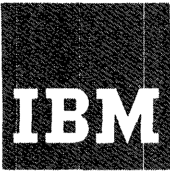
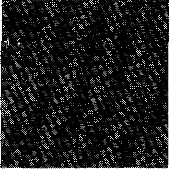
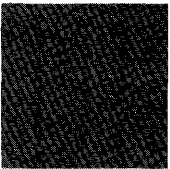
File No. S360-21
Form C28-6514-4

OS



Systems Reference Library

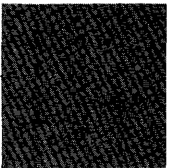




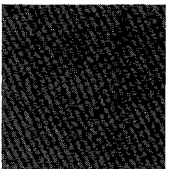
IBM System/360 Operating System

Assembler Language

This publication contains specifications for the IBM System/360 Operating System Assembler Language.

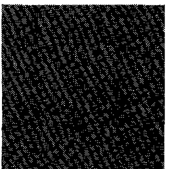


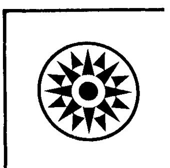
The assembler language is a symbolic programming language used to write programs for the IBM System/360. The language provides a convenient means for representing the machine instructions and related data necessary to program the IBM System/360. The IBM System/360 Operating System Assembler Program processes the language and provides auxiliary functions useful in the preparation and documentation of a program, and includes facilities for processing the assembler macro language.

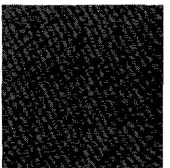


Part I of this publication describes the assembler language.

Part II of this publication describes an extension of the assembler language -- the macro language -- used to define macro-instructions.







PREFACE

This publication is a reference manual for the programmer using the assembler language and its features.

Part I of this publication presents information common to all parts of the language followed by specific information concerning the symbolic machine instruction codes and the assembler program functions provided for the programmer's use. Part II contains a description of the macro language and procedures for its use.

Appendixes A through I follow Part II. Appendixes A through F are associated with Parts I and II and present such items as a summary chart for constants, instruction listings, character set representations, and other aids to programming. Appendix G contains macro-language summary charts, and Appendix H is a sample program. Appendix I is a features comparison chart of System/360 assemblers.

Knowledge of IBM System/360 machine operations, particularly storage addressing, data formats, and machine instruction formats and functions, is prerequisite to using this publication, as is experience with programming concepts and techniques or completion of basic courses of instruction in these areas. IBM System/360 machine operations are discussed in the publication IBM System/360: Principles of Operation, Form A22-6821. Information on program assembling, linkage editing, executing, interpreting listings, and assembler programming considerations is provided in IBM System/360 Operating System: Assembler (E) Programmer's Guide, Form C28-6595.

The following publications are referred to in this publication:

IBM System/360 Operating System: Introduction, Form C28-6534

IBM System/360 Operating System: Linkage Editor, Form C28-6538

IBM System/360 Operating System: Control Program Services, Form C28-6541

IBM System/360 Operating System: Concepts and Facilities, Form C28-6535

IBM System/360 Operating System: Data Management, Form C28-6537

IBM System/360 Operating System: Utilities, Form C28-6586

MAJOR REVISION (February 1966)

This publication is a major revision of the previous edition, Form C28-6514-3, which is now obsolete. This new edition should be reviewed in its entirety since changes have been made throughout. Most of the changes are additional specifications to previously existing material; one new conditional assembly instruction, ACTR, has been added, and the appendixes have been revised.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.

A form for readers' comments appears at the back of this publication. It may be mailed directly to IBM. Address any additional comments concerning this publication to the IBM Corporation, Programming Systems Publications, Department D58, PO Box 390, Poughkeepsie, N. Y. 12602

© International Business Machines Corporation, 1964

Continuation Lines

When it is necessary to continue a statement on another line, the following rules apply.

1. Enter a continuation character (not blank, and not part of the statement coding) in column 72 of the line.

2. Continue the statement on the next line, starting in column 16. All columns to the left of column 16 must be blank.

3. When more than one line is needed, each line to be continued must have a character (not blank, and not part of the statement coding) entered in column 72.

4. Only two continuation lines may be used for a statement except in a macro-instruction, which allows as many as necessary.

Statement Boundaries

Source statements are normally contained in columns 1-71 of statement lines and columns 16-71 of any continuation lines. Therefore, columns 1, 71, and 16 are referred to as the "begin," "end," and "continue" columns, respectively. (This convention may be altered by use of the Input Format Control (ICTL) assembler instruction discussed later in this publication. The continuation character, if used, always immediately follows the "end" column.

Statement Format

Statements may consist of one to four entries in the statement field. They are, from left to right: a name entry, an operation entry, an operand entry, and a comments entry. These entries must be separated by one or more blanks, and must be written in the order stated.

The coding form (Figure 2-1) is ruled to provide an 8-character name field, a 5-character operation field, and a 56-character operand and/or comments field.

If desired, the programmer may disregard these boundaries and write the name, operation, operand, and comment entries in other positions, subject to the following rules:

1. The entries must not extend beyond statement boundaries (either the conventional boundaries, or as designated by the programmer via the ICTL instruction).

2. The entries must be in proper sequence, as stated previously.

3. The entries must be separated by one or more blanks.

4. If used, a name entry must be written starting in the begin column.

5. The name and operation entries must be completed in the first line of the statement, including at least one blank following the operation entry.

A description of the name, operation, operand, and comments entries follows:

Name Entries: The name entry is a symbol created by the programmer to identify a statement. A name entry is usually optional. The symbol must consist of eight characters or less, and be entered with the first character appearing in the begin column. If the begin column is blank, the assembler program assumes no name has been entered. No blanks may appear in the symbol.

Operation Entries: The operation entry is the mnemonic operation code specifying the machine operation, assembler, or macro-instruction operation desired. An operation entry is mandatory and cannot appear in a continuation line. It must start at least one position to the right of the begin column. Valid mnemonic operation codes for machine and assembler operations are contained in Appendixes D and E of this publication. Valid operation codes consist of five characters or fewer for machine or assembler-instruction operation codes, and eight characters or fewer for macro-instruction operation codes. No blanks may appear within the operation entry.

Operand Entries: Operand entries are the coding that identifies and describes data to be acted upon by the instruction, by indicating such things as storage locations, masks, storage-area lengths, or types of data.

Depending on the needs of the instruction, one or more operands may be written. Operands are required for all machine instructions.

Operands must be separated by commas, and no blanks may intervene between operands and the commas that separate them.

The operands may not contain embedded blanks, except as follows:

> If character representation is used to specify a constant, a literal, or immediate data in an operand, the character string may contain blanks, e.g., C'A D'.

Comments Entries: Comments are descriptive items of information about the program that are to be inserted in the program listing. All 256 valid characters (see "Character Set" in this section), including blanks may be used in writing a comment. The entry may follow the operand entry and must be separated from it by a blank; comments entries cannot extend beyond the end column (column 71).

An entire statement field may be used for a comment by placing an asterisk in the begin column. Extensive comments entries may be written by using a series of lines with an asterisk in the begin column of each line or by using continuation lines.

In statements where an optional operand entry is omitted but a comments entry is desired, the absence of the operand entry must be indicated by a comma preceded and followed by one or more blanks, as follows:

Name	Operation	Operand
	END	, COMMENT

Statement Example: The following example illustrates the use of name, operation, operand, and comments entries. A compare instruction has been named by the symbol COMP; the operation entry (CR) is the mnemonic operation code for a register-to-register compare operation, and the two operands (5,6) designate the two general registers whose contents are to be compared. The comments entry reminds the programmer that he is comparing "new sum" to "old" with this instruction.

Name	Operation	Operand
COMP	CR	5,6 NEW SUM TO OLD

Identification-Sequence Field

The identification-sequence field of the coding form (columns 73-80) is used to enter program identification and/or statement sequence characters. The entry is optional. If the field, or a portion of it, is used for program identification, the identification is punched in the source cards and reproduced in the printed listing of the source program.

To aid in keeping source statements in order, the programmer may number the cards in this field. These characters are punched into their respective cards, and during assembly the programmer may request the assembler to verify this sequence by use of the Input Sequence Checking (ISEQ) assembler instruction. This instruction is discussed in Section 5, under "Program Control Instructions."

Summary of Statement Format

The entries in a statement must always be separated by at least one blank and must be in the following order: name, operation, operand(s), comment.

Every statement requires an operation entry. Name and comment entries are optional. Operand entries are required for all machine instructions and most assembler instructions.

The name and operation entries must be completed in the first statement line, including at least one blank following the operation entry.

The name and operation entries must not contain blanks. Operand entries must not have blanks preceding or following the commas that separate them.

A name entry must always start in the begin column.

If the column after the end column is blank, the next line must start a new statement. If the column after the end column is not blank, the following line will be treated as a continuation line.

All entries must be contained within the designated begin, end, and continue column boundaries.

```
14+BETA-(GAMMA-LAMBDA)
```

When the assembler program encounters terms in parentheses in combination with other terms, it first reduces the combination of terms inside the parentheses to a single value which may be absolute or relocatable, depending on the combination of terms. This value then is used in reducing the rest of the combination to another single value.

Terms in parentheses may be included within a set of terms in parentheses:

```
A+B-(C+D-(E+F)+10)
```

The innermost set of terms in parentheses is evaluated first. Five levels of parentheses are allowed; a level of parentheses is a left parenthesis and its corresponding right parenthesis. Parentheses which occur as part of an operand format do not count in this limit. An arithmetic combination of terms is evaluated as described in the next section "Expressions."

EXPRESSIONS

This subsection discusses the expressions used in coding operand entries for source statements. Two types of expressions, absolute and relocatable, are presented along with the rules for determining these attributes of an expression.

As shown in Figure 2-2, an expression is composed of a single term or an arithmetic combination of terms. The following are examples of valid expressions:

```
*                       BETA*10
AREA1+X'2D'             B'101'
*+32                    C'ABC'
N-25                    29
FIELD+332               L'FIELD
FIELD                   LAMBDA+GAMMA
(EXIT-ENTRY+1)+GO       TEN/TWO
=F'1234'
ALPHA-BETA/(10+AREA*L'FIELD)-100
```

The rules for coding expressions are:

1. An expression may not start with an arithmetic operator, (+-/*). Therefore, the expression -A+BETA is invalid. However, the expression 0-A+BETA is valid.

2. An expression may not contain two terms or two operators in succession.

3. An expression may not consist of more than 16 terms.

4. An expression may not have more than five levels of parentheses.

5. A multi-term expression may not contain a literal.

Evaluation of Expressions

A single term expression, e.g., 29, BETA, *, L'SYMBOL, takes on the value of the term involved.

A multi term expression, e.g., BETA+10, ENTRY-EXIT, 25*10+A/B, is reduced to a single value, as follows:

1. Each term is given its value.

2. Every expression is computed to 32 bits.

3. Arithmetic operations are performed left to right. Multiplication and division are done before addition and subtraction, e.g., A+B*C is evaluated as A+(B*C), not (A+B)*C. The computed result is the value of the expression.

4. Division always yields an integer result; any fractional portion of the result is dropped. E.g., 1/2*10 yields a zero result, whereas 10*1/2 yields 5.

5. Division by zero is valid and yields a zero result.

Parenthesized multiterm expressions used in an expression are processed before the rest of the terms in the expression, e.g., in the expression A+BETA*(CON-10), the term CON-10 is evaluated first and the resulting value used in computing the final value of the expression.

Negative values are carried in 2s complement form. Final values of expressions are the truncated rightmost 24 bits of the results. The value of an expression before truncation must be in the range -2^{24} through $2^{24}-1$. A negative result is considered to be a 3-byte positive value. Intermediate results have a range of -2^{31} through $2^{31}-1$.

Absolute and Relocatable Expressions

An expression is called absolute if its value is unaffected by program relocation.

An expression is called relocatable if its value changes upon program relocation.

The two types of expressions, absolute and relocatable, take on these characteristics from the term or terms composing them.

ABSOLUTE EXPRESSION: An absolute expression may be an absolute term or any arithmetic combination of absolute terms. An absolute term may be a non-relocatable symbol, any of the self-defining terms, or the length attribute reference. As indicated in Figure 2-2, all arithmetic operations are permitted between absolute terms.

An absolute expression may contain relocatable terms (RT) -- alone or in combination with absolute terms (AT) -- under the following conditions:

1. There must be an even number of relocatable terms in the expression.

2. The relocatable terms must be paired. Each pair of terms must have the same relocatability attribute, i.e., they appear in the same control section in this assembly (see "Program Sectioning and Linking," Section 3). Each pair must consist of terms with opposite signs. The paired terms do not have to be contiguous, e.g., RT+AT-RT.

3. No relocatable term may enter into a multiply or divide operation. Thus, RT-RT*10 is invalid. However, (RT-RT)*10 is valid.

The pairing of relocatable terms (with opposite signs and the same relocatability attribute) cancels the effect of relocation. Therefore the value represented by the paired terms remains constant, regardless of program relocation. For example, in the absolute expression A-Y+X, A is an absolute term, and X and Y are relocatable terms with the same relocatability attribute. If A equals 50, Y equals 25, and X equals 10, the value of the expression would be 35. If X and Y are relocated by a factor of 100 their values would then be 125 and 110. However, the expression would still evaluate as 35 (50-125+110=35).

