File No. S360(Mod.20)-21
Order No. GC26-3602-6

CPS

IBM Systems Reference Library

IBM System/360 Model 20 Card Programming Support Basic Assembler Language

This reference publication provides programmers with the information required to write programs in the Basic Assembler language of the IBM System/360 Model 20.

The Basic Assembler language provides the user with a convenient means of making full use of the operational capabilities of the Model 20. Programs written in the Basic Assembler language (source programs) are translated into machine-language by means of the Basic Assembler program.

The description of the language includes rules for writing source programs and explanations of the instructions for controlling the Basic Assembler program. In addition, this publication includes a number of tables for convenient reference and conversion. Time and storage requirements are listed in a separate section. An extensive sample program is given to illustrate Basic Assembler language programming.

The description of the card and tape versions of the Basic Assembler program is confined to the aspects that affect the planning and writing of source programs.

Readers of this publication should be thoroughly familiar with the contents of the SRL publication IBM System/360 Model 20, Functional Characteristics, Order No. GA26-5847. Titles and abstracts of other Model 20 SRL publications are contained in the publication IBM System/360 Model 20, Bibliography, Order No. GA26-3565.

Seventh Edition (May, 1969; reprinted January, 1971)

This is a reprint of GC26-3602-5 incorporating changes issued in Technical Newsletter GN33-8612, dated April 6, 1970.

This edition applies to the following program version and modification levels of IBM System/360 Model 20, Card Programming Support, Basic Assembler, and to all subsequent versions and modifications until otherwise indicated in new editions and Technical Newsletters.

Program Number	Version/Modification
360T-AS-001	3/7
360T-AS-110	2/0
360U-AS-130	2/2
360U-AS-153	2/0

Changes are continually made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest SRL Newsletter, Order No. GN20-0361, for the editions that are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Laboratory, Publications Department, P.O. Box 24, Uithoorn, Netherlands.

Prerequisite to using this publication is a thorough knowledge of IBM System/360 Model 20 machine operations, particularly storage addressing, data formats, and machine instruction formats and functions. It is assumed that the reader has experience with programming concepts and techniques or has completed basic courses of instruction in these areas.

Publications closely related to this one are:

IBM System/360 Model 20:

Functional Characteristics, Form A26-5847.

Card Programming Support, Basic Assembler (Card), Operating Procedures, Form C26-3802.

Card Programming Support, Basic Assembler (Tape), Operating Procedures, Form C24-9011.

Card Programming Support, Input/Output Control System, Form C26-3603.

Input/Output Control System for the Communications Adapter, Form C26-3606.

Input/Output Control System for the Binary Synchronous Communications Adapter, Form C33-4001.

Card Programming Support, Basic Utility Programs, Functions and Operating Procedures, Form C26-3604.

Titles and abstracts of other Model 20 SRL publications are contained in the IBM System/360 Model 20 Bibliography, Form A26-3565.

CONTENTS

Computer programs may be expressed either in machine language, in other words, language directly interpreted by the computer, or in a symbolic language, which is more meaningful to the programmer. The symbolic language, however, must be translated into machine language before the computer can execute the program. This function is accomplished by an associated processing program.

Of the various symbolic programming languages, Assembler languages are closest to machine language in form and content.

The Basic Assembler language discussed in this manual is a symbolic programming language for the IBM System/360 Model 20. It enables the programmer to use all Model 20 machine functions, as if he were coding in Model 20 machine language.

The Basic Assembler program translates or processes programs written in Basic Assembler language into machine language for execution by the computer. The program written in the Basic Assembler language used as input to the Basic Assembler program is called the source program; the machine-language program produced as output from the Basic Assembler program is called the object program. The translation or processing procedure performed by the Basic Assembler program to produce the object program is called assembling or assembly.

Four versions of the Basic Assembler program are available:

a. Two card versions. These are two-pass programs for a Model 20 system that includes only card input/output devices. One of the versions permits the assembly of the macro instructions associated with the Input/Output Control System for the Binary Synchronous Communications Adapter (BSCA IOCS).

b. Two tape versions. These versions differ from the card versions by being one-pass programs and by using magnetic tape as an intermediate storage medium, thus reducing card-handling and assembly time.

Note: The CPS Input/Output Control System (IOCS) routines can be assembled by means of either version.

DEFINITIONS

Terms used in this publication are defined in the glossary provided in Appendix H.

BASIC ASSEMBLER LANGUAGE STATEMENTS

Program statements (source statements) written in Basic Assembler language may consist of: a name to identify the statement; a symbolic operation code (mnemonic) to identify the function the statement represents; one or more items called operands, to designate the data or storage locations used in the operation; and comments.

Programs written in Basic Assembler language may consist of up to five types of instructions: definition instructions, program linking instructions, Basic Assembler control instructions, input/output instructions (including IBM-supplied I/O macro instructions), and machine instructions. There are predefined mnemonic codes for all instructions in the Basic Assembler language.

Definition instructions are used to reserve storage, to define constants, and to equate symbols to the attributes of an expression.

Program linking instructions are used to link program sections for joint execution.

Basic Assembler control instructions are used to begin assembly, end assembly, and set the location counter.

Input/output instructions designate the units used as I/O devices, and control their operation. The use of IOCS macro instructions saves programming time because it relieves the user of having to code, test, and provide linkages to his own I/O routines.

Machine instructions direct the computer to execute certain operations. The Basic Assembler produces an equivalent internal machine instruction in the object program from each machine instruction in the source program.

BASIC ASSEMBLER LANGUAGE FEATURES

Variety in Data Representation

Decimal, hexadecimal, or character representation of machine-language binary values may be employed by the programmer in writing source statements. The programmer selects the representation best suited to his purpose.

Base Register Address Calculation

The Model 20 Basic Assembler language provides for two methods of addressing:

1. The address may be specified as a displacement plus a base register the contents of which are added to the displacement. The base register may be one of the general registers 8 through 15 or one of the pseudo base registers 0 through 3. (If a Submodel 5 is used, pseudo registers 0-7 are available. However, 0-3 are the only pseudo registers recognized in CPS programs.)

   a. When using a general register, the register contents can be controlled by the programmer.

   b. When using a pseudo base register, the register contents are assumed to be fixed (i.e., 0, 4096, 8192, and 12288). This corresponds to what is termed direct addressing in the Model 20 SRL publication Functional Characteristics, Form A26-5847.

2. The address may be specified symbolically without the use of a base register. In this case, the Basic Assembler assumes the clerical burden of computing storage locations in terms of a base address and a displacement.

Relocatability

The object programs produced by the Basic Assembler may be in a format enabling relocation from the originally assigned storage area to any other suitable area.

Program Linking

The linking facilities of the Basic Assembler language and program allow symbols to be defined in one assembly and referred to in another, thus effecting a link between separately assembled programs. This permits reference to data and/or transfer of control between programs. A discussion of linking is contained under Program Linking.

Program Listings

A listing of the source-program statements and the resulting object-program statements is produced by the Basic Assembler for each source program it assembles. The programmer can partly control the form and contents of the listing.

Error Indications

As a source program is assembled, it is analyzed for actual or potential errors in the use of the Basic Assembler language. Detected errors are indicated in the program listing.

MINIMUM SYSTEM CONFIGURATION

The minimum system configuration for assembling and executing Basic Assembler programs is as follows. The configuration applies to all versions of the program except where indicated.

Submodel 2

- An IBM 2020 Central Processing Unit, Model B2 for the normal version, or C2 for the BSCA version (4096 or 8192 bytes of main storage);
- one of the following card units:
  IBM 2560 Multi-Function Card Machine, Model A1,
  IBM 2520 Card Read-Punch, Model A1,
  IBM 2501 Card Reader, Model A1 or A2 with either an IBM 2520 Card Punch, Model A2 or A3, or an IBM 1442 Card Punch, Model 5;
- an IBM 2415 Magnetic Tape Unit, Model 1 or 4 (for the tape versions only);
- one of the following printers:
  IBM 1403 Printer, Model N1, 2, or 7,
  IBM 2203 Printer, Model A1;

Submodel 3

- an IBM 2020 Central Processing Unit, Model B3 (4096 bytes of main storage);
- an IBM 2560 Multi-Function Card Machine, Model A2;
- an IBM 2203 Printer, Model A2.

Submodel 4

- an IBM 2020 Central Processing Unit, Model B4 (4096 bytes of main storage);
- an IBM 2560 Multi-Function Card Machine, Model A2;

- an IBM 2203 Printer, Model A2.

Submodel 5

- an IBM 2020 Central Processing Unit, Model C5 (8192 bytes of main storage);
- one of the following card units: IBM 2560 Multi-Function Card Machine, Model A1, IBM 2520 Card Read Punch, Model A1, IBM 2501 Card Reader, Model A1 or A2 with either an IBM 2520 Card Punch, Model A2 or A3, or an IBM 1442 Card Punch, Model 5;
- an IBM 2415 Magnetic Tape Unit, Model 1 or 4 (for the tape versions only);
- one of the following printers: IBM 1403 Printer, Model N1, 2, or 7, IBM 2203 Printer, Model A1.

Note 1: CPS does not support main storage sizes of 24K and 32K, but CPS programs will run on Models DC5 and E5 although only 16K bytes are used. (The maximum value of the location counter is X'3FFF'. Therefore, the Basic Assembler will not permit references to addresses greater than this.)

Note 2: If 7-track tapes are used, the data-conversion feature is required.

MAXIMUM SYSTEM CONFIGURATION

Basic Assembler object programs may be produced for the following maximum system configurations.

Submodel 2

- An IBM 2020 Central Processing Unit, Model D2 (16,384 bytes of main storage); with or without IBM Binary Synchronous Communications Adapter, Feature No. 2074;
- two IBM 2311 Disk Storage Drives, Model 11 or 12 (both must be the same model);
- an IBM 2415 Magnetic Tape Unit, Model 1 through 6;
- an IBM 2501 Card Reader, Model A1 or A2;
- an IBM 1442 Card Punch, Model 5;
- one of the following card units: IBM 2520 Card Read-Punch, Model A1, IBM 2520 Card Punch, Model A2 or A3, IBM 2560 Multi-Function Card Machine, Model A1;
- one of the following printers: IBM 1403 Printer, Model N1, 2, or 7, IBM 2203 Printer, Model A1;
- one of the following magnetic character readers: IBM 1419 Magnetic Character Reader, Model 1 or 31, IBM 1259 Magnetic Character Reader, Model 1, 31, or 32;
- an IBM 2152 Printer-Keyboard.

Submodel 3

- an IBM 2020 Central Processing Unit, Model D3 (16,384 bytes of main storage);
- an IBM 2560 Multi-Function Card Machine, Model A2;
- an IBM 2203 Printer, Model A2.

Submodel 4

- an IBM 2020 Central Processing Unit, Model D4 (16,384 bytes of main storage); with or without IBM Binary Synchronous Communications Adapter, Feature No. 2074;
- two IBM 2311 Disk Storage Drives, Model 12;
- an IBM 2560 Multi-Function Card Machine, Model A2;
- an IBM 2203 Printer, Model A2;
- an IBM 2152 Printer-Keyboard.

Submodel 5

- an IBM 2020 Central Processing Unit, Model D5 (16,384 bytes of main storage); with or without IBM Binary Synchronous Communications Adapter, Feature No. 2074;
- four IBM 2311 Disk Storage Drives, Model 11 or 12;
- an IBM 2415 Magnetic Tape Unit, Model 1 through 6;
- an IBM 2501 Card Reader, Model A1 or A2;
- an IBM 1442 Card Punch, Model 5;
- one of the following card units: IBM 2520 Card Read-Punch, Model A1, IBM 2520 Card Punch, Model A2 or A3, IBM 2560 Multi-Function Card Machine, Model A1;
- one of the following printers: IBM 1403 Printer, Model N1, 2, or 7, IBM 2203 Printer, Model A1;
- one of the following magnetic character readers:

IBM 1419 Magnetic Character Reader, Model 1 or 31,
IBM 1259 Magnetic Character Reader, Model 1, 31, or 32;

• an IBM 2152 Printer-Keyboard.

Note: CPS does not support main storage sizes of 24K and 32K, but CPS programs will run on Models DC5 and E5 although only 16K bytes are used.

LANGUAGE COMPATIBILITY

The IBM System/360 Model 20 Basic Assembler language is compatible with the Basic Assembler language for the other models of the IBM System/360, except where differences in machine design make it necessary to include some instructions in the Model 20 Basic Assembler language that are not contained in the System/360 Basic Assembler language. The mnemonics of these Model 20 instructions are:

BAS
BASR
CIO
HPR
SPSW
TIOB
XIO

The use of the CIO, SPSW, TIOB, and XIO instructions in Model 20 programs can be avoided by using IOCS macro instructions to satisfy input/output requirements.

Programs that are written in the Model 20 Basic Assembler language and contain statements with blank operands cannot be assembled by other System/360 Assembler programs.

In addition, the use and the functions of registers 0 through 3 in Model 20 programming differ from the corresponding registers on other models of the IBM System/360.

CODING CONVENTIONS

Statements in Basic Assembler language can be written in free format; in other words, the statement components need not begin in a specified column of the coding sheet. (The name of a statement, which must begin in column 25, is an exception to this rule.) However, the statement components must be separated from each other by at least one blank column.

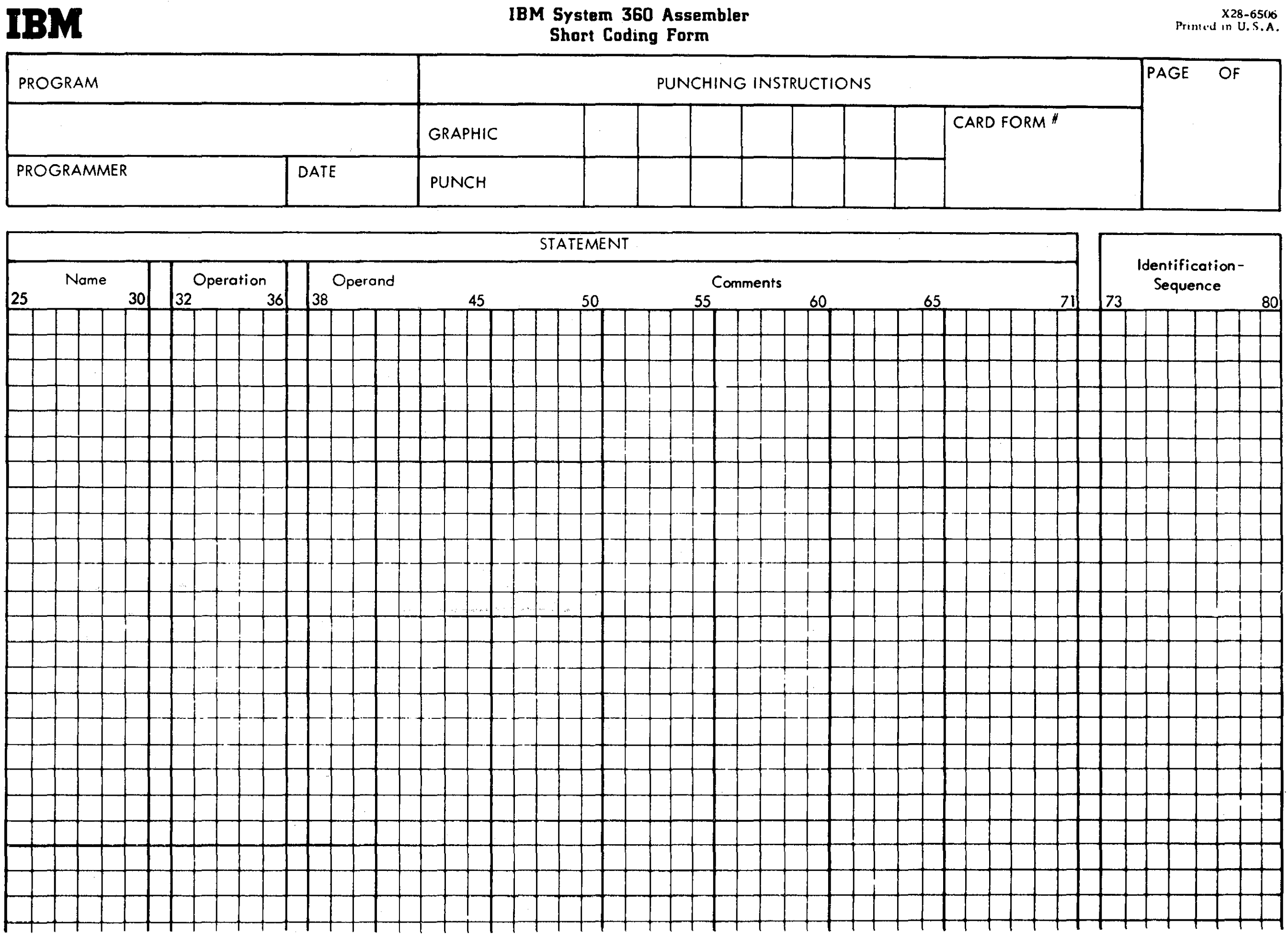
For the purpose of clarity, most programmers do not use the free format but prefer to begin each type of statement component in a specific column of the coding sheet.

The Coding Form

The coding form shown in Figure 1 is designed to satisfy this preference. This form -- the IBM System/360 Assembler Short Coding Form (No. X28-6506-2) -- contains a statement field which extends from column 25 to column 71 and is broken down into three sub-fields: the name field (cols. 25-30), the operation field (cols. 32-36), and the operand field (cols. 38-71).

The column numbers on the coding form refer to the column numbers on the cards into which the source program is to be punched.

For the purpose of alignment, each entry in one of the sub-fields should begin in the leftmost column of the sub-field. Thus, the operation entry should begin in column 32 and the operand entry should begin in column 38. (Note that the name entry must begin in column 25.) Figure 2 shows a coding form with a number of typical statements in the Basic Assembler language.



IBM

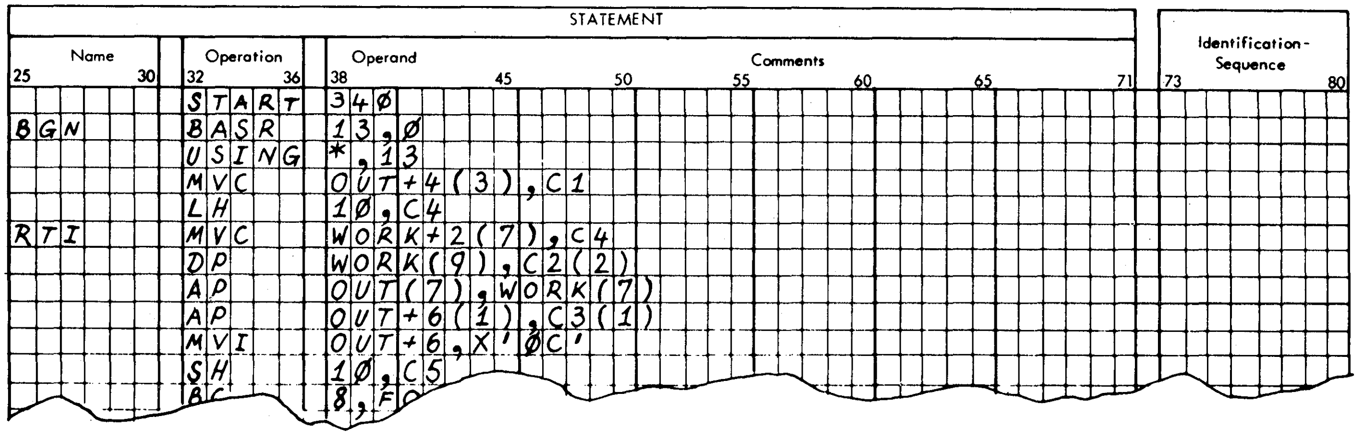
IBM System 360 Assembler
Short Coding Form

X28-6506
Printed in U.S.A.

PROGRAM	PUNCHING INSTRUCTIONS		PAGE OF
	GRAPHIC	CARD FORM #	
PROGRAMMER / DATE	PUNCH		

STATEMENT				Identification-Sequence
Name 25–30	Operation 32–36	Operand 38	Comments 45 50 55 60 65 71	73–80

Figure 1. The IBM System/360 Assembler Short Coding Form



Name	Operation	Operand
	START	34Ø
BGN	BASR	13,Ø
	USING	*,13
	MVC	OUT+4(3),C1
	LH	1Ø,C4
RTI	MVC	WORK+2(7),C4
	DP	WORK(9),C2(2)
	AP	OUT(7),WORK(7)
	AP	OUT+6(1),C3(1)
	MVI	OUT+6,X'ØC'
	SH	1Ø,C5
	BC	8,F

Figure 2. Typical Statements on a Short Coding Form

STATEMENT FORMATS AND COMPONENTS

A source program that is written in the Basic Assembler language is composed of a sequence of statements. These statements have the following format:

```
┌──────┬───────────┬────────────┬──────────┐
| Name | Operation | Operand(s) | Comments |
└──────┴───────────┴────────────┴──────────┘
|<---------Instruction---------->|
```

Each source statement is punched into a separate card. The deck of cards that contains all the statements of one source program is referred to as the source program deck.

A statement may consist of (1) an instruction only, or (2) an instruction and a comments portion. Instruction entries and comments entries are described in two separate sections below.

Instruction Entries

The instruction entry must contain an operation entry, and may contain a name and an operand entry. These three types of entry are described in the subsequent sections.

The Name Entry: The name entry consists of a symbol that is placed in the name field of the coding form to identify the associated statement. The use of such names is optional.

In the Basic Assembler language, names must conform to the following rules.

1. The first character of the name must be alphabetic.

2. The name must not be longer than four characters.

3. The name must not contain special characters or embedded blanks.

4. The name must begin in column 25 of the coding form and in column 25 of the source card.

5. The name must be separated from the operation entry by at least one blank.

Examples of valid names:

RNT1
C345
A
BGN

Examples of invalid names:

3NBR	(the first character is not alphabetic)
START	(the symbol contains more than 4 characters)
RL+8	(the symbol contains a special character)

A programming example that demonstrates the use of the name entry is shown in Figure 3.

Note 1: For all joint assemblies (i.e., whenever the programmer uses the IOCS and wishes to assemble the generated IOCS routines with his source program) user programs must not contain a name that begins with the letter I followed by three numerical characters (0-9). In addition to this, the name assigned to a file must not appear in the name field of any statement in the source program.

Note 2: User programs for joint assemblies with the BSCA Basic Assembler must not contain a name that begins with the letters ID followed by two numerical characters. User programs for both joint and separate assemblies of the BSCA Basic Assembler must not include the type codes of the BSCA macro instructions in a name field.

The Operation Entry: The operation entry consists of a mnemonic operation code that represents a machine instruction, a Basic Assembler instruction, or an IOCS macro instruction.

A mnemonic operation code consists of up to five alphabetic characters. It must be separated from the name entry and the operand entry by at least one blank column each.

To understand the terms used in this publication, a clear distinction must be made between (1) a machine instruction written in Basic Assembler language and (2) a Basic Assembler instruction.

A machine instruction written in the Basic Assembler language is an instruction to the computer. General descriptions of these instructions are contained in the section Machine Instruction Statements. Detailed descriptions of machine instructions are contained in the SRL publication IBM System/360 Model 20, Functional Characteristics, Form A26-5847.

A Basic Assembler instruction is an instruction to the Basic Assembler program. The functions of Basic Assembler instructions are summarized in Appendix A. Detailed descriptions are contained in the pertinent sections of this publication.

The IOCS macro instructions are summarized in the section Input/Output Macro Instructions. Detailed descriptions of these macro instructions are contained in the SRL publication IBM System/360 Model 20 Card Programming Support, Input/Output Control System, Form C26-3603.

The following are examples of valid operation codes:

LH	load halfword
AH	add halfword
MVC	move characters
ORG	reset location counter
TIOB	test I/O and branch

The Operand Entry: The operand entry provides the Basic Assembler program or the computer with the information required to carry out the instruction specified in the operation field.

An operand entry may consist of one or two operands. Operands are used to designate storage addresses, to specify register numbers, or to define I/O devices, immediate data, masks, and lengths of storage areas.

An operand may consist of a symbol (name), a constant, or a compound expression. Two examples of compound expressions are shown below.

X'BF'	-- defines the hexadecimal constant BF, which is equal to decimal 191.
GAMA-150	-- designates the storage address of GAMA minus 150 bytes.

Each operand entry must be separated from the associated operation entry by at least one blank column. In addition, each operand entry must be delimited by at least one blank column; i.e., any associated comments entry must be separated from the operand entry by at least one blank column.

For example, the AH instruction requests the computer to add a halfword to the contents of a register. The operand, therefore, must specify (1) the number of the register and (2) the storage address of this halfword, as shown in the sample statement

```
    AH    8,VALX
```

The above statement specifies that the value (halfword) stored at the location whose address is VALX be added to the contents of register 8.

The operand entry of the AH instruction in the above example consists of two operands: the register number 8 and the symbolic address VALX. These two operands must be separated from each other by a comma.

Note: Operand entries that consist of two operands must conform to the format

```
    operand1,operand2
```

and must not contain a blank column. When the computer encounters a blank column in an operand entry, it considers the operand entry to be terminated.

The attributes and functions of symbols and expressions that may appear in the operand field of a statement are described in a later section.

Comments Entries

The comments entry in a statement provides for the insertion of explanatory information into a program listing. Comments do not affect the assembly or the execution of a program, but they facilitate the reading and understanding of a program listing by explaining the purpose or function of a particular statement.

Any valid character, including blanks, can be used in a comment. Comments entries are punched into a statement card to the right of the operand entry and separated from it by at least one blank column. Comments entries must not extend beyond column 71.

If the desired comments entry cannot be accommodated in the space available on the right of the operand entry, or if comments consist of general information that pertains to a sequence of statements, the "comments card" can be used.

Comments cards must contain an asterisk in column 25; columns 1-24 and 26-71 are available for comments. Any number of comments cards may be inserted anywhere in a source-program deck.

Identification-Sequence Entries

The identification-sequence field (columns 73-80 of the coding form) can be used to specify identifying information and/or to provide the statements of a program with sequence numbers. Some typical identification-sequence entries are shown in the example below.

```
Example 1:    SALE0001
              SALE0002
                 .
                 .
                 .
                 .
              SALE0813
```

```
Example 2:   MAIN001
             MAIN002
                .
                .
                .
             MAIN097
             ROUT1/01
                .
                .
                .
             ROUT1/65
             MAIN098
                .
                .
             MAIN466

Example 3:   MILLER
             MILLER
                .
                .
                .
             MILLER
```

Any identification-sequence entry is printed in the program listing as it is read. Identification-sequence entries do not affect the assembly or the execution of the program.

Sample Sequence of Statements

Figure 3 shows a sample sequence of statements in the Basic Assembler language. This example illustrates the writing and the general function of the statements and their components as discussed in the preceding sections.

The comments entries in Figure 3 refer to the subsequent notes.

Note 1:

The instruction CALC SR 9,10 causes the contents of register 10 to be subtracted from the contents of register 9. When this subtraction has been completed, control is transferred to the physically next statement. (Refer to Note 2.)

Note 2:

The instruction BC 12,RES1 causes a test to determine if the contents of register 9 -- the register whose contents were changed by means of the preceding instruction -- are equal to or less than zero.

If they are, this BC instruction causes a branch to the symbolic address RES1. (Refer to Note 3.)

If the contents of register 9 are greater than zero (positive), the BC instruction causes the physically next statement (SH instruction) to be executed. (Refer to Note 4.)

Note 3:

The instruction RES1 STH 9,OUTA is executed only if the contents of register 9 were found to be less than or equal to zero (refer to Note 2).

This STH instruction causes the contents of register 9 to be transferred to an (output) area named OUTA. When this transfer has been completed, the physically next statement of the program (not shown in this example) is executed.

Note 4:

The instruction SH 9,CON2 is executed only if the contents of register 9 were found to be greater than zero (refer to Note 2).

This SH instruction causes the value stored at the symbolic address CON2 to be subtracted from the current contents of register 9. When this subtraction has been completed, the physically next statement is executed. (Refer to Note 5.)

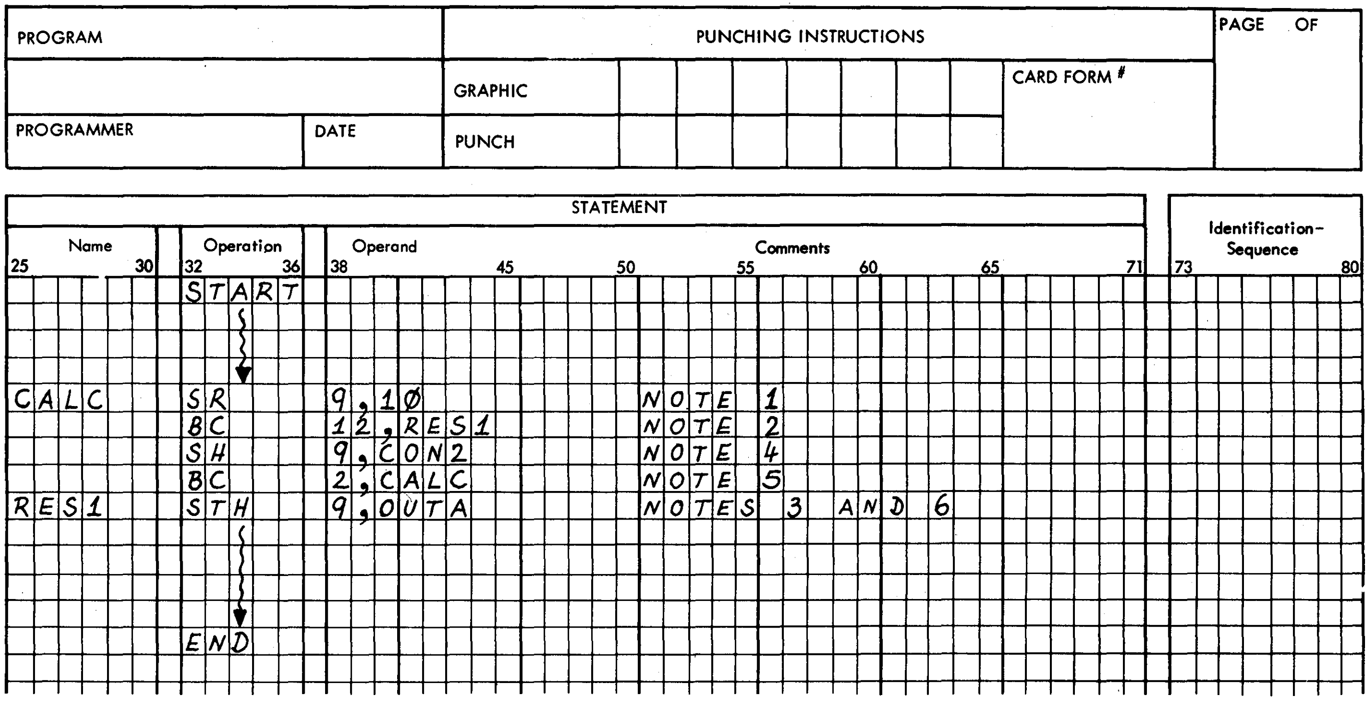
Note 5:

The instruction BC 2,CALC causes a conditional branch to the symbolic address CALC, which is the address of the SR instruction referred to in Note 1.

Note that this BC instruction is executed only if the contents of register 9 were found to be greater than zero in the test caused by the instruction BC 12,RES1.

Note 6:

The program "loops" through the statement sequence beginning with the instruction CALC SR 9,10 and ending with the instruction BC 2,CALC until the contents of register 9 are found to be less than or equal to zero. When this is the case, the instruction BC 12,RES1 causes an exit from the loop to the instruction RES1 STH 9,OUTA (refer to Note 3).



PROGRAM | PUNCHING INSTRUCTIONS | PAGE OF

GRAPHIC | CARD FORM #

PROGRAMMER | DATE | PUNCH

STATEMENT

Name (25–30)	Operation (32–36)	Operand (38)	Comments	Identification-Sequence (73–80)
	START			
CALC	SR	9,1Ø	NOTE 1	
	BC	12,RES1	NOTE 2	
	SH	9,CON2	NOTE 4	
	BC	2,CALC	NOTE 5	
RES1	STH	9,OUTA	NOTES 3 AND 6	
	END			

Figure 3. Sample Sequence of Statements

THE LANGUAGE STRUCTURE

THE CHARACTER SET

The following 44 characters can be used in statements written in the Basic Assembler language.

26 alphabetic characters:	A through Z
10 numerical characters:	0 through 9
8 special characters:	*+-,)('blank

The punch combinations that represent these characters are shown in Appendix F. However, constants and character self-defining terms may contain any of the 256 punch combinations listed in Appendix F.

SELF-DEFINING TERMS

A self-defining term is a term whose value is not assigned by the Basic Assembler program, but is inherent in the term itself. Thus, the decimal digit 3, representing the value 3, is a self-defining term.

