	R1, R2	12
Load Complement	LCR	RR	R1, R2	13
Load Halfword	LH	RX	R1, D2(X2, B2)	48
Load Multiple	LM	RS	R1, R3, D2(B2)	98
Load Negative	LNR	RR	R1, R2	11
Load Positive	LPR	RR	R1, R2	10
Load PSW	LPSW	SI	D1(B1)	82
Move	MVI	SI	D1(B1), I2	92
Move	MVC	SS	D1(L, B1), D2(B2)	D2
Move Numerics	MVN	SS	D1(L, B1), D2(B2)	D1
Move with Offset	MVO	SS	D1(L1, B1), D2(L2, B2)	F1
Move Zones	MVZ	SS	D1(L, B1), D2(B2)	D3
Multiply	MR	RR	R1, R2	1C
Multiply	M	RX	R1, D2(X2, B2)	5C
Multiply Halfword	MH	RX	R1, D2(X2, B2)	4C
OR	OR	RR	R1, R2	16
OR	O	RX	R1, D2(X2, B2)	56
OR	OI	SI	D1(B1), I2	96
OR	OC	SS	D1(L, B1), D2(B2)	D6
Pack	PACK	SS	D1(L1, B1), D2(L2, B2)	F2
Set Program Mask	SPM	RR	R1	04
Set System Mask	SSM	SI	D1(B1)	80
Shift Left Double	SLDA	RS	R1, D2(B2)	8F
Shift Left Single	SLA	RS	R1, D2(B2)	8B
Shift Left Double Logical	SLDL	RS	R1, D2(B2)	8D
Shift Left Single Logical	SLL	RS	R1, D2(B2)	89
Shift Right Double	SRDA	RS	R1, D2(B2)	8E
Shift Right Single	SRA	RS	R1, D2(B2)	8A
Shift Right Double Logical	SRDL	RS	R1, D2(B2)	8C
Shift Right Single Logical	SRL	RS	R1, D2(B2)	88
Start I/O	SIO	SI	D1(B1)	9C
Store	ST	RX	R1, D2(X2, B2)	50
Store Character	STC	RX	R1, D2(X2, B2)	42
Store Halfword	STH	RX	R1, D2(X2, B2)	40
Store Multiple	STM	RS	R1, R3, D2(B2)	90
Subtract	SR	RR	R1, R2	1B
Subtract	S	RX	R1, D2(X2, B2)	5B
Subtract Halfword	SH	RX	R1, D2(X2, B2)	4B
Subtract Logical	SLR	RR	R1, R2	1F
Subtract Logical	SL	RX	R1, D2(X2, B2)	5F
Supervisor Call	SVC	RR	I	0A
Test and Set	TS	SI	D1(B1)	93
Test Channel	TCH	SI	D1(B1)	9F
Test I/O	TIO	SI	D1(B1)	9D
Test Under Mask	TM	SI	D1(B1), I2	91
Translate	TR	SS	D1(L, B1), D2(B2)	DC
Translate and Test	TRT	SS	D1(L, B1), D2(B2)	DD
Unpack	UNPK	SS	D1(L1, B1), D2(L2, B2)	F3

DECIMAL FEATURE INSTRUCTIONS

Add Decimal	AP	SS	D1(L1, B1), D2(L2, B2)	FA
Compare Decimal	CP	SS	D1(L1, B1), D2(L2, B2)	F9
Divide Decimal	DP	SS	D1(L1, B1), D2(L2, B2)	FD
Edit	ED	SS	D1(L, B1), D2(B2)	DE
Edit and Mark	EDMK	SS	D1(L, B1), D2(B2)	DF
Multiply Decimal	MP	SS	D1(L1, B1), D2(L2, B2)	FC
Subtract Decimal	SP	SS	D1(L1, B1), D2(L2, B2)	FB
Zero and Add	ZAP	SS	D1(L1, B1), D2(L2, B2)	F8