An absolute expression reduces to a single absolute value.

The following examples illustrate absolute expressions. A is an absolute term; X and Y are relocatable terms with the same relocatability attribute.

```
A-Y+X
A
A*A
X-Y+A
*-Y  (a reference to the location counter
     must be paired with another relocatable
     term from the same control section,
     i.e., with the same relocatability
     attribute)
```

RELOCATABLE EXPRESSIONS: A relocatable expression is one whose value would change by n if the program in which it appears is relocated n bytes away from its originally assigned area of storage. All relocatable expressions must have a positive value.

A relocatable expression may be a relocatable term. A relocatable expression may contain relocatable terms -- alone or in combination with absolute terms -- under the following conditions:

1. There must be an odd number of relocatable terms.
2. All the relocatable terms but one must be paired. Pairing is described in "Absolute Expression."
3. The unpaired term must not be directly preceded by a minus sign.
4. No relocatable term may enter into a multiply or divide operation.

A relocatable expression reduces to a single relocatable value. This value is the value of the odd relocatable term, adjusted by the values represented by the absolute terms and/or paired relocatable terms associated with it. The relocatability attribute is that of the odd relocatable term.

For example, in the expression W-X+W-10, W and X are relocatable terms with the same relocatability attribute. If initially W equals 10 and X equals 5, the value of the expression is 5. However, upon relocation this value will change. If a relocation factor of 100 is applied, the value of the expression is 105. Note that the value of the paired terms, W-X, remains constant at 5 regardless of relocation. Thus, the new value of the expression, 105, is the result of the value of the odd term (W) adjusted by the values of W-X and 10.

The following examples illustrate relocatable expressions. A is an absolute term, W and X are relocatable terms with the same relocatability attribute, Y is a relocatable term with a different relocatability attribute.

```
Y-32*A   W-X+*          =F'1234'(literal)
W-X+Y                   A*A+W-W+Y
* (reference to         W-X+W
  location counter)     Y
```

Just as machine instructions are used to request the computer to perform a sequence of operations during program execution time, so assembler instructions are requests to the assembler to perform certain operations during the assembly. Assembler-instruction statements, in contrast to machine-instruction statements, do not always cause machine-instructions to be included in the assembled program. Some, such as DS and DC, generate no instructions but do cause storage areas to be set aside for constants and other data. Others, such as EQU and SPACE, are effective only at assembly time; they generate nothing in the assembled program and have no effect on the location counter.

The following is a list of assembler instructions.

Symbol Definition Instruction
EQU - Equate Symbol

Data Definition Instructions
DC - Define Constant
DS - Define Storage
CCW - Define Channel Command Word

* Program Sectioning and Linking Instructions
START - Start Assembly
CSECT - Identify Control Section
CXD - Cumulative Length of External Dummy Section
DSECT - Identify Dummy Section
DXD - Define External Dummy Section
ENTRY - Identify Entry-Point Symbol
EXTRN - Identify External Symbol
COM - Identify Blank Common Control Section

* Base Register Instructions
USING - Use Base Address Register
DROP - Drop Base Address Register

Listing Control Instructions
TITLE - Identify Assembly Output
EJECT - Start New Page
SPACE - Space Listing
PRINT - Print Optional Data

Program Control Instructions
ICTL - Input Format Control
ISEQ - Input Sequence Checking
ORG - Set Location Counter
LTORG - Begin Literal Pool
CNOP - Conditional No Operation
COPY - Copy Predefined Source Coding
END - End Assembly
PUNCH - Punch a Card
REPRO - Reproduce Following Card

* Discussed in Section 3.

SYMBOL DEFINITION INSTRUCTION

EQU -- EQUATE SYMBOL

The EQU instruction is used to define a symbol by assigning to it the length, value, and relocatability attributes of an expression in the operand field. The format of the EQU instruction statement is as follows:

Name	Operation	Operand
A symbol	EQU	An expression

The expression in the operand field may be absolute or relocatable. Any symbols appearing in the expression must be previously defined.

The symbol in the name field is given the same length, value, and relocatibility attributes as the expression in the operand field. The length attribute of the symbol is that of the leftmost (or only) term of the expression. In the case of EQU to * or to a self-defining term, the length attribute is 1. The value attribute of the symbol is the value of the expression.

The EQU instruction is the means of equating symbols to register numbers, immediate data, and other arbitrary values. The following examples illustrate how this might be done:

Name	Operation	Operand
REG2	EQU	2 (general register)
TEST	EQU	X'3F'(immediate data)

To reduce programming time, the programmer can equate symbols to frequently used expressions and then use the symbols as operands in place of the expressions. Thus, in the statement:

Name	Operation	Operand
FIELD	EQU	ALPHA-BETA+GAMMA

FIELD is defined as ALPHA-BETA+GAMMA and may be used in place of it. Note, however, that ALPHA, BETA, and GAMMA must all be previously defined. If the final result of the expression is negative, it is treated as if it were positive.

The assembler will assign a length attribute of 1 in an EQU to * statement.

DATA DEFINITION INSTRUCTIONS

There are three data definition instruction statements: Define Constant (DC), Define Storage (DS), and Define Channel Command Word (CCW).

These statements are used to enter data constants into storage, to define and reserve areas of storage, and to specify the contents of channel command words. The statements may be named by symbols so that other program statements can refer to the fields generated from them. The discussion of the DC instruction is far more extensive than that of the DS instruction, because the DS instruction is written in the same format as the DC instruction and may specify some or all of the information that the DC instruction provides. Only the function and treatment of the statements vary. For this reason, the DC instruction is presented first and discussed in more detail than the DS instruction.

DC -- DEFINE CONSTANT

The DC instruction is used to provide constant data in storage. It may specify one constant or a series of constants, thereby relieving the programmer of the necessity to write a separate data definition statement for each constant desired. Furthermore, a variety of constants may be specified: fixed-point, floating-point, decimal, hexadecimal, character, and storage addresses. (Data constants are generally called constants unless they are created from storage addresses, in which case they are called address constants.) The format of the DC instruction statement is as follows:

Name	Operation	Operand
A symbol or blank	DC	One or more operands in the format described below, each separated by a comma

Each operand consists of four subfields: the first three describe the constant, and the fourth subfield provides the constant or constants. The first and third subfields may be omitted, but the second and fourth must be specified. Note that more than one constant may be specified in the fourth subfield for most types of constants. Each constant so specified must be of the same type; the descriptive subfields that precede the constants apply to all of them. No blanks may occur within any of the subfields (unless provided as characters in a character constant or a character self-defining term), nor may they occur between the subfields of an operand. Similarly, blanks may not occur between operands and the commas that separate them when multiple operands are being specified.

The subfields of each DC operand are written in the following sequence:

1	2	3	4
Duplication Factor	Type	Modifiers	Constant(s)

Although the constants specified in one operand must have the same characteristics, each operand may specify different types of constants. For example, in a DC instruction with three operands, the first operand might specify four decimal constants, the second a floating-point constant, and the third a character constant.

The symbol that names the DC instruction is the name of the constant (or first constant if the instruction specifies more than one). Relative addressing (e.g., SYMBOL+2) may be used to address the various constants if more than one has been specified, because the number of bytes allocated to each constant can be determined.

The value attribute of the symbol naming the DC instruction is the address of the leftmost byte (after alignment) of the first, or only, constant. The length attribute depends on two things: the type of constant being defined and the presence of a length specification. Implied lengths are assumed for the various constant types

The implied length of BLCON is two bytes. A reference to BLCON would cause the entire two bytes to be referenced.

When bit-length specification is used in association with multiple constants (see "Operand Subfield 4: Constant" following), each succeeding constant in the list is assembled starting at the next available bit. Figure 5-3 illustrates this.

As coded:

Name	Operation	Operand
BLMCON	DC	FL.10'673,21,57'

In storage:

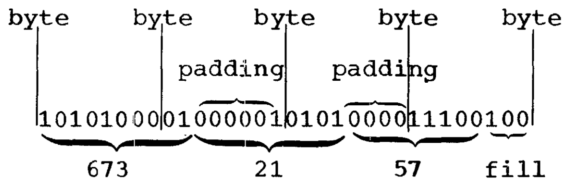


Figure 5-3. Bit-Length Specification (Multiple Constants)

The symbol used as a name entry in a DC assembler instruction takes on the length attribute of the first constant in the list; therefore the implied length of BLMCON in Figure 5-3 is two bytes.

If duplication is specified, filling occurs once at the end of the field occupied by the duplicated constant(s).

When bit-length specification is used in association with multiple operands, assembly of the constant(s) in each succeeding operand starts at the next available bit. Figure 5-4 illustrates this.

As coded:

Name	Operation	Operand
BLMOCON	DC	FL.7'9',CL.10'AB',XL.14'C4'

In storage:

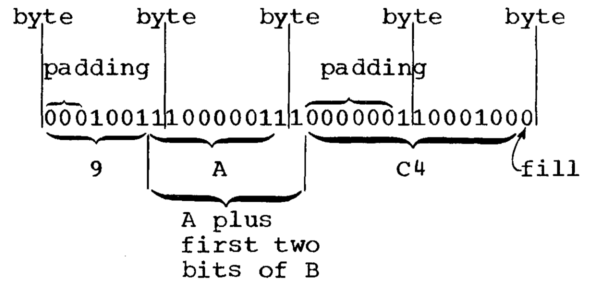


Figure 5-4. Bit-Length Specification (Multiple Operands)

In Figure 5-4, three different types of constants have been specified, one to an operand. Note that the character constant 'AB' which normally would occupy 16 bits is truncated on the right to fit the 10-bit field designated. Note that filling occurs only at the end of the field occupied by all the constants.

SCALE MODIFIER: This modifier is written as Sn, where n is either a decimal value or an absolute expression enclosed by parentheses. Any symbol in the expression must be previously defined. The decimal self-defining term or the parenthesized expression may be preceded by a sign; if none is present, a plus sign is assumed. The maximum values for scale modifiers are summarized in Appendix F.

A scale modifier may be used with fixed-point (F, H) and floating-point (E, D) constants only. It is used to specify the amount of internal scaling that is desired, as follows.

Scale Modifier for Fixed-Point Constants: the scale modifier specifies the power of two by which the constant must be multiplied after it has been converted to its binary representation. Just as multiplication of a decimal number by a power of 10 causes the decimal point to move, multiplication of a binary number by a power of two causes the binary point to move. This multiplication has the effect of moving the binary point away from its assumed position in the binary field; the assumed position being to the right of the rightmost position.

Thus, the scale modifier indicates either of the following: (1) the number of binary positions to be occupied by the fractional portion of the binary number, or (2) the number of binary positions to be deleted from the integral portion of the binary number. A positive scale of x shifts the integral portion of the number x binary positions to the left, thereby re-

serving the rightmost x binary positions for the fractional portion. A negative scale shifts the integral portion of the number right, thereby deleting rightmost integral positions. If a scale modifier does not accompany a fixed-point constant containing a fractional part, the fractional part is lost.

In all cases where positions are lost because of scaling (or the lack of scaling), rounding occurs in the leftmost bit of the lost portion. The rounding is reflected in the rightmost position saved.

Scale Modifier for Floating-Point Constants: Only a positive scale modifier may be used with a floating-point constant. It indicates the number of hexadecimal positions that the fraction is to be shifted to the right. Note that this shift amount is in terms of hexadecimal positions, each of which is four binary positions. (A positive scaling actually indicates that the point is to be moved to the left. However, a floating-point constant is always converted to a fraction, which is hexadecimally normalized. The point is assumed to be at the left of the leftmost position in the field. Since the point cannot be moved left, the fraction is shifted right.)

Thus, scaling that is specified for a floating-point constant provides an assembled fraction that is unnormalized, i.e., contains hexadecimal zeros in the leftmost positions of the fraction. When the fraction is shifted, the exponent is adjusted accordingly to retain the correct magnitude. When hexadecimal positions are lost, rounding occurs in the leftmost hexadecimal position of the lost portion. The rounding is reflected in the rightmost hexadecimal position saved.

EXPONENT MODIFIER: This modifier is written as En, where n is either a decimal self-defining term or an absolute expression enclosed by parentheses. Any symbols in the expression must be previously defined. The decimal value or the parenthesized expression may be preceded by a sign; if none is present, a plus sign is assumed. The maximum values for exponent modifiers are summarized in Appendix F.

An exponent modifier may be used with fixed-point (F, H) and floating-point (E, D) constants only. The modifier denotes the power of 10 by which the constant is to be multiplied before its conversion to the proper internal format.

This modifier is not to be confused with the exponent of the constant itself, which is specified as part of the constant and is explained under "Operand Subfield 4: Constant." The exponent modifier affects each constant in the operand, whereas the exponent written as part of the constant only pertains to that constant. Thus, a constant may be specified with an exponent of +2, and an exponent modifier of +5 may precede the constant. In effect, the constant has an exponent of +7.

Note that there is a maximum value, both positive and negative, listed in Appendix F for exponents. This applies to the exponent modifier and to the sum of the exponent modifier and the exponent specified as part of the constant.