The three types of self-defining terms are decimal, hexadecimal, and character terms. They can be used to specify immediate data, masks, registers or addresses, and constants.

Self-defining terms must not be confused with data constants, which are described in the section Definition Instructions. There is a clear distinction in the use of each: the Basic Assembler program assembles the value of a self-defining term, but it assembles the address of a data constant.

A self-defining term is considered absolute because its value is not changed on program relocation.

Decimal Self-Defining Terms

A decimal self-defining term is an unsigned decimal number with a maximum of five digits, e.g., 007, 11900, or 3. Its value must not exceed 16383. A decimal self-defining term is assembled as its binary equivalent.

Hexadecimal Self-Defining Terms

A hexadecimal self-defining term is a sequence of up to four hexadecimal digits enclosed in apostrophes and preceded by the prefix X (e.g., X'9',X'A4',X'20B3'). The highest hexadecimal self-defining term is 3FFF. This value corresponds to the maximum decimal self-defining term 16383. Each hexadecimal digit is assembled as its 4-bit binary equivalent, as shown in Figure 4.

Hexadecimal Digit	Binary Equivalent
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Figure 4. Table of Hexadecimal Self-Defining Terms

A hexadecimal-to-decimal conversion table is shown in Appendix G.

Character Self-Defining Terms

A character self-defining term consists of a single character, enclosed in apostrophes and preceded by the prefix C (e.g., C'A', C'/', C'5', C' '). Any of the 256 EBCDIC punch combinations shown in Appendix F can be used for character specification. However, ampersands and apostrophes that are to be specified as self-defining characters must be doubled within the enclosing apostrophes. Thus, a single apostrophe must be written as C'''' and a single ampersand as C'&&'.

Each character self-defining term is assembled as its 8-bit EBCDIC code equivalent (see Appendix F).

SYMBOLS

Symbols are used to refer to locations in main storage by name rather than by the actual address.

A symbol may be placed in the name field of one statement and in the operand entry of another statement. However, if a symbol is to be placed in the operand entry of a statement, it must be "defined" elsewhere in the program.

A symbol is considered "defined" when it appears

(a) in the name field of some statement within the same program, or

(b) in the operand of an EXTRN statement within the same program.

A prerequisite for defining a symbol by method (b) is that the same symbol appear in the operand entry of an ENTRY statement and in the name field of some statement in another program section. (Refer to the section Program Linking for further information about the use of EXTRN and ENTRY statements.)

Defining Symbols

The Basic Assembler program maintains an internal table -- the symbol table -- where it stores all symbols that are used as names within a program. Each symbol in the table is associated with a storage address, which is the setting of the location counter at the time the symbol is read. A program-generated length attribute and a name identification are added. The length attribute depends on the basic instruction format. The name identification indicates whether the symbol is relocatable or absolute, and whether it is external (defined in a separately assembled program section). Thus, a symbol entered in the name field of a statement is considered to be defined.

All symbols that are used as expressions, i.e., as operands of a statement, must be defined. Normally, this can be done at the most convenient position in the program. The ORG and the EQU instructions, however, require the symbols in their operands to be previously defined. Otherwise, the Basic Assembler identifies these statements in the program listing by the diagnostic message U (undefined).

Relocatable and Absolute Symbols

In general, a symbol is considered to be relocatable because relocatability is its inherent purpose. (Refer to the section Absolute and Relocatable Programming.) However, for the convenience of relating the meaning of the stored information to its symbolic address, a symbol can be equated to an absolute address by means of the Basic Assembler instruction EQU, which is described in the section EQU-Equate Symbol.

The Basic Assembler program generates the relocatable or absolute attribute of a symbol as part of the name identification. This attribute is then stored with the symbol in the symbol table.

External Symbols

Limited main storage availability may require a program to be divided into a number of sections, each of which can be assembled separately.

In one program section, the operand entry of a statement may contain a symbol that is defined in a different program section. This symbol must be introduced by an EXTRN statement into the section in which it is not defined. In the program section where the symbol is defined, it must be specified in an ENTRY statement. The Basic Assembler instructions, ENTRY and EXTRN, are described in the section Program Linking.

Restrictions on Symbols

Each symbol can represent one specific storage address only. Therefore, it must not be defined twice. The number of symbols that can be specified in a program depends on the available storage capacity, as shown in Figure 5.

Storage Capacity	Number of Symbols Allowed in the Source Program	
4096	165	--
8192	847	805*
12288	1530	1487*
16384	2213	2170*

* for the BSCA version

Figure 5. Number of Symbols versus Storage Capacity

If the number of symbols exceeds the applicable maximum, a symbol-table overflow occurs. The card versions of the Basic Assembler program require an additional assembly run to compensate for the overflow; the tape versions (after an informative halt) deal with the situation automatically. Detailed explanations are supplied in the section The Basic Assembler Program.

Relative Addressing

To avoid a symbol-table overflow, the number of symbols can be reduced by means of relative addressing.

The term relative addressing refers to the method of specifying storage locations by means of a defined symbol plus or minus a displacement, or by means of the setting of the location counter plus or minus a displacement. The following examples show some relative addresses.

```
FLDA-200    (symbol minus displacement)
*+12        (location counter plus
             displacement)
FLDB+X'F'   (symbol plus hexadecimal
             displacement)
```

Note: The asterisk (*) represents the value of the location counter after the preceding instruction has been read in, and, if required, boundary alignment has taken place.

The use of relative addressing is illustrated in the example below. In this example, statement sequence A uses different symbols to refer to five different storage locations: BGN, SYM, AUG, ADD, and SUM. In statement sequence B, these five storage locations are referred to by only two different symbols (BGN and AUG) and by three relative addresses: AUG+2 (for ADD), *+6 (for SYM), and AUG+4 (for SUM).

```
Sequence A   BGN   BASR    13,0
                   USING   *,13
                   LH      12,AUG
                   AH      12,ADD
                   BC      2,SYM
                   SR      12,12
             SYM   STH     12,SUM
                   BASR    14,15
             AUG   DS      H
             ADD   DS      H
             SUM   DS      H

Sequence B   BGN   BASR    13,0
                   USING   *,13
                   LH      12,AUG
                   AH      12,AUG+2
                   BC      2,*+6
                   SR      12,12
                   STH     12,AUG+4
                   BASR    14,15
             AUG   DS      H
                   DS      H
                   DS      H
```

The relative address AUG+2 can be used to replace the symbol ADD because the storage area referred to by ADD (see statement ADD DS H) begins directly behind the storage area AUG, which is two bytes long (see statement AUG DS H). The same applies to the replacement of the symbol SUM by the relative address AUG+4.

The branch address SYM is replaced by the relative address *+6 (current setting of the location counter plus 6 bytes). This relative address causes a branch to the location six bytes beyond the BC instruction; in other words, to the first byte of the instruction STH 12,AUG+4.

EXPRESSIONS

An expression is any symbol or self-defining term, relocatable or absolute, used in the operand entry of a statement.

Compound Expressions

An expression that consists of more than one symbol or self-defining term and connected by plus or minus signs is referred to as a compound expression.

Examples:

```
BETA-10+200
FLD+X'2D'
*-GAMA+200
```

Restrictions. The Basic Assembler program considers an expression to be terminated by a blank or a comma, depending on the type of expression. An expression must not

- begin with a plus or minus sign,
- comprise more than three symbols and/or self-defining terms,
- have a negative value at object time (if it is absolute),
- contain another relocatable symbol if an external symbol is part of the expression,
- contain any self-defining term with a value >4095 if used as operand of a machine instruction, and
- exceed 16383 (decimal).

Evaluation of Expressions

The Basic Assembler replaces symbolic expressions with their numerical equivalents by evaluating compound expressions, executing arithmetic calculations, and inserting the results into the instruction.

Absolute Expressions

An expression is considered absolute if it contains

(1) only self-defining terms and/or absolute symbols, or

(2) one positive and one negative relocatable symbol.

Some examples of absolute expressions are shown below. (The symbols PHS1 and PHS2 are considered relocatable.)

```
2510               PHS2-PHS1
PHS2+2510-PHS1     2510-PHS2+PHS1
```

Relocatable Expressions

The value of a relocatable expression is changed by the Basic Assembler program on program relocation, in other words, the relocation factor is applied to its numerical equivalent to compute the new storage address.

Relocatable expressions must conform to the following rules:

- A relocatable expression must contain either one or three relocatable symbols.
- If a relocatable expression contains three relocatable symbols, one and only one of these symbols must be preceded by a minus sign.
- If a relocatable expression contains only one relocatable symbol, this symbol must not be negative.

Some examples of valid relocatable expressions are shown below. (R stands for "relocatable symbol".)

```
R+1, R-8, R-R+R, *-X'D0'
```

The following examples show some invalid relocatable expressions.

R+R	(contains two relocatable symbols)
R+R+R	(one of the relocatable symbols should be negative)
16-R	(the relocatable symbol must not be negative)
R-R-R	(two negative relocatable symbols)

LOCATION COUNTER

The Basic Assembler program uses a counter to record the address assigned to each statement read into main storage. This counter is referred to as the location counter.

At assembly time, as soon as an instruction statement has been read into main storage, and, if required, boundary alignment has taken place, the location counter is incremented by the number of bytes occupied by that statement. The location counter then indicates the next available storage location.

```
LOC.|                                   | SOURCE STATEMENTS                                             |OBJ.
CTR |  OBJECT CODE                      |                                                               |CRD
0154                                     INDA   START 340                                      STMT01  001
0154    0DD0                                    BASR  13,0              LOAD BASE REG.         STMT02  002
0156                                            USING *,13              ASSIGN BASE RE.        STMT03  002
0156    47F0 D048                               BC    15,CALC           CIRCLE THE CONST       STMT04  002
000A                                     R10    EQU   10                                       STMT05  002
015A                                     PRT    DS    CL17                                     STMT06  002
016B    0000 0000 0000 0000 00           WORK   DC    XL9'0'                                   STMT07  003
0174    0000 0000 0000 00                ACCU   DC    XL7'0'                                   STMT08  003
017B    2400 0C                          CPTL   DC    X'24000C'                                STMT09  003
017E    025C                             RATE   DC    X'025C'                                  STMT10  003
0180    0000 0000 0000 5C                ROUN   DC    X'00000000000005C'                       STMT11  003
0188    0152                             CNT    DC    H'338'                                   STMT12  003
018A    0001                             DECR   DC    H'1'                                     STMT13  003
018C    4020 6B20 2020 6B20 2020 6B      MASK   DC    X'40206B2020206B2020206B'                STMT14  003
0197    2020 214B 2020                          DC    X'2020214B2020'                          STMT15  003
019E    48A0 D032                        CALC   LH    R10,CNT           LOAD COUNT             STMT16  003
01A2    D202 D022 D025                          MVC   ACCU+4(3),CPTL    LOAD ACCU              STMT17  003
01A8    D206 D017 D01E                   LOOP   MVC   WORK+2(7),ACCU    LOAD WORK              STMT18  004
01AE    FD81 D015 D028                          DP    WORK,RATE         COMPUTE INTEREST       STMT19  004
01B4    FA66 D01E D015                          AP    ACCU,WORK(7)      INCREMENT CAPITAL      STMT20  004
01BA    FA66 D01E D02A                          AP    ACCU,ROUN         ROUND DECIMAL          STMT21  004
01C0    920C D024                               MVI   ACCU+6,X'0C'      RESTORE LAST DIGIT     STMT22  004
01C4    4BA0 D034                               SH    R10,DECR          DECRFASE COUNT         STMT23  004
01C8    4720 D052                               BC    2,LOOP            TEST FOR COMPLETION    STMT24  004
01CC    D210 D004 D036                          MVC   PRT,MASK          MASK TO PRINT AREA     STMT25  004
01D2    DE10 D004 D01E                          ED    PRT,ACCU          EDIT RESULT            STMT26  004
01D8    D040 D004 0011                   FINE   XIO   PRT(X'40'),17     PRINT RESULT           STMT27  004
01DE    4710 D0A0                               BC    1,PERR            TEST PRINTER NOT OK    STMT28  005
01E2    4740 D082                               BC    4,FINE            TEST PRINTER WORKNG    STMT29  005
01E6    9A40 D090                               TIOB  *,X'40'           TEST END OF I/O        STMT30  005
01EA    9A41 D0A0                               TIOB  PERR,X'41'        TEST PRINTER ERROR     STMT31  005
01EE    9900 0999                        HALT   HPR   X'999',0          DISPLAY 999            STMT32  005
01F2    47F0 D098                               BC    15,HALT           LOCK RESTART           STMT33  005
01F6    9900 0111                        PERR   HPR   X'111',0          DISPLAY 111            STMT34  005
01FA    47F0 D082                               BC    15,FINE           REPEAT PRINT           STMT35  005
0154                                            END   INDA                                     STMT36  006
```

Figure 6A. Assignment of Storage Addresses

Location Counter Setting		Instruction	Length	Statement
In Hex	In Decimal			
0154	340	START	none	01
0154	340	BASR	2 bytes	02
0156	342	USING	none	03
0156	342	BC	4 bytes	04
015A	346	EQU	none	05
015A	346	DS	17 bytes	06
016B	363	DC	9 bytes	07
0174	372	|	|	
|	|	|	|	
|	|	|	|	
V	V	V	V	
019E	414	LH	4 bytes	16
01A2	418	MVC	6 bytes	17
01A8	424	MVC	6 bytes	18
01AE	430	DP	6 bytes	19
01B4	436	|		
		|		
		V		
		etc.		

Figure 6B. Assignment of Storage Addresses

ASSIGNED ADDRESSES

If a printer is attached to the Model 20 during the assembly of a source program, a program listing is produced, as shown in Figure 6A. The listing includes all statements translated into machine language. To the left of the machine-language statements, the listing contains the address assigned to each statement; i.e., the current setting of the location counter at the time the statement is read into main storage.

In the example in Figure 6B, the location counter is initially set to 340, which is the address of the next sequential storage location. The next program statement, the BASR instruction, is stored beginning at location 340. Since two bytes are required for the BASR instruction, the location counter is incremented to 342. Then follows the USING statement, which does not require any storage space. Therefore, the address 342 is assigned to the BC instruction that follows the USING statement. After storing the BC statement, which requires 4 bytes, the location counter points to storage address 346. This procedure is continued until the entire program is assembled.

Location-Counter Overflow

The location-counter setting is limited to the storage capacity specified in the control card. The control card is described in the SRL publications IBM System/360 Model 20 Card Programming Support, Basic Assembler, Operating Procedures, Forms C26-3802 and C24-9011 for the card or tape versions, respectively.

If, for example, the specification in the control card is 4096 bytes and the program to be assembled exceeds this capacity, the location counter is reset to 0 at the point where the specified storage capacity is exceeded -- even if the storage capacity that is actually available is greater than 4K. The respective statement is identified by an error message (L).

The largest number the location counter can accommodate is $2^{14}-1$ or, in hexadecimal notation, 3FFF. The leftmost digits of any value greater than 3FFF are truncated.

Reference to the Location Counter

At any point in the source program, the programmer may refer to the current setting of the location counter by using an asterisk in the operand entry. The example in Figure 7 illustrates a print routine, which includes a method of stopping the processing flow until the execution of a previously initiated output operation has been completed.

The instruction TIOB *,X'40' tests to determine if the attached 1403 printer is still busy with the execution of the last print command. The second operand (X'40') specifies the unit and the function. The first operand specifies the address to which the program is to branch if the printer is busy.

During the assembly of this instruction, the Basic Assembler program replaces the asterisk by the actual branch address, which is the current setting of the location counter, 1078. During execution, the program repeatedly branches to the same instruction until the printer is no longer busy and sequential processing of the subsequent instructions can continue.

Note: The same effect can be obtained by the insertion of a symbol in the operand entry that is also inserted in the name field:

```
    TEST    TIOB    TEST,X'40'
```

The symbol TEST, as a branch address, also repeatedly refers the program to the same statement until the busy condition no longer exists.

Location Counter	Name	Operation	Operand
1060	HALT	HPR	X'99',0
1064	FIN	XIO	PRT(X'40'),17
1070		BC	1,HALT
1074		BC	4,FIN
1078		TIOB	*,X'40'
1082		TIOB	HALT,X'41'

Figure 7. Use of an Asterisk in the Operand Entry of a Statement

Resetting the Location Counter

The Basic Assembler instruction ORG can be used to reset the location counter to any desired value. This is described in the section ORG -- Resetting the Location Counter.

STORAGE ADDRESSES

A storage address is the address of the leftmost byte of the area referred to. The length of an addressed area is either explicitly stated in the operand entry, or is implied in the constant by which the addressed area has been defined. Registers are fixed length areas and are, therefore, exempt from this rule.

The two ways of specifying storage addresses in a program written in Basic Assembler language are:

1. effective addressing, allowing for symbolic (or implied) addressing and explicit addressing; and

2. absolute (or direct) addressing.

An address is generated as a storage field of 16 bits. The four high-order bits (the B-field) indicate the base register. The twelve low-order bits (the D-field) indicate the displacement, which is the difference (in bytes) between the contents of the base register (or the address represented by a symbol) and the referenced storage location. D1(B1) and D2(B2) designate addresses that are part of the first and the second operand, respectively.

Addition of the contents of the base register to the displacement gives the actual address of a location in main storage. (Refer to the section Base Registers.)

EFFECTIVE ADDRESSING

Effective addresses are identified by a 1-bit in the leftmost position of the B-field, which signals that at least one of the general registers 8 through 15 must be used as a base register. At assembly time, the address of a location in main storage is split into two parts,

a) a fixed value contained in the base register, and

b) a displacement, which is the difference between the actual storage address and the contents of the base register.

At object time, the contents of the general register specified by the B-field of an address are added to the contents of the D-field to form the actual address in main storage.

SYMBOLIC (IMPLIED) ADDRESSING

Symbolic (or implied) addressing is used when a symbol is given in the operand entry of a statement, rather than the explicit specification of a base register and a displacement. The equivalent value of the symbol is assigned by the location counter. The symbol must be defined elsewhere in the program. (Refer to the section Symbols.)

When the Basic Assembler program encounters a symbol during assembly, it scans the symbol table, finds the associated address, and assembles this address into the instruction. If the operand consists of a compound expression, such as ALFA-BETA+GAMA, the address equivalents of all symbols are looked up, the arithmetic operations are executed, and the result is assembled into the instruction.

The computed address integer is not stored as it is, but is first split into a base register and a displacement. This is explained in the following example.

If the address equivalent of the abovementioned compound expression (ALFA-BETA+GAMA) were 6319, the Basic Assembler would split this address by selecting a base register containing the closest value to 6319. For example, if the three base registers 9, 10, and 11 were used and contained the values 4000, 5000, and 7000, respectively, the Basic Assembler would select register 10, because this register would cause the smallest displacement, which is the difference between the actual storage address and the contents of the base register selected by the Basic Assembler for address generation. Thus, the displacement resulting from the splitting of 6319 is 1319. The address 6319, assembled into the instruction, therefore, has the following format: 1319 (10); or A527 in hexadecimal notation, as it is printed on the program listing produced during the assembly. "A" represents the base register and "527" represents the displacement, 1319.

A displacement calculated by the Basic Assembler cannot be greater than 4095. For the calculation of addresses higher than 4095, additional base registers must be used.

The rules followed by the Basic Assembler in the selection of a suitable base register are as follows:

1. If more than one register would produce a valid displacement (not exceeding 4095), the Basic Assembler uses the register that produces the smallest displacement.

2. If two or more registers produce the same displacement, the Basic Assembler uses the highest-numbered register.

3. If none of the specified registers produces a valid displacement, the address field in the instruction that contains the invalid operand is set to zero. An appropriate error message appears in the program listing.

The advantages of symbolic addressing are the simplicity of the method itself and the resulting relocatability of the program.

EXPLICIT ADDRESSING

Explicit addressing requires the specification of a base register and a displacement in the operand entry of a statement.

Example: MVI 800(8),X'A'

The above statement causes the immediate data (X'A') to be stored in the location identified by D1=800 and B1=8.

Indexing

Explicit addressing provides a special technique of address modification, called indexing. Using the indexing method, the programmer can conveniently deal with a storage area step by step.

Assume that a table of 100 integers, each of which is 5 bytes long, is contained in main storage. These integers are to be transferred one-by-one to the output area OUTA.

```
      START  350
BGN   BASR   9,0
      USING  *,9
       .
       .
       .
      LH     10,TADR
      AH     10,TLEN
      STH    10,TLIM
      LH     8,TADR
LOOP  MVC    OUTA(5),0(8)
      XIO    OUTA(X'40'),5
      AH     8,INCR
      CH     8,TLIM
      BC     12,LOOP
      HPR    X'99',0
       .
       .
       .
TLIM  DS     H
TLEN  DC     H'495'
INCR  DC     H'5'
TADR  DC     Y(TAB)
TAB   DS     100CL5
OUTA  DS     100CL5
      END    BGN
```

In the above routine, register 9 is used as a base register. The maximum table address (TAB+495) is computed in register 10 and then stored at location TLIM. Register 8 is loaded with the address of the first table entry (see the section Address Constants). The expression 0(8) thus designates the first table entry (TAB), which is moved to OUTA.

The data stored in OUTA is printed. (For simplicity, the necessary edit and test routines are omitted.) The subsequent instruction is used to increase by five the contents of register 8, causing the address 0(8) to point to the position of the second table entry (TAB+5). The contents of register 8 are then compared with the maximum table address at location TLIM. If the value in register 8 is lower than, or equal to, the compared value in TLIM, the program branches to LOOP to fetch another table argument. Otherwise the program halts.

Normally, if base registers are used for address generation, a symbol in the operand entry of a statement should not be accompanied by an explicit base register designation. It is possible, however, to specify a symbolic address accompanied by an explicit base register designation, instead of using the expression 0(8) in the previous example. If TAB(8) is given as the second operand of the MVC instruction, the address is computed by adding the (normal) displacement value of TAB to the contents of register 8. The statement is flagged with a warning message.

In the previous example the instructions

```
      LH     8,TADR
LOOP  MVC    OUTA(5),0(8)
```

may be replaced by

```
      SR     8,8
      AR     8,9
LOOP  MVC    OUTA(5),TAB(8)
```

Note: In the statements following this MVC statement, the program again uses the base register that was originally designated.

This program can be simplified further if absolute addressing is used. In the following example pseudo register 0 is used as a base register.

```
      START  350
BGN   USING  *,0
        .
        .
        .
      SR     8,8
LOOP  MVC    OUTA(5),TAB(8)
      XIO    OUTA(X'40'),5
      AH     8,INCR
      CH     8,TLIM
      BC     12,LOOP
      HPR    X'99',0
        .
        .
        .
TLIM  DC     H'495'
INCR  DC     H'5'
TAB   DS     100CL5
OUTA  DS     100CL5
      END    BGN
```

Register 8 is initially set to zero. Thus, TAB(8) refers to the first table entry. When the last MVC instruction has been executed, register 8 contains the value 500 and the program halts.

ABSOLUTE (DIRECT) ADDRESSING

Absolute addresses are identified by a zero in the leftmost bit position of the B-field. In absolute addressing, the 14 low-order bits of the combined B and D-field represent the complete address value and refer directly to byte locations in main storage. Absolute addresses are specified by decimal integers or absolute symbols in the operand entry of a statement.

Example:

Name	Operation	Operand
	STH	13,2440

The above statement causes the contents of register 13 to be stored in position 2440 of main storage.

Absolute addresses are also split into base register and displacement by the Basic Assembler program, as described in the section Effective Addressing. This addressing method, however, requires the specification of pseudo-registers to be used as base registers. A program that contains absolute addresses is not relocatable.

GENERAL AND PSEUDO-REGISTERS

General Registers

The Model 20 uses eight auxiliary storage units which are referred to as general registers. Each of these general registers has a length of one halfword (two bytes). The general registers are numbered from 8 to 15 and are used for temporary storage of information during execution of indexing, fixed-point arithmetic, address generation, and logical operations.

Information that requires the use of registers can be transferred

(1) from register to register,

(2) from register to main storage, or

(3) from main storage to register.

The direction of the information flow is implied in the machine-instruction format. (Refer to the section Machine Instruction Statements.)

When general registers are used for addressing, they are referred to as base registers. Base registers are assigned by a USING statement, as explained in the section Base Registers.

An advantage of using general registers for fixed-point arithmetic is that data need not be packed prior to computation. All calculations are executed in binary form.

Examples of the use of general registers:

AR 9,10	The contents of register 10 are added to the contents of register 9. The result is contained in register 9.
LH 12,AREA	The first 2 bytes of the field AREA are loaded into register 12. (Note that in this case the field AREA must be aligned at a halfword boundary.)
STH 13,OUTA	The contents of register 13 are stored in the field OUTA. (Note that in this case the field OUTA must be aligned at a halfword boundary.)

Restriction

When using the IOCS, the following restrictions on general registers apply.

- Register 15 must not be used by the programmer at any time.

- Register 14 is available only for restricted use, since its contents are changed each time a macro instruction is executed.

- Registers 11-15 are used by the 1419 IOCS.

Pseudo-Registers

In addition to the eight general registers there are four pseudo-registers numbered 0 to 3. (If a Submodel 5 is used, pseudo registers 0-7 are available. However, 0-3 are the only pseudo registers recognized in CPS programs.) The pseudo-registers are assumed to have the following permanent contents:

Register	Assumed Contents
0	0
1	4096
2	8192
3	12288

The pseudo-registers may be used only for storage addressing, i.e., as base registers. The advantage, in comparison to the use of general registers, is that pseudo-registers need not be loaded with a base address. Thus, program execution is faster and the general registers are available for other purposes. However, pseudo-registers can be used only for the specification of absolute addresses. Additional information is given in the section Absolute Addressing.

BASE REGISTERS

Base registers are general registers that are used for addressing main storage locations. The contents of a base register are subtracted from each storage address during program assembly; the remainder is referred to as the displacement. The base-register number, together with the displacement, is assembled into the instruction.

At least one general register must be assigned as a base register at the beginning of a relocatable program. In addition, this register must be loaded with the desired base address, which is normally the start address of the program.

USING -- USE BASE REGISTER

The USING statement is used to assign base registers. It also informs the Basic Assembler program of the anticipated contents of the respective base registers.

Example:

Name	Operation	Operand
	USING	*,11

The above statement designates register 11 as a base register and informs the Basic Assembler program that it may expect register 11 to contain the current value of the location counter.

Note: A name entry is not used. If a symbol appears in the name field of the USING statement, it is disregarded by the Basic Assembler program -- if it conforms to symbol specifications. Otherwise, it is identified by a diagnostic message in the program listing.

All registers that are assigned by means of USING statements must be loaded. This can be achieved by means of BASR instructions.

BASR -- BRANCH AND STORE REGISTER

The BASR (Branch and Store Register) instruction causes bits 16 to 31 of the Program Status Word (PSW) to be stored in the register defined in the first operand. Since bits 16 to 31 of the PSW contain the address of the next sequential instruction, this address is loaded into the specified register. Then the program branches to the address contained in the register specified in the second operand. The branch address is determined prior to the storing of bits 16 to 31 of the PSW.

For example, the statement BASR 12,12 causes register 12 to be loaded with the current value of the location counter. This is followed by a branch to the address previously contained in register 12.

Thus, in the above USING-statement example, register 11 can be loaded as follows:

Name	Operation	Operand
	BASR	11,0

Register 11 now contains the address of the next storage location; that is, the current value of the location counter at assembly time. The second operand, which normally specifies the register that contains the branch address, prevents branching because it refers to register 0. Accordingly, the first instructions of a program may be the following:

```
       START   356
       BASR    11,0
       USING   *,11
BGN    .............
```

The largest displacement that can be calculated by the Basic Assembler is 4095. Therefore, an additional base register mustbe assigned for each additional 4096 bytes of main storage required.

Additional base registers may be specified also for other programming purposes, such as creating defined areas (dummy sections) in main storage where certain program subroutines can be executed or where intermediate data is stored. However, if several base registers are specified by subsequent USING statements, an adequate method of loading these base registers must be found.

Figure 8 illustrates one such method.

Location-Counter Reference	Name	Operation	Operand
1000		START	1000
1000		BASR	11,0
1002		USING	*,11
1002		BC	15,PRGM
1006		USING	*+4098-6,12
1006	ALFA	DC	Y(*+4098-6)
1008		USING	*+6192-8,13
1008	BETA	DC	Y(*+6192-8)
1010		USING	*+4500-10,14
1010	GAMA	DC	Y(*+4500-10)
	PRGM	LH	12,ALFA
1016		LH	13,BETA
1020		LH	14,GAMA

Figure 8. Example of Loading Base Registers

Explanation: The following base registers are assigned by USING statements: 11,12, 13, and 14. In this example, the base registers are loaded with the following base addresses.

Register 11 --	1002
Register 12 --	5098
Register 13 --	7192
Register 14 --	5500

Base register 12 is assigned and loaded to deal with addresses higher than the maximum address the Basic Assembler can generate by using base register 11, which is

4095(11) = 4095 + 1002 = 5097.

The next higher address is generated as

0(12) = 0 + 5098 = 5098

Base register 11 is loaded when the BASR instruction is executed. Note that 1002 is the address of the first machine instruction after the BASR statement.

To load registers 12 to 14, the desired addresses are supplied to the Basic Assembler by means of address constants, which are then loaded into the respective register by subsequent LH instructions. Since the location counter is being referred to, the addresses specified by address constants are incremented first by the start address of the program (1000), and then by the length of each instruction. Therefore, the accumulated instruction lengths must be subtracted when the address constants are set up. The expressions contained within the parentheses of the address constants can also be used in the first operand of the respective USING statement.

Accordingly, the address constants have the following values:

```
ALFA = 1006 + 4098 - 6 = 5098
BETA = 1008 + 6192 - 8 = 7192
GAMA = 1010 + 4500 - 10= 5500
```

The contents of a base register can be altered whenever required; but the Basic Assembler program must be informed of the change by means of a USING statement.

Example:

Name	Operation	Operand
	USING	ALFA,9
	|	
	|	
	|	
	V	
	USING	ALFA+1000,9

To use absolute addressing, a pseudo-register must be specified in the second operand of the USING statement. In addition, the first operand must be an asterisk; otherwise, the USING statement will be identified by a diagnostic message in the program listing.

The pseudo-registers need not be loaded. They are assumed to contain at any time the values described in the section Pseudo-Registers.

```
The statements:  START 0
                 USING *,0
                 USING *+4096,1
                 USING *+8192,2
                 USING *+12288,3
                 ORG   *+316
```

inform the Basic Assembler program that pseudo-registers 0 through 3, the contents of which are 0, 4096, 8192 and 12288 are to be used as base registers.

For example, in this case storage address 3091 is split into displacement C13(hexadecimal equivalent for 3091) and base register 0, and assembled as 0C13. In like manner, storage address 6000 is assembled as 1770, address 10000 as 2710, and address 16000 as 3E80.

A program cannot be relocated if pseudo-registers are used as base registers. This disadvantage, however, may be outweighed by having all the general registers available for other purposes.

DROP -- RELEASE BASE REGISTER

If a general register has been assigned the functions of a base register, it cannot be used for other programming purposes unless the programmer cancels the assignment. This can be done by means of a DROP statement.

Example:

Name	Operation	Operand
	USING	ADDR,11
	|	
	|	
	|	
	|	
	V	
	DROP	11

After the DROP statement in the above example, register 11 can be used as an index register, an accumulator for arithmetic operations, etc. A name entry is not used in the DROP statement. If a name is specified, it is disregarded by the Basic Assembler program -- if it conforms to symbol specifications. Otherwise, the statement is identified by a diagnostic message in the program listing.

ABSOLUTE AND RELOCATABLE PROGRAMMING

A program is relocatable if it fulfills the following conditions:

1. It must contain all of the loader information produced by the Basic Assembler program (i.e., the punching of ESD and RLD cards must not be suppressed during the assembly of such programs).
2. At least one of the general registers 8 to 15 must be used for address generation.
3. It must not contain absolute expressions to refer to areas that are to be relocated.

A program is absolute if at least one of pseudo-registers 0 to 3 is specified and used for address generation throughout the program.

Absolute programming has the advantage of saving general registers for programming purposes other than address generation. In addition, the Basic Assembler program is not required to split the specified absolute addresses if pseudo-register 0 is specified in an appropriate USING statement. Absolute programming does not restrict the application of symbolic addressing.

Absolute programming must not be used under the following conditions:

1. If (1) the IOCS is used, and (2) the source program and the symbolic IOCS routines are to be assembled separately.
2. If subsequent parts of a program are loaded and executed together. In this case, only the program loaded first may be absolute.

Extensive programs that exceed the available main storage capacity must be subdivided into sections that are assembled separately.

Since the Basic Assembler program is no longer required during object program execution, storage availability is increased, which may allow the loading and simultaneous execution of more than one object program.

Two jointly executed program sections may contain the same symbols, provided these symbols are defined in only one of the two programs. In addition, these two program sections must be linked together by means of EXTRN and ENTRY statements. These statements are described below.