DIRECT CONTROL FEATURE INSTRUCTIONS

Read Direct	RDD	SI	D1(B1), I2	85
Write Direct	WRD	SI	D1(B1), I2	84

PROTECTION FEATURE INSTRUCTIONS

Insert Storage Key	ISK	RR	R1, R2	09
Set Storage Key	SSK	RR	R1, R2	08

FLOATING-POINT FEATURE INSTRUCTIONS

Add Normalized (Long)	N ADR	RR	R1, R2	2A
Add Normalized (Long)	N AD	RX	R1, D2(X2, B2)	6A
Add Normalized (Short)	N AER	RR	R1, R2	3A
Add Normalized (Short)	N AE	RX	R1, D2(X2, B2)	7A
Add Unnormalized (Long)	AWR	RR	R1, R2	2E
Add Unnormalized (Long)	AW	RX	R1, D2(X2, B2)	6E
Add Unnormalized (Short)	AUR	RR	R1, R2	3E
Add Unnormalized (Short)	AU	RX	R2, D2(X2, B2)	7E
Compare (Long)	CDR	RR	R1, R2	29
Compare (Long)	CD	RX	R1, D2(X2, B2)	69
Compare (Short)	CER	RR	R1, R2	39
Compare (Short)	CE	RX	R1, D2(X2, B2)	79
Divide (Long)	N DDR	RR	R1, R2	2D
Divide (Long)	N DD	RX	R1, D2(X2, B2)	6D
Divide (Short)	N DER	RR	R1, R2	3D
Divide (Short)	N DE	RX	R1, D2(X2, B2)	7D
Halve (Long)	HDR	RR	R1, R2	24
Halve (Short)	HER	RR	R1, R2	34
Load and Test (Long)	LTDR	RR	R1, R2	22
Load and Test (Short)	LTER	RR	R1, R2	32
Load Complement (Long)	LCDR	RR	R1, R2	23
Load Complement (Short)	LCER	RR	R1, R2	33
Load (Long)	LDR	RR	R1, R2	28
Load (Long)	LD	RX	R1, D2(X2, B2)	68
Load Negative (Long)	LNDR	RR	R1, R2	21
Load Negative (Short)	LNER	RR	R1, R2	31
Load Positive (Long)	LPDR	RR	R1, R2	20
Load Positive (Short)	LPER	RR	R1, R2	30
Load (Short)	LER	RR	R1, R2	38
Load (Short)	LE	RX	R1, D2(X2, B2)	78
Multiply (Long)	N MDR	RR	R1, R2	2C
Multiply (Long)	N MD	RX	R1, D2(X2, B2)	6C
Multiply (Short)	N MER	RR	R1, R2	3C
Multiply (Short)	N ME	RX	R1, D2(X2, B2)	7C
Store (Long)	STD	RX	R1, D2(X2, B2)	60
Store (Short)	STE	RX	R1, D2(X2, B2)	70
Subtract Normalized (Long)	N SDR	RR	R1, R2	2B
Subtract Normalized (Long)	N SD	RX	R1, D2(X2, B2)	6B
Subtract Normalized (Short)	N SER	RR	R1, R2	3B
Subtract Normalized (Short)	N SE	RX	R1, D2(X2, B2)	7B
Subtract Unnormalized (Long)	SWR	RR	R1, R2	2F
Subtract Unnormalized (Long)	SW	RX	R1, D2(X2, B2)	6F
Subtract Unnormalized (Short)	SUR	RR	R1, R2	3F
Subtract Unnormalized (Short)	SU	RX	R1, D2(X2, B2)	7F

READER'S COMMENTS

Programmer's Introduction to the IBM System/360 Architecture, Instructions, and Assembler Language (C20-1646-1)

Your comments regarding this publication will help us improve future editions. Please comment on the usefulness and readability of the publication, suggest additions and deletions, and list specific errors and omissions.

USEFULNESS AND READABILITY

fold

fold

SUGGESTED ADDITIONS AND DELETIONS

ERRORS AND OMISSIONS (give page numbers)

fold

fold

Name______________________________

Title or Position________________________

Address_____________________________

FOLD ON TWO LINES, STAPLE AND MAIL
No Postage Necessary if Mailed in U.S.A.

C20-1646-1

fold fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
112 East Post Road
White Plains, N. Y. 10601

Attention: Technical Publications

fold fold