Operand Subfield 4: Constant

This subfield supplies the constant (or constants) described by the subfields that precede it. A data constant (all types except A,Y,S, Q and V) is enclosed by apostrophes. An address constant (types A, Y, S, Q and V) is enclosed by parentheses. To specify two or more constants in the subfield, the constants must be separated by commas and the entire sequence of constants must be enclosed by the appropriate delimiters (i.e., apostrophes or parentheses). Thus, the format for specifying the constant(s) is one of the following:

Single Constant	Multiple Constants*
'constant'	'constant,...,constant'
(constant)	(constant,...,constant)

* Not permitted for character, hexadecimal, and binary constants.

All constant types except character (C), hexadecimal (X), binary (B), packed decimal (P), and zoned decimal (Z), are aligned on the proper boundary, as shown in Appendix F, unless a length modifier is specified. In the presence of a length modifier, no boundary alignment is performed. If an operand specifies more than one constant, any necessary alignment applies to the first constant only. Thus, for an operand that provides five full-word constants, the first would be aligned on a full-word boundary, and the rest would automatically fall on full-word boundaries.

The total storage requirement of an operand is the product of the length times the number of constants in the operand times the duplication factor (if present) plus any bytes skipped for boundary alignment of the first constant. If more than one operand is present, the storage requirement is derived by summing the requirements for each operand.

If an address constant contains a location counter reference, the location counter value that is used is the storage address of the first byte the constant will occupy. Thus, if several address constants in the same instruction refer to the location counter, the value of the location counter varies from constant to constant. Similarly, if a single constant is specified (and it is a location counter reference) with a duplication factor, the constant is duplicated with a varying location counter value.

The following text describes each of the constant types and provides examples.

Character Constant -- C: Any of the valid 256 punch combinations may be designated in a character constant. Only one character constant may be specified per operand. Since multiple constants within an operand are separated by commas, an attempt to specify two character constants would result in interpreting the comma separating them as a character.

Special consideration must be given to representing apostrophes and ampersands as characters. Each single apostrophe or ampersand desired as a character in the constant must be represented by a pair of apostrophes or ampersands. Only one apostrophe or ampersand appears in storage.

The maximum length of a character constant is 256 bytes. No boundary alignment is performed. Each character is translated into one byte. Double apostrophes or double ampersands count as one character. If no length modifier is given, the size in bytes of the character constant is equal to the number of characters in the constant. If a length modifier is provided, the result varies as follows:

1. If the number of characters in the constant exceeds the specified length, as many rightmost bytes and/or bits as necessary are dropped.

2. If the number of characters is less than the specified length, the excess rightmost bytes and/or bits are filled with blanks.

In the following example, the length attribute of FIELD is 12:

Name	Operation	Operand
FIELD	DC	C'TOTAL IS 110'

However, in this next example, the length attribute is 15, and three blanks appear in storage to the right of the zero:

Name	Operation	Operand
FIELD	DC	CL15'TOTAL IS 110'

In the next example, the length attribute of FIELD is 12, although 13 characters appear in the operand. The two ampersands count as only one byte.

Name	Operation	Operand
FIELD	DC	C'TOTAL IS &&10'

Note that in the next example, a length of four has been specified, but there are five characters in the constant.

Name	Operation	Operand
FIELD	DC	3CL4'ABCDE'

The generated constant would be:

ABCDABCDABCD

On the other hand, if the length had been specified as six instead of four, the generated constant would have been:

ABCDE ABCDE ABCDE

Note that the same constant could be specified as a literal.

Name	Operation	Operand
	MVC	AREA(12),=3CL4'ABCDE'

Hexadecimal Constant -- X: A hexadecimal constant consists of one or more of the hexadecimal digits, which are 0-9 and A-F. Only one hexadecimal constant may be specified per operand. The maximum length of a hexadecimal constant is 256 bytes (512 hexadecimal digits). No boundary alignment is performed.

Constants that contain an even number of hexadecimal digits are translated as one byte per pair of digits. If an odd number

of digits is specified, the leftmost byte has the leftmost four bits filled with a hexadecimal zero, while the rightmost four bits contain the odd (first) digit.

If no length modifier is given, the implied length of the constant is half the number of hexadecimal digits in the constant (assuming that a hexadecimal zero is added to an odd number of digits). If a length modifier is given, the constant is handled as follows:

1. If the number of hexadecimal digit pairs exceeds the specified length, the necessary leftmost bits (and/or bytes) are dropped.

2. If the number of hexadecimal digit pairs is less than the specified length, the necessary bits (and/or bytes) are added to the left and filled with hexadecimal zeros.

An eight-digit hexadecimal constant provides a convenient way to set the bit pattern of a full binary word. The constant in the following example would set the first and third bytes of a word to 1s:

Name	Operation	Operand
	DS	0F
TEST	DC	X'FF00FF00'

The DS instruction sets the location counter to a full word-boundary.

The next example uses a hexadecimal constant as a literal and inserts 1s into bits 24 through 31 of register 5.

Name	Operation	Operand
	IC	5,=X'FF'

In the following example, the digit A would be dropped, because five hexadecimal digits are specified for a length of two bytes:

Name	Operation	Operand
ALPHACON	DC	3XL2'A6F4E'

The resulting constant would be 6F4E, which would occupy the specified two bytes. It would then be duplicated three times, as requested by the duplication factor. If it had merely been specified as X'A6F4E', the resulting constant would have had a hexadecimal zero in the leftmost position:

0A6F4E

Binary Constant -- B: A binary constant is written using 1s and 0s enclosed in apostrophes. Only one binary constant may be specified in an operand. Duplication and length may be specified. The maximum length of a binary constant is 256 bytes.

The implied length of a binary constant is the number of bytes occupied by the constant including any padding necessary. Padding or truncation takes place on the left. The padding bit used is a 0.

The following example shows the coding used to designate a binary constant. BCON would have a length attribute of 1.

Name	Operation	Operand
BCON	DC	B'11011101'
BTRUNC	DC	BL1'100100011'
BPAD	DC	BL1'101'

BTRUNC would assemble with the leftmost bit truncated, as follows:

00100011

BPAD would assemble with five zeros as padding, as follows:

00000101

Fixed-Point Constants -- F and H: A fixed-point constant is written as a decimal number, which may be followed by a decimal exponent if desired. The number may be an integer, a fraction, or a mixed number (i.e., one with integral and fractional portions). The format of the constant is as follows:

1. The number is written as a signed or unsigned decimal value. The decimal point may be placed before, within, or after the number, or it may be omitted, in which case the number is assumed to be an integer. A positive sign is assumed if an unsigned number is specified. Unless a scale modifier accompanies a mixed number or fraction, the fractional portion is lost, as explained under "Subfield 3: Modifiers."

2. The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal self-defining term specifying the exponent of the factor 10. The exponent may be in the range -85 to +75. If an unsigned exponent is specified, a plus sign is assumed. The exponent causes the value of the constant to be adjusted by the power of 10 that it specifies before the constant is converted to its binary form. The exponent may exceed the permissible range for exponents, provided that the sum of the exponent and the exponent modifier does not exceed that range.

The number is converted to a binary number, and scaling is performed if specified. The binary number is then rounded and assembled into the proper field, according to the specified or implied length. The resulting number will not differ from the exact value by more than one in the last place. If the value of the number exceeds the length specified or implied, the sign is lost, the necessary leftmost bits are truncated to the length of the field, and the value is then assembled into the whole field. Any duplication factor that is present is applied after the constant is assembled. A negative number is carried in 2s complement form.

An implied length of four bytes is assumed for a full-word (F) and two bytes for a half-word (H), and the constant is aligned to the proper full-word or half-word if a length is not specified. However, any length up to and including eight bytes may be specified for either type of constant by a length modifier, in which case no boundary alignment occurs.

Maximum and minimum values, exclusive of scaling, for fixed-point constants are:

Length	Max	Min
8	$2^{63}-1$	-2^{63}
4	$2^{31}-1$	-2^{31}
2	$2^{15}-1$	-2^{15}
1	$2^{7}-1$	-2^{7}

A field of three full-words is generated from the statement shown below. The location attribute of CONWRD is the address of the leftmost byte of the first word, and the length attribute is 4, the implied length for a full-word fixed-point constant. The expression CONWRD+4 could be used to address the second constant (second word) in the field.

Name	Operation	Operand
CONWRD	DC	3F'658474'

The next statement causes the generation of a two-byte field containing a negative constant. Notice that scaling has been specified in order to reserve six bits for the fractional portion of the constant.

Name	Operation	Operand
HALFCON	DC	HS6'-25.46'

The next constant (3.50) is multiplied by 10 to the -2 before being converted to its binary format. The scale modifier reserves 12 bits for the fractional portion.

Name	Operation	Operand
FULLCON	DC	HS12'3.50E-2'

The same constant could be specified as a literal:

Name	Operation	Operand
	AH	7,=HS12'3.50E-2'

The final example specifies three constants. Notice that the scale modifier requests four bits for the fractional portion of each constant. The four bits are provided whether or not the fraction exists.

Name	Operation	Operand
THREECON	DC	FS4'10,25.3,100'

Floating-Point Constants -- E and D: A floating-point constant is written as a decimal number, which may be followed by a decimal exponent, if desired. The number may be an integer, a fraction, or a mixed number (i.e., one with integral and fractional portions). The format of the constant is as follows:

1. The number is written as a signed or unsigned decimal value. The decimal point may be placed before, within, or after the number, or it may be omitted, in which case, the number is assumed to be an integer. A positive sign is assumed if an unsigned number is specified.

2. The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal value specifying the exponent of the factor 10. The exponent may be in the range -85 to +75. If an unsigned exponent is specified, a plus sign is assumed. The exponent may exceed the permissible range for exponents, provided that the sum of the exponent and the exponent modifier does not exceed that range.

Machine format for a floating-point number is in two parts: the portion containing the exponent, which is sometimes called the characteristic, followed by the portion containing the fraction, which is sometimes called the mantissa. Therefore, the number specified as a floating-point constant must be converted to a fraction before it can be translated into the proper format. For example, the constant 27.35E2 represents the number 27.35 times 10 to the 2nd. Represented as a fraction, it would be .2735 times 10 to the 4th, the exponent having been modified to reflect the shifting of the decimal point. The exponent may also be affected by the presence of an exponent modifier, as explained under "Operand Subfield 3: Modifiers." Thus, the exponent is also altered before being translated into machine format.

The exponent is then translated into its binary equivalent, and the fraction is converted to a binary number. Scaling is performed if specified; if not, the fraction is normalized (leading hexadecimal zeros are removed). Rounding of the fraction is then performed according to the specified or implied length, and the number is stored in the proper field. The resulting number will not differ from the exact value by more than one in the last place. Within the portion of the floating-point field allocated to the fraction, the hexadecimal point is assumed to be to the left of the leftmost hexadecimal digit, and the fraction occupies the leftmost portion of the field. Negative fractions are carried in true representation, not in the 2s complement form.

An implied length of four bytes is assumed for a full word (E) and eight bytes is assumed for a double word (D). The constant is aligned at the proper word or double-word boundary if a length is not specified. However, any length up to and including eight bytes may be specified for either type of constant by a length modifier, in which case no boundary alignment occurs.

Any of the following statements could be used to specify 46.415 as a positive, full-word, floating-point constant; the last is a machine-instruction statement with a literal operand. Note that the last two constants contain an exponent modifier.

Name	Operation	Operand
	DC	E'46.415'
	DC	E'46415E-3'
	DC	E'+464.15E-1'
	DC	E'+.46415E+2'
	DC	EE2'.46415'
	AE	6,=EE2'.46415'

The following would each be generated as double-word floating-point constants.

Name	Operation	Operand
FLOAT	DC	DE+4'+46,-3.729,+473'

Decimal Constants -- P and Z: A decimal constant is written as a signed or unsigned decimal value. If the sign is omitted, a plus sign is assumed. The decimal point may be written wherever desired or may be omitted. Scaling and exponent modifiers may not be specified for decimal constants. The maximum length of a decimal constant is 16 bytes. No word boundary alignment is performed.

The placement of a decimal point in the definition does not affect the assembly of the constant in any way, because, unlike fixed-point and floating-point constants, a decimal constant is not converted to its binary equivalent. The fact that a decimal constant is an integer, a fraction, or a mixed number is not pertinent to its generation. Furthermore, the decimal point is not assembled into the constant. The programmer may determine proper decimal point alignment either by defining his data so that the point is aligned or by selecting machine-instructions that will operate on the data properly (i.e., shift it for purposes of alignment).

If zoned decimal format is specified (Z), each decimal digit is translated into

one byte. The translation is done according to the character set shown in Appendix A. The rightmost byte contains the sign as well as the rightmost digit. For packed decimal format (P), each pair of decimal digits is translated into one byte. The rightmost digit and the sign are translated into the rightmost byte. The bit configuration for the digits is identical to the configurations for the hexadecimal digits 0-9 as shown in Section 3 under "Hexadecimal Self-Defining Value." For both packed and zoned decimals, a plus sign is translated into the hexadecimal digit C, and a minus sign into the digit D.

If an even number of packed decimal digits is specified, one digit will be left unpaired because the rightmost digit is paired with the sign. Therefore, in the leftmost byte, the leftmost four bits will be set to zeros and the rightmost four bits will contain the odd (first) digit.

If no length modifier is given, the implied length for either constant is the number of bytes the constant occupies (taking into account the format, sign, and possible addition of zero bits for packed decimals). If a length modifier is given, the constant is handled as follows:

1. If the constant requires fewer bytes than the length specifies, the necessary number of bytes is added to the left. For zoned decimal format, the decimal digit zero is placed in each added byte. For packed decimals, the bits of each added byte are set to zero.