The EXTRN Statement

For the joint execution of two programs (A and B), EXTRN statements must be used in program B to introduce symbols that are used in program B but defined in program A.

Example:

```
|Name|Operation|Operand                    |
|----+---------+---------------------------|
|    |EXTRN    |F1                         |
```

The EXTRN statement in the above example introduces F1 as a symbol that is defined in another program section.

A name entry is not used in the EXTRN statement. If a symbol is entered in the name field, it is disregarded by the Basic Assembler program -- provided it conforms to symbol specifications. Otherwise, it is identified by a diagnostic message in the program listing.

Only one operand -- a relocatable symbol -- may be specified in an EXTRN statement. Each additional external symbol must be introduced by an additional EXTRN statement.

If an external symbol is to be used, the following action is required:

1. An address constant must be created for the external symbol.

2. The address constant must be loaded into a general register.

3. The external symbol must be referred to in the program by means of the above general register.

The maximum number of EXTRN statements to be used within one program sequence is 14. Symbols contained in statements in excess of this number are indicated as undefined in the program listing.

An EXTRN statement must immediately follow a START statement, an ENTRY statement, or another EXTRN statement. If an EXTRN statement is incorrectly placed, it is identified by a warning message. If it contains an incorrect operand, it is identified by an error message. In either case, the statement is not used.

The ENTRY Statement

An EXTRN statement in program B requires an ENTRY statement with the same operand in program A, where the appropriate symbol is defined.

Example:

```
|Name|Operation|Operand                    |
|----+---------+---------------------------|
|PRGA|START    |2000                       |
|    |ENTRY    |F1                         |
|    |  |      |                           |
|    |  |      |                           |
|    |  |      |                           |
|    |  V      |                           |
|F1  |DC       |XL2'F0F0'                  |
|    |END      |PRGA                       |
```

The above ENTRY statement permits program B, which has been loaded and stored behind program A, to use the contents of the field F1.

The Basic Assembler ENTRY statement follows the same syntax rules as the EXTRN statement. The START statement of a program can also be used instead of an ENTRY statement; that is, program names need not be introduced as linkage symbols by ENTRY statements.

The order in which independently assembled programs are loaded determines the extent of their linkability by means of the relocatable program loader. Programs containing the entry points must be loaded ahead of the programs containing the corresponding external links.

```
                  MAIN PROGRAM                       |                 SUBROUTINE
Name  | Operation | Operand                          | Name  | Operation | Operand
CRDT  | START     |                   ┌──────────────┼─>CVB  | START     | 1000
      | ENTRY     | F1                |              |       | EXTRN     | F1
      | EXTRN     | CVB               |              |       | BASR      | 11,0
      | BASR      | 8,0               |              |       | USING     | *,11
      | USING     | *,8               |              |       | MVC       | WAN,KOO
      | LH        | 12,YCVB           |              |       | LH        | 10,YF1
GET   | XIO       | INPT(X'12'),80    |              |       | MVN       | WAN+1(1),0(10)
      |   |       |                   |              |       |   |       |
      |   |       |                   |              |       |   |       |
      |   |       |                   |              |       |   |       |
      |   V       |                   |              |       |   V       |
      | MVC       | F1,INPT+10        |              |       | AH        | 13,WAN
      | BASR      | 9,12 <────────────┘          ┌───┼───────┼──BCR      | 15,9
      | SR        | 13,14<───────────────────────┘   | WAN   | DS        | H
      |   |       |                                  | KOO   | DC        | H'0'
      |   |       |                                  |       |           |
      |   |       |                                  |       |           |
      |   V       |                                  |       |           |
      | BC        | 15,GET                           | YF1   | DC        | Y(F1)
F1    | DC        | C'00'                            |       | END       | CVB
YCVB  | DC        | Y(CVB)                           |       |           |
      | END       | CRDT                             |       |           |
```

Figure 9. Sample of Program Linkage

However, a program may refer to the names of programs loaded subsequently, by means of the Include Segment (ICS) card of the Relocatable-Program Loader. This is described in the SRL publication IBM System/360 Model 20 Card Programming Support, Basic Utility Programs, Functions and Operating Procedures, Form C26-3604.

SAMPLE PROGRAM

A sample program that illustrates program linking is shown in Figure 9.

The main program in Figure 9 is assumed to deal with data in binary form. Since the data obtained by means of the XIO statement is in unpacked decimal form, the subroutine is used to convert the data into binary. To achieve this, the main program must be loaded first, using the Relocatable Program Loader, including an ICS card to allow reference to the subroutine which is loaded after the main program. (The two programs in this example are considered to be separately assembled.)

Program linkage is achieved as follows. Through the ICS card, the loader reserves a storage area for the subsequent program while loading the main program. The address of the reserved area is loaded into register 12 during execution of the main program to allow branching to CVB, which the EXTRN statement declares to be an externally defined symbol.

An ENTRY statement in the subroutine is not required for CVB because the START statement, in this case, serves the same purpose. During execution of the main program, the data that is read from cards (XIO instruction) is stored in the field INPT. For conversion into binary form, the applicable data section is moved into F1. Then the program branches into the subroutine (BASR instruction).

The contents of F1 are available to the subroutine because F1 is declared to be an external symbol by the EXTRN statement, and an entry is provided by an appropriate statement in the main program. In addition, the address of F1 is loaded into register 10 during execution of the subroutine. Explicit addressing with base register 10 and a displacement of 0 (MVN instruction) enables the subroutine to make use of the required data.

The contents of F1 are processed until the final step (AH instruction) results in a binary value contained in register 13. Then, a branch back to the main program is performed (BCR 15,9) and the binary value in register 13 is at the disposal of the main program.

EQU -- EQUATE SYMBOL

The Basic Assembler instruction EQU is used to equate a symbol to the attributes of an expression.

The EQU statement consists of (1) the name entry, (2) the operation code EQU, and (3) an expression as an operand. All symbols appearing in the operand of the EQU statement must have been previously defined.

Example:

Name	Operation	Operand
REG5	EQU	5

The symbol REG5 is equated to the absolute value 5 and thus becomes absolute. To the Basic Assembler program, it is of no further significance whether REG5 or the value 5 is specified in the operand of a statement elsewhere in the program.

To reduce programming time, symbols can be equated to frequently used compound expressions, as shown in the following example:

Name	Operation	Operand
CALC	EQU	A-B+C

DC -- DEFINE CONSTANT

Constants are data supplied to the program by the Basic Assembler statement DC (define constant).

The object program refers to these constants by their symbolic addresses, i.e., each DC statement is normally identified by a symbol that points towards the storage location of the constant. A DC statement may have only one operand which has the following components:

```
Type     Length     Constant
        Modifier
```

The type is written as a single letter, C, X, H, or Y. The length modifier is written as a decimal integer, preceded by the letter L. It must not be specified for H and Y-type constants.

The four types of constants are shown in Figure 10.

The length of a constant must not exceed 16 bytes including the bytes skipped for boundary alignment. Constants exceeding these lengths must be defined by subsequent DC statements. For example, the character constant C'THIS PARAMETER COMBINATION IS INVALID' should be defined as

```
PRT1 DC   C'THIS PARAMETER C'
     DC   C'OMBINATION IS IN'
     DC   C'VALID'
```

The symbol PRT1 in the statement below still refers to the complete sentence; i.e., it causes the complete sentence to be transferred to position FLDA.

```
MVC      FLDA(37),PRT1
```

Character Constants

Character constants may consist of any of the 256 EBCDIC characters. Each character

Type of Constant	Code	Machine Format of the Constant	Alignment at
Character	C	8-bit code for each character	byte boundary
Hexadecimal	X	4-bit code for each hexadecimal digit	byte boundary
Halfword	H	16-bit binary equivalent of the specified value (signed)	halfword boundary
Address	Y	16-bit binary equivalent of symbolic or absolute storage address	halfword boundary

Figure 10. Types of Constants

in these constants occupies one byte of main storage.

DC statements that define character constants may comprise all of the three operand components: type, length modifier, and the constant.

Example:

Name	Operation	Operand
CON1	DC	CL4'ABCD'

In the above example, the name entry and length modifier are optional and may be omitted. This statement causes the constant ABCD to be generated in main storage.

The length modifier (L4) coincides with the number of characters in the constant. Therefore, it has no effect because the Basic Assembler program assumes the length of the constant to be implied if the length modifier is omitted. However, if the length modifier disagrees with the number of characters in the constant, the constant is modified as follows.

1. If the length modifier is smaller than the number of characters in the constant, rightmost digits of the constant are dropped to achieve agreement with the modifier.

2. If the length modifier is greater than the number of characters in the constant, the excess rightmost bytes are filled with blanks until the length of the constant agrees with the length modifier.

The constant must be enclosed by apostrophes. The length of the constant must not exceed 16 bytes. Apostrophes and ampersands that are to appear within constants must be written twice but are counted only once.

Example:
```
Statement:     CON2       DC   C'''TOTAL 10'''
Generated as: 'TOTAL 10'       (implied length
                                10 bytes)

Statement:     CON2       DC   CL1'YY'
Generated as:  Y               (explicit
                                length one
                                byte)
```

In the last example, the specification of the length modifier (L1) causes the last character Y to be truncated. This statement will be identified by a warning message in the program listing.

Hexadecimal Constants

Hexadecimal constants are used to introduce data characters each of which occupies half a byte of main storage. DC statements that define hexadecimal constants may comprise all of the three operand components: type, length modifier, and the constant.

Example:

Name	Operation	Operand
MASK	DC	XL3'A345BF'

In the above example, the name entry and length modifier are optional and may be omitted. This statement causes the constant A345BF to be generated in main storage. Each pair of digits is translated into one byte. Thus, the length modifier, L3, coincides with the length of the constant and has no effect because the implied length is half the number of hexadecimal digits specified if the length modifier is omitted. However, if the length modifier is not equal to half the number of hexadecimal digits, the constant is modified as follows:

1. If the length modifier is smaller than the number of pairs of hexadecimal digits the leftmost digits of the constant are dropped to achieve agreement with the modifier.

2. If the length modifier is greater than the number of pairs of hexadecimal digits, the excess leftmost bytes are filled with zeros until the length of the constant agrees with the length modifier.

The constant may consist of any number of valid hexadecimal characters, 0 to 9 and A to F, but must not exceed 32 digits. If an odd number of digits is specified, a hexadecimal zero is added to the leftmost position.

Examples:
```
Statement:     TRIX       DC   X'3AF'
generated as:  03AF

Statement:     INCR       DC   XL4'BA05'
generated as:  0000BA05

Statement:     TRNC       DC   XL2'AFE696'
generated as:  E696
```

In the last example, the specification of the length modifier (L2) causes truncation of the digits AF. The truncation causes the statement to be identified by a warning message in the program listing.

A hexadecimal constant can be used to set the binary bits of a halfword. The constant in the following example sets the eight leftmost bits of a halfword to 1's. Since a hexadecimal constant is not boundary aligned, the preceding DS statement is applied to force this condition. (For a discussion of DS statements refer to the section DS -- Define Storage.)

Name	Operation	Operand
	DS	0H
TEST	DC	X'FF00'

Halfword Constants

A halfword constant is a signed integer, aligned at a halfword boundary. The operand must not contain a length code.

Example:

Name	Operation	Operand
WORK	DC	H'-24'

The name entry is optional and can be omitted. The above statement causes the generation of one halfword in main storage, containing the value -24.

The highest allowable value for a halfword constant is 32767, the lowest, -32768. If a specified number exceeds either value, the constant is set to zero and the statement is identified by a warning message in the program listing. Unsigned numbers are considered to be positive.

Address Constants

An address constant is a relocatable or absolute expression, enclosed in parentheses with the prefix Y. It is used for indexing (i.e., generating and incrementing address values to scan main storage) and for program linking. The operand must not contain a length modifier.

Example:

Name	Operation	Operand
ADTA	DC	Y(TABL)

In the above example, the address of TABL is stored at position ADTA. If ADTA is now loaded into a register, an AH instruction can be used to update or increment this address by any desired value. This is demonstrated in Figure 14 and in the section Indexing.

The routine PRGM in Figure 14 calculates certain values, which are then stored in the 480-byte table defined by TABL. The program loads the first value to be stored into register 10 (statement 034A) and branches to LOOP (statement 0330).

The statement named LOOP stores the value of register 10 in the location designated by register 8, which is the table address ADTA loaded into register 8 by the statement named RTN. Thus, the first calculation result is stored in the first table position.

The AH statement then increments the contents of register 8 (i.e., the table address) by four, the implied length of each position. The contents of register 8 now point to the second table position.

Successive repetitions of the procedure continue until the table is filled or the program is terminated by the TM instruction.

The use of the address constant to link two or more simultaneously executed program parts is discussed in the section Program Linking.

An absolute expression is specified in the operand of an address constant if a branch to an absolute address is performed during the course of a program. But the program must be relocatable. Obviously, the absolute address should be updated upon program relocation to avoid branching to the wrong statement. This updating is guaranteed by the address constant. One method of accomplishing this updating is demonstrated by the following example.

```
    BC    15,0
    ORG   *-2
    DC    Y(3215)
```

In the normal branch instruction BC 15, 3215, the address 3215 would not be altered upon program relocation. Therefore, the second operand is set to zero as the branch instruction is assembled.

On its own, this imperative branch instruction would be invalid because it instructs the computer to branch and, at the same time, prevents the branch by setting the branch address to zero. However, the Basic Assembler program does not consider this statement incorrect since all syntax requirements are satisfied. The second operand of the BC instruction can be omitted, provided the comma is written.

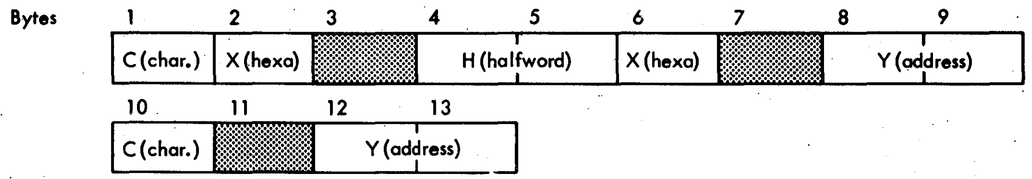


Figure 11. Uneconomical Storing of a Sequence of Constants

The ORG statement reduces the value of the location counter by two bytes so that it points to the location of the second operand of the BC instruction, which is updated if the program is relocated.

Sequence of Definition of Constants

Halfword and address constants are automatically aligned at halfword boundaries by advancing the location counter to the proper value (multiple of 2) when either type of constant is encountered in the source program.

For economical use of main storage, the sequence in which constants are defined is important. The following example shows the definition of a sequence of constants. It is assumed that the first storage position of these constants is not boundary aligned. C and X-type constants have an implied length of one byte.

```
DC   C (character type)
DC   X (hexadecimal type)
DC   H (halfword type)
DC   X (hexadecimal type)
DC   Y (address type)
DC   C (character type)
DC   Y (address type)
```

They are stored as shown in Figure 11.

As shown in Figure 11 three bytes are not used. A more economical specification sequence is

```
DC   C (character type)
DC   H (halfword type)
DC   Y (address type)
DC   Y (address type)
DC   X (hexadecimal type)
DC   X (hexadecimal type)
DC   C (character type)
```

resulting in the storage allocation shown in Figure 12.

DS -- DEFINE STORAGE

The DS (Define Storage) statement is used to reserve storage for work areas, I/O areas, tables, etc. These storage areas are not set to zeros or blanks. The location counter is incremented during assembly by the number of bytes implied in the operand of the DS statement, leaving the respective storage positions unused when the object program is loaded. The program later refers to this area by the symbolic address of the DS statement. The DS statement can also be used to effect boundary alignment of the subsequent program sections. The DS statement has only one operand. It has the following format:

Duplication Factor	Type	Length Modifier

The duplication factor is written as a decimal integer; the type is written as a single letter, C or H. The length modifier is written as a decimal integer, preceded by the letter L. It may only be specified for C-type constants. The maximum value is 256. The storage area that can be reserved by a DS statement is limited only by the capacity of the location counter.

H-Type Operand

The H-type operand is employed to reserve a storage area the subfields of which have an implied length of two bytes.

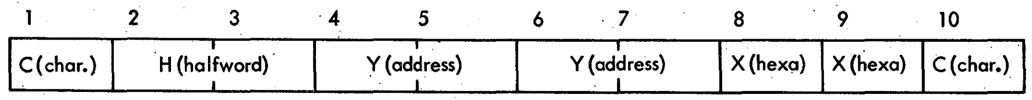


Figure 12. Economical Storing of a Sequence of Constants

Example:

Name	Operation	Operand
INA1	DS	20H

This statement causes 20 halfwords (40 bytes) of main storage to be reserved, beginning at a halfword boundary. The leftmost byte of this area carries the symbolic address INA1. Each storage field referred to by this address has the implied length of two bytes. The knowledge of the implied length is important if INA1 is specified as an operand of a machine instruction that requires the inclusion of a length factor.

C-Type Operand

For reservation of storage areas with subfields of different implied lengths, the C-type operand is used.

Example:

Name	Operation	Operand
INA2	DS	100CL3

This statement reserves 100 fields of main storage with a length of 3 bytes each, a total of 300 bytes, addressable through the symbol INA2. This reserved area is not boundary aligned.

The length modifier of a DS C-type statement may have any value from 1 through 256. Additional examples of DS statements are shown below.

AREA	DS	CL100	defines one field of 100 bytes.
FLD1	DS	80C	defines 80 fields of one byte each.

While the Basic Assembler is processing a DS statement, it discontinues the punching of the current TXT and RLD cards. Punching is resumed with a new TXT card for the location following the reserved area(or areas). Therefore, all DS statements of a program should be grouped together to reduce the number of TXT cards punched.

Duplication Factor

Data fields frequently contain values that will be loaded into a register in the course of a program. These data fields must be aligned at a halfword boundary.

If the data is defined as character or hexadecimal constants, i.e., data is not automatically boundary aligned, it may be difficult to verify this alignment, especially in a complex program. In such a case, it is better to force boundary alignment, as a precaution, thus removing the need to verify.

In the following example, a storage area named AREA is defined, with an implied length of 128 bytes. The preceding DS statement with a duplication factor of zero sets the location counter to a halfword boundary.

Name	Operation	Operand
	DS	0H
AREA	DS	CL128

A duplication factor of zero is also used to assign a name and a length attribute to a storage area without actually reserving it. Subsequent DS or DC statements then establish subfields within the larger area by assigning addresses to these subfields and generating data.

In the example in Figure 13, the name PAYR is assigned to an area of 50 bytes. No space is actually reserved at this point, but subsequent DS statements subdivide and reserve the storage within the area PAYR. The symbols PYNO, REGH, etc., which are specified in the name fields of the DS statements, allow reference to subsections of the area PAYR. The address PAYR still implies the length of 50 bytes and refers to the area as a whole.

Name	Operation	Operand
PAYR	DS	0CL50
	DS	2H
PYNO	DS	CL6
LNAM	DS	CL10
FNAM	DS	CL10
REGH	DS	H
OVTM	DS	H
STRT	DS	CL4
OVRT	DS	CL4
SLRY	DS	CL6
	DS	H

Figure 13. Reservation of Main Storage

Basic Assembler control instructions are used to begin assembly (START), end assembly (END), and set the location counter to a value at a halfword boundary (ORG).

START -- START PROGRAM

When a program is loaded, a start address normally specifies the point where the first byte of information is to be stored. Bytes 0 to 155 of main storage lie within an area that contains information required for the execution of a program. This information must not be overwritten. Therefore, the lowest usable start address is 156 (hexadecimal 009C). Before a source program or the Basic Assembler program can be loaded, a program to execute the loading functions is required. Such a program (the Absolute-Program Loader or the Relocatable-Program Loader) is stored from location 156 upward. The Absolute-Program Loader, for example, requires 160 bytes of main storage, which increases the possible start address for the source program to 316 (hexadecimal 013C). A start address of 156 can be used in this case, provided the subsequent 160 bytes are reserved for the Absolute-Program Loader by means of an appropriate DS or ORG statement.

The start address is specified in a START statement. The operand of the START statement specifies the tentative loading point in the form of an absolute address. The value of the location counter is incremented to represent this address as soon as the START statement is read by the Basic Assembler program. If the START statement is omitted, the location counter is automatically set to 340. (A START statement without an operand should not be used and is flagged with a C.)

Example:

```
    START 1000
```

This statement causes the location counter to be advanced to 1000. Since the START statement does not consume any storage space itself, the specified start address is assigned to the instruction that follows the START statement. If a symbol is entered in the name field of a START statement, it is considered to be the program name and is entered in the symbol table, together with the start address of the program. In addition, the Basic Assembler program causes the name to be punched into columns 73 to 76 of each object program card.

Note: For the purpose of boundary alignment, the start address should be an even number. If it is an odd number, the Basic Assembler program advances the location counter to the next higher even value above the specified start address.

END -- END OF PROGRAM

A program written in Basic Assembler language must be terminated by an END statement, which supplies the branch address required for program execution after the program is loaded.

The operand of the END statement contains the address of the point to which control is to be transferred on completion of the loading process. This is normally the address of the first machine instruction in the problem program.

Example:

Name	Operation	Operand
PBL1	START	340
BGN	BASR	10,0
	|	
	|	
	V	
	END	BGN

In the above example, the start address for program execution is BGN. When the END card is read, the address contained in the operand of the END statement is loaded into register 12 by the Absolute-Program Loader, followed by a branch to the address in register 12, which initiates program execution.

If it is desired to load more than one program for simultaneous execution, the Relocatable-Program Loader must be used and a load terminate (LDT) card must be supplied. In this case, the loader program disregards the END card. For further details refer to the SRL publication _IBM System/360 Model 20 Card Programming Support, Basic Utility Programs, Functions and Operating Procedures_, Form C26-3604.

ORG -- RESET LOCATION COUNTER

The ORG statement is used to reset the location counter to any desired value.

The statement ORG *+500 causes the present value of the location counter to be incremented by 500. The operand of an ORG statement is invalid if it is not a relocatable expression, if the expression consists of or includes a symbol that has not been previously defined, or if the name field of the statement contains an invalid symbol. A valid symbol in the name field is disregarded by the Basic Assembler program. Invalid operands are identified by error messages in the program listing. The location counter should not be reset to a value lower than the start address of the program unless it is to be loaded by the Absolute-Program Loader. The ORG statement can be used when source and object programs exceed the available main storage capacity and must, therefore, be assembled and executed in separate phases.

The program shown in Figure 14 executes certain scientific-mathematical calculations and stores the results in a 480-byte table, which is printed out later. The calculations are assumed to consist of two phases, each of which requires 3200 bytes. This means that (with an available storage capacity of 4096 bytes) each calculation routine must be assembled separately. The resulting object programs are executed one after the other.

For this reason, the table area where the results of the calculations are stored, was reserved at the first available positions of main storage, followed immediately by the constants and program routines required for all successive calculation phases. This information occupies storage locations 013C to 0340 (the addresses are given in hexadecimal notation to facilitate reference to Figure 14), and will not be overwritten when subsequent program phases for execution of the assembly are loaded.

The statement PRGM MVC X(5),Y, which is stored at position 0344, is the first statement of the calculation routine. All other statements of this procedure have been omitted, except those that demonstrate the chaining of the various program routines.

When the first result has been calculated, it is loaded into register 10 (statement 034A). The program then branches to LOOP (statement 0330). The following program segment stores the result in the first position of the table area (a detailed explanation is given in the section DC -- Define Constant), and tests a switch to determine if the program must go through the calculation routine again to compute another result.

If this is the case, the program branches to PRGM. Otherwise, the calculation phase has been completed and the program branches to the loader area (statement 033C) to read calculation phase 2 into main storage.

Calculation phase 2, as a separately assembled program, also begins with a START instruction. However, since the loader does not use it, register 14 has the same contents as during the previous assembly. Therefore, the BASR instruction can be omitted and the START address becomes 318. However, the USING instruction is required.

Now the previous program part must be linked to the subsequent one. The ORG statement is used to reset the location counter to position 0344. This is the start address of the calculation routine phase 1, which is no longer required and can be overwritten by phase 2.

Since the operand of the ORG statement must be relocatable and hexadecimal 0344 is an absolute address, the location counter is set to 0(*-318) and the desired address 0344, which is equal to PH2, is added. The operand *-318+PH2 thus obtained is relocatable. The address 0344 can be determined only from the program listing, after the assembly of phase 1. It must then be inserted into a previously prepared statement card.

The location counter setting of 0344 causes the subsequent program (a) to be loaded, starting at this position, and (b) to overwrite phase 1.

By following this procedure, any number of programs can be assembled separately and then be linked for successive execution.

```
   013C                          PBL1   START 316                         001
   013C   0DE0                   BGN    BASR  14,0                        002
   013E                                 USING *,14                        002
   013E   47F0 E1EA                     BC    15,RTN                      002
   0142                          TABL   DS    120CL4     DEFINE RESULT TABLE    002
   0322   0142                   ADTA   DC    Y(TABL)                     003
   0324   0004                   FOUR   DC    H'4'                        003
   0326   F0                     SWIT   DC    CL1'0'                      003
   0008                          R8     EQU   8                           003
   0009                          R9     EQU   9                           003
   000A                          R10    EQU   10                          003
   0328   4880 E1E4              RTN    LH    R8,ADTA    LOAD TABLE ADDRESS     003
   032C   47F0 E206                     BC    15,PRGM                     003
   0330   40A0 8000              LOOP   STH   R10,0(0,R8) BRING RES INTO TABL   003
   0334   4A80 E1E6                     AH    R8,FOUR    INCRM TABLE ADDR       003
   0338   9101 E1E8                     TM    SWIT,1     TEST FOR PROGRAM END   003
   033C   4710 009C                     BC    1,156      LOAD PHASE 2           003
   0340   47E0 E206                     BC    14,PRGM    REEXECUTE PHASE 1      003
                                 *
                                 *
                                 *START CALCULATION PHASE 1
                                 *
                                 *
U  0344   D204 0000 0000         PRGM   MVC   X(5),Y                      003
                                 *
                                 *
                                 *THIS PROGRAM PHASE REQUIRES  CA. 3200 BYTES
                                 *
                                 *
                                 *
                                 *
U  034A   48A0 0000                     LH    R10,RES      LOAD RESULT INTO R10   003
   034E   47F0 E1F2                     BC    15,LOOP      INITIATE TABLE ENTRY   003
   013C                                 END   BGN                         005
```

Figure 14 (Part 1). Programmed Routine for Table Look-up and Program Linking

```
                                 *START CALCULATION PHASE 2
   013E                                 START 318                               001
   013E                                 USING *,14                              002
   000A                          R10    EQU   10                                002
   0326                          SWIT   EQU   X'0326'           SYMBOL LINKING  002
   0330                          LOOP   EQU   SWIT+10           SYMBOL LINKING  002
   0344                          PH2    EQU   X'0344' ADD OPERND AFT ASSBLY PH1 002
   0344                                 ORG   *-318+PH2  JUMP TO FIRST AVAIL LOC 002
                                 *
                                 *
   0344   92F1 0326              PBL2   MVI   SWIT,C'1'    BEGINNING OF CALC PH2  002
                                 *
                                 *
                                 *THIS PROGRAM PHASE REQUIRES  CA. 3200 BYTES
                                 *
                                 *
                                 *
U  0348   48A0 0000                     LH    R10,RES       LOAD RESULT INTO R10  002
   034C   47F0 0330                     BC    15,LOOP       INITIATE TABLE ENTRY  002
   0344                                 END   PBL2                              003
```

Figure 14 (Part 2). Programmed Routine for Table Look-up and Program Linking

Input/output operations can be caused in two ways:

1. by means of the Input/Output Control System (IOCS), or

2. by writing I/O routines using the Basic Assembler I/O instructions.

The use of IOCS allows the writing of macro instructions, as explained in a subsequent section. The second method, the writing of individual I/O routines, is explained in the following paragraphs.

Three types of I/O instructions are available in the Basic Assembler language:

1. XIO instructions (execute input and output).

2. CIO instructions (control input and output).

3. TIOB instructions (test input and output and branch).

The XIO statement has an SS format, and CIO and TIOB statements have SI formats, as explained in the section Machine Instruction Formats.

All three instructions include the unit and function (UF) specification field. Data in this field must be specified in hexadecimal notation.

XIO -- EXECUTE INPUT/OUTPUT

The operand entry of an XIO instruction is written

```
D1(UF,B1),D2(B2)
```

or when using symbolic addressing,

```
Symbol 1 (UF), Symbol 2.
```

U designates the Unit used as the I/O device and F designates the assigned Function, i.e., the operation to be executed.

For example, a 2501 reader is attached and X'12' is specified in the UF field of the XIO instruction. The hexadecimal digit 1 tells the Basic Assembler program that the 2501 is used and the hexadecimal digit 2 indicates that the unit must read a card. A complete list of all UF codes is provided in Appendix C.

Depending on the specification in the UF field of the XIO instruction, the second symbol designates the amount of data to be handled during the I/O operation; i.e., the number of card columns to be read or punched, or the number of characters to be printed. Samples of XIO instructions are shown in Figure 15.

Note: If the XIO statement refers to a card unit, the value in the second operand must not exceed 80. If it refers to a printer, the maximum value is 144 for a 2203 Printer; 132 for a 1403 Model 2 or N1; and 120 for a 1403 Model 7.

Note	Name	Operation	Operand
	CARD	EQU	80
	LINE	EQU	100
1.	OUT	XIO	FLDA(X'40'),LINE
2.	OUT	XIO	OUTB(X'40'),20
3.	PNCH	XIO	OUTA(X'36'),CARD
4.	INPT	XIO	IN1(X'23'),16
5.	INPT	XIO	EXAR(X'24'),CARD

Figure 15. Sample of XIO Instructions

1. Prints 100 characters on the attached 1403 or 2203 printer.

2. Prints 20 characters on the attached 1403 or 2203 printer.

3. Punches 80 columns on the attached 1442 Card Punch, Model 5.

4. Reads the first 16 columns of a card from the secondary hopper of the attached 2560 MFCM.

5. Punches 80 columns of a card from the primary hopper of the attached 2560 MFCM or 2520.

CIO -- CONTROL INPUT/OUTPUT

CIO instructions are used to control the operation of attached I/O devices. With card I/O devices, the CIO instruction is used for stacker or print-head selection; with a printer, the CIO instruction is used to cause spacing or skipping.

The instruction is written in the following format:

```
CIO  D1(B1),UF
  or
CIO  S1,UF (S=symbol)
```

Stacker Selection

For stacker selection of card I/O devices, the unit specifications in the UF field is always a 2. The function specification can be a 0, 1, or 2, depending on the attached I/O device and the function desired. The stacker is specified by the first operand of the CIO statement. A summary of I/O instructions, including the associated unit and function specification codes, is given in Appendix C.

Examples:

```
1.  CIO 4,X'21'
2.  CIO 3,X'22'
3.  CIO 2,X'20'
```

1. It is assumed that a preceding read instruction caused the feeding of a card from the secondary hopper of the attached MFCM. The sample statement causes this card to be ejected into stacker 4.

2. The card presently in the punch or pre-print station of the attached MFCM is ejected into stacker 3.

3. If the attached unit is a 2520 and a preceding read or punch instruction caused the feeding of a card, this card is ejected into stacker 2. If the attached unit is a 2560 MFCM and a preceding read instruction caused the feeding of a card from the primary hopper, this card is ejected into stacker 2.

If the I/O unit is a 2560 MFCM and stacker selection is not specified, stacker 1 is automatically selected for cards from the primary hopper and stacker 5 for cards from the secondary hopper. Therefore, CIO statements that assign these functions are not required.

In addition, if 6, 7, 14, or 15 is specified in the first operand of a CIO instruction that refers to a 2560 MFCM, the selected cards are ejected into stacker 1.

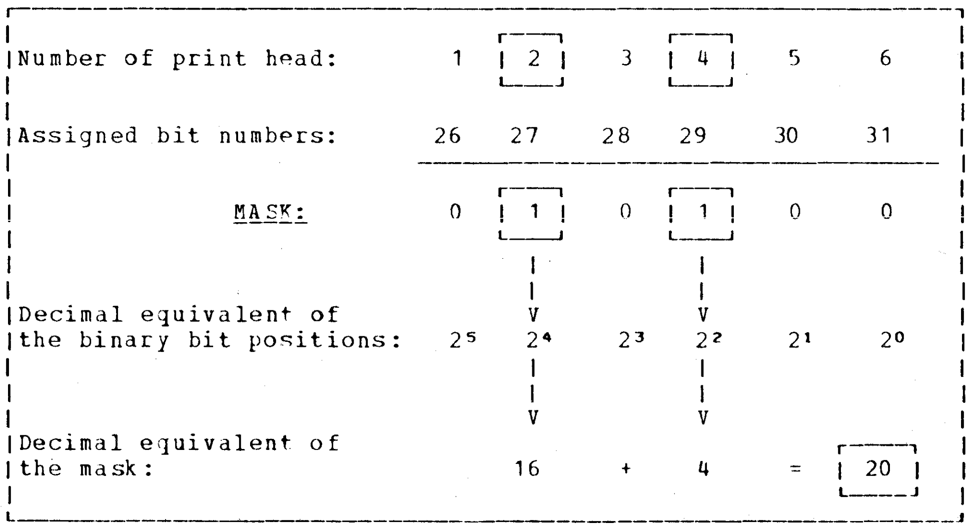
In the programming sequence, the CIO statement for a 2560 MFCM should follow a read instruction, if possible. In addition, it must precede the next read, punch, or punch-and-feed instruction for the same hopper. For punch-card stacker selection, the relevant CIO instruction must be placed before the next read, punch, or punch-feed instruction, regardless of the referenced hopper.

The punch-card stacker select function (X'22') is specified for stacker assignment, if the respective card is in the punch unit or in the pre-print station of the MFCM.

For a 2520, the CIO statement is required only to assign stacker 2. In the programming sequence, this statement should precede the read, punch, or punch-and-feed instruction.

Print-Head Selection

Print heads are selected by using bits 26 to 31 of the machine-instruction format as a mask. The mask is specified as a decimal integer in the first operand of the pertinent CIO statement and sets the bits assigned to the individual print heads to one. This is illustrated in Figure 16.



Number of print head: 1 2 3 4 5 6
Assigned bit numbers: 26 27 28 29 30 31
MASK: 0 1 0 1 0 0
Decimal equivalent of the binary bit positions: 2^5 2^4 2^3 2^2 2^1 2^0
Decimal equivalent of the mask: 16 + 4 = 20

Figure 16. Sample of a Mask for Print-Head Selection

In the above example, print heads 2 and 4 are selected because the bits assigned to these are set to 1 by the mask. The decimal equivalent of the mask is specified in the first operand of the CIO statement as follows:

```
    CIO 20,X'23'
```

The operand X'23' refers to a card I/O device and specifies the print-head-select function.

The highest decimal number that can be used as a mask for print head selection is 63, which activates all available print heads. The mask can also be expressed in hexadecimal notation or in the format D1(B1).

Spacing and Skipping

A CIO statement that refers to a printer must contain the unit address (U) hexadecimal 4. If a spacing function is requested, the first operand specifies the number of space to be performed. This can be expressed in decimal or hexadecimal form, or as D1(B1). The maximum number of spaces allowed is 3.

The appropriate function codes are shown in the summary of I/O instructions in Appendix C.

Example:

```
    CIO 2,X'4C'
```

This statement causes the immediate spacing of 2 lines on both carriages of an attached 2203 Printer.

If a skipping function is requested, the first operand specifies the channel number of the carriage control tape that identifies the line at which the skipping is terminated.

Example:

```
    CIO 6,X'45'
```

This statement causes the skipping of a page on the attached 1403 Printer, up to the line identified by a punch in channel 6 of the carriage control tape.

Serial I/O Channel

All CIO statements that refer to the serial I/O channel must contain the unit address hexadecimal 6. For the appropriate function specification refer to Appendix C. The use of the first operand D1(B1) is described in the following SRL publication:

IBM System/360 Model 20, 1419 Magnetic Character Reader, Form A24-1499.

Communications Adapters

A CIO statement that refers to the Model 20 Communications Adapter or the Binary Synchronous Communications Adapter must contain the unit address hexadecimal 5. For the appropriate function specification, refer to Appendix C.

The first operand, D1(B1), of a CIO statement that refers to one of the communications adapters is ignored. However, it must be contained in the statement, and must resemble a valid address.

TIOB -- TEST INPUT/OUTPUT AND BRANCH

TIOB statements are used to test the operational conditions of the attached I/O devices or the proper execution of an I/O function; e.g., print error, last card, feed error, device busy.

If a busy condition exists, a branch is performed to the address specified in the first operand of the pertinent statement. Otherwise, the subsequent program statement is processed.

The operands of a TIOB statement are written in the following form:

```
D1(B1),UF

   or

S1,UF
```

Examples:

```
    1.  TIOB   AREA,X'24'
    2.  TIOB   *,X'40'
    3.  TIOB   HALT,X'33'
```

1. This instruction causes a branch to position AREA after the last card has been read on the attached I/O device with the device address 2.
2. This statement causes the program to loop until the attached printer has completed the current print cycle.
3. This instruction causes a branch to the procedure named HALT if a punch error has occurred on the attached 1442 Card Punch.

A summary of the Basic Assembler I/O instructions, together with the associated function specification codes, is provided in Appendix C.

SEQUENCE OF I/O INSTRUCTIONS

The proper sequence of the input-output instructions for different cases is shown in the following examples:

```
CARRIAGE CONTROL:     TIOB   *,X'46'             TEST CARRIAGE BUSY
                      TIOB   *,X'40'             TEST PRINTER BUSY
                      CIO    1,X'45'             SKIP TO CHANNEL ONE IMMEDIATELY
           OR         CIO    3,X'44'             SPACE THREE TIMES IMMEDIATELY

PRINTER CONTROL:      TIOB   *,X'46'             TEST CARRIAGE BUSY
                      TIOB   *,X'40'             TEST PRINTER BUSY
                      XIO    PRT(X'41'),120(0)   PRINT AND SPACE SUPPRESS
                      BC     4,*-6               BRANCH IF PRINTER WORKING
                      BC     1,HLT               BRANCH IF PRINTER NOT OPERATIVE

CARD READER CONTROL:  TIOB   *,X'20'             TEST READER BUSY
                      XIO    CRD(X'22'),80(0)    READ THE CARD
                      BC     4,*-6               BRANCH IF READER WORKING
                      BC     1,HLT               BRANCH IF READER NOT OPERATIVE
                      TIOB   END,X'24'           BRANCH IF LAST CARD
                      TIOB   HLT,X'25'           BRANCH IF FEED ERROR

PUNCH CONTROL:        TIOB   *,X'20'             TEST READER/PUNCH BUSY
                      CIO    2,X'22'             SELECT STACKER TWO
                      XIO    PCH(X'25'),80(0)    PUNCH SECONDARY CARD
                      BC     5,HLT               BRANCH IF PUNCH NOT OPERATIVE
                      TIOB   HLT,X'21'           TEST READER/PUNCH ERROR
                      TIOB   END,X'24'           TEST LAST CARD
                      TIOB   HLT,X'25'           TEST FEED ERROR
```

INPUT/OUTPUT MACRO INSTRUCTIONS

A major part of most programs written in Basic Assembler language consists of the routines required to read data into the system and to produce the output of the processing performed on the input data. IBM provides the user of the Model 20 Basic Assembler language with a library of tested I/O routines, which is part of the IBM System/360 Model 20 Card Programming Support, Input/Output Control System (CPS IOCS).

Macro Instruction	Function
GET	Makes the next record available in the area specified by the user.
PUT	Makes a record (in an area specified by the user) available for an I/O operation.
OPEN	Opens the file, i.e., ensures that all information necessary to handle a file has been provided.
CLOSE	Closes the file, i.e., ensures proper handling of the file after all records have been processed.
CRDPR	Moves the information to be printed on a card from the work area into the specified print area. Used only in connection with a 2560 MFCM.
CNTRL	Causes the performance of certain I/O functions, e.g., skipping, spacing, stacker selection.
LOM	Starts processing of files in non-overlap mode.
EOM	Starts processing of files in overlap mode, in case of a preceding LOM macro instruction.
PRTOV	Checks for printer overflow conditions.
WAITC	Causes the problem program to wait for the completion of all pending card I/O operations before processing the next sequential instruction.

Figure 17. Summary of IOCS Macro Instructions

In the source program, the IOCS routines are called by statements referred to as macro instructions. The use of IOCS macro instructions saves programming time because it relieves the user of coding, testing, and providing linkages to his own I/O routines. In addition, the IOCS routines take advantage of the time-sharing capability of the Model 20, thereby optimizing throughput.

For detailed information on the Model 20 IOCS, refer to the SRL publication _IBM System/360 Model 20 Card Programming Support, Input/Output Control System_, Form C26-3603.

Figure 17 contains a summary of the IOCS macro instructions and their functions.

Additional macro instructions, and the associated I/O routines, are available to users of the Communications Adapter and the 1419 Magnetic Character Reader. For detailed information refer to the following SRL publications:

IBM System/360 Model 20:

Input/Output Control System for the Communications Adapter, Form C26-3606;

Input/Output Control System for the Binary Synchronous Communications Adapter, Form C33-4001;

Input-Output Control System for the IBM 1419 Magnetic Character Reader, Form C26-3607.

I/O ROUTINES -- INCLUDING INTERRUPTS

A user program which enables the interrupt mode with an SPSW statement that changes the channel mask bit of the current program status word from 1 to 0 must ensure that the pending interrupts caused by the loader do not interfere with the execution of the object program.

Both the Absolute-Program Loader and the Relocatable-Program Loader cause two pending interrupts. Interrupt 1 is caused when the program is read on a 2501, 2520, or 2560; interrupts 1 and 2 are caused when the program is read on a 2520 or a 2560.

Interrupt 1: Associated with the _last read instruction_ of the loader, interrupt 1 is pending when the execution of the object program begins. This interrupt becomes effective after the first SPSW instruction in the user program has been processed. The program in this case branches to the programmed interrupt routine, although the condition on which the interrupt routine is based has not occurred.

An example of the programming sequence that enables the interrupt mode through an SPSW statement is shown in Figure 18. For this purpose, the first two TIOB statements in the figure may be disregarded.

Interrupt 2: This interrupt is issued at the end of program loading. After the END card of the object program has been read, an XIO instruction in the loader program causes a _dummy punch cycle_ that moves the

END card from the pre-punch station to the punch unit of the punching device, prior to execution of the object program. The dummy cycle is effected by specifying X'40' (blank) to be punched into column 1, which results in nothing being punched.

Interrupt 2 also occurs after the first SPSW instruction in the user program after the dummy punch instruction has been executed.

The XIO dummy instruction may cause a mispunching of the END card during the initial phase of the object program. While the XIO instruction is being executed, the loader transfers control to the object program and, thereby, initiates processing. If the I/O device used for loading is a 2560 MFCM or a 2520 card read-punch and the loader area is overwritten before execution of the dummy punch instruction has been completed, a character other than blank may be punched into column 1 of the END card, which makes the END card invalid.

Mispunching of the END card can be avoided by using a TIOB instruction as the first statement in the user's program, as shown in Figure 18.

The mispunching of the END card can also be avoided if the loader area is not overwritten during the initial processing phase. (The initial processing phase is terminated after execution of the first XIO statement in the user program that refers to the 2560 that is used for program loading).

Name	Operation	Operand
BEG	TIOB	*,X'22' WAIT, when loading from a 2520
	TIOB	*,X'20' WAIT, when loading from a 2560
MVNP	MVC	148(4),AXPW GET AUXILIARY NEW PSW
	SPSW	AXPW ENABLE INTERRUPT MODE
AXPW	DC	X'0100' THIS PSW BRANCHES TO
	DC	Y(SNPS) MAIN PROGRAM
SNPS	MVC	148(4),SYMB defining the address of
	┌──>	user's PSW.
	│	
	└──MAIN PROGRAM	

Figure 18. Sample Routine for Compensation of Pending Interrupts Caused by the Loader

This section describes the coding of the machine instructions written in Basic Assembler language and translated into machine language. The machine-language format and the functions of each machine instruction are described and the use of each instruction is illustrated by an example.

A machine instruction is a direction given to the computer to cause the execution of a certain operation. In Basic Assembler language, these instructions are written in the form of mnemonic codes, which are translated by the Basic Assembler program into System/360 internal or machine code, respectively. The codes are printed in the leftmost part of the program listing, next to the location counter reference.

Machine instructions are divided into four groups, according to basic operand format:

1. RR instructions (register to register), length: 2 bytes.
2. RX instructions (register to storage or storage to register), length: 4 bytes.
3. SI instructions (storage - immediate), length: 4 bytes.
4. SS instructions (storage to storage), length: 6 bytes.

A summary of these formats, together with their associated operation codes, is shown in Figure 19.

All machine-instruction statements are automatically aligned at halfword boundaries. All bytes skipped are filled with hexadecimal zeros.

Any machine instruction can be identified by a symbol, which can be used as a branch address in operand(s) of other statement(s).

Notes Referring to Figure 19

1. R1 and R2 are absolute expressions that specify general registers. The general register numbers are 8 through 15.
2. D1 and D2 are absolute expressions that specify displacements. A value of 0 through 4095 may be specified.
3. B1 and B2 are absolute expressions that specify base registers. Register numbers are 0-3 and 8-15.
4. M1 is an absolute expression representing a condition code.
5. L, L1, and L2 are absolute expressions that specify field lengths. An L expression can specify a value of 1 - 256. L1 and L2 expressions can specify a value of 1 - 16. In all cases, the assembled value will be one less than the specified value.
6. I2 is an absolute expression that provides immediate data. The value of the expression may be 0 - 255.
7. S1 and S2 are absolute or relocatable expressions that specify an address.
8. SI instruction fields that are crossed out in the machine formats are not examined during instruction execution. The fields are not written in the symbolic operand, but are assembled as binary zeros.
9. UF is an absolute expression representing an input/output unit address and a function.

	Basic Machine Format	Assembler Operand Field Format	Applicable Instructions
RR	8 Operation Code | 4 R1 | 4 R2	R1,R2 (See note 1)	AR ,BASR,SR
	8 Operation Code | 4 M1 | 4 R2	M1,R2 (See notes 1 and 4)	BCR
RX	8 Operation Code | 4 R1 | 4 X2 | 4 B2 | 12 D2	R1,D2(0,B2) R1,S2 (See notes 1,2,3,and 7)	STH,LH,CH,AH,SH, BAS
	8 Operation Code | 4 M1 | 4 X2 | 4 B2 | 12 D2	M1,D2(0,B2) M1,S2 (See notes 2,3,4,and 7)	BC
SI	8 Operation Code | 8 I2 | 4 B1 | 12 D1	D1(B1),I2 S1,I2 (See notes 2,3,6,and 7)	CLI,MVI,NI,OI,TM,HPR
	8 Operation Code | 8 -- | 4 B1 | 12 D1	D1(B1) S1 (See notes 2,3,7,and 8)	SPSW
	8 Operation Code | 8 UF | 4 B1 | 12 D1	D1(B1),UF S1,UF (See notes 2,3,and 7-9)	TIOB CIO (D1(B1) detailed specification)

SS	8 Operation Code | 4 L1 | 4 L2 | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(L1,B1),D2(L2,B2)
		S1(L1),S2(L2)
		(See notes 2,3,5,and 7)
			PACK,UNPK,MVO,AP,
			CP,DP,MP,SP,ZAP

	8 Operation Code | 8 L | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(L,B1),D2(B2) S1(L),S2 (See notes 2,3,5,and 7)	CLC,MVC,MVN, MVZ,TR,ED
	8 Operation Code | 8 UF | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(UF,B1),D2(B2) S1(UF),S2 (See notes 2,3,7,and 9)	XIO (D2(B2) detailed specification)

Figure 19. Machine Instruction Formats

MACHINE-INSTRUCTION MNEMONIC CODES

The mnemonic operation codes (shown in Appendix B and Figure 19) are designed to be easily-remembered codes that indicate the functions of the instructions. The normal format of the code is shown below; the items in brackets are not necessarily present in all codes:

Verb [Modifier] [Data Type] [Machine Format]

The verb, which is usually one or two characters, specifies the function. For example, A represents Add, and MV represents Move. The function may be further defined by a modifier and the data type. For example, the modifier L indicates a logical function and the C indicates a character as data type, as in CLC for Compare Logical Character.

The letters R and I are added to the codes to indicate, respectively, RR and SI machine instruction formats. Thus, AR indicates Add in the RR format. Functions involving character and decimal data types imply the SS format.

INSTRUCTION FORMATS

A distinction must be made between the instruction format in Basic Assembler language and the instruction format in machine language, as translated by the Basic Assembler program.

Example:

Instruction in Basic Assembler Language		Instruction in Machine Language						
Op Cd	Operands	Op Cd	L1	L2	B1	D1	B2	D2
DP	WORK(9),C2(2)	FD	8	1	D	0DC	D	0EF

In the above example, the DP instruction causes the dividend that is contained in the field WORK, with an explicit length of 9 bytes, to be divided by the divisor, contained in the field C2, with an explicit length of 2 bytes.

Assuming register 13 has been assigned as base register by an appropriate USING statement, the Basic Assembler program translates this instruction into the format FD81 D0DC D0EF, as shown. The mnemonic operation code becomes FD; the WORK length code (9) is contained in the L1 field; and the C2 length code (2) is contained in the L2 field in the assembled instruction. (Each assembled length code is one less than the length of the statement in Basic Assembler language because the length code 1 is assembled as 0, thus permitting a length of 16 within the 4-bit L1 and L2 fields.) The operand addresses are split in a base register and a displacement, which are contained in the B and D-fields respectively (see the section _Storage Addresses_).

RR FORMAT

This is the shortest of the four instruction formats and requires the least processing time. It is used to specify register-to-register operation; i.e., data is transferred from one register to another. In Basic Assembler language, such a statement is written as

```
Op-Code  R1,R2   (R=register)
   or
Op-Code  M1,R2   (M=mask)
```

Example:

AR 9,10 The contents of register 10 are added to the contents of register 9.

The operand format M1,R2 is used together with the Branch-on-Condition-Register (BCR) operation code. It is applied if the program reaches a decision point where, under a certain condition, a branch must be performed. In this case, the branch address is contained in the register specified (R2).

Example:

BCR 8,15

The binary equivalent of 8(1000) is used as a mask to test the condition code in the Program Status Word. The branch is executed if the condition code is 00. (Refer to the section _The Condition Code_).

RX FORMAT

This format is used to cause data flow between a register and main storage. The direction of the flow is determined by the operation code. The Store Halfword (STH) instruction transfers data from a register to storage; the AH instruction causes information in main storage to be added to the contents of a register. The address specified in the second operand of an RX instruction can be in explicit or implied form.

In Basic Assembler language, the instruction is written as:

```
Op-Code  R1,D2(X2,B2) when using explicit
                      addressing,
   or

Op-Code  R1,S2        when using implied
                      addressing.
                      (S = symbol)
```

When specifying an explicit address, the X2 sub-field of the operand D2(X2,B2) must be set to 0.

Example:

```
STH   9,AREA+4(0,12)
```

The contents of register 9 are stored at the location (AREA+4)+(contents of register 12). However, this statement is valid only if AREA is defined as an absolute symbol with an address value not exceeding 4095.

Branch instructions in the RX format (operation code BC) are written as:

```
Op-Code  M1,D2(0,B2)  when using explicit
                      addressing,
   or

Op-Code  M1,S2        when using implied
                      addressing.
```

The field M1 is used as a mask to test the condition code. The subsequent section describes how this mask is set up.

The Condition Code

The condition code in the Program Status Word occupies 2 bits. Therefore, it can be used to represent four conditions: 00, 01, 10, or 11.

The corresponding masks reflecting these settings are:

Condition Code	Mask
00	8
01	4
10	2
11	1

This means, for example, that a branch instruction to be performed on condition 8 is executed if the condition code setting is 00. Accordingly, a branch can be requested in a program if the condition code setting is either 01 or 11. The corresponding mask, in this case, would be 4+1=5.

Thus, the maximum value of a mask is 8+4+2+1=15. Specifying a branch on condition 15 means that the branch must be performed, whatever the condition code setting is. Such a branch is called unconditional. Masks can also be specified in hexadecimal notation. Figure 20 contains examples of branches testing the condition code.

Examples:

	Name	Operation Code	Operand
1.	CALC	SR	9,10
		BC	8,OUT
2.	CMPR	CLI	FLDA,X'D0'
		BC	2,BGN
3.	SUM	AP	FLDA(10),FLDB(5)
		BC	15,TABL(0,8)

Figure 20. Branches Testing the Condition Code

Explanation:

1. Fixed-point arithmetic instructions, like the above SR instruction, set the condition code to reflect the status of the result whether or not the result is equal to, less than, or greater than 0. The branch to the location OUT is executed if the result of the preceding mathematical operation is 0. Otherwise, the next sequential instruction in the program is processed.

2. The CLI instruction causes the value stored at the location FLDA to be compared with the hexadecimal self-defining term D0. The branch is performed if the contents of FLDA are greater than D0. Otherwise, the next sequential instruction in the program is processed.

3. Variable-length arithmetic instructions also set the condition code to reflect the status of the result (see paragraph 1). The subsequent branch instruction is unconditional, i.e., the program branches under any condition to the location represented by TABL+(contents of register 8).

Thus, the interpretation of the condition code setting depends on the type of operation caused by the preceding instruction. A summary of the relation of the situation to the condition code setting is given in Appendix E.

Indexing with RX Instructions

The index field X2 of the instruction format Op-Code R1,D2(X2,B2) must always be set to 0. The base register B2 in an RX instruction, however, can be used as index register, if (1) explicit addressing is applied and (2) the D2 displacement is absolute.

In Figure 14, the method of indexing with address constants has been demonstrated by a table look-up procedure. The table address was loaded into a register and successively updated, thus pointing to the subsequent table positions. The same effect is achieved when TABL is used as displacement D2 in an RX instruction and register R8 is used as B2 to increment TABL.

For example, the table area of 480 bytes is set up in storage with the address TABL, as in Figure 14. Each entry in this area is also considered to have an implied length of four bytes. It is assumed that this table will be filled with successive entries of results computed during processing.

To use TABL as a displacement in an RX instruction, it must be made absolute to retain the relocatability of the program. TABL must be equated with an <u>absolute</u> expression that references the location counter.

<u>Example</u>:

```
TABL  EQU  *-NULL
```

NULL represents the value of 0 to avoid altering the address of TABL, assigned by the Basic Assembler program, when TABL is being defined. In addition, to make *-NULL an absolute expression, NULL must represent a <u>relocatable</u> 0. This can be done as follows:

```
     START  340
NULL EQU    *-340
```

Here, the expression in the operand of the EQU statement becomes relocatable because it contains an odd number of relocatable expressions.

Adopting the above procedures, the program routine could be as shown in Figure 21.

SI FORMAT

This format is used to load immediate data that are specified in the instruction into storage.

In Basic Assembler language, such instructions are written as:

Op-Code	D1(B1),I2	in case of explicit addressing, and
Op-Code	S1,I2	in case of implied addressing.

Name	Operation	Operand	Comments
PBL1	START	340	
NULL	EQU	*-340	0=RELOCATABLE
BGN	BASR	14,0	LOAD BASE REGISTER
	USING	*,14	ASSIGN BASE REGISTER
	SR	8,8	INITIALIZE INDEX REGISTER
PRGM	MVC	X(5),Y	START COMPUTATION
	|		
	|		
	V		
	STH	10,TABL(0,8)	RESULT INTO TABLE
	AH	8,INCR	INCREMENT INDEX REGISTER
	CH	8,LIMT	TEST FOR TABLE END
	BC	4,PRGM	COMPUTE ANOTHER RESULT
	ED	PRTA,MASK	INITIATE PRINTOUT
	|		
	|		
	V		
TABL	EQU	*-NULL	TABL=ABSOLUTE
	DS	120CL4	DEFINE TABLE AREA
PRTA	DS	120CL4	DEFINE PRINT AREA
X	DS	10H	DEFINE AREA X
Y	DC	H'00000175'	DEFINE AREA Y
INCR	DC	H'4'	DEFINE INCREMENT VALUE
LIMT	DC	H'480'	DEFINE TABLE LIMIT
	END	PBL1	

Figure 21. Sample Program Using TABL as Displacement

The field I2 represents the immediate data, which can be any single self-defining term with a maximum length of 8 bits.

Examples:

```
CLI  JACK,C'6'
TM   MIND,X'40'
MVI  PARA,X'AF'
```

Some of the input/output instructions the programmer uses to write his own I/O routines are also in the SI format. The field I2 in this case is designated UF and is used to specify the I/O unit and its function.

Accordingly, these instructions are written as follows:

```
Op-Code  D1(B1),UF
```

or

```
Op-Code  S1,UF
```

The Set Program Status Word (SPSW) instruction causes the current program status word to be replaced by a new PSW stored at the position referenced in the operand of the SPSW instruction. Since the current PSW contains the address of the next sequential instruction to be processed, the SPSW instruction is equal to a branch instruction.

In Basic Assembler language, this instruction is written as follows:

```
Op-Code  D1(B1) or S1
```

Example:

Name	Operation	Operand
	SPSW	NPSW
NPSW	DC	X'0100'
	DC	Y(BEGN)

In this example, the new PSW contained in the field NPSW is transferred to the internal location of the current PSW. The constant X'0100' replaces the leftmost 16 bits and the address constant replaces the rightmost 16 bits of the 4-byte PSW. Then the program branches to the address specified by bit positions 16 to 31 of the new PSW storage position BEGN.

The termination of object program execution is achieved by a Halt-and-Proceed (HPR) instruction. This instruction also belongs to the SI-type formats and is written as shown in the following example:

```
HPR  X'999',0
```

The operation code HPR is translated into the machine code 99 which is displayed in the UL register panels of the CPU. This code also appears in the UL register panels in case of a programmed halt during execution of an assembly.

To indicate that the program has reached the HPR instruction (completion of object program execution), the address X'999' specified in the first operand of this instruction is displayed in the STR register panels of the CPU.

The second operand of the HPR instruction is ignored and, though assembled, has no influence on the program. Normally, zero is specified as the second operand of the HPR instruction. It can be omitted, however, if the comma is written to satisfy syntax requirements.

SS FORMAT

This format is used to cause data flow from one area of storage to another. It requires specification of the field lengths for the data to be acted upon.

With one exception, which is explained later, the SS instructions form two major groups. The first group includes instructions that require specification of length codes for both operands. The second group requires a length specification for the first operand only.

In Basic Assembler language, the first group of instructions is written as follows:

Op-Code	D1(L1,B1),D2(L2,B2)	when explicit addressing is used, or
Op-Code	S1(L1),S2(L2)	when implied addressing is used.

L1 and L2 in the above format designate the length fields. The operation codes belonging to this group are summarized in Figure 19.

Examples:

```
PACK  AREA(9),INPT+5(9)
MVO   400(10,8),RES1(13)
```

The length code of an expression can be omitted if the length of a field is implied in its name.

Example:

Name	Operation	Operand
	CP	FLDA,FLDB
FLDA	DC	C'0000'
FLDB	DC	XL4'0'

Field A and B each have the implied length of four bytes. An explicit length specification, therefore, is redundant. If a symbol with an implied field length is accompanied by an explicit length code, the implied length is disregarded.

In explicit addressing, the length code becomes redundant if the length is implied in the symbol specified as the displacement.

Example:

Name	Operation	Operand
	CP	FLDA(2),AREA(,8)
AREA	DC	CL2'0'

The fields enclosed in parentheses are referred to as sub-fields. In the above example, the first sub-field of the second operand was omitted because the displacement AREA implies the length of two bytes. Note that the comma separating the sub-fields must be specified in spite of the first sub-field having been omitted. Otherwise, the expression in parentheses is assumed to be a length code and the displacement AREA is considered an implied address.

The second group of SS instructions requires the length specification in the first operand only. The operation codes for this group are summarized in Figure 19.

In Basic Assembler language, these instructions are written as follows:

```
Op-Code  D1(L,B1),D2(B2)  when using ex-
                          plicit addressing,
     or

Op-Code  S1(L),S2         when using implied
                          addressing.
```

The length may be explicit or implied, but the comma separating the sub-fields in the first operand must be entered, even if the length code in an explicit address is omitted.

Example:

```
MVC   FLDA(5),WORK(8)
```

The expression 5 in the first operand is evaluated as a length code and the expression 8 in the second operand is considered to be a base register, even though the two operands appear to specify the same items.

The Execute Input/Output (XIO) instruction is written as follows:

```
Op-Code  D1(UF,B1),D2(B2)  when using ex-
                           plicit address-
                           ing, or

Op-Code  S1(UF),S2         when using im-
                           plied addressing.
```

The length code in the first operand is replaced by the unit and functions specification.

Example:

```
XIO  AREA(X'22'),50
```

This instruction causes 50 card columns to be read on the assigned card-reading device. The data is read into the storage location named AREA. For detailed explanations, refer to the section Input/Output Instructions.

TYPES OF MACHINE OPERATIONS

There are 3 types of operations:

1. Binary arithmetic operations.

2. Decimal arithmetic operations.

3. Non-arithmetic operations.

These operations differ not only in their internal logic but also in the format of data, use of registers, and format of instructions.

Some operations set a condition code in bits two and three of the Program Status Word (PSW). This condition code indicates the relationship (less than/greater than, zero, negative, positive etc.) between the two operands as a result of the last operation effecting the condition code setting. For details about the PSW see the SRL publication IBM System/360 Model 20 Functional Characteristics, Form A26-5847.

BINARY ARITHMETIC

Binary arithmetic is used by binary instructions for operands like addresses, indexes, counters, and binary data. The length of each operand is one halfword including the sign. Negative numbers are given in the twos-complement form. The first operand must be in one of the general registers. The other operand may be either in a register or in main storage. For detailed information refer to the SRL publication IBM System/360 Model 20 Functional Characteristics, Form A26-5847.

Data Format

Binary numbers have a fixed length of one halfword=16 bits. The first (leftmost) bit contains the sign, the other 15 bits the binary value. Binary numbers may be stored in one of the general registers or in main storage. In main storage, the address of the left byte must be even.

Binary halfword

Sign	Binary Value
0	1 15

Representation of Numbers

Binary numbers are represented as signed integers. Positive numbers are represented in true form with a 0-bit as sign. Negative numbers are in the twos-complement form with a 1-bit as sign. The twos-complement form is found by reversing each bit (0 to 1 and 1 to 0) and adding a 1 to the rightmost bit.

A zero is always positive by definition. The absolute value of the lowest possible negative number is higher by one than the highest possible positive number.

Highest possible positive number:

```
┌────────────────┐
│0111111111111111│=2¹⁵-1=+32767
└────────────────┘
 0             15
```

Lowest possible negative number:

```
┌────────────────┐
│1000000000000000│=-(2¹⁵)=-32768
└────────────────┘
 0             15
```

Machine Formats of Instructions for Binary Operations

Binary operations are in the RR or RX-Format.

RR-Format

```
┌─────────┬────┬────┐
│Op-code  │R1  │R2  │
└─────────┴────┴────┘
 0       7   11   15
```

R1 indicates a general register containing the first binary number and R2 a general register containing the second binary number. R1 and R2 may refer to the same register. The result of an instruction in the RR-Format replaces the first operand.

RX-Format

```
┌─────────┬────┬────┬────┬────────────┐
│Op-code  │R1  │0000│B2  │     D2     │
└─────────┴────┴────┴────┴────────────┘
 0       7   11   15   19 20        31
```

R1 indicates a general register containing the first operand. The address of the second operand is indicated by the fields B2 and D2 in one of two ways. Either they give the address directly ($0 \leq B2 \leq 3$) or an effective address is formed by adding the contents of the register named in the B2-field ($8 \leq B2 \leq 15$) to the relative address given in the D2-field.

The result of an operation in the RX-Format replaces the first operand. Exception: After "Store Halfword" the result replaces the second operand.

Condition Code After Binary Operations

Condition code	00	01	10	11
Add register	zero	<zero	>zero	-
Subtract register	zero	<zero	>zero	-
Compare halfword*	equal	low	high	-
Add halfword	zero	<zero	>zero	-
Subtract halfword	zero	<zero	>zero	-

*first operand compared to second.

Binary Arithmetic Error Conditions

Error conditions that may occur during the execution of binary operations are:

1. Operation code invalid.
2. Addressing error:
   a. An instruction address or an operand address refers to the protected first 144 bytes of main storage (addresses 0 to 143).
   b. An instruction address or an operand address is outside available storage.
   c. The last (highest) main-storage position contains any part of an instruction that is to be executed.
   d. The R1 or R2 fields of a binary instruction contain binary values 0 through 7.

3. Specification error:

   a. The low-order bit of an instruction address is one, i.e., no halfword boundary.
   b. The halfword second operand is not located on a halfword boundary.
   c. Bits 12 through 15 of an RX format instruction are not all zero.

4. Binary overflow check.

5. CPU parity error.

INSTRUCTIONS FOR BINARY ARITHMETIC

Name	Op-code	Format	Mnemonic
Add Register	1A	RR	AR
Subtract Register	1B	RR	SR
Store Halfword	40	RX	STH
Load Halfword	48	RX	LH
Compare Halfword	49	RX	CH
Add Halfword	4A	RX	AH
Subtract Halfword	4B	RX	SH

Add Register

Format: RR Op-code 1A

Machine instruction: AR R1,R2

Function: The contents of the first operand field are added to the contents of the second operand field. The result is stored in the register specified by the first operand. The second operand remains unchanged.