2. If the constant requires more bytes than the length specifies, the necessary number of leftmost digits or pairs of digits is dropped, depending on which format is specified.

Examples of decimal constant definitions follow.

Name	Operation	Operand
	DC	P'+1.25'
	DC	Z'-543'
	DC	Z'79.68'
	DC	PL3'79.68'

The following statement specifies both packed and zoned decimal constants. The length modifier applies to each constant in the first operand (i.e., to each packed decimal constant). Note that a literal could not specify both operands.

Name	Operation	Operand
DECIMALS	DC	PL8'+25.8,-3874, +2.3',Z'+80,-3.72'

The last example illustrates the use of a packed decimal literal.

Name	Operation	Operand
	UNPK	OUTAREA,=PL2'+25'

ADDRESS CONSTANTS: An address constant is a storage address that is translated into a constant. Address constants are normally used for initializing base registers to facilitate the addressing of storage. Furthermore, they provide the means of communicating between control sections of a multisection program. However, storage addressing and control section communication are also dependent on the use of the USING assembler instruction and the loading of registers. Coding examples that illustrate these considerations are provided in Section 3 under "Programming with the Using Instruction."

An address constant, unlike other types of constants, is enclosed in parentheses. If two or more address constants are specified in an operand, they are separated by commas, and the entire sequence is enclosed by parentheses. There are four types of address constants: A, Y, S, and V. A relocatable address constant may not be specified with bit lengths.

Complex Relocatable Expressions: A complex relocatable expression can only be used to specify an A-type or Y-type address constant. These expressions contain two or more unpaired relocatable terms and/or negative relocatable terms in addition to any absolute or paired relocatable terms that may be present. A complex relocatable expression might consist of external symbols and designate an address in an independent assembly that is to be linked and loaded with the assembly containing the address constant.

A-Type Address Constant: This constant is specified as an absolute, relocatable, or complex relocatable expression. (Remember that an expression may be single term or multiterm.) The value of the expression is calculated to 32 bits as explained in Section 2 with one exception: the maximum

value of the expression may be $2^{31}-1$. The value is then truncated on the left, if necessary, to the specified or implied length of the field and assembled into the rightmost bits of the field. The implied length of an A-type constant is four bytes, and alignment is to a full-word boundary unless a length is specified, in which case no alignment will occur. The length that may be specified depends on the type of expression used for the constant; a length of .1-4 bytes may be used for an absolute expression, while a length of only 3 or 4 may be used for a relocatable or complex relocatable expression.

In the following examples, the field generated from the statement named ACONST contains four constants, each of which occupies four bytes. Note that there is a location counter reference in one. The value of the location counter will be the address of the first byte allocated to the fourth constant. The second statement shows the same set of constants specified as literals (i.e., address constant literals).

Name	Operation	Operand
ACONST	DC	A(108,LOOP, END-STRT,*+4096)
	LM	4,7,=A(108,LOOP, END-STRT,*+4096)

Note: When the location counter reference occurs in a literal, as in the LM instruction above, the value of the location counter is the address of the first byte of the instruction.

Y-Type Address Constant: A Y-type address constant has much in common with the A-type constant. It too is specified as an absolute, relocatable, or complex relocatable expression. The value of the expression is also calculated to 32 bits as explained in Section 2. However, the maximum value of the expression may be only $2^{15}-1$. The value is then truncated, if necessary, to the specified or implied length of the field and assembled into the right-most bits of the field. The implied length of a Y-type constant is two bytes, and alignment is to a half-word boundary unless a length is specified, in which case no alignment will occur. The maximum length of a Y-type address constant is two bytes. If length specification is used, a length of two bytes may be designated for a relocatable or complex expression and .1 to 2 bytes for an absolute expression.

Warning: Specification of relocatable Y-type address constants should be avoided in programs destined to be executed on machines having more than 32,767 bytes of storage capacity. In any case Y-type address constants should not be used in programs to be executed under Operating System/360 control.

S-Type Address Constant: The S-type address constant is used to store an address in base-displacement form.

The constant may be specified in two ways:

1. As an absolute or relocatable expression, e.g., S(BETA).

2. As two absolute expressions, the first of which represents the displacement value and the second, the base register, e.g., S(400(13)).

The address value represented by the expression in (1) will be broken down by the assembler into the proper base register and displacement value. An S-type constant is assembled as a half word and aligned on a half-word boundary. The leftmost four bits of the assembled constant represents the base register designation, the remaining 12 bits the displacement value.

If length specification is used, only two bytes may be specified. S-type address constants may not be specified as literals.

Q-Type Address Constant: This constant is used to reserve storage for the offset of an external dummy section. This offset is added to the address of the block of storage allocated to external dummy sections to access the desired section. The constant is specified as one relocatable symbol which has been previously defined in a DXD or DSECT statement. The implied length of a Q-type address constant is four bytes and boundary alignment is to a full word; a length of 1-4 bytes may be specified. No bit length specification is permitted in a Q-type constant. In the following example the constant VALUE has been previously defined in a DXD or DSECT statement. To access VALUE the value of A is added to the base address of the block of storage allocated for external dummy sections. Q-type address constants may not be specified in literals.

Name	Operation	Operand
A	DC	Q(VALUE)

V-Type Address Constant: This constant is used to reserve storage for the address of an external symbol that is used for effecting branches to other programs. The constant may not be used for external data references. The constant is specified as one relocatable symbol, which need not be identified by an EXTRN statement. Whatever symbol is used is assumed to be an external symbol by virtue of the fact that it is supplied in a V-type address constant.

Note that specifying a symbol as the operand of a V-type constant does not constitute a definition of the symbol for this assembly. The implied length of a V-type address constant is four bytes, and boundary alignment is to a full word. A length modifier may be used to specify a length of either three or four bytes, in which case no such boundary alignment occurs. In the following example, 12 bytes will be reserved, because there are three symbols. The value of each assembled constant will be zero until the program is loaded.

The one to three operands may include an operand from each of the following groups:

1. ON - A listing is printed.

   OFF - No listing is printed.

2. GEN - All statements generated by macro-instructions are printed.

   NOGEN - Statements generated by macro-insructions are not printed with the exception of MNOTE with a severity code (other than *) which will print regardless of NOGEN. However, the macro-instruction itself will appear in the listing.

3. DATA - Constants are printed out in full in the listing.

   NODATA - Only the leftmost eight bytes are printed on the listing.

A program may contain any number of PRINT statements. A PRINT statement controls the printing of the assembly listing until another PRINT statement is encountered.

Until the first PRINT statement (if any) is encountered, the following is assumed:

Name	Operation	Operand
	PRINT	ON, NODATA, GEN

For example, if the statement:

Name	Operation	Operand
	DC	XL256'00'

appears in a program, 256 bytes of zeros are assembled. If the statement:

Name	Operation	Operand
	PRINT	DATA

is the last PRINT statement to appear before the DC statement, all 256 bytes of zeros are printed in the assembly listing. However, if:

Name	Operation	Operand
	PRINT	NODATA

is the last PRINT statement to appear before the DC statement, only eight bytes of zeros are printed in the assembly listing.

Whenever an operand is omitted, it is assumed to be unchanged and continues according to its last specification.

The hierarchy of print control statements is:

1. ON and OFF
2. GEN and NOGEN
3. DATA and NODATA

Thus with the following statement nothing would be printed.

Name	Operation	Operand
	PRINT	OFF, DATA, GEN

PROGRAM CONTROL INSTRUCTIONS

The program control instructions are used to specify the end of an assembly, to set the location counter to a value or word boundary, to insert previously written coding in the program, to specify the placement of literals in storage, to check the sequence of input cards, to indicate statement format, and to punch a card. Except for the CNOP and COPY instructions, none of these assembler instructions generate instructions or constants in the object program.

ICTL -- INPUT FORMAT CONTROL

The ICTL instruction allows the programmer to alter the normal format of his source program statements. The ICTL statement must precede all other statements in the source program and may be used only once. The format of the ICTL instruction statement is as follows:

Name	Operation	Operand
Blank	ICTL	1-3 decimal values of the form b,e,c

Operand b specifies the begin column of the source statement. It must always be specified, and must be from 1-40, inclusive. Operand e specifies the end column of the source statement. The end column, when specified, must be from 41-80, inclusive; when not specified, it is assumed to be 71. The column after the end column is used to indicate whether the next card is a continuation card. Operand c specifies the continue column of the source statement. The continue column, when specified, must be from 2-40 and must be greater than b. If the continue column is not specified, or if column 80 is specified as the end column, the assembler assumes that there are no continuation cards, and all statements must be contained on a single card. The operand forms b,,c and b, are invalid.

If no ICTL statement is used in the source program, the assembler assumes that 1, 71, and 16 are the begin, end, and continue columns, respectively.

The next example designates the begin column as column 25. Since the end column is not specified, it is assumed to be column 71. No continuation cards are recognized because the continue column is not specified.

Name	Operation	Operand
	ICTL	25

ISEQ -- INPUT SEQUENCE CHECKING

The ISEQ instruction is used to check the sequence of input cards. The format of the ISEQ instruction statement is as follows:

Name	Operation	Operand
Blank	ISEQ	Two decimal values of the form l,r; or blank

The operands l and r, respectively, specify the leftmost and rightmost columns of the field in the input cards to be checked. Operand r must be equal to or greater than operand l. Columns to be checked must not be between the begin and end columns.

Sequence checking begins with the first card following the ISEQ statement. Comparison of adjacent cards makes use of the eight-bit internal collating sequence. (See Appendix A.) Each card checked must be higher than the preceding card.

An ISEQ statement with a blank operand terminates the operation. Checking may be resumed with another ISEQ statement.

Sequence checking is only performed on statements contained in the source program. Statements inserted by the COPY assembler-instruction or generated by a macro-instruction are not checked for sequence. Also macro-definitions in a macro library are not checked.

PUNCH -- PUNCH A CARD

The PUNCH assembler-instruction causes the data in the operand to be punched into a card. One PUNCH statement produces one punched card. As many PUNCH statements may be used as are necessary. The format is:

Name	Operation	Operand
Blank	PUNCH	1 to 80 characters enclosed in apostrophes

Using character representation, the operand is written as a string of up to 80 characters enclosed in apostrophes. All characters, including blank, are valid. The position immediately to the right of the left apostrophe is regarded as column one of the card to be punched. Substitution is performed for variable symbols in the operand. Special consideration must be given to representing apostrophes and ampersands as characters. Each apostrophe or ampersand desired as a character in the constant must be represented by a pair of apostrophes or ampersands. Only one apostrophe or ampersand appears in storage.

PUNCH statements may occur anywhere within a program, except before macro definitions. They may occur within a macro definition but not between the end of a

macro definition and the beginning of the next macro definition. If a PUNCH statement occurs before the first control section, the resultant card will precede all other cards in the object program card deck; otherwise the card will be punched in place. No sequence number or identification is punched in the card.

REPRO -- REPRODUCE FOLLOWING CARD

The REPRO assembler-instruction causes data on the following statement line to be punched into a card. The data is not processed; it is punched in a card, and no substitution is performed for variable symbols. No sequence number or identification is punched on the card. One REPRO instruction produces one punched card. The REPRO instruction may not appear before a macro definition. REPRO statements that occur before all statements composing the first or only control section will punch cards which precede all other cards of the object deck. The format is:

Name	Operation	Operand
Blank	REPRO	Blank

The line to be reproduced may contain any combination of up to 80 valid characters. Characters may be entered starting in column 1 and continuing through column 80 of the line. Column 1 of the line corresponds to column 1 of the card to be punched.

ORG -- SET LOCATION COUNTER

The ORG instruction is used to alter the setting of the location counter for the current control section. The format of the ORG instruction statement is:

Name	Operation	Operand
Blank	ORG	A relocatable expression or blank

Any symbols in the expression must have been previously defined. The unpaired relocatable symbol must be defined in the same control section in which the ORG statement appears.

The location counter is set to the value of the expression in the operand. If the operand is omitted, the location counter is set to the next available (unused) location for that control section.

An ORG statement must not be used to specify a location below the beginning of the control section in which it appears. The following is invalid if it appears less than 500 bytes from the beginning of the current control section.

Name	Operation	Operand
	START ORG	1000 *-500

If it is desired to reset the location counter to a value that is one byte beyond the highest location yet assigned (in the control section), the following statement would be used:

Name	Operation	Operand
	ORG	

If previous ORG statements have reduced the location counter for the purpose of redefining a portion of the current control section, an ORG statement with an omitted operand can then be used to terminate the effects of such statements and restore the location counter to its highest setting plus one.

LTORG -- BEGIN LITERAL POOL

The LTORG instruction causes all literals since the previous LTORG (or start of the program) to be assembled at appropriate boundaries starting at the first doubleword boundary following the LTORG statement. If no literals follow the LTORG statement, alignment of the next instruction (which is not a LTORG instruction) will occur. Bytes skipped are not zeroed. The format of the LTORG instruction statement is:

Name	Operation	Operand
Symbol or blank	LTORG	Not used

The symbol represents the address of the first byte of the literal pool. It has a length attribute of 1.