The sign is determined by the rules of algebra. A zero result is always positive. A sum consisting of more than 15 numeric bits plus the sign causes an overflow. In detail, this is what happens: First all 16 bits of both operands are added. The result is correct if the addition results in a carry out of both the sign-bit position and the high order numeric-bit position or in no carry at all. However, if the addition causes a carry out of only one of the two positions a binary overflow will take place.

Note: An overflow will change the sign of the result.

Condition Code:

00	Result=zero
01	Result<zero
10	Result>zero

Example: Assume register 8 contains hexadecimal 0123 and register 9 contains hexadecimal 0532.

Source statement:

```
Op-code    R1 R2
----------------
     AR      8,9
```

From this source statement the Basic Assembler creates the following object code:

Op-code	R1	R2
1A	8	9

After execution register 8 contains hexadecimal 0655. The condition code is 10.

Subtract Register

Format: RR Op-code 1B

Machine instruction: SR R1,R2

Function: The contents of the second operand field are subtracted from the contents of the first operand field. The result will be in the register specified by R1. Both operands and the result consist of 15 numeric bits plus the sign. The second operand remains unchanged.

The subtraction is performed by adding the twos-complement of the second operand to the first operand. All 16 bits of both operands are added. If this results in a carry out of both the sign-bit position and the high order numeric-bit position or in no carry at all, then the result is correct. If there is, however, a carry out of only one of the two positions a binary overflow will occur.

A register may be cleared to zero by subtraction from itself.

There is no twos-complement for the highest negative number. This number remains unchanged when a complementation is performed. Nonetheless, the subtraction is still executed correctly.

Condition Code:

00	Result=zero
01	Result<zero
10	Result>zero

Example: Assume register 8 contains hexadecimal 047F and register 9 contains hexadecimal 00D7.

Source statement:

```
Op-code R1 R2
-------------
 SR      8,13
```

From this source statement the Basic Assembler creates the following object code:

Op-code	R1	R2
1B	8	D

After execution register 8 contains hexadecimal 03A8. The condition code is 10.

Store Halfword

Format: RX Op-code 40

Machine instruction: STH R1,D2(0,B2)

Function: The contents of the register specified by R1 are stored in the halfword at the main-storage location addressed by B2 and D2. The first operand remains unchanged.

Condition Code: No change.

Example: Assume register 9 contains hexadecimal 68AF, register 11 contains hexadecimal 001E, and the displacement in the second operand is hexadecimal 29E (decimal 670).

Source statement:

```
Op-code R1 D2 X2=0 B2
---------------------
 STH     9,670(0,11)
```

From this source statement the Basic Assembler creates the following object code:

Op-code	R1	X2=0	B2	D2
40	9	0	B	29E

After execution the field starting at storage location hexadecimal 2BC (decimal 700) contains 68AF.

Load Halfword

Format: RX Op-code 48

Machine instruction: LH R1,D2(0,B2)

Function: The halfword at the main storage location addressed by B2 and D2 is placed into the 16 bit positions of the register specified by R1. The second operand remains unchanged.

Condition Code: No change.

Example: Assume register 9 contains hexadecimal AAAA, register 12 contains 0032, the displacement in the second operand is 1F4 (decimal 500), and the field starting at storage location hexadecimal 226 (decimal 550) contains 80AF.

Source statement:

```
Op-code R1 D2 X2=0 B2
---------------------
 LH      9,500(0,12)
```

From this source statement the Basic Assembler creates the following object code:

Op-code	R1	X2=0	B2	D2
48	9	0	C	1F4

After execution register 9 contains hexadecimal 80AF.

Compare Halfword

Format: RX Op-code 49

Machine instruction: CH R1,D2(0,B2)

Function: The 16 bits of the register specified by R1 are compared with the halfword at the main storage location addressed by B2 and D2. The comparison is algebraic, i.e., the signs must be taken into consideration. Both operands remain unchanged. A condition code is set.

Condition Code:

00	First operand=second operand
01	First operand<second operand
10	First operand>second operand

Example: Assume register 9 contains hexadecimal 0001, the displacement in the second operand is hexadecimal 690 (decimal 1680), and register 13 contains hexadecimal 0026, and the halfword at storage location hexadecimal 6B6 is AF99.

Source statement:

```
Op-code R1 D2   X2=0 B2
-----------------------
 CH      9,1680(0,13)
```

From this source statement the Basic Assembler creates the following object code:

Op-code	R1	X2=0	B2	D2
49	9	0	D	690

After comparison the resulting condition code setting will be: 10.

Add Halfword

Format: RX Op-code 4A

Machine instruction: AH R1,D2(0,B2)

Function: The halfword in main storage, addressed by B2 and D2, is added to the 16 bits of the register specified by R1. The sign is determined by the rules of algebra. A zero result is positive by definition.

If the resulting sum is larger than 15 bits plus the sign, an overflow occurs. All 16 bits of both operands are added. If there is a carry out of both the sign-bit position and the high-order numeric bit position or if there is no carry at all, the result is correct. A binary overflow will occur if there is a carry out of only one position. A condition code is set.

Condition Code:

00	Result=zero
01	Result<zero
10	Result>zero

Example: Assume register 9 contains hexadecimal 047F, register 11 contains hexadecimal 0028, the displacement in the second operand is 1EA (decimal 490), and the field at storage location hexadecimal 212 (530) contains hexadecimal 1F29.

Source statement:

```
Op-code R1 D2  X2=0 B2
----------------------
 AH      9,490(0,11)
```

From this source statement the Basic Assembler creates the following object code:

Op-code	R1	X2=0	B2	D2
4A	9	0	B	1EA

After execution register 9 contains hexadecimal 23A8 and the condition code is 10.

Subtract Halfword

Format: RX Op-code 4B

Machine instruction: SH R1,D2(0,B2)

Function: This instruction is identical to the Add Halfword instruction with the following exception: The twos-complement of the second operand, addressed by B2 and D2, is added in place of the true value.

Condition Code:

00	Result=zero
01	Result<zero
10	Result>zero

Example: Assume register 9 contains hexadecimal 047F, register 11 contains hexadecimal 0050, the displacement in the second operand is hexadecimal 320 (decimal 800), and the field starting at storage location hexadecimal 370 (decimal 880) contains hexadecimal 00D7.

Source statement:

```
Op-code R1 D2  X2=0 B2
----------------------
 SH      9,800(0,11)
```

From this source statement the Basic Assembler creates the following object code:

Op-code	R1	X2=0	B2	D2
4B	9	0	B	320

After execution register 9 contains hexadecimal 03A8 and the condition code is 10.

DECIMAL ARITHMETIC

Decimal arithmetic can be performed only with data in packed format. Packed format means that there are two digits in one byte except for the low order byte. It contains one digit and the sign.

Data is transferred to and from the external I/O devices in zoned format. Thus, the data has to be packed and unpacked before and after processing respectively. In zoned format, each byte contains a zone in the left halfbyte and a digit in the right halfbyte except the last one which contains the sign and a digit. The address in an instruction always specifies the left-most byte of the operand. The length field in an assembled instruction indicates how many bytes are part of the operand in addition to the addressed (left) byte.

Data Format

Decimal operations are performed in main storage. The operands have a length from

1-16 bytes. A field may start at any address including an odd one. In zoned format there may be a maximum of 16 digits, in the packed format a maximum of 31 digits plus the sign in a field. The two operands may be of different length. Multiplicand and divisor are restricted to a maximum of 15 digits plus the sign.

The values in the operand fields are assumed to be right aligned, with leading zeros where required. The operands are processed as integers from right to left. If a result extends beyond the field indicated by the address and the length field, the extending (high order) part is ignored and the condition code is set to 11.

Fields specified in a decimal-arithmetic instruction may overlap only if the rightmost bytes coincide. Exception: with the ZAP instruction an overlap to the right is permissable.

Representation of Numbers

Decimal numbers consist of binary coded digits and a sign. The decimal digits 0-9 are represented in the four bit code by the bit combinations 0000-1001. The combinations 1010-1111 are reserved for representations of a sign (+,-). 1011 and 1101 represent a minus, the other four combinations a plus. The representations 1100, 1101, 1010, and 1011 are created during calculations in main storage. Negative numbers are represented in true form. The two decimal formats are:

Packed decimal number

Byte		Byte		Byte	
Digit	Digit	Digit	Digit	Digit	Sign

Zoned decimal number

Byte		Byte		Byte	
Zone	Digit	Zone	Digit	Sign	Digit

Machine Formats of Instructions for Decimal Arithmetic

Decimal operations have the SS format:

SS-Format

Op-code	L1	L2	B1	D1	B2	D2
0	7 11	15	19	31	35	47

The fields B1 and D1 give the main-storage address of the left byte of the first operand field; L1 gives its length. In the Basic Assembler created object code, the number of bytes in a field is equal to the length code minus one.

The instruction fields B2, D2, and L2 give the respective information for the second operand.

The address of the leftmost byte is found by adding the contents of the register specified in the B-field and the contents of the D-field.

The result of a decimal operation replaces the first operand. It cannot occupy more storage area than indicated in the B,D, and L fields. The second operand remains unchanged. Exception: overlapping fields.

The general registers are not affected by decimal operations.

Condition Code after Decimal Operations

The results of the decimal operations listed in the table below set a condition code.

	00	01	10	11
ZAP	zero	<zero	>zero	-
CP*	equal	low	high	-
AP	zero	<zero	>zero	overflow
SP	zero	<zero	>zero	overflow

*First operand compared to second.

All other decimal operations leave the condition code unchanged.

Decimal Arithmetic Error Conditions

The following error conditions may occur during the execution of decimal arithmetic operations:

1. Operation code invalid.

2. Addressing error:

   a. An instruction address or an operand address refers to the protected first 144 bytes of main storage.
   b. An instruction address or an operand address is outside available storage.
   c. An instruction occupies the last two (highest) main-storage positions.

3. Specification error:

   a. The low-order bit of an instruction address is one, i.e., no halfword boundary.
   b. For Zero and Add, Compare Decimal, Add Decimal, and Subtract Decimal instructions the length code L2 is greater than the length code L1.
   c. For Multiply Decimal and Divide Decimal instructions, the length code L2 is greater than 7 or greater than or equal to the length code L1.

4. Data error:

   a. A sign or digit code of an operand in the Zero and Add, Compare Decimal, Add Decimal, Subtract Decimal, Multiply Decimal, or Divide Decimal instruction is incorrect, or the operand fields in these instructions overlap incorrectly.
   b. The first operand in a Multiply Decimal instruction has insufficient high-order zeros.

5. Decimal divide check:
   The resultant quotient in a Divide Decimal instruction exceeds the specified data field instruction (including division by zero) or the dividend has no leading zero.

6. CPU parity error.

INSTRUCTIONS FOR DECIMAL ARITHMETIC

Name	Op-code	Format	Mnemonic
Move with Offset	F1	SS	MVO
Pack	F2	SS	PACK
Unpack	F3	SS	UNPK
Zero and Add	F8	SS	ZAP
Compare Decimal	F9	SS	CP
Add Decimal	FA	SS	AP
Subtract Decimal	FB	SS	SP
Multiply Decimal	FC	SS	MP
Divide Decimal	FD	SS	DP

Move with Offset

Format: SS Op-code F1

Machine instruction:
MVO D1(L1,B1),D2(L2,B2)

Function: The contents of the second operand field are moved to the location specified by the first operand. The move is executed with an offset of half a byte (one digit) to the left. The right half-byte of the first operand remains unchanged. There is no check for validity. The fields need not have equal lengths. Leading zeros are inserted if the first operand is longer than the second. If the second operand is longer than the first, the high-order digits of the second operand are ignored.

The move proceeds from right to left one byte at a time. The second operand may overlap the first excluding the rightmost byte of the first operand.

Condition Code: No change.

Example: Assume register 12 contains hexadecimal 0250, register 15 contains hexadecimal 040F, the displacement given in both operands is zero, storage location hexadecimal 40F contains hexadecimal 123456, and storage location hexadecimal 250 contains hexadecimal 77 88 99 0C.

Source statement:

```
Op-code D1 L1 B1D2 L2 B2
-------------------------
 MVO    0(4,12),0(3,15)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L1	L2	B1	D1	B2	D2
F1	3	2	C	000	F	000

After execution the field at location hexadecimal 250-253 contains hexadecimal 01 23 45 6C.

Pack

Format: SS Op-code F2

Machine instruction:
PACK D1(L1,B1),D2(L2,B2)

Function: The unpacked content of the second operand field is packed and placed into the first operand field. The second operand field must contain an unpacked decimal number. It may have a maximum size of 16 bytes. There is no check for validity of digits and sign.

The lengths of the fields need not be equal. Leading zeros are inserted if the first operand field is too long for the result. The high-order digits of the second operand are ignored if the first operand field is too short for the result. The fields are processed from right to left one byte at a time.

Condition Code: No change.

Example: Assume register 11 contains hexadecimal 044A, register 9 contains hexadecimal 02C0, the displacement in the first operand is hexadecimal 244, in the second operand it is hexadecimal 180, and that storage location hexadecimal 440-444 contains hexadecimal F1 F2 F3 F4 C5.

Source statement:

```
Op-code D1 L1 B1  D2 L2 B2
--------------------------
 PACK   580(4,11),384(5,9)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L1	L2	B1	D1	B2	D2
F2	3	4	B	244	9	180

After execution the field at storage location hexadecimal 68E contains 00 12 34 5C.

Unpack

Format: SS Op-code F3

Machine instruction:
UNPK D1(L1,B1),D2(L2,B2)

Function: The packed contents of the second operand field are changed to zoned format and stored in the first operand field. The second operand field must contain a packed decimal number. Sign and digits are not checked for validity.

After processing, the zoned decimal number in the first operand contains the sign (high-order four bits) and one digit in the rightmost byte. Each of the other bytes contains a zone and a digit.

The fields are processed from right to left. If the first operand field is too long it is filled with leading zeros. If the first operand field is too short to contain all the digits of the second operand, the leading digits are ignored. The operands may overlap but you must exercise caution.

Condition Code: No change.

Example: Assume register 10 contains hexadecimal 0FA0, the displacement in the first operand is hexadecimal FB4, that in the second operand is hexadecimal 65, and location hexadecimal 1004-1007 contains hexadecimal 01 23 45 6D.

Source statement:

```
Op-code D1  L1 B1  D2 L2 B2
---------------------------
 UNPK   4020(5,10),100(4,10)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L1	L2	B1	D1	B2	D2
F3	4	3	A	FB4	A	65

After execution location hexadecimal 1F54-1F58 contains F2 F3 F4 F5 D6.

Zero and Add Packed

Format: SS Op-code F8

Machine instruction:
ZAP D1(L1,B1),D2(L2,B2)

Function: The first operand field is zeroed out and the contents of the second operand field are placed into the first operand field. This operation is equivalent to an addition into a zero-field. The second operand must be in packed format.

A zero result is positive by definition. The second operand may be shorter than the first operand. If the second operand is longer, then a machine stop occurs and the instruction is not executed.

Processing proceeds from right to left. All digits and the sign of the second operand are checked for validity. High order zeros are supplied if needed. The fields may overlap if the rightmost byte of the first operand is coincident with, or to the right of, the rightmost byte of the second operand.

Condition Code:

00	Result=zero
01	Result<zero
10	Result>zero

Example: Assume register 10 contains hexadecimal 01F4, the displacement in the first operand is hexadecimal 294, that in the second operand is hexadecimal 37A, and storage location hexadecimal 56E contains 01 23 4D.

Source statement:

```
Op-code D1 L1 B1  D2 L2 B2
--------------------------
 ZAP    660(4,10),890(3,10)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L1	L2	B1	D1	B2	D2
F8	3	2	A	294	A	37A

After execution location 487-48A contains 00 01 23 4D.

Compare Decimal Packed

Format: SS Op-code F9

Machine instruction:
CP D1(L1,B1),D2(L2,B2)

Function: The contents of the first operand field are compared to the contents of the second operand field and the result is indicated by a new condition code.

The comparison proceeds from right to left and is algebraic, i.e. the sign and all digits are compared one byte at a time. (Negative values are smaller than positive values).

A negative zero is equal to a positive zero. The sign and all digits are checked for validity. A halt occurs if the second operand field is longer than the first operand field and the instruction is not executed. If the second operand field is shorter it is extended with leading zeros.

The contents of both operand fields do not change. An overflow cannot occur. The two fields may overlap if the rightmost bytes coincide. Therefore, it is possible to compare a number to itself.

Note the difference between "Compare Decimal Packed" and "Compare Logical Characters" (CLC).

CP: comparison proceeds from right to left, the sign, zero, and invalid characters are considered, and fields of unequal length are extended.

CLC: Comparison proceeds from left to right, the sign and invalid characters are not considered.

Condition Code:

00	First operand=second operand
01	First operand<second operand
10	First operand>second operand

Example: Assume register 12 contains hexadecimal 0040, register 11 contains hexadecimal 02F0, the displacement in the first operand is hexadecimal 640, that in the second operand is hexadecimal 3E8, location hexadecimal 680-682 contains 01000C, and location 6D8-6D9 contains 99 99C.

Source statement:

```
Op-code D1  L1 B1   D2 L2 B2
----------------------------
 CP     1600(3, 12),1000(2,11)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L1	L2	B1	D1	B2	D2
F9	2	1	C	640	B	3E8

After compariso the condition code is 10.

Add Decimal Packed

Format: SS Op-code FA

Machine instruction:
AP D1(L1,B1),D2(L2,B2)

Function: The contents of the second operand field are added to the contents of the first operand field. The result replaces the first operand.

The sign is determined by the rules of algebra. A zero result is positive by definition. Exception: It is possible that a remaining zero result after an overflow has a negative sign. A condition code is set.

If the second operand field is longer than the first a program error halt occurs and the instruction is not executed. If the second operand field is shorter than the first it is expanded with leading zeros and addition will take place normally. Signs and digits are checked for validity. Addition proceeds from right to left. The result is in packed format.

The two fields may overlap if the rightmost bytes coincide. Thus, it is possible to double a number.

Condition Code:

00	Result=zero
01	Result<zero
10	Result>zero
11	Overflow

Example: Assume register 8 contains hexadecimal 0014 storage location 329 (hexadecimal) contains 00 22 2D, storage location 500 (hexadecimal) contains 01 00 0C, the displacement in the first operand is 315 (hexadecimal), and that in the second operand is 4EC (hexadecimal).

Source statement:

```
Op-code D1 L1 B1  D2 L2 B2
__________________________
 AP     789(3,8),1260(3,8)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L1	L2	B1	D1	B2	D2
FA	2	2	8	315	8	4EC

After execution storage location 329 (hexadecimal) contains 00 77 8C.

Subtract Decimal Packed

Format: SS Op-code FB

Machine instruction:
SP D1(L1,B1),D2(L2,B2)

Function: The contents of the second operand field are subtracted from the contents of the first operand field. The result is placed into the first operand field. The sign is determined by the rules of algebra. A zero result is positive by definition. Exception: A zero result remaining in case of an overflow may possibly have a minus sign.

If the second operand field is longer than the first a program error halt occurs and the instruction is not executed. If the second operand field is shorter it is expanded with zeros and subtraction will take place normally.

All digits and the signs are checked for validity. The operation proceeds from right to left by reversing the sign of the second operand and then adding the second operand to the first. The result is in packed format.

The fields may overlap if the rightmost bytes coincide. Thus it is possible to clear a field to zero.

Condition Code:

00	Result=zero
01	Result<zero
10	Result>zero
11	Overflow

Example: Assume register 9 contains (hexadecimal) 00C8, register 8 contains (hexadecimal) 012C, storage location 898 (hexadecimal) contains 012C, storage location CE4 (hexadecimal) contains 008C, the displacement in the first operand is 7D0 (hexadecimal), and that in the second operand is BB8 (hexadecimal).

Source statement:

```
Op-code  D1 L1 B1 D2 L2 B2
__________________________
 SP    2000(2,9),3000(2,8)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L1	L2	B1	D1	B2	D2
FB	1	1	9	7D0	8	BB8

After execution storage location 898 (hexadecimal) contains 00A0. The condition code is 10.

Multiply Decimal Packed

Format: SS Op-code FC

Machine instruction:
MP D1(L1,B1),D2(L2,B2)

Function: The multiplicand in the first operand field is multiplied by the multiplier in the second operand field. The product is placed into the first operand field. The second operand may have a maximum of 15 digits (L2=7) plus the sign and must be shorter than the first operand. If L2 > 7 or L2 ≥ L1 a program error halt occurs and the instruction is not executed.

The length of the product is equal to the sum of the lenqths of multiplier and multiplicand. Therefore the multiplicand must be expanded with leading zeros by the number of bytes of the multiplier. Otherwise a halt occurs. An overflow is not possible. The product may have a maximum length of 30 digits plus the sign. It contains at least one leading zero.

The factors and the result are considered to be signed integers. The sign is determined by the rules of algebra. The operand fields may overlap if their rightmost bytes coincide. Thus, it is possible to square a number.

Note: You can save computing time by using the larger of the two factors as the second operand.

Condition Code: No change.

Example:

1. Multiplicand x multiplier = product
   MAND x MOR = PROD
2. Length MAND + length MOR = length PROD
3. The MAND must be right-aligned and have leading zeros before the multiplication is executed.

	Name	Operation	Operand
		.	
		.	
		.	
1		ZAP	PROD,MAND
2		MP	PROD,MOR
		.	
		.	
		.	
	MOR	DS	CL3
	MAND	DS	CL2
	PROD	DS	CL5
		.	
		.	
		.	

Assume the Basic Assembler has allocated storage location (hexadecimal) 1C92 to statement MOR. Then, MAND has location 1C95 amd PROD has location 1C97. Further assume that the storage locations implicitly addressed by MOR and MAND contain 37219D and 425C respectively and register 12 contains (hexadecimal) 1194. (The Basic Assembler automatically calculates the displacement shown in the object coding by subtracting the contents of register 12 from the address value of the implicit address).

Source statement:

```
Op-code D1 L1 B1 D2 L2 B2
-------------------------
 ZAP         PROD,MAND
```

Basic Assembler produced object code:

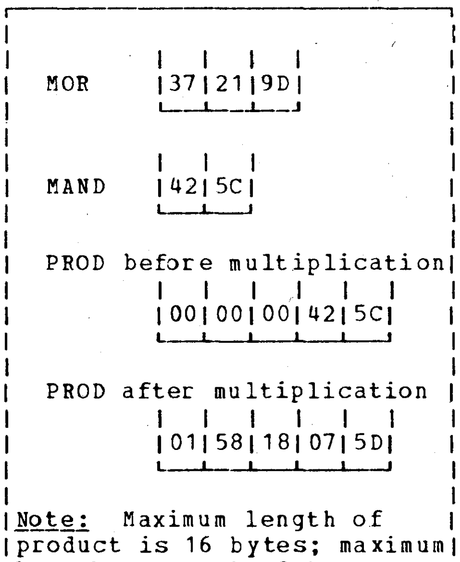
Op-code	L1	L2	B1	D1	B2	D2
F8	4	1	C	B03	C	B01

and

```
Op-code D1 L1 B1 D2 L2 B2
-------------------------
 MP          PROD,MOR
```

Op-code	L1	L2	B1	D1	B2	D2
FC	4	2	C	B03	C	AFE

The result of the two instructions is shown in Figure 22.



Note: Maximum length of product is 16 bytes; maximum length of MOR is 8 bytes.

Figure 22. Decimal Multiplication

Divide Decimal Packed

Format: SS Op-code FD

Machine instruction:
DP D1(L1,B1),D2(L2,B2)

Function: The dividend in the first operand field is divided by the divisor in the second operand field. The quotient and the remainder are placed into the first operand field.

The quotient occupies the left part of the first operand, i.e. the address of the quotient is the same as the address of the dividend. The remainder occupies the right part of the first operand and has a length equal to that of the divisor.

The quotient and the remainder together occupy the entire dividend field (first operand). This means the dividend field must be large enough to accomodate a divisor of maximum length and a quotient of maximum length. In the extreme case the dividend field has to be expanded with zeros to the left by the number of bytes of the divisor.

The length of the quotient field (in bytes) is L1-L2. The divisor field may have a maximum of 15 digits plus the sign and must be smaller than the dividend field.

If L2 > 7 or L2 ≥ L1 a halt occurs and the operation is not executed. The dividend must have at least one leading zero or a halt occurs and the operation is not executed.

Dividend, divisor, quotient, and remainder are signed integers. The sign is determined according to the rules of algebra from the signs of dividend and divisor. The sign of the remainder is always identical to the sign of the dividend. This also holds true if the quotient or the remainder are zero.

If the quotient contains more than 29 digits plus the sign, or if the dividend has no leading zero, then a halt occurs and the operation is not executed. The divisor and the dividend remain unchanged and there is no overflow. The two operands may overlap if their rightmost bytes coincide.

Condition code: No change.

Example:

1. Dividend : Divisor = Quotient
   DEND : DOR = QUOT

2. Length of processing field = length QUOT + length DOR

   maximum length of processing field (PROFE) = length DEND + length DOR (packed bytes).

3. The dividend must be right-aligned with at least one leading zero before the division is performed.

Name	Operation	Operand
	.	
	.	
	.	
	ZAP	PROFE,DEND
	DP	PROFE,DOR
	.	
	.	
	.	
DEND	DS	CL4
DOR	DS	CL2
PROFE	DS	CL5
	.	
	.	
	.	

Assume the Basic Assembler has allocated storage locations as follows: DEND hexadecimal A09, PROFE hexadecimal F40, and DOR hexadecimal CAC. Register 9 contains hexadecimal 0400. The Basic Assembler automatically calculates the displacements for the two operands by subtracting the contents of register 9 from the respective storage address values. The source and object codings for the ZAP and DP are:

Source statement:

```
Op-code D1 L1 B1 D2 L2 B2
-------------------------
 ZAP       PROFE,DEND
```

Basic Assembler produced object code:

Op-code	L1	L2	B1	D1	B2	D2
F8	4	3	9	758	9	609

and

Source statement:

```
Op-code D1 L1 B1 D2 L2 B2
-------------------------
 DP        PROFE,DOR
```

Basic Assembler produced object code:

Op-code	L1	L2	B1	D1	B2	D2
FD	4	1	9	758	9	8AC

The results of the two instructions are shown in Figure 23.

```
┌──────────────────────────────┐
│                              │
│   DEND   │27│95│34│3C│       │
│                              │
│   PROFE  │00│27│95│34│3C│    │
│                              │
│   DOR    │21│3C│             │
│                              │
│   Quotient:                  │
│   PROFE  │13│12│3C│14│4C│    │
│                              │
│Quotient and remainder each   │
│have their own sign.          │
│Note:  Maximum length of      │
│quotient is 16 bytes; maxi-   │
│mum length of DOR is 8        │
│bytes.                        │
└──────────────────────────────┘
```

Figure 23. Decimal Division

NON-ARITHMETIC OPERATIONS

There are special instructions for the non-arithmetic processing of data. The operands are processed one byte at a time. In some cases the left four bits and the right four bits of a byte are treated separately.

Processing of data fields in main storage proceeds from left to right. A field may start at any address excluding the reserved areas.

In non-arithmetic operations the operand fields are considered to contain alphameric data. An exception is the Edit-instruction which requires packed decimal numbers in the second operand field.

Data Format

The data are either in main storage or in the instruction itself. They may be a single character or an entire field. If two operands are used they must be of equal length. Exception: the Edit-instruction. The two formats for non-arithmetic data are:

Fixed Length

```
┌─────────┐
│single   │
│character│
└─────────┘
 0       7
```

Variable Length

```
┌─────────┬─────────┬──        ──┬─────────┐
│character│character│            │character│
└─────────┴─────────┴──        ──┴─────────┘
 0       7 8      15
```

In storage-to-storage (SS) operations, the fields may start at any address with exception of the first 144 bytes, which are reserved. The maximum length of a field is 256 bytes. Immediate data is limited to a length of one byte.

The EDIT operation only handles data of packed format. The other instructions handle all bit combinations.

Storage-to-storage instructions may have overlapping operands. The result of overlapping depends on the particular operation. Overlapping does not influence the operation if the operands remain unchanged (e.g. in a comparison). If one or both change, however, execution of the operation may be influenced by the overlapping and by the manner in which the data are rounded off and stored.

Machine Formats of Instructions for Non-Arithmetic Operations

Non-arithmetic instructions are either in the SI- or the SS-format.

SI-Format

```
┌───────┬────────┬───┬─────────┐
│Op-code│   I2   │B1 │   D1    │
└───────┴────────┴───┴─────────┘
 0     7       15  19        31
```

The address of the first operand field is the sum of the contents of the B1-and D1-fields. The operand has a length of one byte. The second operand also has a length of one byte but it is contained directly in the instruction. The result is placed into the first operand field. The general registers are not affected by an SI-instruction.

SS-Format

```
┌───────┬────────┬───┬─────────┬───┬─────────┐
│Op-code│   L    │B1 │   D1    │B2 │   D2    │
└───────┴────────┴───┴─────────┴───┴─────────┘
 0     7       15  19        31  35        47
```

The address of the each operand field is the sum of the contents the respective B- and D-fields. The first and second operand fields must have the same length.

The result of an operation in the SS-Format is placed into the first operand field. The contents of the general registers remain unchanged.

Condition Code After Non-Arithmetic Operations

The results of the operations determine the condition code. Move-operations do not set a code. In case of the EDIT-instruction the condition code indicates the status of the field to be transferred into the mask.

Table of condition codes:

	00	01	10	11
Test under Mask	zero	mixed	--	one
And	zero	not zero	--	--
Compare Logical	equal	low	high	--
Or	zero	not zero	--	--
Edit	zero	<zero	>zero	--

Error Conditions

Error conditions which may occur during the execution of non-arithmetic operations are:

1. Operation code invalid

2. Addressing error
   a. An instruction address or an operand address refers to the protected first 144 bytes of main storage (addresses 0 to 143).
   b. An instruction address or an operand address is outside available storage.
   c. The last (highest) main-storage position contains any part of an instruction that is to be executed.

3. Specification error
   The low-order bit of an instruction address is one, i.e., no halfword boundary.

4. Data error
   An invalid digit code is contained within the second operand field of an Edit operation.

5. CPU parity error.

INSTRUCTIONS FOR NON-ARITHMETIC OPERATIONS

Name	Format	Operation Code
Move Immediate (MVI)	SI	92
Move Characters (MVC)	SS	D2
Move Numerics (MVN)	SS	D1
Move Zones (MVZ)	SS	D3
Compare Logical (CLI)	SI	95
Compare Logical (CLC)	SS	D5
Edit (ED)	SS	DE
And (NI)	SI	94
Or (OI)	SI	96
Test under Mask (TM)	SI	91
Halt & Proceed (HPR)	SI	99
Translate (TR)	SS	DC

Move Immediate

Format: SI Op-code 92

Machine Instruction:
```
MVI   D1(B1),I2
```

Function: ~~The byte from I2 is placed directly into the storage location addressed by B1 and D1.~~

Condition Code: No change.

Example: Assume register 10 contains (hexadecimal) 082E, storage location A22 (hexadecimal) contains A, the displacement in the first operand is 1F4, and the immediate data is the $.

Source statement:

```
Op-code D1 B1  I2
-----------------
 MVI    500(10),C'$'
```

From this source statement the Basic Assembler produces the following object code:

Op-code	I2	B1	D1
92	5B	A	1F4

After execution storage location A22 contains $.

Move Characters

Format: SS Op-code D2

Machine instruction:
```
MVC   D1(L,B1),D2(B2)
```

Function: The contents of the second operand field are placed into the first operand field. Processing is performed from left to right one byte at a time.

The two operand fields may overlap. If the first operand field is to the left of the second operand field, then transfer will proceed correctly. If the first operand field is exactly one byte to the right of the second operand field, then this byte will be propagated throughout the first operand field.

Condition Code: No change.

Example: Assume register 11 contains (hexadecimal) 0258, register 15 contains (hexadecimal) 04B0, storage location 3E8 (hexadecimal) contains optional data, storage location 7D0 (hexadecimal) contains C9 C2 D4, the displacement in the first operand is 190 (hexadecimal), and that in the second operand is 320 (hexadecimal).

Source statement:

```
Op-code  D1 L B1 D2 B2
----------------------
 MVC     400(3,11),800(15)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L	B1	D1	B2	D2
D2	2	B	190	F	320

After execution storage location 3E8 contains C9 C2 D4.

Move Zones

Format: SS Op-code D3

Machine instruction:
MVZ D1(L,B1),D2(B2)

Function: The high-order four bits (the zones) of each byte in the second operand field are placed into the high-order four bits of the first operand field. The low order four bits (the numerics) of each byte remain unchanged. Movement is from left to right one byte at a time. The digits are not checked for validity. The operand fields may overlap.

Condition Code: No change.

Example: Assume register 10 contains (hexadecimal) 0890, storage location 8F4-8F7 (hexadecimal) contains F4 F3 F2 C1, the displacement in the first operand is 64 (hexadecimal), and that in the second operand is 66 (hexadecimal).

Source statement:

```
Op-code  D1 L B1  D2 B2
-----------------------
 MVZ     100(1,10),102(10)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L	B1	D1	B2	D2
D3	0	A	064	A	066

After execution storage location 8F4-8F7 contains F4 F3 F2 F1.

Move Numerics

Format: SS Op-code D1

Machine instruction:
MVN D1(L,B1),D2(B2)

Function: The low order four bits (the numerics) of each byte in the second operand field are placed, from left to right, into the corresponding low order four bits of the first operand field. The high order four bits (the zones) of each operand remain unchanged. The digits are not checked for validity. The operand fields may overlap.

Condition Code: No change.

Example: Assume register 15 contains (hexadecimal) 7DA, storage location 8A4-8A7 (hexadecimal) contains F4 F3 F2 C1, storage location 96A (hexadecimal) contains F9 F8 F7 D6, the displacement in the first operand is C8 (hexadecimal), and that in the second operand is 190 (hexadecimal).

Source statement:

```
Op-code  D1 L B1  D2  B2
------------------------
 MVN     200(4,15),400(15)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L	B1	D1	B2	D2
D1	3	F	0C8	F	190

After execution storage location 8A4-8A7 contains F9 F8 F7 C6.

Compare Logical Immediate

Format: SI Op-code 95

Machine instruction:
CLI D1(B1),I2

Function: The eight-bit symbol of the immediate-data operand (the second operand) is compared to the contents of the first operand field. The result sets the condition code. The two bytes are treated as eight-bit unsigned binary values. This results in the following order of comparison:

Special characters, lower case letters, upper case letters, digits (System/360 collating sequence).

All 256 bit combinations are valid.

Condition Code:

00:	first operand=second operand
01:	first operand<second operand
10:	first operand>second operand

Example: Assume register 15 contains (hexadecimal) 01F4, storage location 5DC (hexadecimal) contains F9, the displacement in the first operand is 3E8 (hexadecimal), and the immediate data is the letter A.

Source statement:

```
Op-code  D1  B1  I2
____________________
 CLI    1000(15),C'A'
```

From this source statement the Basic Assembler produces the following object code:

Op-code	I2	B1	D1
95	C1	F	3E8

After execution the condition code setting is 10.

Compare Logical Characters

Format: SS Op-code D5

Machine instruction:
CLC D1(L,B1),D2(B2)

Function: The contents of the first operand field are compared with those of the second operand field. The fields may have a maximum length of 256 bytes. The comparison is terminated as soon as inequality is encountered.

All bits are treated alike as part of an unsigned binary quantity. The order of comparison is the System/360 collating sequence:

Special characters, lower case letters, upper case letters, digits.

Comparison proceeds from left to right. All 256 bit combinations are valid.

Condition Code:

00:	first operand=second operand
01:	first operand<second operand
10:	first operand>second operand

Example: Assume register 11 contains (hexadecimal) 0320 storage location AF0-AF3 (hexadecimal) contains D1 D6 C8 D5, storage location 708-70B (hexadecimal) contains D1 D6 C5 E8, the displacement in the first operand is 7D0 (hexadecimal), and that in the second operand is 3E8 (hexadecimal).

Source statement:

```
Op-code  D1  L B1   D2  B2
___________________________
 CLC    2000(4,11),1000(11)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L	B1	D1	B2	D2
D5	3	B	7D0	B	3E8

After having compared the third character the condition code setting will be 10.

Edit

Format: SS Op-code DE

Machine instruction:
ED D1(L,B1),D2(B2)

Function: The format of the source field (the second operand) is changed from packed to zoned and is edited under control of the pattern field (the first operand). The edited result replaces the pattern. The two fields must not overlap. Editing includes sign and punctuation control and the suppressing and protecting of leading zeros. It also facilitates programmed blanking of all-zero fields. Several numbers may be edited in one operation, and numeric information may be combined with alphabetic information. The length field applies to the pattern (the first operand). It may have a maximum of 256 bytes. The pattern has unpacked format and may contain any character. The source (the second operand) has packed format and must contain valid decimal digit-and sign-codes. Its left half-byte must always contain one of the digits 0-9. The right half-byte may be a digit or a sign.

Both operands are processed left to right one character at a time. Overlapping pattern- and source-fields give unpredictable results.

A so-called S-trigger controls the Edit-operation. Depending on various conditions during the operation the trigger is set either to ON or OFF. This setting determines whether a source digit or a fill character is inserted into the result field.

As mentioned before, the pattern may contain any unpacked character. However, three Bit-combinations have special significance:

0010 0000	(hexadecimal 20) = digit-select character
0010 0010	(hexadecimal 22) = field-separator character
0010 0001	(hexadecimal 21) = significance-start character.

The digit-select character indicates a position in the result field into which the corresponding digit of the source field or a fill character is to be inserted.

The field-separator character is used if several source fields are to be inserted into one pattern. By setting the S-trigger to OFF it causes every source field to be treated separately. The field-separator character is always replaced by the fill character.

The significance-start character sets the S-trigger to ON. Now every character in the pattern is replaced by the respective digit of the source field or the fill character.

The S-trigger is set to OFF (0):

1. At the beginning of an Edit-operation.
2. By the field-separator character in the pattern.
3. By a positive sign (1010, 1100, 1110, 1111).

The S-trigger is set to ON (1):

1. By a valid digit (1-9) of the source field.
2. By the significance-start character in the pattern.
3. By a negative sign (1011, 1101).

During the processing of the left half-byte the sign of the right half-byte is checked and set accordingly. If a sign coincides with a valid digit or with a significance-start character in one position of the result field, the the sign takes precedence and the S-trigger is set to OFF (0).

The new S-trigger setting always takes effect with the subsequent position.

The fill character, which under certain conditions, is placed into the result field, is always the first (left) character of a pattern; it is retained in the pattern (exception: the digit-select character and the significance-start character).

The S-trigger in OFF position causes:

1. The digit-select character (hexadecimal 20) and/or the significance-start character (hexadecimal 21) to be replaced by a valid digit (1-9) from the source field.
2. The fill character to be stored in place of a zero in the source field.
3. The fill character to be stored in place of any character in the pattern (exception: the digit select and the significance start characters).

The S-trigger in ON position causes:

1. The digit-select and/or the significance-start character to be replaced by every digit (0-9) from the source field.
2. A character in the pattern to remain unchanged (exception: the digit-select, field-separator, and significance-start characters).

All digits in the result field receive the zone 1111.

<u>Condition Code</u>:
The condition code is set to:

1. <u>00</u> if the source field contains only zeros. The setting of the S-trigger has no effect.
2. <u>01</u> if the source field is not zero and the S-trigger is set to ON (1). (Negative result).
3. <u>10</u> if the source field is not zero and the S-trigger is set to OFF (0). (Positive result).

If several fields are edited with one pattern, then the condition code refers to the field being processed. If the pattern has a field-separator in the last place, then the condition code is set to zero.

The following symbols are used in the example:

Symbol	Meaning
b (hexadecimal 40)	blank character
((hexadecimal 21)	significance-start character
) (hexadecimal 22)	field-separator character
d (hexadecimal 20)	digit-select character

If the number to be edited is a negative number, then the CR (hexadecimal C3D9) is commonly used in the last two bytes of the pattern. Since the minus sign does not reset the S-trigger, the CR will be left unchanged in the pattern. The CR stems from business application. It stands for credit and indicates payments due.

Example: (The numbers are given in decimal notation with the hexadecimal equivalent in parentheses.)

Assume that register 12 contains 1000 (03E8),
D1 is 0 (00),
D2 is 200 (C8),
storage location 1000-1012 (3E8-3F4) contains bdd,dd(.ddbCR (unpacked),
storage location 1200-1203 (4B0-4B3) contained 0257426C (packed).

Source statement:

Op-code	D1 L B1 D2 B2
ED	0(13,12),200(12)

From this source statement the Basic Assembler produces the following object code:

Op-code	L	B1	D1	B2	D2
DE	C	C	000	C	0C8

Processing proceeds left to right one character at a time as shown in Figure 24.

Condition code=10; result greater than zero.

After execution location 1000-1012 (3E8-3F4) contains bb2,574.26bbb.

If the contents of location 1200-1203 are 00 00 02 6D, the following results are obtained:

(before) Loc 1000-1012 (3E8-3F4)
bdd,dd(.ddbCR
(after) Loc 1000-1012 (3E8-3F4)
bbbbbb.26bCR

Condition code=1; result less than zero.

In this case the significance-start character in the pattern causes the decimal point to be left unchanged. The minus sign

Pattern	Digit	S-trigger	Rule	Location 1000-1012
b		0	leave[1]	bdd,dd(.ddbCR
d	0	0	fill	bbd,dd(.ddbCR
d	2	1	digit	bb2,dd(.ddbCR[2]
,		1	leave	same
d	5	1	digit	bb2,5d(.ddbCR
d	7	1	digit	bb2,57(.ddbCR
(	4	1	digit	bb2,574.ddbCR
.		1	leave	same
d	2	1	digit	bb2,574.2dbCR
d	6+	0	digit	bb2,574.26bCR[3]
b		0	fill	same
C		0	fill	bb2,574.26bbR
R		0	fill	bb2,574.26bbb

Figure 24. Processing of Edit-Instruction

Notes:
1. This character is saved as the fill character.
2. First non-zero digit sets S-trigger to one.
3. The plus sign in this byte sets the S-trigger to zero.

does not reset the S-trigger so that the CR symbol is also preserved.

And Immediate

Format: SI Op-code 94

Machine instruction:
NI D1(B1),I2

Function: The immediate data in the second operand field and the contents of the storage location addressed in the first operand field are connected by the logical AND. The result (logical product) is placed into the first operand field.

The connective AND is applied bit by bit. If there is a 1-bit in both operands, then the 1-bit in the first operand remains unchanged. Otherwise the 1-bit in the first operand will be changed to a 0-bit.

Condition Code: If all eight bits in the result field are zero, the condition code is set to 00. Otherwise it is set to 01.

Example: (The numbers are given in decimal notation with the hexadecimal equivalent in parentheses).

Assume that
register 8 contains 4096(1000),
D1 is 1000(3E8),
I2 is 2720(AA), in binary notation: 1010 1010,
location 5096(1060) contains 240(F0), in binary notation: 1111 0000.

Source statement:

```
Op-code  D1 B1  I2
--------------------
 NI     1000(8),X'AA'
```

From this source statement the Basic Assembler produces the following object code:

Op-code	I2	B1	D1
94	AA	8	3E8

After execution storage location 5096(1060) contains 160(A0) or in binary notation 1010 0000.

Condition code setting is 01.

Or Immediate

Format: SI Op-code 96

Machine instruction:
OI D1(B1),I2

Function: The immediate data in the second operand field and the contents of the storage location addressed in the first operand field are connected by the inclusive OR. The result (logical sum) is placed into the first operand field.

The inclusive OR is applied bit by bit. A 0-bit in both operand fields will set the bit in the result field (first operand) to zero. Otherwise the resulting bit will always be one.

Condition Code: If all bits are zero, then the condition code is 00. Otherwise the code is set to 01.

Example: (The numbers are given in decimal notation with the hexadecimal equivalent in parentheses).

Assume that
register 8 contains 4096(1000),
D1 is 1000(3E8),
I2 is 2720(AA), in binary notation: 1010 1010,
storage location 5096(1060) contains 240(F0), in binary notation: 1111 1010.

Source statement:

```
Op-code  D1  B1 I2
-------------------
 OI     1000(8),X'AA'
```

From this source statement the Basic Assembler produces the following object code:

Op-code	I2	B1	D1
96	AA	8	3E8

After execution storage location 5096(1060) contains 250(FA) or in binary notation: 1111 1010.

Condition code is 01.

Test Under Mask

Format: SI Op-code 91

Machine instruction:
TM D1(B1),I2

Function: The bit combination in the first operand field is compared with the mask in the I2-field. The result of the comparison sets the condition code.

The eight bits of the mask correspond bit by bit to the eight bits defined by the first operand. A comparison with a bit in the first operand is performed only if the corresponding bit in the mask contains a "1". If the bit in the mask is "0", the corresponding bit in the first operand field will not be tested.

Condition Code:

00: all bits tested were zero (also, if all bits in the mask were zero, i.e., no test).
01: some (not all) of the bits tested were one.
11: all bits tested were one.

Example: (The numbers are given in decimal notation with the hexadecimal equivalent in parentheses).

Assume that
register 8 contains 2000(07D0),
D1 is 650(28A),
I2 is 217(D9) or in binary notation:
1101 1001,
storage location 2650(A5A) contains 204(CC) or in binary notation: 1100 1100.

Source statement:

```
Op-code  D1 B1  I2
------------------
 TM     650(8),X'D9'
```

From this source statement the Basic Assembler produces the following object code:

Op-code	I2	B1	D1
91	D9	8	28A

Condition code is 01.

Halt and Proceed

Format: SI Op-code 99

Machine instruction:
HPR D1(B1),0

Function: This instruction is used to halt the CPU. All input/output operations are continued to completion.

Execution of the program may be resumed with the next sequential instruction by pressing the Start key on the CPU.

This instruction uses the SI-Format in which the I2 field is ignored. The direct or effective address derived from the B1-D1 fields may be used to identify the Halt and Proceed instruction.

Condition Code: No change.

Example: (The numbers are given in decimal notation with the hexadecimal equivalent in parentheses).

Assume that
register 10 contains 450(01C2),
D1 is 140(080),
The halt number 590(24E) is shown on the E-S-T-R registers on the console as 024E.

Source statement:

```
Op-code  D1 B1 I2
-----------------
 HPR    140(10),0
```

From this source statement the Basic Assembler produces the following object code:

Op-code	I2	B1	D1
99	00	A	08C

Translate

Format: SS Op-code DC

Machine instruction:
TR D1(L,B1),D2(B2)

Function: This operation allows you to replace the values of one operand field by the corresponding values of a table. Every byte in the first operand field is used to look up a value in a table. The binary value of a byte is added to the starting address (given by the B2/D2 field) of the table. The sum is the place of the table-value wanted. This table-value replaces the byte in the first operand used to locate the table-value.

Processing proceeds from left to right until the end of the first operand is reached. The maximum length may be 256. The table must contain as many bytes as indicated by the highest binary value used for searching.

Condition Code: No change.

Example: (The numbers are given in decimal notation with the hexadecimal equivalent in parentheses).

Assume that
register 10 contains 0(0000),
register 12 contains 0(0000),
D1 is 1000 (3E8),

D2 is 2000(7D0),
storage location 1000-1012(3E8-3F4) contains the EBCDIC characters 542156037835 and location 2000-2009(7D0-7D9) contains the EBCDIC characters 6MB0Ib3-2 where b=blank.

Source statement:

```
Op-code D1  L B1   D2  B2
--------------------------
 TR    1000(12,10),2000(12)
```

From this source statement the Basic Assembler produces the following object code:

Op-code	L	B1	D1	B2	D2
DC	0B	A	3E8	C	7D0

After execution storage location 1000-1012 (3E8-3F4) contains the EBCDIC characters bIBMb360-20b where b=blank.

BRANCHING

Normally the CPU processes instructions in the order of their location in main storage. Branching operations allow a departure from this sequence. The machine can make logical decisions on the basis of certain conditions. For example:

- The program continues in its normal sequence.
- The program branches to a subroutine.
- Part of the program is repeated (loop).

The branch address may be obtained from one of the general registers or it may be specified in an instruction. The branch address is independent of the updated instruction address.

Branching is determined either by the condition code in the Program Status Word (PSW) or by the contents of the general registers used in the operations.

During a branching operation the rightmost half of the PSW, the updated instruction address, may be stored before the instruction address is replaced by the branch address. The stored information may be used to link the new instruction sequence with the preceding sequence.

The condition code set by certain instructions and the branch instruction are used to make logical decisions within a program. The branch operation itself does not change the condition code.

Machine Formats of Instructions for Branch Operations

Branching instructions can be in the RR or the RX format.

RR Format

```
┌────────┬────┬────┐
│Op-code │R1  │R2  │
└────────┴────┴────┘
 0       7   11   15
```

The R1 field may specify a general register into which the updated instruction address is to be stored as link information, or may contain a mask which is employed to identify the bit values of the condition code. In the latter case it is referred to as the M1 field.

The R2 field specifies the general register that contains the branch address.

RX Format

```
┌────────┬────┬────┬────┬──────────────┐
│Op-code │ R1 │0000│ B2 │      D2      │
└────────┴────┴────┴────┴──────────────┘
 0       7   11   15   19 20          31
```

The R1 field may specify a general register into which the updated instruction address is to be stored as link information, or may contain a mask (then called M1 field) that is employed to identify the bit values of the condition code.

The direct or effective address derived from the B2-D2 fields is the branch address.

Error Conditions

Error conditions which may occur during a branch operation are:

1. Operation code invalid.
2. Addressing error.
   a. An instruction address or a branch address refers to the protected first 144 bytes of main storage.
   b. An instruction address or a branch address is outside available storage.
   c. The R1 field of a Branch and Store instruction contains binary values zero through seven, or the R2 field of an RR format branch instruction contains binary values one through seven.
   d. An instruction part is located in the last (highest) two main storage positions.

3. Specification error.

   a. The low-order bit of an instruction address is one, i.e., no halfword boundary.
   b. Bits 12 through 15 of an RX format instruction are not all zero.

4. CPU parity error.

INSTRUCTIONS FOR BRANCH OPERATIONS

The branch instructions, their operation codes, formats, and mnemonics are shown the following table:

Name	Format	Op-Code
Branch on Condition (BCR)	RR	07
Branch on Condition (BC)	RX	47
Branch & Store (BASR)	RR	0D
Branch & Store (BAS)	RX	4D

Branch on Condition Register

Format: RR   Op-code   07

Machine instruction:
BCR   M1,R2

Function: The condition code is tested against the four bits in the mask M1. If the condition is met, a branch occurs to the address in main storage specified by R2. Otherwise, the next sequential instruction is executed.

There is a corresponding bit in the mask for each of the four possible condition code settings as shown below:

```
|  07  |     M1      |   R2   |
└──────┼──┬──┬──┬────┼────────┘
       |00|01|10|11|
```

The condition for a branch is met if the mask bit corresponding to the current condition code setting is a 1-bit. It is possible to connect several conditions by specifying a 1-bit in the corresponding mask-bit positions. An unconditional branch occurs if all four bits in the mask are 1-bits. The branch instruction is ignored if all four bits in the mask are 0-bits or if R2 is zero.

Condition code: No change.

Example: Assume register 9 contains decimal 555 (hexadecimal 22B), the condition code in the PSW is 01, and the mask is given as hexadecimal 6.

Source statement:
```
 Op-code  M1  R2
 ---------------
 BCR     X'6',9
```

Basic Assembler produced object code:

Op-code	M1	R2
07	0110	9

A branch to the main storage location 22B will take place.

Branch on Condition

Format RX   Op-code   47

Machine instruction:
BC   M1,D2(0,B2)

Function: The condition code is tested against the mask M1 (four bits). If the condition is met, a branch occurs to the address in main storage specified by B2/D2. Otherwise the next sequential instruction is executed.

For each of the four condition code settings there is a corresponding bit of the mask as shown below:

```
|  47  |     M1      |  0000  | B2 |   D2   |
└──────┼──┬──┬──┬────┼────────┴────┴────────┘
       |00|01|10|11|
```

The condition for a branch is met if the corresponding condition code exists for at least one defined bit in the mask.

It is possible to connect several conditions by defining several bits in the mask accordingly. An unconditional branch occurs if all four bits in the mask are one. The branch instruction is ignored if all four bits in the mask are zero.

Condition Code: No change

Example: Assume that
D2 is 875 decimal (36B hexadecimal),
Register 11 contains 0000,
Condition code in the PSW: 00.

Source statement:
```
Op-code  M1   D2  0 B2
----------------------
 BC     X'8',875(0,11)
```

Basic Assembler produced object code:

Op-code	M1	0	B2	D2
47	8	0	B	36B

A branch to main storage location 36B (hexadecimal notation) takes place (branch on equal).

Branch and Store Register

Format: RR Op-code 0D

Machine instructions:
BASR R1,R2

Function: A branch is taken to the address specified by the contents of the register in the R2-field. Next, the rightmost 16 bits of the PSW (the address of the next sequential instruction before the branch is taken) are loaded into the general register specified in the R1 field. This is to link the new instruction sequence with the preceding sequence. If R2 contains all zeros, then only the next sequential instruction is loaded into the register specified by the R1 field and no branching takes places.

Condition Code: No change.

Example:
The contents of the register 10 are arbitrary.
Assume that register 12 contains hexadecimal 0362 (decimal 866),
PSW 16-31 contains hexadecimal 026D (decimal 621).

Source statement:

Op-code	R1 R2
BASR	10,12

Basic Assembler produced object code

Op-code	R1	R2
0D	A	C

After execution register 10 contains 026D and a branch is taken to storage location 362 (hexadecimal).

Branch and Store

Format: RX Op-code 4D

Machine instruction:
BAS R1,D2(0,B2)

Function: The rightmost 16 bits of the PSW, the updated instruction address, are stored as link information in the general register specified by R1. Next, the address specified by B2/D2 is stored as an instruction address in the PSW. This amounts to a branch to the address specified by B2/D2.

Condition Code: No change.

Example:
The contents of register 10 are arbitrary.
Assume that register 11 contains hexadecimal 044C,
PSW 16-31 contains 036C,
D2 is hexadecimal 12C (decimal 300).

Source statement:

Op-code	R1 D2 0 B2
BAS	10,300(0,11)

Basic Assembler produced object code:

Op-code	R1	X=0	B2	D2
4D	A	0	B	12C

After execution register 10 contains hexadecimal 036C and a branch to storage location hexadecimal 578 is taken.

The Basic Assembler program is available in both card and tape versions.

The card versions are used if only card I/O devices are included in the system configuration. The tape versions can be used if an IBM 2415 Magnetic Tape Unit Model 1 or 4 is available, in addition to the card I/O units.

BASIC ASSEMBLER (CARD VERSIONS)

The card versions require two passes. During the first pass the Basic Assembler program (phase 1) produces pass information required during pass 2. This information is punched into columns 1-24 of the source cards or into the corresponding columns of duplicated source cards. In addition, a listing of all source statements is supplied if a printer is attached to the system and if an appropriate entry has been made in the control card.

During the second pass, the source cards containing the pass information are processed by the Basic Assembler program (phases 2 and 3). Then the symbol table generated in storage is punched into cards, if desired. At the end of the assembly the following output is obtained:

- a Clear-Storage card and an Absolute-Program Loader card for loading of the object program.
- TXT cards containing the source statements, translated into machine language.
- ESD and RLD cards containing information for program linking and relocation.
- A program listing, as shown in Figure 34.

Note: The first three items above are referred to as the object deck.

In order to assemble a source program written in Basic Assembler language, the source deck must be supplemented by a control (CTL) card, specifying the system configuration used for the assembly and the desired output. The CTL card as well as the card handling required during an assembly is described in the SRL publication IBM System/360 Model 20, Card Programming Support, Basic Assembler (Card Versions), Operating Procedures, Form C26-3802.

The control card can also be used to specify a diagnostic run. In this case, the punching of all cards is suppressed. The only output produced is a listing of all statements in Basic Assembler language. Most of the erroneous statements are identified by diagnostic messages.

Error Elimination

For the card versions of the Basic Assembler program, a reassembly feature is provided that permits the reassembly of a partially or completely assembled program in less time than would be required by the repetition of the total assembly. For a reassembly, at least pass 1, phase 1, and pass 2, phase 2 of the Basic Assembler program (i.e., the punching and/or printing of the symbol table) must be completed.

A reassembly can be executed to correct erroneous statements and/or to compensate for a symbol-table overflow, which occurs if the number of symbols specified in the source program exceeds the limit in regard to the storage capacity used. Refer also to the sections Symbols and Expressions.

When a reassembly is to be performed, the same amount of main storage must be specified to the Basic Assembler program as for the original assembly.

The symbol-table overflow can be eliminated by:

- making use of relative addressing, described in the section referenced above, thereby reducing the number of symbols in the program;
- performing an additional assembly run, as described in a subsequent section; or
- subdividing the program into segments and performing a separate assembly for each segment.

A program that is to be reassembled can be changed in any manner. New symbols can be added, existing symbols can be redefined (if there is room in the symbol table), existing symbols can be deleted except from the symbol table, and new statements can be added to the program. A statement that is to be changed must be repunched, leaving columns 1 through 24 blank.

Additional Assembly Run. This increases the number of symbols permitted in regard to the storage capacity used during an assembly.

During pass 2 of the original assembly, the portion of the object deck already assembled is completed. On completion of pass 2, a programmed halt occurs to enable the user to remove this portion of the object deck.

When the system is restarted after an overflow, the Basic Assembler generates a new control card that contains the USING table and the value of the location counter at the time the overflow occurred. After generation of this control card, the remaining portion of the source deck is duplicated.

The duplicated source cards contain the following:

Pass Information (Columns 1-24): For example, a diagnostic message or the punch 12-11-0-7-8, the operation code, and one or more pointers designating the location of storage addresses of related symbols.

Source Statement (Columns 25-71): The identification sequence field (columns 73 to 80) is not duplicated.

The new control card and the duplicated source cards are the input for the first (or only) additional assembly run. If another symbol-table overflow occurs, this first additional assembly run is considered to be the original assembly run and another additional assembly run can be performed.

This again increases, at the rate permitted for a new assembly, the number of symbols that can be used in the program.

BASIC ASSEMBLER (TAPE VERSIONS)

The tape versions of the Basic Assembler program use tape as an intermediate storage medium, which reduces card handling time. The Basic Assembler program and the first source program (both contained in punched cards) are read into the system during the initial run. Intermediate information is not punched into cards (as with the pass information of the card version) but is written on tape, from which it can be retrieved by the program when required.

Once the appropriate tape version of the Basic Assembler is written on a work tape, it can be used for the assembly of any number of source programs during the same run. Each source program is read in after the object deck for the preceding program has been punched. The subsequent source decks must be separated by blank cards.

For the assembly of a source program with the tape version of the Basic Assembler program, a control card similar to the control card of the card version, must be created. The control card and the card handling required during an assembly run are described in the SRL publication IBM System/360 Model 20, Card Programming Support, Basic Assembler (Tape Versions), Operating Procedures, Form C24-9011.

The input decks of the tape versions of the Basic Assembler consist of (1) the Basic Assembler pre-phase and (2) the five Basic Assembler phase decks. The pre-phase is used to read and evaluate the control card and to write the Basic Assembler program onto tape. The first four Basic Assembler phases are used to read the cards containing the source program, to check the statement formats, to translate the program into machine language, to print the program listings, and to punch the object program deck.

The fifth Basic Assembler phase is used to deal with a possible symbol-table overflow. Otherwise it is not used.

In case of a symbol-table overflow, the tape versions of the Basic Assembler program automatically initiate a routine to compensate for the overflow. The punching of the object program is discontinued at the point where the overflow occurs. Phase 5 of the program causes the generation of additional intermediate information, which is required by the Basic Assembler program to initiate another assembly run. The assembly is then repeated, from the beginning, to process the subsequent part of the source program and punch the remaining object cards.

The printed output produced by the tape versions of the Basic Assembler is the same as the printed output produced by the card versions of the Basic Assembler.

DIAGNOSTIC MESSAGES

Errors in the syntax of source statements and other violations of programming conventions are marked by diagnostic messages in the program listing to the left of the statements involved. These diagnostic messages, produced by both versions of the Basic Assembler program, are subdivided into two groups:

1. Warning messages.

2. Error messages.

Warning messages indicate violations of programming rules that do not affect execution of the assembly. The pertinent message codes are D, L, R, T, and W.

Error messages identify incorrect statements that prevent the Basic Assembler program from completing an assembly. The pertinent message codes are C, M, N, O, S, and U. A summary of all diagnostic messages is provided in Appendix D.

LOADING OBJECT PROGRAMS

Two routines for the loading of object programs are available: (1) the Absolute-Program Loader and (2) the Relocatable-Program Loader.

The Absolute-Program Loader is punched into a single card by the Basic Assembler program when the object deck is punched. Any loader control cards that may have been produced by the Basic Assembler (ESD and RLD) are ignored by the Absolute-Program Loader.

If the program is to be relocated on loading, the operator must replace the Absolute-Program Loader card with the deck containing the Relocatable-Program Loader. The loading routines are described in detail in the SRL publication, IBM System/360 Model 20 Card Programming Support, Basic Utility Programs, Functions and Operating Procedures, Form C26-3604.

This section lists the storage and time requirements for the assembly of source programs and the execution of object programs.

MAIN STORAGE REQUIREMENTS

Assembly of Source Programs: Figure 25 shows the main storage requirements for the assembly of source programs containing the maximum number of symbols.

Storage Capacity	Number of Symbols in Source Program
4096	165
8192	847
12288	1530
16384	2213

Figure 25. Main Storage Requirements for Assembly

Execution of Object Programs: The Absolute-Program Loader requires 160 bytes of main storage (including the load/read area). The Relocatable-Program Loader requires approximately 500 bytes. The remaining portion of main storage is available for object program execution.

Note: If the source program contains external symbols, additional storage is required for the External Symbol Identification table.

TIME REQUIREMENTS -- CARD VERSION

Assembly of Source Programs: Figure 26 shows the times required to assemble a source program consisting of 600 cards, including 165 symbols, on two basic input/output configurations. The available main storage is 4096 bytes. The times given apply to IBM Model 20, Submodel 2.

If an IBM Model 20 Submodel 3 or 4 is used, the time requirements shown in Figure 26 will increase by approximately 50%. For an IBM Model 20 Submodel 5 the time requirements will decrease by approximately 10%.

The time requirements depend on the distribution of symbols and on the type of cards (i.e., original or duplicated source cards) into which the pass information is punched.

The total times shown in Figure 26 do not include card handling time or the time required for loading the two Basic Assembler decks (approximately 10 to 15 seconds).

I/O Configuration	Time (in Minutes)
2560 MFCM and 2203 Printer	Pass 1: 4 to 7 Pass 2: 4 to 5 TOTAL: 8 to 12
2501 Card Reader 2520 Card Punch 1403 Printer	Pass 1: 4 to 6 Pass 2: 2 to 3 TOTAL: 6 to 9

Figure 26. Summary of Time Requirements for Assembly, Card Version

Execution of Object Programs: The time required for the execution of an object program depends on the length of the program and on the types of operations employed.

TIME REQUIREMENTS -- TAPE VERSION

The time required for the assembly of source programs depends on the distribution of symbols and on the model of the 2415 used during the assembly. The average time requirement for a source program comprising 600 cards and 165 symbols is from 6.2 to 8 minutes, when using a storage capacity of 4096 bytes.

WRITING A PROGRAM IN BASIC ASSEMBLER LANGUAGE

This section illustrates the writing of a program in Basic Assembler language, from the first approach to the specified problem, through the subsequent steps of writing the statements and executing the assembly and the object program, and concludes with the result printed as final output.

STATING THE PROBLEM

The sample problem used is as follows. In 1627, an Indian sold Manhattan Island for twenty-four dollars. Determine the resulting capital in 1965 if this money had been immediately transferred to a bank at an interest of 4% per annum. The interest earned each year should be rounded to the nearest cent.

WRITING THE SOURCE PROGRAM

THE FLOWCHART

To establish a guide line that defines the steps to be taken towards a solution, a flowchart can be developed, as shown in Figure 27.

INITIALIZING THE PROGRAM (STMT1-STMT3)

According to the flowchart, initializing the program is the first step. This means (1) incrementing the location counter to a tentative loading point and (2) loading and assigning a base register.

These first instructions can now be entered on an IBM coding form, as shown in Figure 28. The operand of the START instruction (STMT1) causes the location counter setting to be incremented to 340 (hexadecimal 154). The next statement causes the address 342 (hexadecimal 156) to be loaded into register 13 (STMT2) and the USING statement assigns to register 13 the attributes of a base register (STMT3).

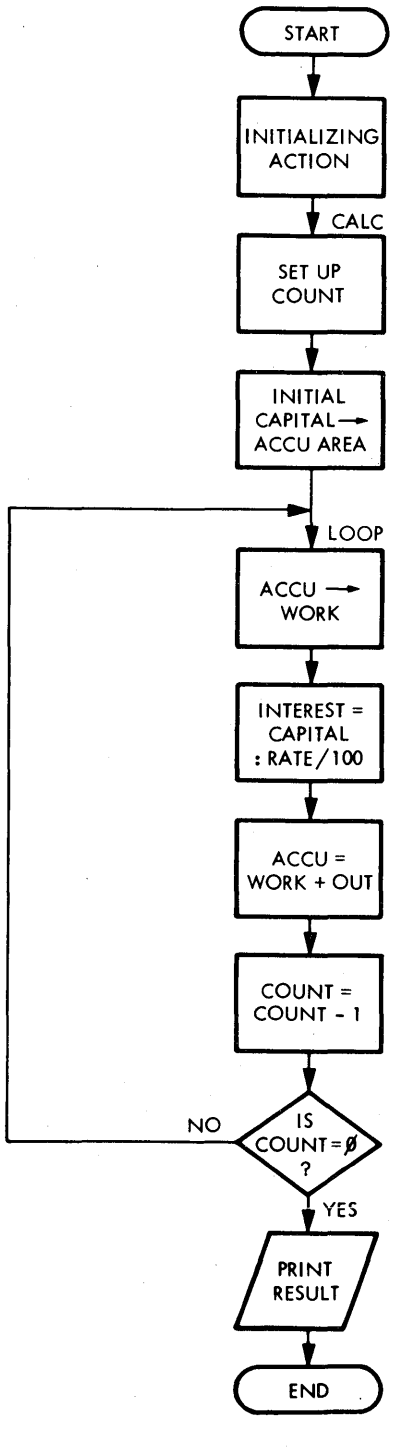
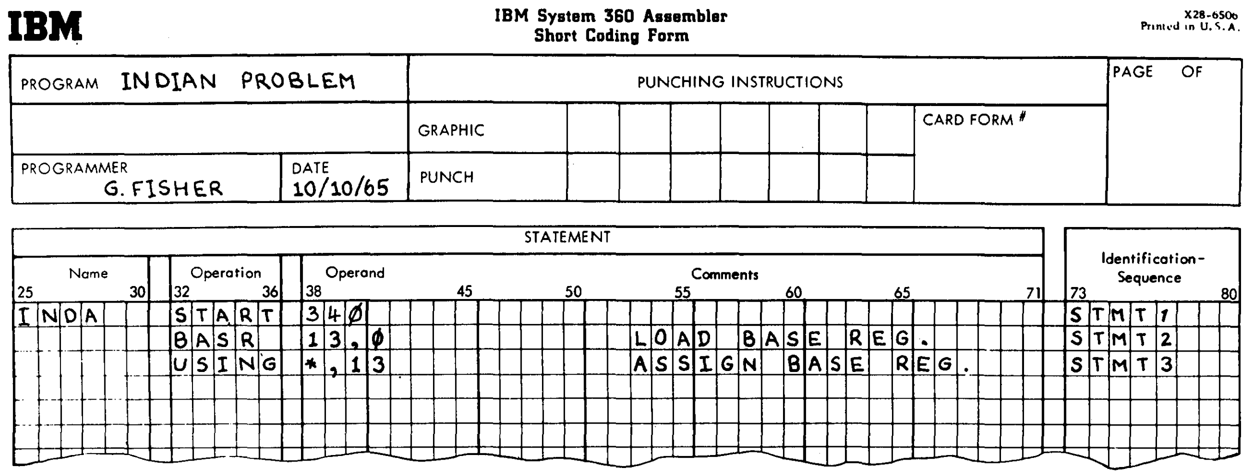


Figure 27. Sample Program Flowchart



IBM

IBM System 360 Assembler
Short Coding Form

X28-6506
Printed in U.S.A.

PROGRAM INDIAN PROBLEM	PUNCHING INSTRUCTIONS		PAGE OF
	GRAPHIC	CARD FORM #	
PROGRAMMER G. FISHER	DATE 10/10/65	PUNCH	

STATEMENT