Special Addressing Consideration

Any literals used after the last LTORG statement in a program are placed at the end of the first control section. If there are no LTORG statements in a program, all literals used in the program are placed at the end of the first control section. In these circumstances the programmer must ensure that the first control section is always addressable. This means that the base address register for the first control section should not be changed through usage in subsequent control sections. If the programmer does not wish to reserve a register for this purpose, he may place a LTORG statement at the end of each control section thereby ensuring that all literals appearing in that section are addressable.

Duplicate Literals

If duplicate literals occur within the range controlled by one LTORG statement, only one literal is stored. Literals are considered duplicates only if their specifications are identical. A literal will be stored, even if it appears to duplicate another literal, if it is an A-type address constant containing any reference to the location counter.

The following examples illustrate how the assembler stores pairs of literals, if the placement of each pair is controlled by the same LTORG statement.

```
X'F0'
        Both are stored
C'0'

XL3'0'
        Both are stored
HL3'0'

A(*+4)
        Both are stored
A(*+4)

X'FFFF'
        Identical; the first is stored
X'FFFF'
```

CNOP -- CONDITIONAL NO OPERATION

The CNOP instruction allows the programmer to align an instruction at a specific half-word boundary. If any bytes must be skipped in order to align the instruction properly, the assembler ensures an unbroken instruction flow by generating no-operation instructions. This facility is useful in creating calling sequences consisting of a linkage to a subroutine followed by parameters such as channel command words (CCW).

The CNOP instruction ensures the alignment of the location counter setting to a half-word, word, or double-word boundary. If the location counter is already properly aligned, the CNOP instruction has no effect. If the specified alignment requires the location counter to be incremented, one to three no-operation instructions are generated, each of which uses two bytes.

The format of the CNOP instruction statement is as follows:

Name	Operation	Operand
Blank	CNOP	Two absolute expressions of the form b,w

Any symbols used in the expressions in the operand field must have been previously defined.

Operand b specifies at which byte in a word or double word the location counter is to be set; b can be 0, 2, 4, or 6. Operand w specifies whether byte b is in a word (w=4) or double word (w=8). The following pairs of b and w are valid:

b,w	Specifies
0,4	Beginning of a word
2,4	Middle of a word
0,8	Beginning of a double word
2,8	Second half word of a double word
4,8	Middle (third half word) of a double word
6,8	Fourth half word of a double word

A macro-definition consists of:

1. A macro-definition header statement.

2. A macro-instruction prototype statement.

3. Zero or more model statements, COPY statements, MEXIT, MNOTE, or conditional assembly instructions.

4. A macro-definition trailer statement.

Except for MEXIT, MNOTE, and conditional assembly instructions, this section of the publication describes all of the statements that may be used to prepare macro-definitions. Conditional assembly instructions are described in Section 9. MEXIT and MNOTE instructions are described in Section 10.

Macro-definitions appearing in a source program must appear before all PUNCH and REPRO statements and all statements which pertain to the first control section. Specifically, only the listing control instructions (EJECT, PRINT, SPACE, and TITLE), ICTL and ISEQ instructions, and comments statements may occur before the macro-definitions. All but the ICTL instruction may appear between macro-definitions if there is more than one definition in the source program.

MACRO -- MACRO-DEFINITION HEADER

The macro-definition header statement indicates the beginning of a macro-definition. It must be the first statement in every macro-definition. The format of this statement is:

Name	Operation	Operand
Blank	MACRO	Blank

MEND -- MACRO-DEFINITION TRAILER

The macro-definition trailer statement indicates the end of a macro-definition. It must be the last statement in every macro-definition. The format of this statement is:

Name	Operation	Operand
Blank	MEND	Blank

MACRO-INSTRUCTION PROTOTYPE

The macro-instruction prototype statement (hereafter called the prototype statement) specifies the mnemonic operation code and the format of all macro-instructions that refer to the macro-definition. It must be the second statement of every macro-definition. The format of this statement is:

Name	Operation	Operand
A symbolic parameter or blank	A symbol	Zero or more symbolic parameters, separated by commas

The symbolic parameters are used in the macro-definition to represent the name field and operands of the corresponding macro-instruction. A description of symbolic parameters appears under "Symbolic Parameters."

The name field of the prototype statement may be blank, or it may contain a symbolic parameter.

The symbol in the operation field is the mnemonic operation code that must appear in all macro-instructions that refer to this macro-definition. The mnemonic operation code must not be the same as the mnemonic operation code of another macro-definition in the source program or of a machine or assembler instruction as listed in Appendix G.

The operand field may contain 0 to 200 symbolic parameters separated by commas. If there are no symbolic parameters, comments may not appear.

The following is a prototype statement.

Name	Operation	Operand
&NAME	MOVE	&TO,&FROM

Statement Format

The prototype statement may be written in a format different from that used for assembler language statements. The normal format is described in Part I of this publication. The alternate format described here allows the programmer to write an operand on each line, and allows the interspersing of operands and comments in the statement.

In the alternate format, as in the normal format, the name and operation fields must appear on the first line of the statement, and at least one blank must follow the operation field on that line. Both types of statement formats may be used in the same prototype statement.

The rules for using the alternate statement format are:

1. If an operand is followed by a comma and a blank, and the column after the end column contains a nonblank character, the operand field may be continued on the next line starting in the continue column. More than one operand may appear on the same line.

2. Comments may appear after the blank that indicates the end of an operand, up to and including the end column.

3. If the next line starts after the continue column, the information entered on the next line is considered comments, and the operand field is considered terminated. Any subsequent continuation lines are considered comments.

Note: A prototype statement may be written on as many continuation lines as necessary. When using normal format, the operands of a prototype statement must begin on the first statement line or in the continue column of the second line.

The following examples illustrate: (1) the normal statement format, (2) the alternate statement format, and (3) the combination of both statement formats.

Name	Operation	Operand Comments	
NAME1	OP1	OPERAND1,OPERAND2,OPERAN D3 THIS IS THE NORMAL STATEMENT FORMAT	X X
NAME2	OP2	OPERAND1, THIS IS THE AL OPERAND2,OPERAND3, TERNA TE STATEMENT FORMAT	X X
NAME3	OP3	OPERAND1, THIS IS A COMB OPERAND2,OPERAND3,OPERAN D4,OPERAND5 INATION OF BOTH STATEMENT FORMATS	X X X

MODEL STATEMENTS

Model statements are the macro-definition statements from which the desired sequences of assembler language statements are generated. Zero or more model statements may follow the prototype statement. A model statement consists of one to four fields. They are, from left to right, the name, operation, operand, and comments fields.

The name field may be blank, or it may contain a symbol or symbolic parameter. (Neither an * nor .* may be substituted in the begin column of a model statement.)

The operation entry may contain any machine, or assembler instruction as listed in Section 5, or macro-instruction mnemonic operation code, except COPY, END, ICTL, ISEQ, and PRINT; or it may contain a variable symbol. Variable symbols may not be used to generate the following mnemonic operation codes, nor may variable symbols be used in the name and operand entries of these instructions: COPY, END, ICTL, or ISEQ. Variable symbols may not be used to generate CSECT, DSECT, PRINT, REPRO, START or macro-instruction mnemonic operation codes. Variable symbols may not be used to generate the name and operation code of the ACTR instruction or operation codes not listed in Section 5.

Variable symbols may also be used outside macro-definitions to generate mnemonic operation codes with the preceding restrictions. Although COPY statements may not be used as model statements, they may be part of a macro-definition. The use of COPY statements is described under "COPY statements."

The operand entry may contain ordinary symbols or variable symbols. Model statement fields must follow the rules for

the macro-instruction are the symbol HERE, then HERE replaces each occurrence of &A in the macro-definition. However, if &A is a SET symbol, the value assigned to &A can be changed, and a different value can replace each occurrence of &A in the macro-definition.

The same variable symbol may not be used as a symbolic parameter and as a SET symbol in the same macro-definition.

The following illustrates this rule.

Name	Operation	Operand
&NAME	MOVE	&TO,&FROM

If the statement above is a prototype statement, then &NAME, &TO, and &FROM may not be used as SET symbols in the macro-definition.

The same variable symbol may not be used as two different types of SET symbols in the same macro-definition. Similarly, the same variable symbol may not be used as two different types of SET symbols outside macro-definitions.

For example, if &A is a SETA symbol in a macro-definition, it cannot be used as a SETC symbol in that definition. Similarly, if &A is a SETA symbol outside macro-definitions, it cannot be used as a SETC symbol outside macro-definitions.

The same variable symbol may be used in two or more macro-definitions and outside macro-definitions. If such is the case, the variable symbol will be considered a different variable symbol each time it is used.

For example, if &A is a variable symbol (either SET symbol or symbolic parameter) in one macro-definition, it can be used as a variable symbol (either SET symbol or symbolic parameter) in another definition. Similarly, if &A is a variable symbol (SET symbol or symbolic parameter) in a macro-definition, it can be used as a SET symbol outside macro-definitions.

All variable symbols may be concatenated with other characters in the same way that symbolic parameters may be concatenated with other characters. The rules for concatenating symbolic parameters with other characters are in Section 7 under the subsection "Symbolic Parameters."

Variable symbols in macro-instructions are replaced by the values assigned to them, immediately prior to the start of processing the definition. If a SET symbol is used in the operand field of a macro-instruction, and the value assigned to the SET symbol is equivalent to the sublist notation, the operand is not considered a sublist.

ATTRIBUTES

The assembler assigns attributes to macro-instruction operands and to symbols in the program. These attributes may be referred to only in conditional assembly instructions.

There are six kinds of attributes. They are: type, length, scaling, integer, count, and number. Each kind of attribute is discussed in the paragraphs that follow.

If an outer macro-instruction operand is a symbol before substitution, then the attributes of the operand are the same as the corresponding attributes of the symbol. The symbol must appear in the name field of an assembler language statement or in the operand field of an EXTRN statement in the program. The statement must be outside macro-definitions and must not contain any variable symbols.

If an inner macro-instruction operand is a symbolic parameter, then the attributes of the operand are the same as the attributes of the corresponding outer macro-instruction operand.

If a macro-instruction operand is a sublist, the programmer may refer to the attributes of either the sublist or each operand in the sublist. The type, length, scaling, and integer attributes of a sublist are the same as the corresponding attributes of the first operand in the sublist.

All the attributes of macro-instruction operands may be referred to in conditional assembly instructions within macro-definitions. However, only the type, length, scaling, and integer attributes of symbols may be referred to in conditional assembly instructions outside macro-definitions. Symbols appearing in the name field of generated statements are not assigned attributes.

Each attribute has a notation associated with it. The notations are:

Attribute	Notation
Type	T'
Length	L'
Scaling	S'
Integer	I'
Count	K'
Number	N'

The programmer may refer to an attribute in the following ways:

1. In a statement that is outside macro-definitions, he may write the notation for the attribute immediately followed by a symbol. (e.g., T'NAME refers to the type attribute of the symbol NAME.)
2. In a statement that is in a macro-definition, he may write the notation for the attribute immediately followed by a symbolic parameter. (e.g., L'&NAME refers to the length attribute of the characters in the macro-instruction that correspond to symbolic parameter &NAME; L'&NAME(2) refers to the length attribute of the second operand in the sublist that corresponds to symbolic parameter &NAME.)

Type Attribute (T')

The type attribute of a macro-instruction operand, or a symbol is a letter.

The following letters are used for symbols that name DC and DS statements and for outer macro-instruction operands that are symbols that name DC or DS statements.

- A — A-type address constant, implied length, aligned, (also in CXD statement)
- B — Binary constant.
- C — Character constant.
- D — Long floating-point constant, implied length, aligned.
- E — Short floating-point constant, implied length, aligned.
- F — Full-word fixed-point constant, implied length, aligned.
- G — Fixed-point constant, explicit length.
- H — Half-word fixed-point constant, implied length, aligned.
- K — Floating-point constant, explicit length.
- P — Packed decimal constant.
- Q — Q-type address constant, implied length, aligned.
- R — A-, S-, Q-, V-, or Y-type address constant, explicit length.
- S — S-type address constant, implied length, aligned.
- V — V-type address constant, implied length, aligned.
- X — Hexadecimal constant.
- Y — Y-type address constant, implied length, aligned.
- Z — Zoned decimal constant.

The following letters are used for symbols (and outer macro-instruction operands that are symbols) that name statements other than DC or DS statements, or that appear in the operand field of an EXTRN statement.

- I — Machine instruction
- J — Control section name
- M — Macro-instruction
- T — External symbol
- W — CCW assembler instruction

The following letters are used for inner and outer macro-instruction operands only.

- N — Self-defining term
- O — Omitted operand

The following letter is used for inner and outer macro-instruction operands that cannot be assigned any of the above letters. This includes inner macro-instruction operands that are symbols. This letter is also assigned to symbols that name EQU and LTORG statements, to any symbols occurring more than once in the name field of source statements, and to all symbols naming statements with expressions as modifiers.

- U — Undefined

The attributes of A, B, C and D are undefined in the following example:

Name	Operation	Operand
A	DC	3FL (A-B)'75'
B	DC	(A-B) F'15'
C	DC	&X'1'
D	DC	FL(3-2)'1'

The programmer may refer to a type attribute in the operand field of a SETC instruction, or in character relations in the operand fields of SETB or AIF instructions.

Length (L'), Scaling (S'), and Integer (I') Attributes

The length, scaling, and integer attributes of macro-instruction operands, and symbols are numeric values.

The length attribute of a symbol (or of a macro-instruction operand that is a symbol) is as described in Part I of this publication.

Conditional assembly instructions must not refer to the length attributes of symbols or macro-instruction operands whose type attributes are the letters M, N, O, T, or U.

The symbolic parameter &NAME is used in the name field of the prototype statement (statement 1) and the first model statement (statement 2). In the macro-instruction (statement 3) a sequence symbol (.SYM) corresponds to the symbolic parameter &NAME. &NAME is not replaced by .SYM, and, therefore, the generated statement (statement 4) does not contain an entry in the name field.

LCLA,LCLB,LCLC -- DEFINE SET SYMBOLS

The format of these instructions is:

Name	Operation	Operand
Blank	LCLA, LCLB, or LCLC	One or more variable symbols, that are to be used as SET symbols, separated by commas

The LCLA, LCLB, and LCLC instructions are used to define and assign initial values to SETA, SETB, and SETC symbols, respectively. The SETA, SETB, and SETC symbols are assigned the initial values of 0, 0, and null character value, respectively.

The programmer should not define any SET symbol whose first four characters are &SYS.

All LCLA, LCLB, or LCLC instructions in a macro-definition must appear immediately after the prototype statement, and GBLA, GBLB or GBLC instructions, or LCLA, LCLB, or LCLC instructions. All LCLA, LCLB, or LCLC instructions outside macro-definitions must appear after all macro-definitions in the source program, after all GBLA, GBLB, and GBLC instructions outside macro-definitions, before all conditional assembly instructions, and PUNCH and REPRO statements outside macro-definitions, and before the first control section of the program.

SETA -- SET ARITHMETIC

The SETA instruction may be used to assign an arithmetic value to a SETA symbol. The format of this instruction is:

Name	Operation	Operand
A SETA symbol	SETA	An arithmetic expression

The expression in the operand field is evaluated as a signed 32-bit arithmetic value which is assigned to the SETA symbol in the name field. The minimum and maximum allowable values of the expression are -2^{31} and $+2^{31}-1$, respectively.

The expression may consist of one term or an arithmetic combination of terms. The terms that may be used alone or in combination with each other are self-defining terms, variable symbols, and the length, scaling, integer, count, and number attributes. Self-defining terms are described in Part I of this publication.

Note: A SETC variable symbol may appear in a SETA expression only if the value of the SETC variable is one to eight decimal digits. The decimal digits will be converted to a positive arithmetic value.

The arithmetic operators that may be used to combine the terms of an expression are + (addition), - (subtraction), * (multiplication), and / (division).

An expression may not contain two terms or two operators in succession, nor may it begin with an operator.

The following are valid operand fields of SETA instructions:

```
&AREA+X'2D'     I'&N/25
&BETA*10        &EXIT-S'&ENTRY+1
L'&HERE+32      29
```

The following are invalid operand fields of SETA instructions:

```
&AREAX'C'      (two terms in succession)
&FIELD+-       (two operators in succession)
-&DELTA*2      (begins with an operator)
*+32           (begins with an operator;
                 two operators in succession)
NAME/15        (NAME is not a valid term)
```

Evaluation of Arithmetic Expressions

The procedure used to evaluate the arithmetic expression in the operand field of a SETA instruction is the same as that

used to evaluate arithmetic expressions in assembler language statements. The only difference between the two types of arithmetic expressions is the terms that are allowed in each expression.

The following evaluation procedure is used:

1. Each term is given its numerical value.

2. The arithmetic operations are performed moving from left to right. However, multiplication and/or division are performed before addition and subtraction.

3. The computed result is the value assigned to the SETA symbol in the name field.

The arithmetic expression in the operand field of a SETA instruction may contain one or more sequences of arithmetically combined terms that are enclosed in parentheses. A sequence of parenthesized terms may appear within another parenthesized sequence. Only five levels of parentheses are allowed and an expression may not consist of more than 16 terms. Parentheses required for sublist notation, substring notation, and subscript notation count toward this limit.

The following are examples of SETA instruction operand fields that contain parenthesized sequences of terms.

```
(L'&HERE+32)*29
&AREA+X'2D'/(&EXIT-S'&ENTRY+1)
&BETA*10*(I'&N/25/(&EXIT-S'&ENTRY+1))
```

The parenthesized portion or portions of an arithmetic expression are evaluated before the rest of the terms in the expression are evaluated. If a sequence of parenthesized terms appears within another parenthesized sequence, the innermost sequence is evaluated first.

Using SETA Symbols

The arithmetic value assigned to a SETA symbol is substituted for the SETA symbol when it is used in an arithmetic expression. If the SETA symbol is not used in an arithmetic expression, the arithmetic value is converted to an unsigned integer, with leading zeros removed. If the value is zero, it is converted to a single zero.

The following example illustrates this rule:

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&TO,&FROM
		LCLA	&A,&B,&C,&D
1	&A	SETA	10
2	&B	SETA	12
3	&C	SETA	&A-&B
4	&D	SETA	&A+&C
	&NAME	ST	2,SAVEAREA
5		L	2,&FROM&C
6		ST	2,&TO&D
		L	2,SAVEAREA
		MEND	
	HERE	MOVE	FIELDA,FIELDB
	HERE	ST	2,SAVEAREA
		L	2,FIELDB2
		ST	2,FIELDA8
		L	2,SAVEAREA

Statements 1 and 2 assign to the SETA symbols &A and &B the arithmetic values +10 and +12, respectively. Therefore, statement 3 assigns the SETA symbol &C the arithmetic value -2. When &C is used in statement 5, the arithmetic value -2 is converted to the unsigned integer 2. When &C is used in statement 4, however, the arithmetic value -2 is used. Therefore, &D is assigned the arithmetic value +8. When &D is used in statement 6, the arithmetic value +8 is converted to the unsigned integer 8.

The following example shows how the value assigned to a SETA symbol may be changed in a macro-definition.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&TO,&FROM
		LCLA	&A
1	&A	SETA	5
	&NAME	ST	2,SAVEAREA
2		L	2,&FROM&A
3	&A	SETA	8
4		ST	2,&TO&A
		L	2,SAVEAREA
		MEND	
	HERE	MOVE	FIELDA,FIELDB
	HERE	ST	2,SAVEAREA
		L	2,FIELDB5
		ST	2,FIELDA8
		L	2,SAVEAREA

Therefore, if the type attribute is not the letter F, statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attribute is the letter F, statement 3 (the next sequential statement) is processed.

AGO -- UNCONDITIONAL BRANCH

The AGO instruction is used to unconditionally alter the sequence in which source program or macro-definition statements are processed by the assembler. The assembler assigns a maximum count of 4096 AIF and AGO branches that may be executed in the source program or in a macro-definition. When a macro-definition calls an inner macro-definition, the current value of the count is saved and a new count of 4096 is set up for the inner macro-definition. When processing in the inner definition is completed and a return is made to the higher definition, the saved count is restored. The format of this instruction is:

Name	Operation	Operand
A sequence symbol or blank	AGO	A sequence symbol

The statement named by the sequence symbol in the operand field is the next statement processed by the assembler.

The statement named by the sequence symbol may precede or follow the AGO instruction.

If an AGO instruction is part of a macro-definition, then the sequence symbol in the operand field must appear in the name field of a statement that is in that definition. If an AGO instruction appears outside macro-definitions, then the sequence symbol in the operand field must appear in the name field of a statement outside macro-definitions.

The following example illustrates the use of the AGO instruction.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&T,&F
1		AIF	(T'&T EQ 'F').FIRST
2		AGO	.END
3	.FIRST	AIF	(T'&T NE T'&F).END
	&NAME	ST	2,SAVEAREA
		L	2,&F
		ST	2,&T
		L	2,SAVEAREA
4	.END	MEND	

Statement 1 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is the letter F, statement 3 is the next statement processed by the assembler. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler.

Statement 2 is used to indicate to the assembler that the next statement to be processed is statement 4 (the statement named by sequence symbol .END).

ACTR -- CONDITIONAL ASSEMBLY LOOP COUNTER

The ACTR instruction is used to assign a maximum count (different from the standard count of 4096) to the number of AGO and AIF branches executed within a macro-definition or within the source program. The format of this instruction is as follows:

Name	Operation	Operand
Blank	ACTR	Any valid SETA expression

This statement, which can only occur immediately after the global and local declarations, causes a counter to be set to the value in the operand field. The counter is checked for zero or a negative value; if it is not zero or negative, it is decremented by one each time an AGO or AIF branch is executed. If the count is zero before decrementing, the assembler will take one of two actions:

1. If processing is being performed inside a macro definition, the entire nest of macro definitions will be terminated and the next source statement will be processed.

2. If the source program is being processed, an END card will be generated.

An ACTR instruction in a macro-definition affects only that definition; it has no effect on the number of AIF and AGO branches that may be executed in macro-definitions called.

ANOP -- ASSEMBLY NO OPERATION

The ANOP instruction facilitates conditional and unconditional branching to statements named by symbols or variable symbols.

The format of this instruction is:

Name	Operation	Operand
A sequence symbol	ANOP	Blank

If the programmer wants to use an AIF or AGO instruction to branch to another statement, he must place a sequence symbol in the name field of the statement to which he wants to branch. However, if the programmer has already entered a symbol or variable symbol in the name field of that statement, he cannot place a sequence symbol in the name field. Instead, the programmer must place an ANOP instruction before the statement and then branch to the ANOP instruction. This has the same effect as branching to the statement immediately after the ANOP instruction.

The following example illustrates the use of the ANOP instruction.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&T,&F
		LCLC	&TYPE
1		AIF	(T'&T EQ 'F').FTYPE
2	&TYPE	SETC	'E'
3	.FTYPE	ANOP	
4	&NAME	ST&TYPE	2,SAVEAREA
		L&TYPE	2,&F
		ST&TYPE	2,&T
		L&TYPE	2,SAVEAREA
		MEND	

Statement 1 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler. If the type attribute is the letter F, statement 4 should be processed next. However, since there is a variable symbol (&NAME) in the name field of statement 4, the required sequence symbol (.FTYPE) cannot be placed in the name field. Therefore, an ANOP instruction (statement 3) must be placed before statement 4.

Then, if the type attribute of the first operand is the letter F, the next statement processed by the assembler is the statement named by sequence symbol .FTYPE. The value of &TYPE retains its initial null character value because the SETC instruction is not processed. Since .FTYPE names an ANOP instruction, the next statement processed by the assembler is statement 4, the statement following the ANOP instruction.

CONDITIONAL ASSEMBLY ELEMENTS

The following chart summarizes the elements that can be used in each conditional assembly instruction. Each row in this chart indicates which elements can be used in a single conditional assembly instruction. Each column is used to indicate the conditional assembly instructions in which a particular element can be used.

The intersection of a column and a row indicates whether an element can be used in an instruction, and if so, in what fields of the instruction the element can be used. For example, the intersection of the first row and the first column of the chart indicates that symbolic parameters can be used in the operand field of SETA instructions. For example, the intersection of the first row and the first column of the chart indicates that symbolic parameters can be used in the operand field of SETA instructions.

Instruction	Type of Instruction	Program Interruption Possible						Condition Code Set			
		A	S	Ov	P	Op	Other	00	01	10	11
Add	RX	x	x	F				Sum=0	Sum<0	Sum>0	Overflow
Add	RR			F				Sum=0	Sum<0	Sum>0	Overflow
Add Decimal	SS, Decimal	x		D	x	x	Data	Sum=0	Sum<0	Sum>0	Overflow
Add Halfword	RX	x	x	F				Sum=0	Sum<0	Sum>0	Overflow
Add Logical	RX	x	x					Sum=0 (H)	Sum 0 (H)	Sum= 0 (I)	Sum 0 (I)
Add Logical	RR							Sum=0 (H)	Sum= 0 (H)	Sum= 0 (I)	Sum 0 (I)
Add Normalized, Long	RX, Floating Pt.	x	x	E		x	B, C	R	L	M	P
Add Normalized, Long	RR, Floating Pt.		x	E		x	B, C	R	L	M	P
Add Normalized, Short	RX, Floating Pt.	x	x	E		x	B, C	R	L	M	P
Add Normalized, Short	RR, Floating Pt.		x	E		x	B, C	R	L	M	P
Add Unnormalized, Long	RX, Floating Pt.	x	x	E		x	C	R	L	M	P
Add Unnormalized, Long	RR, Floating Pt.		x	E		x	C	R	L	M	P
Add Unnormalized, Short	RX, Floating Pt.	x	x	E		x	C	R	L	M	P
Add Unnormalized, Short	RR, Floating Pt.		x	E		x	C	R	L	M	P
Add Logical	RX	x	x					J	K		
And Logical	SS	x			x			J	K		
And Logical	RR							J	K		
And Logical Immediate	SI	x			x			J	K		
Branch and Link	RX							N	N	N	N
Branch and Link	RR							N	N	N	N
Branch on Condition	RX							N	N	N	N
Branch on Condition	RR							N	N	N	N
Branch on Count	RX							N	N	N	N
Branch on Count	RR							N	N	N	N
Branch on Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on High	RX, Ext. Mnemonic							N	N	N	N
Branch on Index High	RX, Ext. Mnemonic							N	N	N	N
Branch on Index Low or Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on Low	RX, Ext. Mnemonic							N	N	N	N
Branch if Mixed	RX, Ext. Mnemonic							N	N	N	N
Branch on Minus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on Not High	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Low	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Minus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Ones	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Plus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Zeros	RX, Ext. Mnemonic							N	N	N	N
Branch if Ones	RX, Ext. Mnemonic							N	N	N	N
Branch on Overflow	RX, Ext. Mnemonic							N	N	N	N
Branch on Plus	RX, Ext. Mnemonic							N	N	N	N
Branch if Zeros	RX, Ext. Mnemonic							N	N	N	N
Branch on Zero	RX, Ext. Mnemonic							N	N	N	N
Branch Unconditional	RX, Ext. Mnemonic							N	N	N	N
Branch Unconditional	RR, Ext. Mnemonic							N	N	N	N
Compare Algebraic	RX	x	x					Z	AA	BB	
Compare Algebraic	RR							Z	AA	BB	
Compare Decimal	SS, Decimal	x				x	Data	Z	AA	BB	
Compare Halfword	RX	x	x					Z	AA	BB	
Compare Logical	RX	x	x					Z	AA	BB	
Compare Logical	RX	x	x					Z	AA	BB	
Compare Logical	SS	x						Z	AA	BB	
Compare Logical Immediate	SI	x						Z	AA	BB	
Compare, Long	RX, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Long	RR, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Short	RX, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Short	RR, Floating Pt.		x			x		Z	AA	BB	
Convert to Binary	RX	x	x				Data, F	N	N	N	N
Convert to Decimal	RX	x	x		x			N	N	N	N