```
Name      Operation  Operand                       Comments                    Identification-
                                                                               Sequence
25    30  32     36  38       45      50      55      60      65          71  73          80
INDA      START      34Ø                                                       STMT1
          BASR       13,Ø                     LOAD BASE REG.                   STMT2
          USING      *,13                     ASSIGN BASE REG.                 STMT3
```

Figure 28. Initialization Routine

DATA CONSTANTS AND WORK AREAS
(STMT4-STMT15)

Next, we must introduce the data and set up the required work areas. Knowing that the program must execute arithmetic calculations, including several division operations, it appears to be the most convenient approach to define our data in packed decimal form, as required for decimal arithmetic. In addition, we know that DP instructions require the dividend to have a certain number of leading zeros. Therefore, we define the work areas as a string of hexadecimal zeros.

The following data constants and work areas are required:

1. The capital (24.000) allowing for an additional decimal position, which can be used for rounding to the nearest cent (STMT9).

2. The divisor (25) for calculation of the 4% interest (STMT10).

3. The parameter (5) for rounding the last decimal position (STMT11).

4. The count (338) to control the number of calculations executed (STMT12).

5. The parameter (1) to decrement the count (STMT13).

6. The mask required when transforming the result into unpacked format for printing and for insertion of the necessary commas and the decimal point (STMT14; STMT15).

7. A print (PRT; 17 bytes) large enough to accommodate the mask (STMT6).

8. An area (ACCU; 7 bytes) to accumulate the computed interest and the resulting new capital (STMT8).

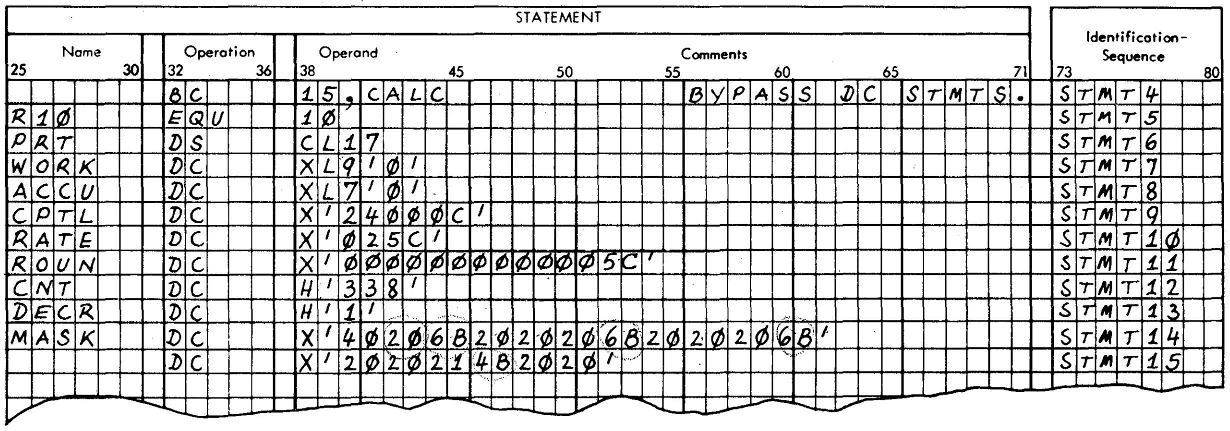
9. A work area for execution of the division and rounding (STMT7), with a length of 9 bytes, which is equal to the length of the divisor plus the length of the dividend.

Figure 29 shows how these constants and areas are defined.

The BC statement (STMT4) in Figure 26 is required during execution of the object program so that it can branch around the constants.

Register 10 is specified by R10 in the operand of a program statement (STMT5) which facilitates the reading of the statements. The constant ROUN is used to round.

The constant MASK provides a basis for the ED (Edit) instruction that transforms data to be printed into unpacked format and inserts the necessary decimal signs. Information to be printed is edited into a field that contains the mask. The mask causes leading zeros to be suppressed by its first character (hexadecimal 40). Each decimal digit printed must be represented by the select character, 20, 21, or 22 in the mask -- whichever is applicable. Commas and decimal points are specified by the characters 6B and 4B, respectively, placed in the position where they should appear in the printed data.



Name	Operation	Operand / Comments	Identification-Sequence
	BC	15,CALC BYPASS DC STMTS.	STMT4
R1Ø	EQU	1Ø	STMT5
PRT	DS	CL17	STMT6
WORK	DC	XL9'Ø'	STMT7
ACCU	DC	XL7'Ø'	STMT8
CPTL	DC	X'24ØØØC'	STMT9
RATE	DC	X'Ø25C'	STMT10
ROUN	DC	X'ØØØØØØØØØØØ5C'	STMT11
CNT	DC	H'338'	STMT12
DECR	DC	H'1'	STMT13
MASK	DC	X'4Ø2Ø6B2Ø2Ø2Ø6B2Ø2Ø2Ø6B'	STMT14
	DC	X'2Ø2Ø214B2Ø2Ø'	STMT15

Figure 29. Introduction of Data and Work Areas

Before a mask can be set up, the maximum size of the expected result must be determined. In our program example, we have analyzed the result and decided to reserve twelve decimal positions. This means, that the largest result expected is of the format:

X,XXX,XXX,XXX.XX

If the result should be shorter, zeros are replaced by blanks (hexadecimal 40 in the first position of the mask).

The mask may then be determined as follows:

```
 X  ,  X  X  X  ,  X  X  X  ,  X  X  X  .  X  X
 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
40 20 6B 20 20 20 6B 20 20 20 6B 20 20 21 4B 20 20
```

The digit preceding the decimal point is specified as 21. This code is the initial start character and causes zero suppression to be disregarded from here on. This allows printing of the decimal point, in case the result is less than 1.

PROGRAM ROUTINE (STMT16-STMT24)

Now we can concentrate on the program routine itself. According to the flow-chart, we first set up the count. As shown in Figure 32 (STMT16), this is done by loading register 10 with the constant 338 (1965 - 1627). This statement must be named CALC to link it with the branch instruction preceding the DC statements. The initial capital of 24.000 is moved into the ACCU area used to accumulate the intermediate interest amounts and incremented capital (STMT17). Thus, ACCU now has the contents shown in Figure 30.

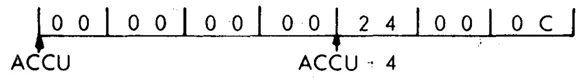
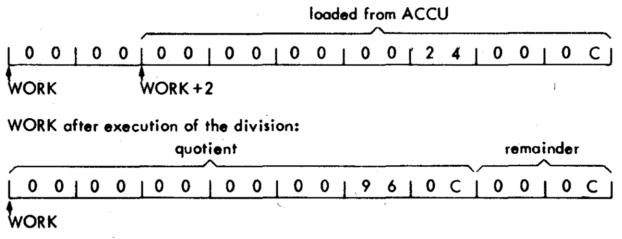


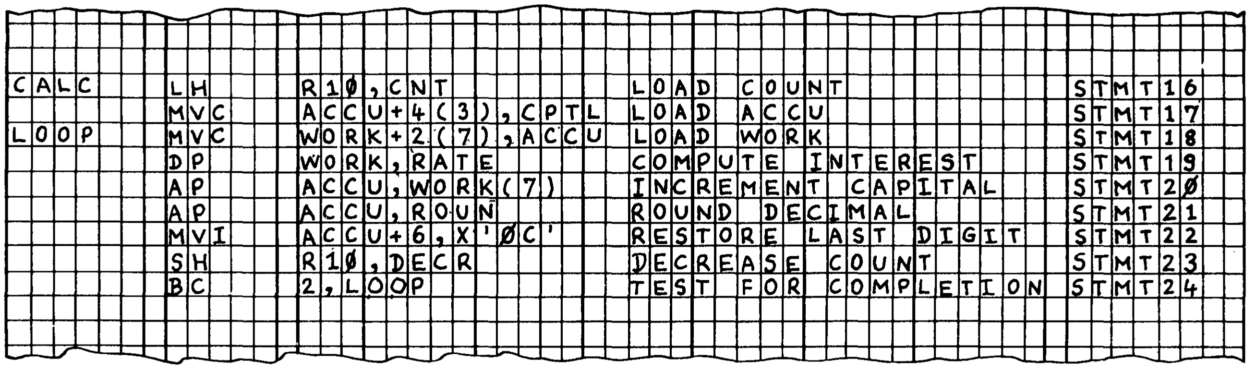
Figure 30. Contents of ACCU After Execution of STMT17

The next step is to bring the contents of ACCU (accumulated capital) into the work area for computation of the interest (STMT18). This is the first of the instructions to be executed 338 times and, therefore, becomes the entry point for the program loop (see flow-chart). The contents of the area WORK are then divided by 25 (STMT19). On execution of the division, the quotient, including leading zeros, is placed into the leftmost portion of the dividend field and the remainder into the rightmost portion of the dividend field. Thus, the first calculation is executed as shown in Figure 31.



loaded from ACCU
00 | 00 | 00 | 00 | 00 | 00 | 24 | 00 | 0C
WORK
WORK+2
WORK after execution of the division:
quotient
remainder
00 | 00 | 00 | 00 | 00 | 96 | 0C | 00 | 0C
WORK

Figure 31. Execution of the First Calculation Step



```
CALC     LH     R1Ø,CNT               LOAD COUNT               STMT16
         MVC    ACCU+4(3),CPTL        LOAD ACCU                STMT17
LOOP     MVC    WORK+2(7),ACCU        LOAD WORK                STMT18
         DP     WORK,RATE             COMPUTE INTEREST         STMT19
         AP     ACCU,WORK(7)          INCREMENT CAPITAL        STMT2Ø
         AP     ACCU,ROUN             ROUND DECIMAL            STMT21
         MVI    ACCU+6,X'ØC'          RESTORE LAST DIGIT       STMT22
         SH     R1Ø,DECR              DECREASE COUNT           STMT23
         BC     2,LOOP                TEST FOR COMPLETION      STMT24
```

Figure 32. Calculation Routine

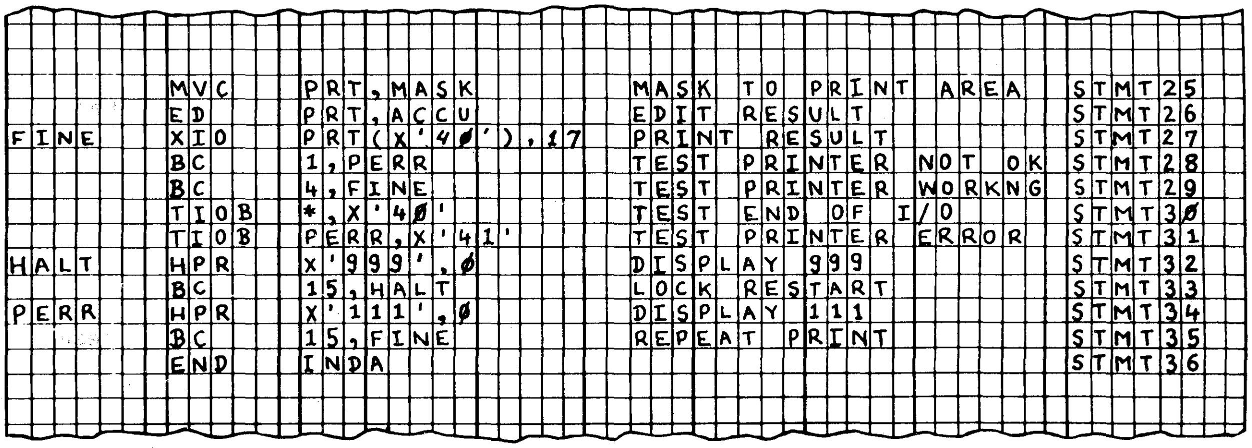
The contents of the leftmost seven bytes of the area WORK (0.960, after the first iteration) are added to the contents of ACCU (STMT20). Accordingly, ACCU now contains 24.960, the capital available after one year of deposit.

Fractions of cents that are equal to or greater than one-half are rounded to the next highest value by adding the constant 0.005 to the contents of ACCU (STMT21). Since the third decimal position contains a zero, the result is not changed. (On the next iteration, however, the computed interest and capital equals 25.958, which results in a rounded total of 25.963.) The original contents of the last byte of ACCU (0C) are then restored in preparation for the next iteration (STMT22).

The counter is then decreased by 1 (STMT23). This instruction also sets the condition code, which indicates whether the result is greater than or equal to zero. If the result is greater than zero, the program branches to LOOP and re-executes the program segment through the condition code test (STMT24). Otherwise, the print routine (Figure 33) is initiated.

OUTPUT (STMT25-STMT35)

STMT25 causes the mask to be moved into the print area. (The length of each operand need not be explicitly stated, because it is implied). The ED instruction (STMT26) causes the editing of the calculated result by moving it into the print area, on top of the mask already contained in this field. The first 4-bit hexadecimal digit of ACCU is placed into the leftmost byte containing a digit select character 20. Although the addressed byte is PRT, the first byte used to store the result is PRT+1.



```
         MVC    PRT,MASK              MASK TO PRINT AREA       STMT25
         ED     PRT,ACCU              EDIT RESULT              STMT26
FINE     XIO    PRT(X'4Ø'),17         PRINT RESULT             STMT27
         BC     1,PERR                TEST PRINTER NOT OK      STMT28
         BC     4,FINE                TEST PRINTER WORKNG      STMT29
         TIOB   *,X'4Ø'               TEST END OF I/O          STMT3Ø
         TIOB   PERR,X'41'            TEST PRINTER ERROR       STMT31
HALT     HPR    X'999',Ø              DISPLAY 999              STMT32
         BC     15,HALT               LOCK RESTART             STMT33
PERR     HPR    X'111',Ø              DISPLAY 111              STMT34
         BC     15,FINE               REPEAT PRINT             STMT35
         END    INDA                                           STMT36
```

Figure 33. Print Routine

Finally, the XIO instruction (STMT27) causes the printing of the result. The first operand specifies the area (PRT) in which the data to be printed is stored. The code in parentheses refers to a 1403 printer (U=4), and specifies printing as the function to be performed (F=0). The second operand gives the number of characters (bytes) to be printed. At this stage, the program could be terminated. However, we would risk a disregard of our print instruction if, for instance, the printer were out of service, or busy with a previously issued I/O instruction. In addition, we should delay processing of the HPR instruction until the previous I/O operation is completed to ensure that no print errors have been detected.

All of these conditions are taken care of by appropriate test and branch instructions, represented by STMT28 through STMT31. STMT28 branches to the instruction that stops the processing of the program if the printer is not operational. STMT29 tests to see if the printer is working ("Working" means that the Model 20 is in the process of setting up mechanical delays and circuitry or still executing a previous XIO instruction, not that it is executing the present XIO instruction.) and causes the re-execution of the XIO instruction until the printer has completed the last I/O operation. STMT30 tests to see if the printer is busy ("Busy" means that the XIO instruction is actually being executed.) and causes the program to loop around the same instruction until the last print operation has been terminated. STMT31 causes a halt, if a print error occurs, and display of code 111 in the STR register panels on the CPU (STMT34). In the latter case, pressing the start key of the CPU causes the print instruction to be re-executed because of the branch address in STMT35.

PROGRAM END (STMT36)

If no print error occurs, the program halts on reaching the HPR instruction (STMT32). If the start key of the CPU is pressed, STMT33 causes the program to re-execute the previous HPR instruction and to return to the same halt.

ASSEMBLING THE SOURCE PROGRAM

CONTROL CARD

When the program has been punched into cards, the source program can be assembled by either version of the Basic Assembler program.

In our case, it is assumed that the card version is used and that the available system configuration includes a 2560 MFCM and a 2501 Card Reader. Therefore, the 2501 will read the Basic Assembler program and the source program.

In addition, the pass information will be punched into duplicated source cards on the attached 2560, and the first run will scan the program statements for possible errors. Thus, the control card will be supplied with the following entries:

```
Columns 1-5://CTL
Column 6:  0 or blank (Indicates a diag-
                       nostic run; all
                       punch operations are
                       bypassed; only the
                       program listing is
                       printed.)
Column 8:  0 or blank (Indicates that 4096
                       bytes of main
                       storage are used for
                       the assembly.)
```

All other columns are left blank.

DIAGNOSTIC RUN

The statement listing printed during the diagnostic run is shown in Figure 34. To demonstrate the identification of incorrect statements by diagnostic messages, two errors have been deliberately included in the source deck (see STMT19 and STMT29).

```
      INDA   START 340                                 STMT01
             BASR  13,0              LOAD BASE REG.    STMT02
             USING *,13              ASSIGN BASE REG.  STMT03
             BC    15,CALC           CIRCLE THE CONST. STMT04
      R10    EQU   10                                  STMT05
      PRT    DS    CL17                                STMT06
      WORK   DC    XL9'0'                              STMT07
      ACCU   DC    XL7'0'                              STMT08
      CPTL   DC    X'24000C'                           STMT09
      RATE   DC    X'025C'                             STMT10
      ROUN   DC    X'00000000000005C'                  STMT11
      CNT    DC    H'338'                              STMT12
      DECR   DC    H'1'                                STMT13
      MASK   DC    X'40206B2020206B2020206B'           STMT14
             DC    X'2020214B2020'                     STMT15
      CALC   LH    R10,CNT           LOAD COUNT        STMT16
             MVC   ACCU+4(3),CPTL    LOAD ACCU         STMT17
      LOOP   MVC   WORK+2(7),ACCU    LOAD WORK         STMT18
M     LOOP   DP    WORK,RATE         COMPUTE INTEREST  STMT19
             AP    ACCU,WORK(7)      INCREMENT CAPITAL STMT20
             AP    ACCU,ROUN         ROUND DECIMAL     STMT21
             MVI   ACCU+6,X'0C'      RESTORE LAST DIGIT STMT22
             SH    R10,DECR          DECREASE COUNT    STMT23
             BC    2,LOOP            TEST FOR COMPLETION STMT24
             MVC   PRT,MASK          MASK TO PRINT AREA STMT25
             ED    PRT,ACCU          EDIT RESULT       STMT26
      FINE   XIO   PRT(X'40'),17     PRINT RESULT      STMT27
             BC    1,PERR            TEST PRINTER NOT OK STMT28
C            BC    4,FINF            TEST PRINTER WORKINGSTMT29
             TIOB  *,X'40'           TEST END OF I/O   STMT30
             TIOB  PERR,X'41'        TEST PRINTER ERROR STMT31
      HALT   HPR   X'999',0          DISPLAY 999       STMT32
             BC    15,HALT           LOCK RESTART      STMT33
      PERR   HPR   X'111',0          DISPLAY 111       STMT34
             BC    15,FINE           REPEAT PRINT      STMT35
             END   INDA                                STMT36
```

Figure 34. Sample Statement Listing Produced During the Diagnostic Run

STMT19 is marked by an M, indicating that the symbol in the name field is defined twice.

STMT29 is marked by a C, indicating that column 72 of the source card is not blank.

Note: In the case of an erroneous comments card (i.e., punched in column 72), columns 1 to 24 are not printed by the card version of the Basic Assembler program. The statement is marked by a C.

Assembly Run

After correcting these two errors, the source program can be assembled. For this purpose, the entry in column 6 of the control card must be changed to 3. This informs the Basic Assembler program that (1) a 2501 is used for reading, (2) a 2560 MFCM is used for punching, and (3) pass information is to be punched into a duplicated source deck. (For detailed information refer to the SRL publication IBM System/360 Model 20, Card Programming Support, Basic Assembler (Card Versions), Operating Procedures, Form C26-3802.

During assembly, the symbol-table image is printed as shown in Figure 36. The program listing is shown in Figure 37. The punched card output, produced during the assembly run, consists of the object deck with an Absolute-Program Loader card preceding it. When these cards have been loaded into main storage, the execution of the object program produces the result shown in Figure 35.

```
13,721,788.77
```

Figure 35. Result Computed by the Problem Program

```
ACCU    10  0174  06     CALC    10  019E  03     CNT     10  0188  01     CPTL    10  017B  02
DECR    10  018A  01     FINE    10  01D8  05     HALT    10  01EE  03     INDA    10  0154  00
LOOP    10  01A8  05     MASK    10  018C  0A     PERR    10  01F6  03     PRT     10  015A  10
RATE    10  017E  01     ROUN    10  0180  06     R10     00  000A  00     WORK    10  0168  08
```

Figure 36. Image of the Symbol Table