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format	
			Explicit	Implicit
Divide	D	5D	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Divide	DR	1D	R1, R2	
Divide Decimal	DP	FD	D1, (L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Divide, Long	DD	6D	R1, D2(X2, B2), or R1, D2(, B2)	R1, S2(X2) or R1, S2
Divide, Long	DDR	2D	R1, R2	
Divide, Short	DE	7D	R1, D2(X2, B2)or R1, D2(, B2)	R1, S2(X2) or R1, S2
Divide, Short	DER	3D	R1, R2	
Edit	ED	DE	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Edit and Mark	EDMK	DF	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Exclusive Or	X	57	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Exclusive Or	XC	D7	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Exclusive Or	XR	17	R1, R2	
Exclusive Or Immediate	XI	97	D1(B1), I2	S1, I2
Execute	EX	44	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) R1, S2
Halve, Long	HDR	24	R1, R2	
Halve, Short	HER	34	R1, R2	
Halt I/O	HIO	9E	D1(B1)	
Insert Character	IC	43	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Insert Storage Key	ISK	09	R1, R2	
Load	L	58	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load	LR	18	R1, R2	
Load Address	LA	41	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load and Test	LTR	12	R1, R2	
Load and Test, Long	LTDR	22	R1, R2	
Load and Test, Short	LTER	32	R1, R2	
Load Complement	LCR	13	R1, R2	
Load Complement, Long	LCDR	23	R1, R2	
Load Complement, Short	LCER	33	R1, R2	
Load Halfword	LH	48	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load, Long	LD	68	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load, Long	LDR	28	R1, R2	
Load Multiple	LM	98	R1, R3, D2(B2)	R1, R3, S2
Load Negative	LNR	11	R1, R2	
Load Negative, Long	LNDR	21	R1, R2	
Load Negative, Short	LNER	31	R1, R2	
Load Positive	LPR	10	R1, R2	
Load Positive, Long	LPDR	20	R1, R2	
Load Positive, Short	LPER	30	R1, R2	
Load PSW	LPSW	82	D1(B1)	
Load, Short	LE	78	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Load, Short	LER	38	R1, R2	
Move Characters	MVC	D2	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Move Immediate	MVI	92	D1(B1), I2	S1, I2
Move Numerics	MVN	D1	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Move with Offset	MVO	F1	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Move Zones	MVZ	D3	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
Multiply	M	5C	R1, D2(X2, B2)or R1, D2(, B2)	R1, S2(X2) or R1, S2
Multiply	MR	1C	R1, R2	
Multiply Decimal	MP	FC	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Mulitply Halfword	MH	4C	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Multiply, Long	MD	6C	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Multiply, Long	MDR	2C	R1, R2	
Multiply, Short	ME	7C	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Multiply, Short	MER	3C	R1, R2	
No Operation	NOP	47(BC 0)	D2(X2, B2) or D2(, B2)	S2(X2) or S2

Operand Format (Divide)

RR Format

x	-0x- Branching and Status Switching	-1x- Fixed-Point Fullword and Logical	-2x- Floating-Point Long	-3x- Floating-Point Short
0		Load Positive......LPR	Load Positive......LPDR	Load Positive......LPER
1		Load Negative......LNR	Load Negative......LNDR	Load Negative......LNER
2		Load and Test......LTR	Load and Test......LTDR	Load and Test......LTER
3		Load Complement....LCR	Load Complement....LCDR	Load Complement....LCER
4	Set Program Mask....SPM	AND................NR	Halve.............HDR	Halve.............HER
5	Branch and Link.....BALR	Compare Logical....CLR		
6	Branch on Count.....BCTR	OR.................OR		
7	Branch/Condition....BCR	Exclusive OR.......XR		
8	Set Key............SSK	Load...............LR	Load..............LDR	Load..............LER
9	Insert Key..........ISK	Compare............CR	Compare...........CDR	Compare...........CER
A	Supervisor Call.....SVC	Add................AR	Add N.............ADR	Add N.............AER
B		Subtract...........SR	Subtract N........SDR	Subtract..........SER
C		Multiply...........MR	Multiply..........MDR	Multiply..........MER
D		Divide.............DR	Divide............DDR	Divide............DER
E		Add Logical........ALR	Add U.............AWR	Add U.............AUR
F		Subtract Logical...SLR	Subtract U.........SWR	Subtract...........SUR

RX Format

x	-4x- Fixed-Point Halfword and Branching	-5x- Fixed-Point Fullword and Logical	-6x- Floating-Point Long	-7x- Floating-Point Short
0	Store.............STH	Store.............ST	Store.............STD	Store.............STE
1	Load Address.......LA			
2	Store Character....STC			
3	Insert Character...IC			
4	Execute...........EX	AND................N		
5	Branch and Link....BAL	Compare Logical....CL		
6	Branch on Count....BCT	OR.................O		
7	Branch/Condition...BC	Exclusive OR.......X		
8	Load..............LH	Load...............L	Load..............LD	Load..............LE
9	Compare...........CH	Compare............C	Compare...........CD	Compare...........CE
A	Add...............AH	Add................A	Add N.............AD	Add N.............AE
B	Subtract..........SH	Subtract...........S	Subtract N.........SD	Subtract N.........SE
C	Multiply..........MH	Multiply...........M	Multiply..........MD	Multiply..........ME
D		Divide.............D	Divide............DD	Divide............DE
E	Convert-Decimal....CVD	Add Logical........AL	Add U.............AW	Add U.............AU
F	Convert-Binary.....CVB	Subtract Logical...SL	Subtract U.........SW	Subtract U.........SU

RS,SI Format

x	-8x- Branching Status Switching and Shifting	-9x- Fixed-Point Logical and Input-Output	-Ax-	-Bx-
0	Set System Mask.....SSM	Store Multiple.....STM		
1		Test under Mask....TM		
2	Load PSW...........LPSW	Move...............MVI		
3	Diagnose	Test and Set.......TS		
4	Write Direct........WRD	AND................NI		
5	Read Direct.........RDD	Compare Logical....CLI		
6	Branch/High.........BXH	OR.................OI		
7	Branch/Low-Equal....BXLE	Exclusive OR.......XI		
8	Shift Right SL......SRL	Load Multiple......LM		
9	Shift Left SL.......SLL			
A	Shift Right S.......SRA			
B	Shift Left S........SLA			
C	Shift Right DL......SRDL	Start I-O.........SIO		
D	Shift Left DL.......SLDL	Test I-O..........TIO		
E	Shift Right D.......SRDA	Halt I-O..........HIO		
F	Shift Left D........SLDA	Test Channel.......TCH		

Operation Code Notes
- U = Unnormalized
- S = Single
- D = Double
- N = Normalized
- SL = Single Logical
- DL = Double Logical

SS Format

x	-Cx-	-Dx- Logical	-Ex-	-Fx- Decimal
0				
1		Move Numeric.......MVN		Move with Offset...MVO
2		Move Characters....MVC		Pack..............PACK
3		Move Zone..........MVZ		Unpack............UNPK
4		AND................NC		
5		Compare Logical....CLC		
6		OR.................OC		
7		Exclusive OR.......XC		
8				Zero and Add.......ZAP
9				Compare............CP
A				Add................AP
B				Subtract...........SP
C		Translate..........TR		Multiply...........MP
D		Translate and Test.TRT		Divide.............DP
E		Edit...............ED		
F		Edit and Mark......EDMK		

APPENDIX E: ASSEMBLER INSTRUCTIONS

Operation	Name Entry	Operand Entry
ACTR	Must not be present	An arithmetic SETA expression
AGO	A sequence symbol or not present	A sequence symbol
AIF	A sequence symbol or not present	A logical expression enclosed in parentheses, immediately followed by a sequence symbol
ANOP	A sequence symbol	Must not be present
CCW	Any symbol or not present	Four operands, separated by commas
CNOP	A sequence symbol or not present	Two absolute expressions, separated by a comma
COM	A sequence symbol or not present	Must not be present
COPY	Must not be present	A symbol
CSECT	Any symbol or not present	Must not be present
CXD	Any symbol or not present	Must not be present
DC	Any symbol or not present	One or more operands, separated by commas
DROP	A sequence symbol or not present	One to sixteen absolute expressions, separated by commas
DS	Any symbol or not present	One or more operands, separated by commas
DSECT	A variable symbol or an ordinary symbol	Must not be present
DXD	A symbol	One or more operands, separated by commas
EJECT	A sequence symbol or not present	Must not be present
END	A sequence symbol or not present	A relocatable expression or not present
ENTRY	A sequence symbol or not present	One or more relocatable symbols, separated by commas
EQU	A variable symbol or an ordinary symbol	An absolute or relocatable expression
EXTRN	A sequence symbol or not present	One or more relocatable symbols, separated by commas
GBLA	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
GBLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
GBLC	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
ICTL	Must not be present	One to three decimal values, separated by commas

TYPE	IMPLIED LENGTH (BYTES)	ALIGN-MENT	LENGTH MODI-FIER RANGE	SPECIFIED BY	NUMBER OF CON-STANTS PER OPERAND	RANGE FOR EX-PONENTS	RANGE FOR SCALE	TRUN-CATION/ PADDING SIDE
C	as needed	byte	.1 to 256 (1)	characters	one			right
X	as needed	byte	.1 to 256 (1)	hexadecimal digits	one			left
B	as needed	byte	.1 to 256	binary digits	one			left
F	4	word	.1 to 8	decimal digits	multi-ple	-85 to +75	-187 to +346	left
H	2	half word	.1 to 8	decimal digits	multi-ple	-85 to +75	-187 +346	left
E	4	word	.1 to 8	decimal digits	multi-ple	-85 to +75	0-14	right
D	8	double word	.1 to 8	decimal digits	multi-ple	-85 to +75	0-14	right
P	as needed	byte	.1 to 16	decimal digits	multi-ple			left
Z	as needed	byte	.1 to 16	decimal digits	multi-ple			left
A	4	word	.1 to 4 (2)	any expression	multi-ple			left
Q	4	word	2-4	relocatable symbol	one			left
V	4	word	3 or 4	relocatable symbol	multi-ple			left
S	2	half word	2 only	one absolute or relocatab-le expression or two absol-ute express-ions: exp (exp)	multi-ple			
Y	2	half word	.1 to 2 (2)	any expression	multi-ple			left

(1) In a DS assembler instruction C and X type constants may have length specification to 65535.
(2) Bit length specification permitted with absolute expressions only. Relocatable A-type constants, 3 or 4 bytes only; relocatable Y-type constants, 2 bytes only.

APPENDIX G: MACRO LANGUAGE SUMMARY

The four charts in this appendix summarize the macro language described in Part II of this publication.

Chart 1 indicates which macro language elements may be used in the name and operand entries of each statement.

Chart 2 is a summary of the expressions that may be used in macro-instruction statements.

Chart 3 is a summary of the attributes that may be used in each expression.

Chart 4 is a summary of the variable symbols that may be used in each expression.