```
0154                                      INDA   START 340                                  STMT01   001
0154   0000                                      BASR  13,0              LOAD BASE REG.     STMT02   002
0156                                             USING *,13              ASSIGN BASE RE.    STMT03   002
0156   47F0 D048                                 BC    15,CALC           CIRCLE THE CONST   STMT04   002
000A                                      R10    EQU   10                                   STMT05   002
015A                                      PRT    DS    CL17                                 STMT06   002
0168   0000 0000 0000 0000 00             WORK   DC    XL9'0'                               STMT07   003
0174   0000 0000 0000 00                  ACCU   DC    XL7'0'                               STMT08   003
017B   2400 0C                            CPTL   DC    X'24000C'                            STMT09   003
017E   025C                               RATE   DC    X'025C'                              STMT10   003
0180   0000 0000 0000 5C                  ROUN   DC    X'00000000000005C'                   STMT11   003
0188   0152                               CNT    DC    H'338'                               STMT12   003
018A   0001                               DECR   DC    H'1'                                 STMT13   003
018C   4020 6B20 2020 6B20 2020 6B        MASK   DC    X'40206B2020206B2020206B'            STMT14   003
0197   2020 214B 2020                            DC    X'2020214B2020'                      STMT15   003
019E   48A0 D032                          CALC   LH    R10,CNT           LOAD COUNT         STMT16   003
01A2   D202 D022 D025                            MVC   ACCU+4(3),CPTL    LOAD ACCU          STMT17   003
01A8   D206 D017 D01E                     LOOP   MVC   WORK+2(7),ACCU    LOAD WORK          STMT18   004
01AE   FD81 D015 D028                            DP    WORK,RATE         COMPUTE INTEREST   STMT19   004
01B4   FA66 D01E D015                            AP    ACCU,WORK(7)      INCREMENT CAPITAL  STMT20   004
01BA   FA66 D01E D02A                            AP    ACCU,ROUN         ROUND DECIMAL      STMT21   004
01C0   920C D024                                 MVI   ACCU+6,X'0C'      RESTORE LAST DIGIT STMT22   004
01C4   4BA0 D034                                 SH    R10,DECR          DECREASE COUNT     STMT23   004
01C8   4720 D052                                 BC    2,LOOP            TEST FOR COMPLETION STMT24  004
01CC   D210 D004 D036                            MVC   PRT,MASK          MASK TO PRINT AREA STMT25   004
01D2   DE10 D004 D01E                            ED    PRT,ACCU          EDIT RESULT        STMT26   004
01D8   D040 D004 0011                     FINE   XIO   PRT(X'40'),17     PRINT RESULT       STMT27   004
01DE   4710 D0A0                                 BC    1,PERR            TEST PRINTER NOT OK STMT28  005
01E2   4740 D082                                 BC    4,FINE            TEST PRINTER WORKNG STMT29  005
01E6   9A40 D090                                 TIOB  *,X'40'           TEST END OF I/O    STMT30   005
01EA   9A41 D0A0                                 TIOB  PERR,X'41'        TEST PRINTER ERROR STMT31   005
01EE   9900 0999                          HALT   HPR   X'999',0          DISPLAY 999        STMT32   005
01F2   47F0 D098                                 BC    15,HALT           LOCK RESTART       STMT33   005
01F6   9900 0111                          PERR   HPR   X'111',0          DISPLAY 111        STMT34   005
01FA   47F0 D082                                 BC    15,FINE           REPEAT PRINT       STMT35   005
0154                                             END   INDA                                 STMT36   006
```

Figure 37. Assembler Produced Program Listing

APPENDIX A. SUMMARY OF BASIC ASSEMBLER INSTRUCTIONS

Description and Function	Name	Operation	Operand
Base Register Instructions			
Use Base Address Register	not used	USING	Reloc. exp.,abs. exp.
Drop Base Address Register	not used	DROP	Simple abs. exp.
Program Linking Instructions			
Identify Entry Point	not used	ENTRY	Relocatable symbol
Identify External Symbol	not used	EXTRN	Relocatable symbol
Definition Instructions			
Equate Symbol	optional	EQU	Expression
Define Constant	optional	DC	TLC[1]
Define Storage	optional	DS	DFL[2]
Assembler Control Instructions			
Start Program	optional	START	Self-defining value
Reset Location Counter	not used	ORG	Relocatable expression
End of Program	not used	END	Relocatable expression

[1]T--Type (C, X, H or Y)
L--Length Modifier
C--Constant

[2]D--Duplication Factor
F--Field (C or H)
L--Length

Mnemonic Code	Name of Instruction	Operation Code[1]	Basic Machine Format	Operand Field Format	Page Number
AH	Add Halfword	4A	RX	R1,D2(X2,B2)	53
AR	Add	1A	RR	R1,R2	51
AP	Add Decimal	FA	SS	D1(L1,B1),D2(L2,B2)	57
BAS	Branch and Store	4D	RX	R1,D2(X2,B2)	71
BASR	Branch and Store	0D	RR	R1,R2	71
BC	Branch on Condition	47	RX	M1,D2(X2,B2)	70
BCR	Branch on Condition	07	RR	M1,R2	70
CH	Compare Halfword	49	RX	R1,D2(X2,B2)	52
CIO	Control I/O	9B	SI	D1(B1),UF	37
CLC	Compare Logical	D5	SS	D1(L,B1),D2(B2)	64
CLI	Compare Logical Immediate	95	SI	D1(B1),I2	64
CP	Compare Decimal	F9	SS	D1(L1,B1),D2(L2,B2)	57
DP	Divide Decimal	FD	SS	D1(L1,B1),D2(L2,B2)	59
ED	Edit	DE	SS	D1(L,B1),D2(B2)	64
HPR	Halt and Proceed	99	SI	D1(B1),I2	68
LH	Load Halfword	48	RX	R1,D2(X2,B2)	52
MP	Multiply Decimal	FC	SS	D1(L1,B1),D2(L2,B2)	58
MVC	Move Characters	D2	SS	D1(L,B1),D2(B2)	62
MVI	Move Immediate	92	SI	D1(B1),I2	62
MVN	Move Numerics	D1	SS	D1(L,B1),D2(B2)	63
MVO	Move With Offset	F1	SS	D1(L1,B1),D2(L2,B2)	55
MVZ	Move Zones	D3	SS	D1(L,B1),D2(B2)	63
NI	And Logical Immediate	94	SI	D1(B1),I2	67
OI	Or Logical Immediate	96	SI	D1(B1),I2	67
PACK	Pack	F2	SS	D1(L1,B1),D2(L2,B2)	56
SH	Subtract Halfword	4B	RX	R1,D2(X2,B2)	53
SP	Subtract Decimal	FB	SS	D1(L1,B1),D2(L2,B2)	58
SPSW	Set PSW	81	SI	D1(B1)	48
SR	Subtract	1B	RR	R1,R2	51
STH	Store Halfword	40	RX	R1,D2(X2,B2)	52
TIOB	Test I/O and Branch	9A	SI	D1(B1),UF	39
TM	Test under Mask	91	SI	D1(B1),I2	67
TR	Translate	DC	SS	D1(L,B1),D2(B2)	68
UNPK	Unpack	F3	SS	D1(L1,B1),D2(L2,B2)	56
XIO	Execute I/O	D0	SS	D1(UF,B1),D2(B2)	37
ZAP	Zero and Add Decimal	F8	SS	D1(L1,B1),D2(L2,B2)	56

[1]Hexadecimal Equivalent of actual Machine Operation Code.

APPENDIX C. SUMMARY OF INPUT/OUTPUT INSTRUCTIONS

Machine	Mnemonic Operation Code	Operand U	F	Function
	XIO	1	2	Read Card
2501	XIO	1	A	*Read Card, Column Binary
Card Reader	TIOB	1	0	Test Reader Busy
Model A1 or A2	TIOB	1	1	Test Reader Error
	TIOB	1	4	Test Last Card
	XIO	2	2	Read Primary Card
	XIO	2	A	* Read Primary Card, Column Binary
	XIO	2	3	Read Secondary Card
	XIO	2	B	* Read Secondary Card, Column Binary
	XIO	2	4	Punch Primary Card
	XIO	2	5	Punch Secondary Card
	XIO	2	6	Punch and Feed Primary Card
	XIO	2	7	Punch and Feed Secondary Card
2560	XIO	2	0	* Write Card
Multi-Function	TIOB	2	0	Test Reader/Punch Busy
Card Machine	TIOB	2	1	Test Reader/Punch Error
	TIOB	2	2	Test Card Printer Busy
	TIOB	2	4	Test Last Card
	TIOB	2	5	Test Feed Error
	CIO	2	0	Primary Card Stacker Select
	CIO	2	1	Secondary Card Stacker Select
	CIO	2	2	Punch Card Stacker Select
	CIO	2	3	* Print Head Select
	XIO	2	2	Read Card
	XIO	2	A	* Read Card, Column Binary
	XIO	2	4	Punch Card
	XIO	2	6	Punch and Feed
2520	TIOB	2	0	Test Reader Busy
Card Read	TIOB	2	1	Test Reader Error
Punch	TIOB	2	2	Test Punch Busy
	TIOB	2	3	Test Punch Error
	TIOB	2	4	Test Last Card
	TIOB	2	5	Test Feed Error
	CIO	2	0	Stacker Select
	XIO	2	6	Punch Card
2520	TIOB	2	2	Test Punch Busy
Card Punch	TIOB	2	3	Test Punch Error
Model A2 or A3	TIOB	2	5	Test Feed Error
	CIO	2	0	Stacker Select
	XIO	3	6	Punch Card
1442	TIOB	3	2	Test Punch Busy
Card Punch	TIOB	3	3	Test Punch Error
Model 5	TIOB	3	5	Test Feed Error
	XIO	4	0	Print
2203 or 1403	XIO	4	1	Print and Space Suppress
Printer	TIOB	4	0	Test Printer Busy
	TIOB	4	1	Test Printer Error
	TIOB	4	2	Test Channel 9
	TIOB	4	3	Test Channel 12
	TIOB	4	4	* Test Channel 9 (upper)
	TIOB	4	5	* Test Channel 12 (upper)

*Optional Feature

Machine	Mnemonic Operation Code	Operand U	Operand F	Function
2203 or 1403 Printer	TIOB	4	6	Test Carriage Busy
	CIO	4	4	Immediate Space
	CIO	4	5	Immediate Skip
	CIO	4	6	Delayed Space
	CIO	4	7	Delayed Skip
	CIO	4	8	* Immediate Space (upper)
	CIO	4	9	* Immediate Skip (upper)
	CIO	4	A	* Delayed Space (upper)
	CIO	4	B	* Delayed Skip (upper)
	CIO	4	C	* Immediate Space (both)
	CIO	4	D	* Immediate Skip (both)
	CIO	4	E	* Delayed Space (both)
	CIO	4	F	* Delayed Skip (both)
Communica-tions Adapter (C.A.)	XIO	5	2	Receive Record
	XIO	5	4	Transmit Record
	TIOB	5	0	Test C.A. Busy
	TIOB	5	1	Test C.A Error
	TIOB	5	5	Test Received EOT
	CIO	5	0	Set Receive Mode
	CIO	5	2	Send EOT
	CIO	5	3	Inhibit Audible Alarm
Binary Synchronous Communications Adapter(BSCA)	XIO	5	0	Transmit and Receive
	XIO	5	1	Receive Initial
	XIO	5	2	Address Prepare
	XIO	5	3	Auto Call
	XIO	5	4	Receive
	XIO	5	8	Transmit
	TIOB	5	0	Test Any Indicator Set
	TIOB	5	8	Test Busy
	CIO	5	0	Disable ITB
	CIO	5	1	Enable ITB
	CIO	5	2	Enable BSCA
	CIO	5	3	Disable BSCA
	CIO	5	6	Store Current Address
	CIO	5	7	Store Sense Information
	CIO	5	8	Store ITB Address
Serial Input/ Output Channel	XIO	6	2	Read I/O Device (Time sharing)
	XIO	6	4	Write I/O Device (Time sharing)
	XIO	6	10	Read I/O Device (Burst)
	XIO	6	12	Write I/O Device (Burst)
	TIOB	6	1	Test I/O Transfer 1
	TIOB	6	2	Test I/O transfer 2
	TIOB	6	3	Test I/O Transfer 3
	TIOB	6	4	Test I/O Transfer 4
	TIOB	6	5	Test I/O Transfer 5
	TIOB	6	6	Test I/O Transfer 6
	TIOB	6	7	Test I/O Transfer 7
	TIOB	6	8	Test I/O Transfer 8
	TIOB	6	9	Test Read Transfer Error
	CIO	6	0	Unit Control
	CIO	6	1	I/O Select
2415	XIO	7	0	Perform Tape Operation
2311	XIO	8	0	Perform Disk Operation

* Optional Feature

MESSAGE	ERRO^R CONDITION
C	1.) Assembly executed using the Basic Assembler (Card): (a) columns 1-24 and/or column 72 not blank, or (b) operation code and/or operand missing. 2.) Assembly executed using the Basic Assembler (Tape): (a) column 72 not blank, or (b) operation code and/or operand missing.
D[1]	1.) This EQU statement is unnamed. 2.) This START, ENTRY, or EXTRN statement is misplaced. (The statement is ignored).
L[1]	The value of the location counter has exceeded the storage size for program execution as specified in the Basic Assembler Control card (card column 9). Notes: An instruction byte may not occupy the last (highest order) available main storage address. A constant or data byte may be located at this position.
M	The name of this statement is defined more than once.
N	The name of this statement does not conform to the rules as follows: • It has more than four characters, or • its first character is not alphabetic, or • it contains an illegal character.
O	This mnemonic operation code is invalid.
R[1]	In this statement 1.) a relocatable expression has been used in an absolute field, or 2.) an absolute expression has been used in a relocatable field, or 3.) the X-Register field in an RX-instruction is not zero, or 4.) a relocatable expression could not be split into a valid base address and a displacement. (A USING statement is either missing, or wrong, or misplaced.)
S	One of the operands in this statement is invalid. This diagnostic message is printed when one or more of the following conditions occur: 1.) An invalid character is used as a delimiter. 2.) The first character of a symbol in the operand entry is not alphabetic. 3.) A delimiter is incorrectly used. 4.) The operand of a START, ORG, or EQU statement is invalid. 5.) A symbol or self-defining value in the operand entry contains an invalid character. 6.) A self-defining value or a symbol in the operand entry contains too many characters. 7.) A symbol or a self-defining value in the operand entry is followed by an invalid character. 8.) A self-defining value exceeds storage capacity. 9.) An ampersand or an apostrophe used within a character constant is incorrectly specified. 10.) A DS duplication factor is too high. 11.) A DS statement contains an invalid operand. 12.) A DC statement is incorrectly specified.

[1]Warning messages that do not suppress the punching of the object deck.

MESSAGE	ERROR CONDITION
T[1]	The symbol table was filled by the name of the last preceding statement. The name of this statement cannot be accommodated.
U[2]	1.) The operand entry contains an undefined symbol. 2.) The operand entry of an EQU, ORG, or END statement contains a symbol that is not previously defined.
W[1]	The length of a constant defined by a DC statement exceeds the explicit length.

[1]Warning messages that do not suppress the punching of the object deck.
[2]U-messages of type (1) do not suppress the punching of the object deck, those of type (2) do suppress punching.

Code Setting:	00	01	10	11
Mask Used to Test the Code:	8	4	2	1
Fixed Point Arithmetic (RR Format)				
Add Register (AR)	Result=0	Result<0	Result>0	--
Subtract Register (SR)	Result=0	Result<0	Result>0	--
Fixed Point Arithmetic (RX Format)				
Add Half-word (AH)	Result=0	Result<0	Result>0	--
Compare Half-word (CH)	Op1=Op2	Op1<Op2	Op1>Op2	--
Subtract Half-word (SH)	Result=0	Result<0	Result>0	--
Decimal Arithmetic				
Add Packed (AP)	Result=0	Result<0	Result>0	overflow
Compare Packed (CP)	Op1=Op2	Op1<Op2	Op1>Op2	--
Subtract Packed (SP)	Result=0	Result<0	Result>0	overflow
Zero Add Packed (ZAP)	Result=0	Result<0	Result>0	--
Logical Operations				
AND Logical Immediate (NI)	Result=0	Result≠0	--	--
Compare Logical (CLC,CLI)	Op1=Op2	Op1<Op2	Op1>Op2	--
Edit and Mark (ED)	source field	source field	source field	
OR Logical Immediate (OI)	Result=0	Result≠0	--	--
Test Under Mask (TM)	Result=0	Result mixed	--	Result all ones
Input/Output Operations				
Execute Input/Output (XIO)	Unit avail.	Unit working	--	Unit not operational
Control Input/Output (CIO) (referring to the 1403/2203 carriage only)	Device available	Device working	--	Device not operational

This appendix lists all System/360 card codes to which a printer graphic is assigned.

EBCDIC CODE	CARD PUNCH COMBINATION	PRINTER GRAPHIC	DECIMAL	HEXADECIMAL
00000000	12,0,9,8,1		0	00
00000001	12,9,1		1	01
00000010	12,9,2		2	02
00000011	12,9,3		3	03
00000100	12,9,4		4	04
00000101	12,9,5		5	05
00000110	12,9,6		6	06
00000111	12,9,7		7	07
00001000	12,9,8		8	08
00001001	12,9,8,1		9	09
00001010	12,9,8,2		10	0A
00001011	12,9,8,3		11	0B
00001100	12,9,8,4		12	0C
00001101	12,9,8,5		13	0D
00001110	12,9,8,6		14	0E
00001111	12,9,8,7		15	0F
00010000	12,11,9,8,1		16	10
00010001	11,9,1		17	11
00010010	11,9,2		18	12
00010011	11,9,3		19	13
00010100	11,9,4		20	14
00010101	11,9,5		21	15
00010110	11,9,6		22	16
00010111	11,9,7		23	17
00011000	11,9,8		24	18
00011001	11,9,8,1		25	19
00011010	11,9,8,2		26	1A
00011011	11,9,8,3		27	1B
00011100	11,9,8,4		28	1C
00011101	11,9,8,5		29	1D
00011110	11,9,8,6		30	1E
00011111	11,9,8,7		31	1F
00100000	11,0,9,8,1		32	20
00100001	0,9,1		33	21
00100010	0,9,2		34	22
00100011	0,9,3		35	23
00100100	0,9,4		36	24
00100101	0,9,5		37	25
00100110	0,9,6		38	26
00100111	0,9,7		39	27
00101000	0,9,8		40	28
00101001	0,9,8,1		41	29
00101010	0,9,8,2		42	2A
00101011	0,9,8,3		43	2B
00101100	0,9,8,4		44	2C
00101101	0,9,8,5		45	2D
00101110	0,9,8,6		46	2E
00101111	0,9,8,7		47	2F
00110000	12,11,0,9,8,1		48	30
00110001	9,1		49	31
00110010	9,2		50	32
00110011	9,3		51	33
00110100	9,4		52	34
00110101	9,5		53	35
00110110	9,6		54	36
00110111	9,7		55	37

00111000	9,8		56	38
00111001	9,8,1		57	39
00111010	9,8,2		58	3A
00111011	9,8,3		59	3B
00111100	9,8,4		60	3C
00111101	9,8,5		61	3D
00111110	9,8,6		62	3E
00111111	9,8,7		63	3F
01000000		blank	64	40
01000001	12,0,9,1		65	41
01000010	12,0,9,2		66	42
01000011	12,0,9,3		67	43
01000100	12,0,9,4		68	44
01000101	12,0,9,5		69	45
01000110	12,0,9,6		70	46
01000111	12,0,9,7		71	47
01001000	12,0,9,8		72	48
01001001	12,8,1		73	49
01001010	12,8,2	¢	74	4A
01001011	12,8,3	.	75	4B
01001100	12,8,4	<	76	4C
01001101	12,8,5	(	77	4D
01001110	12,8,6	+	78	4E
01001111	12,8,7	|	79	4F
01010000	12	&	80	50
01010001	12,11,9,1		81	51
01010010	12,11,9,2		82	52
01010011	12,11,9,3		83	53
01010100	12,11,9,4		84	54
01010101	12,11,9,5		85	55
01010110	12,11,9,6		86	56
01010111	12,11,9,7		87	57
01011000	12,11,9,8		88	58
01011001	11,8,1		89	59
01011010	11,8,2	!	90	5A
01011011	11,8,3	$	91	5B
01011100	11,8,4	*	92	5C
01011101	11,8,5	)	93	5D
01011110	11,8,6	;	94	5E
01011111	11,8,7	¬	95	5F
01100000	11	-	96	60
01100001	0,1	/	97	61
01100010	11,0,9,2		98	62
01100011	11,0,9,3		99	63
01100100	11,0,9,4		100	64
01100101	11,0,9,5		101	65
01100110	11,0,9,6		102	66
01100111	11,0,9,7		103	67
01101000	11,0,9,8		104	68
01101001	0,8,1		105	69
01101010	12,11		106	6A
01101011	0,8,3	,	107	6B
01101100	0,8,4	%	108	6C
01101101	0,8,5	_	109	6D
01101110	0,8,6	>	110	6E
01101111	0,8,7	?	111	6F
01110000	12,11,0		112	70
01110001	12,11,0,9,1		113	71
01110010	12,11,0,9,2		114	72
01110011	12,11,0,9,3		115	73
01110100	12,11,0,9,4		116	74
01110101	12,11,0,9,5		117	75
01110110	12,11,0,9,6		118	76
01110111	12,11,0,9,7		119	77
01111000	12,11,0,9,8		120	78
01111001	8,1		121	79
01111010	8,2	:	122	7A
01111011	8,3	#	123	7B
01111100	8,4	@	124	7C
01111101	8,5	'	125	7D

01111110	8,6	=	126	7E
01111111	8,7	"	127	7F
10000000	12,0,8,1		128	80
10000001	12,0,1		129	81
10000010	12,0,2		130	82
10000011	12,0,3		131	83
10000100	12,0,4		132	84
10000101	12,0,5		133	85
10000110	12,0,6		134	86
10000111	12,0,7		135	87
10001000	12,0,8		136	88
10001001	12,0,9		137	89
10001010	12,0,8,2		138	8A
10001011	12,0,8,3		139	8B
10001100	12,0,8,4		140	8C
10001101	12,0,8,5		141	8D
10001110	12,0,8,6		142	8E
10001111	12,0,8,7		143	8F
10010000	12,11,8,1		144	90
10010001	12,11,1		145	91
10010010	12,11,2		146	92
10010011	12,11,3		147	93
10010100	12,11,4		148	94
10010101	12,11,5		149	95
10010110	12,11,6		150	96
10010111	12,11,7		151	97
10011000	12,11,8		152	98
10011001	12,11,9		153	99
10011010	12,11,8,2		154	9A
10011011	12,11,8,3		155	9B
10011100	12,11,8,4		156	9C
10011101	12,11,8,5		157	9D
10011110	12,11,8,6		158	9E
10011111	12,11,8,7		159	9F
10100000	11,0,8,1		160	A0
10100001	11,0,1		161	A1
10100010	11,0,2		162	A2
10100011	11,0,3		163	A3
10100100	11,0,4		164	A4
10100101	11,0,5		165	A5
10100110	11,0,6		166	A6
10100111	11,0,7		167	A7
10101000	11,0,8		168	A8
10101001	11,0,9		169	A9
10101010	11,0,8,2		170	AA
10101011	11,0,8,3		171	AB
10101100	11,0,8,4		172	AC
10101101	11,0,8,5		173	AD
10101110	11,0,8,6		174	AE
10101111	11,0,8,7		175	AF
10110000	12,11,0,8,1		176	B0
10110001	12,11,0,1		177	B1
10110010	12,11,0,2		178	B2
10110011	12,11,0,3		179	B3
10110100	12,11,0,4		180	B4
10110101	12,11,0,5		181	B5
10110110	12,11,0,6		182	B6
10110111	12,11,0,7		183	B7
10111000	12,11,0,8		184	B8
10111001	12,11,0,9		185	B9
10111010	12,11,0,8,2		186	BA
10111011	12,11,0,8,3		187	BB
10111100	12,11,0,8,4		188	BC
10111101	12,11,0,8,5		189	BD
10111110	12,11,0,8,6		190	BE
10111111	12,11,0,8,7		191	BF
11000000	12,0		192	C0
11000001	12,1	A	193	C1
11000010	12,2	B	194	C2
11000011	12,3	C	195	C3

11000100	12,4	D	196	C4
11000101	12,5	E	197	C5
11000110	12,6	F	198	C6
11000111	12,7	G	199	C7
11001000	12,8	H	200	C8
11001001	12,9	I	201	C9
11001010	12,0,9,8,2		202	CA
11001011	12,0,9,8,3		203	CB
11001100	12,0,9,8,4		204	CC
11001101	12,0,9,8,5		205	CD
11001110	12,0,9,8,6		206	CE
11001111	12,0,9,8,7		207	CF
11010000	11,0		208	D0
11010001	11,1	J	209	D1
11010010	11,2	K	210	D2
11010011	11,3	L	211	D3
11010100	11,4	M	212	D4
11010101	11,5	N	213	D5
11010110	11,6	O	214	D6
11010111	11,7	P	215	D7
11011000	11,8	Q	216	D8
11011001	11,9	R	217	D9
11011010	12,11,9,8,2		218	DA
11011011	12,11,9,8,3		219	DB
11011100	12,11,9,8,4		220	DC
11011101	12,11,9,8,5		221	DD
11011110	12,11,9,8,6		222	DE
11011111	12,11,9,8,7		223	DF
11100000	0,8,2		224	E0
11100001	11,0,9,1		225	E1
11100010	0,2	S	226	E2
11100011	0,3	T	227	E3
11100100	0,4	U	228	E4
11100101	0,5	V	229	E5
11100110	0,6	W	230	E6
11100111	0,7	X	231	E7
11101000	0,8	Y	232	E8
11101001	0,9	Z	233	E9
11101010	11,0,9,8,2		234	EA
11101011	11,0,9,8,3		235	EB
11101100	11,0,9,8,4		236	EC
11101101	11,0,9,8,5		237	ED
11101110	11,0,9,8,6		238	EE
11101111	11,0,9,8,7		239	EF
11110000	0	0	240	F0
11110001	1	1	241	F1
11110010	2	2	242	F2
11110011	3	3	243	F3
11110100	4	4	244	F4
11110101	5	5	245	F5
11110110	6	6	246	F6
11110111	7	7	247	F7
11111000	8	8	248	F8
11111001	9	9	249	F9
11111010	12,11,0,9,8,2		250	FA
11111011	12,11,0,9,8,3		251	FB
11111100	12,11,0,9,8,4		252	FC
11111101	12,11,0,9,8,5		253	FD
11111110	12,11,0,9,8,6		254	FE
11111111	12,11,0,9,8,7		255	FF

The table in this appendix provides for direct conversion of decimal and hexadecimal numbers between 0000 and 4095 (hexadecimal 000 and FFF).

For numbers outside the range of the table, add the following values to the table figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
1000	4096	9000	36864
2000	8192	A000	40960
3000	12288	B000	45056
4000	16384	C000	49152
5000	20480	D000	53248
6000	24576	E000	57344
7000	28672	F000	61440
8000	32768		

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
10	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
21	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
22	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
23	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
24	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
25	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
26	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
27	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
28	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
29	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
30	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
31	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
32	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
33	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
34	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
35	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
36	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
37	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
38	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
39	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023
40	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
41	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
43	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
44	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
45	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
46	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
47	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
48	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
49	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
50	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
51	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
52	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
53	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
54	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
55	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
56	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
57	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
58	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
59	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
60	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
61	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
65	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
70	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
72	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
73	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
74	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
78	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	236o	2364	2365	2366	2367
94	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
A0	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
B0	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B4	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071
C0	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D0	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E0	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3756	3759
EB	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

Absolute Address
A pattern of characters that identifies a unique storage location or device without further modification.

Address
1. An identification, as represented by a name, or number, for a register, location in storage, or other data source or destination.
2. Loosely, any part of an instruction which specifies the location of an operand for the instruction.

Address Constant
A value, or an expression representing a value, interpreted as a storage address.

Address Modification
The process of changing the address part of a machine instruction by means of coded instructions.

Address Register
A register that stores an address.

Allocate
To assign storage locations or areas of storage for specific routines, portions of routines, constants, data, etc.

Alphameric
A generic term for alphabetic letters, numerical digits, and special characters.

Assemble
To prepare an object-language program from a symbolic-language program by substituting machine operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

Assembler
A program that assembles.

Attribute
A characteristic; e.g., attributes of data include record length, record format, data set name, associated device type and volume identification, use, creation date, etc.

Base Register
A register used for addressing purposes.

Basic Assembler Language
A symbolic language for the writing of source programs.

Basic Assembler Program
A program used to translate source programs written in Basic Assembler language into machine language.

Binary
1. A characteristic or property involving a selection, choice, or condition in which there are two possibilities.
2. The number representation system with a base of two.

Binary Code
A code that makes use of two distinct characters, usually 0 and 1.

Binary-Coded Character
One element of a notation system for representing alphameric characters such as decimal digits, alphabetic letters, punctuation marks, etc., by a fixed number of consecutive binary digits.

Binary-Coded Decimal
A decimal notation in which the individual decimal digits are each represented by a binary code group; e.g., in the 8-4-2-1 coded decimal notation, the number twenty-three is represented as 0010 0011, in binary notation, twenty-three is represented as 10111.

Binary Digit
A character used to represent one of the integers smaller than the radix 2.

Binary-to-Decimal Conversion
Conversion of a binary number to the equivalent decimal number; i.e., a base-two number to a base-ten number.

Bit
A binary digit.

Blank Character
Any character or characters used to produce a character space on an output medium.

Branch
1. To depart from the normal sequence of executing instructions in a computer.
2. A machine instruction that can cause a departure as in (1). Synonymous with 'transfer'.

Byte
A sequence of adjacent binary digits operated upon as a unit.

Card Code
The combinations of punched holes which represent characters (letters, digits, etc.) in a punched card.

Card Column
One of the vertical lines of punching positions on a punched card.

Card Field
A fixed number of consecutive card columns assigned to data of a specific nature.

Card Punch
A device to record information in cards by punching holes in the cards to represent letters, digits, and special characters.

Card Reader
A device which reads and translates into internal form the holes in punched cards.

Card Stacker
A mechanism which stacks cards in a poc-

ket after they pass through a machine.

Character
One of a set of elementary symbols which may include decimal digits 0 through 9, the letters A through Z, punctuation marks, and any other symbols acceptable to a computer for reading, writing or storing.

Character Set
A list of characters acceptable for coding to a specific computer or input/output device.

Clear
To put a storage device into a prescribed state, usually that denoting zero or blank.

Coded Decimal
A type of notation in which each decimal digit is identified by a group of binary ones and zeros.

Column Binary
Pertaining to the binary representation of data on punched cards in which adjacent positions in a column correspond to adjacent bits of data.

Command
An instruction in machine language.

Communication
The process of transferring information from one point, person, or piece of equipment to another.

Computer
1. A device capable of solving problems by accepting data, performing prescribed operations on the data, and supplying the results of these operations. Various types of computers are calculators, digital computers, and analog computers.
2. In information processing, usually, an automatic stored-program computer.

Computer Instruction
Same as machine instruction.

Constant
A fixed or invariable value or data item.

Counter
A device such as a register or storage location used to represent the number of occurrences of an event.

Cycle
1. An interval of space or time in which one set of events is completed.
2. Any set of operations that is repeated regularly in the same sequence. The operations may be subject to variations on each repetition.

Data
Any representation, such as character quantities, to which meaning might be assigned.

Data Conversion
The process of changing data from one form of representation to another.

Data Processing
A systematic sequence of operations performed on data.

Data Processing System
A network of machine components capable of accepting information, processing it according to a plan, and producing the desired results.

Decimal
1. A characteristic or property involving a selection, choice or condition in which there are ten possibilities.
2. The number representation system with a base of ten.

Decimal-to-Binary Conversion
The conversion of a decimal number to the equivalent binary number, i.e., a base-ten number to a base-two number.

Decision
A determination of future action.

Decision Block
A flowchart symbol whose interior contains the criterion for decision or branching.

Decision Instruction
An instruction that selects a branch of a program, e.g., a conditional branch instruction.

Deck
A collection of punched cards.

Decrement
The quantity by which a variable is decreased.

Diagnostic
The detection and isolation of a malfunction or a mistake.

Diagram
A schematic representation of a sequence of operations or routines.

Digit
1. Any of the arabic numerals 1 to 9 and the symbol 0.
2. One of the elements that combine to form numbers in a system other than the decimal system.

Displacement
The difference (in bytes) between the contents of a base register (or the address represented by a symbol) and a referenced storage location.

Dummy
The characteristic of having the appearance of a specified thing but not having the capacity to function as such.

EBCDIC
(Extended Binary Coded Decimal Interchange Code). A specific set of 8-bit codes standard throughout System/360.

Edit
To modify the form or format of data; e.g., to insert or delete characters such as page numbers or decimal points.

Effective Address
The absolute address of the current operand. This may differ from that of the instruction in storage.

Error
A general term to indicate that a data value is not correct or that a machine component is malfunctioning.

ESD card
ESD cards contain all information required for the linking of program segments (such as all symbols defined in one segment but referred to in another segment).

Execute
To carry out an instruction or perform a routine.

Explicit Addressing
Specification of an address by a base register and a displacement in the form D(B).

File
A collection of related records treated as a unit, e.g., in inventory control, one line of an invoice forms an item, a complete invoice forms a record, and the complete set of such records forms a file.

Flowchart
A graphical representation for the definition, analysis, or solution of a problem in which symbols are used to represent operations, data, flow, and equipment.

Hexadecimal Number System
A number system using the equivalent of the decimal number sixteen as a base.

Hopper
A device that holds cards and makes them available to a card feed mechanism. Contrast with card stacker.

Identification
A code number or code name which uniquely identifies a record, block, file or other unit of information.

Image
An exact logical duplicate stored in a different medium.

Immediate Address
The designation of an instruction address which is used as data by the instruction of which it is a part.

Implied Address
The address assigned to a symbol by the Basic Assembler program.

Index Register
A register whose content is added to or subtracted from the operand address prior to or during the execution of an instruction.

Indexing
A technique of address modification often implemented by means of index registers.

Initialize
To set certain counters, switches and addresses at specified times in a computer routine.

Input
1. The data to be processed.
2. The state or sequence of states occurring on a specified input channel.
3. The device or collective set of devices used for bringing data into another device.
4. A channel for impressing a state on a device or logic element.

Input Area
The area of internal storage into which data is transferred from external storage.

Input/Output
1. Common abbreviation I/O. A general term for the equipment used to communicate with a computer.
2. The data involved in such communication.
3. The media carrying the data for input/output.

Instruction
A statement that specifies an operation and the values or locations of all operands. In this context, the term instruction is preferable to the terms command or order which are sometimes used as synonyms. Command should be reserved for electronic signals. Order should be reserved for sequence, interpolation and related usage.

Instruction Format
The allocation of bits or characters of a machine instruction to specific functions.

Interrupt
1. A break in the normal flow of a system or routine such that the flow can be resumed from that point at a later time.
2. To cause an interrupt.

Language
1. A defined set of characters which are used to form symbols, words, etc., and the rules for combining these into meaningful communication, e.g., English, French, Algol, FORTRAN, COBOL, etc.
2. A combination of a vocabulary and rules of syntax.

Linkage
The interconnections between a main routine and a closed routine, i.e., entry and exit for a closed routine from the main routine.

Load
To place data into internal storage.

Location
A position in storage that is usually identified by an address.

Loop
A sequence of instructions that is repeated until a terminal condition occurs.

Machine Address
Same as absolute address.

Machine Code
   Same as operation code.
Machine Instruction
   An instruction that the particular machine can recognize and execute.
Machine Language
   A language that is used directly by a given machine.
Macro Instruction
   A statement that is used in a source program and replaced by a specific sequence of machine instructions in the associated object program.
Magnetic Ink
   Ink containing particles of magnetic substance which can be detected or read by automatic devices; e.g., the ink used for printing on some bank checks for magnetic character recognition.
Magnetic Tape
   A tape with a magnetic surface on which data can be stored.
Main Storage
   The fastest general purpose storage of a computer. Also, for the Model 20, storage within the CPU that can be addressed both for reading and writing data.
Mask
   An alphameric character string consisting of one or more digits, used to test or alter the contents of storage positions.
Mnemonic Code
   A mnemonic code resembles the original word and is usually easy to remember, e.g., ED for edit and MVC for move characters.

Name
   An alphameric character string, normally used to identify a program.

Object Program
   A fully assembled program ready to be loaded in the computer.
Operand
   That which is operated upon. An operand is usually identified by an address part of an instruction.
Operation
   1. The act specified by a single computer instruction.
   2. A program step undertaken or executed by a computer, e.g., addition, multiplication, extraction, comparison, shift, or transfer. The operation is usually specified by the operation part of an instruction.
Operation Code
   The code that represents the specific operations of a computer.
Output
   1. Data that has been processed.
   2. The state or sequence of states occurring on a specified output channel.
   3. The device or collective set of devices used for taking data out of a device.
   4. A channel for expressing a state on a device or logic element.
Output Area
   The area of internal storage from which data is transferred to external storage.
Overflow
   1. That portion of data that exceeds the capacity of the allocated unit of storage.
   2. The generation of overflow as in (1).

Pack
   To combine two or more units of information into a single physical unit to conserve storage.
Padding
   A technique used to fill a block of information with dummy records, words or characters.
Printer
   A device which expresses coded characters as hard copy.
Program
   1. The plan for the solution of a problem including data gathering, processing and reporting.
   2. A group of related routines which solve a given problem.
Programming Language
   A language used to prepare computer programs.
Pseudo-Register
   A register with fixed contents used in conjunction with an IBM System/360 Model 20.
Punched Card
   1. A card punched with a pattern of holes to represent data.
   2. A card as in 1. before being punched.

Read
   To transfer information from an input device to internal or auxiliary storage.
Reader
   A device which converts information in one form of storage to information in another form of storage.
Register
   A device capable of storing a specified amount of data such as one halfword.
Relative Address
   An address expressed by a previously defined symbol and a displacement. (e.g., FLD+10).
Relocate
   In programming, to move a routine from one portion of internal storage to another and to automatically adjust the necessary address references so that the routine, in its new location, can be executed.
Reset
   To restore a storage device to pre-

scribed initial state, not necessarily that denoting zeros.

Restart
To return to a previous point in a program and resume operation from that point.

RLD card
RLD cards identify portions of the text that require modification owing to relocation (such as address constants).

Self-Defining Term
A term with an implied value (e.g., 300, X'2A' , C'F')

Source Language
A language that is an input to a given translation process.

Source Program
A program written in a source language.

Special Character
In a character set, a character that is neither a numeral nor a letter, e.g., -* $ = and blank.

Statement
In computer programming, a meaningful expression or generalized instruction in a source language.

Step
1. One instruction in a computer routine.
2. To cause a computer to execute one instruction.

Storage
1. Pertaining to a device into which data can be entered and from which it can be retrieved at a later time.
2. Loosely, any device that can store data.

Storage Capacity
The amount of data (in bytes) that can be contained in a storage device.

Store
1. To enter data into a storage device.
2. To retain data in a storage device.

Subroutine
A routine that can be part of another routine.

Switch
1. A symbol used to indicate a branching point, or a set of instructions to condition a branch.
2. A physical device which can alter flow.

Symbol Table
A mapping for a set of symbols to another set of symbols or numbers.

Symbolic Address
An address expressed in symbols convenient to the programmer.

Symbolic Language
An artificial language used in logical expressions, that avoids all ambiguities and inadequacies of natural languages.

System
1. A collection of consecutive operations and procedures required to accomplish a specific objective.
2. An assembly of objects united to form a functional unit.

Table
A collection of data, each item being uniquely identified either by some label or by its relative position.

Table Look-Up
A procedure for obtaining the function value corresponding to an argument from a table of function values.

Truncate
To cut off at a specified spot (as contrasted with round or pad).

TXT card
TXT cards contain the user program in machine language.

Unpack
To recover the original data from packed data.

Zero Suppression
The elimination of non-significant zeros in a number.

Zone
The 12, 11, or 0 punches in IBM card code.

INDEX

GC26-3602-6

IBM S/360 Model 20 CPS Basic Assemb. Lang. (S360(Mod.20)-21) Printed in U.S.A. GC26-3602-6

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

IBM System/360 Model 20
Card Programming Support
Basic Assembler Language

Order No. GC26-3602-6

READER'S COMMENT FORM

This sheet is for comments and suggestions about this manual. We would appreciate *your* views, favorable or unfavorable, in order to aid us in improving *this* publication. This form will be sent directly to the author's department. Please include your name and address if you wish a reply. Contact your IBM branch office for answers to technical questions about the system or when requesting additional publications. Thank you.

Name

Address

What is your occupation?

How did you use this manual?

As a reference source

As a classroom text

As a self-study text

Your comments* and suggestions:

*** We would especially appreciate your comments on any of the following topics:**

Clarity of the text	Accuracy	Index	Illustrations	Appearance	Paper
Organization of the text	Cross-references	Tables	Examples	Printing	Binding

GC26-3602-6

YOUR COMMENTS, PLEASE . . .

This manual is part of a library that serves as a reference source for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

CUT ALONG THIS LINE

Fold Fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N. Y.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
112 East Post Road
White Plains, N.Y. 10601

Attention: Department 813 U

Fold Fold

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

IBM System/360 Model 20
Card Programming Support
Basic Assembler Language

Order No. GC26-3602-6

READER'S COMMENT FORM

This sheet is for comments and suggestions about this manual. We would appreciate *your* views, favorable or unfavorable, in order to aid us in improving *this* publication. This form will be sent directly to the author's department. Please include your name and address if you wish a reply. Contact your IBM branch office for answers to technical questions about the system or when requesting additional publications. Thank you.

Name

Address

What is your occupation?

How did you use this manual?

As a reference source

As a classroom text

As a self-study text

Your comments* and suggestions:

*** We would especially appreciate your comments on any of the following topics:**

Clarity of the text	Accuracy	Index	Illustrations	Appearance	Paper
Organization of the text	Cross-references	Tables	Examples	Printing	Binding

GC26-3602-6

YOUR COMMENTS, PLEASE . . .

This manual is part of a library that serves as a reference source for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

CUT ALONG THIS LINE

Fold Fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N. Y.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
112 East Post Road
White Plains, N.Y. 10601

Attention: Department 813 U

Fold Fold

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
(USA Only)

IBM World Trade Corporation
United Nations Plaza, New York, New York 10017
ational)