	Variable Symbols										Attributes						
		Global SET Symbols			Local SET Symbols			System Variable Symbols									
Statement	Symbolic Parameter	SETA	SETB	SETC	SETA	SETB	SETC	&SYSNDX	&SYSECT	&SYSLIST	Type	Length	Scaling	Integer	Count	Number	Sequence Symbol
MACRO																	
Prototype Statement	Name Operand																
GBLA		Operand															
GBLB			Operand														
GBLC				Operand													
LCLA					Operand												
LCLB						Operand											
LCLC							Operand										
Model Statement	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand							Name
COPY																	Name
SETA	Operand[2]	Name Operand	Operand[3]	Operand[9]	Name Operand	Operand[3]	Operand[9]	Operand		Operand[2]		Operand	Operand	Operand	Operand	Operand	
SETB	Operand[6]	Operand[6]	Name Operand	Operand[6]	Operand[6]	Name Operand	Operand[6]	Operand[6]	Operand[4]	Operand[6]	Operand[4]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	
SETC	Operand	Operand[7]	Operand[8]	Name Operand	Operand[7]	Operand[8]	Name Operand	Operand	Operand	Operand	Operand						
AIF	Operand[6]	Operand[6]	Operand	Operand[6]	Operand[6]	Operand	Operand[6]	Operand[6]	Operand[4]	Operand[6]	Operand[4]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Name Operand
AGO																	Name Operand
ACTR	Operand[2]	Operand	Operand[3]	Operand[2]	Operand	Operand[3]	Operand[2]	Operand		Operand[2]		Operand	Operand	Operand	Operand	Operand	
ANOP																	Name
MEXIT																	Name
MNOTE	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand							Name
MEND																	Name
Outer Macro		Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand										Name
Inner Macro	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand							Name
Assembler Language Statement	*	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand										Name

1. Variable symbols in macro-instructions are replaced by their values before processing.
2. Only if value is self-defining term.
3. Converted to arithmetic +1 or +0.
4. Only in character relations.
5. Only in arithmetic relations.
6. Only in arithmetic or character relations.
7. Converted to unsigned number.
8. Converted to character 1 or 0.
9. Only if one to eight decimal digits.

Chart 1. Macro Language Elements

APPENDIX I: ASSEMBLER LANGUAGES--FEATURES COMPARISON CHART

Features not shown below are common to all assemblers. In the chart:
Dash = Not allowed.
X = As defined in Operating System/360 Assembler Language Manual.

Feature	Basic Programming Support/360: Basic Assembler	7090/7094 Support Package Assembler	BPS 8K Tape, BOS 8K Disk Assemblers	BOS 16K Disk/Tape Assembler	OS/360 Assembler
No. of Continuation Cards/Statement (exclusive of macro-instructions)	0	0	1	1	2
Input Character Code	EBCDIC	BCD & EBCDIC	EBCDIC	EBCDIC	EBCDIC
ELEMENTS:					
Maximum Characters per symbol	6	6	8	8	8
Character self-defining terms	1 Char. only	X	X	X	X
Binary self-defining terms	- -	- -	X	X	X
Length attribute reference	- -	- -	X	X	X
Literals	- -	- -	X	X	X
Extended mnemonics	- -	X	X	X	X
Maximum Location Counter value	$2^{16}-1$	$2^{24}-1$	$2^{24}-1$	$2^{24}-1$	$2^{24}-1$
Multiple Control Sections per assembly	- -	- -	X	X	X
EXPRESSIONS:					
Operators	+ -*	+ -*/	+ -*/	+ -*/	+ -*/
Number of terms	3	16	3	8	16
Levels of parentheses	- -	- -	1	3	5
Complex relocatability	- -	- -	X	X	X
ASSEMBLER INSTRUCTIONS:					
DC and DS					
Expressions allowed as modifiers	- -	- -	- -	X	X
Multiple operands	- -	- -	- -	- -	X
Multiple constants in an operand	- -	- -	Except Address Consts.	X	X
Bit length specifications	- -	- -	- -	- -	X
Scale modifier	- -	- -	X	X	X
Exponent Modifier	- -	- -	X	X	X
DC types	Except B, P, Z V, Y, S	Except B, V	X	X	X
DC duplication factor	Except A	X	Except S	X	X

Feature	Basic Programming Support/360: Basic Assembler	7090/7094 Support Package Assembler	BPS 8K Tape, BOS 8K Disk Assemblers	BOS 16K Disk/Tape Assembler	OS/360 Assembler
DC duplication factor of zero	- -	- -	Except S	X	X
DC length modifier	Except H, E, D	X	X	X	X
DS types	Only C, H, F, D	X	X	X	X
DS length modifer	Only C	Only C	X	X	X
DS maximum length modifier	256	256	256	65,535	65,535
DS constant subfield permitted	- -	- -	X	X	X
COPY	- -	- -	- -	X	X
CSECT	- -	- -	X	X	X
DSECT	--	X	X	X	X
ISEQ	- -	- -	X	X	X
LTORG	- -	- -	X	X	X
PRINT	- -	- -	X	X	X
TITLE	- -	X	X	X	X
COM	- -	- -	- -	X	X
ICTL	1 operand (1 or 25 only)	1 operand	X	X	X
USING	2 operands (operand 1 relocatable only)	2-17 operands (operand 1 relocatable only)	6 operands	X	X
DROP	1 operand only	X	5 operands	X	X
CCW	operand 2 (relocatable only)	X	X	X	X
ORG	no blank operand	no blank operand	X	X	X
ENTRY	1 operand only	1 operand only	1 operand only	X	X
EXTRN	1 operand only (max 14)	1 operand only	1 operand only	X	X
CNOP	2 decimal digits	2 decimal digits	2 decimal digits	X	X
PUNCH	- -	- -	- -	X	X
REPRO	- -	- -	X	X	X
Macro Instructions	- -	- -	X	X	X

&SYS, restrictions on use 65,77,90
&SYSECT (see current control section name)
&SYSLIST (see macro-instruction operand)
&SYSNDX (see macro-instruction index)
7090/7094 Support Package Assembler 7,135

Absolute terms 15
ACTR instruction
  Format of 85
  Inside macro-definitions 85
  Outside macro-definitions 85
  Use of 85
Address constants 47-48
  A-type 47
  Complex relocatable expressions 47
  Literals not allowed 19
  S-type 48
  V-type 48
  Y-type 48
Address specification 33
Addressing
  Dummy sections 28
  Explicit 23
  External control sections 30
  Implied 23
  Relative 25
AGO instruction
  Example of 85
  Format of 85
  Inside macro-definitions 85
  Operand field of 85
  Outside macro-definitions 85
  Sequence symbol in 85
  Use of 85
AIF instruction
  Example of 84
  Format of 84
  Inside macro-definitions 84
  Invalid operand fields of 84
  Logical expression in 83
  Operand field of 83
  Outside macro-definitions 84
  Sequence symbols in 85
  Use of 84
  Valid operand fields of 84
Alignment, boundary
  CNOP instruction for 56
  Machine instruction 32
Ampersands in
  Character expressions 80
  Macro-instruction operands 68
  MNOTE instruction 89
  Symbolic parameters 65
  Variable symbols 62
ANOP instruction
  Example of 86
  Format of 86
  Sequence symbol in 86
  Use of 86
Arithmetic expressions
  Arithmetic relations 82
  Evaluation procedure 77
  Invalid examples of 77
  Operand sublists 78
  Operators allowed 77
  Parenthesized terms in
    evaluation of 78
    examples of 78
  SETA instruction 77
  SETB instruction 82
  Substring notation 80
  Terms allowed 77
  Valid examples of 77
Arithmetic relations 82
Arithmetic variable 93
Assembler instructions
  Statement 37
  Table 124
Assembler language
  Basic Programming Support 9,135
  Comparison chart 135
  Macro language, relation to 61
  Statement format 13,14
  Structure 15,16
Assembler program
  Basic functions 10
  Output 26
Assembly, terminating an 57
Assembly no operation (see ANOP instruction)
Attributes
  How referred to 74
  Inner macro-instruction operands 73
  Kinds of 73
  Notations 73
  Operand sublists 73
  Outer macro-instruction operands 73
  Summary chart of 130
  Symbols 73
  Use of 73
  (see also specific attributes)

Basic Programming Support Assembler 7,135
Base registers
  Address calculation 10,30,33
  DROP instructions 24
  Loading of 23
  USING instructions 23
Binary constant 44
Binary self-defining term 18
Binary variable 93
Blanks
  Logical expressions 82
  Macro-instruction operands 69

CCW instruction 50
Channel command word, defining 51
Character codes 102
Character constant 43
Character expressions
  Ampersands in 80
  Character relations 82
  Examples of 79,80
  Periods and 79
  Quotation marks in 79
  SETB instructions 82
  SETC instructions 79
Character relations 82
Character self-defining term 18
Character set 15,102

Character variable 93
CNOP instruction 56
Coding form 12
COM instruction 29
Commas, macro-instruction operands 69
Comments statements
  Example of 14,67
  Model statements 66
  Not generated 67
Comparison chart 135
Compatibility
  Assembler language 9
  Macro-definitions 98
Complex relocatable expressions 47
Concatenation
  Character expressions 79,81
  Defined 65
  Examples of 66
  Substring notations 81
Conditional assembly elements, summary charts of 87,129
Conditional assembly instructions
  How to write 72
  Summary of 72
  Use of 72
  (see also specific instructions)
Conditional branch (see AIF instruction)
Conditional branch instruction 35
  Operand format 35
Constants (see also specific types)
  Defining (see DC instructions)
  Summary of 127
Continuation lines 13
Control dictionary 26
Control section location assignment 26
Control sections
  Blank common 29
  CSECT instruction 27
  Defined 26
  First control section, properties of 26
  START instruction 27
  Unnamed 27
COPY instruction 57
COPY statements in macro-definitions
  Format of 67
  Model statements, contrasted 67
  Operand field of 67
  Use of 67
Count attribute
  Defined 75
  Notation 73
  Operand sublists 75
  Use of 75
  Variable symbols 75
CSECT instruction, symbol in, length attribute of 27
Current control section name (&SYSECT)
  Affected by CSECT, DSECT, START 94
  Example of 95
  Use of 95

Data definition instructions 38
  Channel command words 50
  Constants 38
  Storage 49
DC instruction 38
  Constant operand subfield 42
    Address-constant (see Address constants)
    Binary constant 44
    Character constant 43
    Decimal-constant 46
    Fixed-point constant 44
    Floating-point constant 45
    Hexadecimal constant 43
    Type codes for 40
  Exponent modifier 42
  Duplication factor operand subfield 39
  Length modifier 39
    Bit length specification 40
  Operand subfield modifiers 39
  Scale modifier 41
  Type operand subfield 39
Decimal constants 46-47
  Length, maximum 46
  Length modifier 46
  Packed 47
  Zoned 47
Decimal field, integer attribute of 76
Decimal self-defining terms 18
Defining constants (see DC instruction)
Defining storage (see DC instruction, DS instruction)
Defining symbols 17
Dimension, subscripted SET symbols 92
Displacements 33
Double-shift instruction 32
DROP instruction 24,32
DS instruction 49-50
  Defining areas 49
  Forcing alignment 49
DSECT instruction 28
Dummy section location assignment 28,30
Duplication factor 39
  Forcing alignment 49

Effective address, length 34
EJECT instruction 52
END instruction 58
ENTRY instruction 30
Entry point symbol, identification of 30
EQU instruction 37
Equal signs, as macro-instruction operands 68
Error message (see MNOTE instruction)
Explicit addressing 23,33
  Length 34
Exponent modifiers 42
Expressions 20,30
  Absolute 33
  Evaluation 21
  Relocatable 33
  Summary chart of 129
Extended mnemonic codes 35
  Operand format 36
External control section, addressing of 30
External symbol, identification of 30
EXTRN instruction 30

First control section 26
Fixed-point constants 44-45
  Format 44
  Positioning of 45

MNOTE instruction
   Ampersands in 89
   Error message 89
   Example of 89
   Operand field of 88
   Quotation marks in 89
   Severity code 88
   Use of 89
Model statements
   Comments field of 64
   Comments statements 66
   Defined 64
   Name field of 64
   Operand field of 64
   Operation field of 64
   Use of 64

N' (see Number attribute)
Name entries 13
Number attribute
   Defined 75
   Example of 75
   Notation 75
   Operand sublist 75

Operand sublist
   Alternate statement format 69
   Defined 69
   Example of 70
   Use of 69
Operands
   Entries 13
   Fields 32
   Subfields 32,33
   Symbolic 30,32,34
Operating system 11
Operation field 32
ORG instruction 55
Outer macro-instruction defined 70

Paired parentheses 68
Paired quotation 68
Parentheses in
   Arithmetic expressions 78
   Logical expressions 83
   Macro-instruction operands 68
   Operand fields and subfields 33
   Paired 68
Period in
   Character expressions 79
   Comments statements 67
   Concatenation 66
   Sequence symbols 76
Positional macro-definition (see macro-definition)
Positional macro-instruction (see macro-definition and macro-instruction)
Previously defined symbols 17
PRINT instruction 52
Program control instructions 52
Program listings 11
Program sectioning and linking 26
Prototype statement
   Example of 64
   Format of 63
   Keyword (see keyword prototype statement)
   Mixed-mode (see mixed-mode prototype statement)
   Name field of 63
   Operand field of 63
   Operation field of 63
   Statement format 64
   Symbolic parameters in 63
   Use of 63
PUNCH instruction 54

Quotation marks in
   Character expressions 79
   Macro-instruction operands 68
   MNOTE instruction 89
Quoted string 68

Relocatability 15,10
   Attributes 22,30
   Program, general register zero 24
Relocatable expressions 22,32
   In USING instructions 24
Relocatable terms 15
   Pairing of 21
   In relocatable expressions 22
Relative addressing 25
REPRO instruction 55
RR machine-instruction format 32
   Length attribute 32
   Symbolic operands 34
RS machine-instruction format 32
   Address specification 33
   Length attribute 32
   Symbolic operands 34
RX machine-instruction format 32
   Address specification 33
   Length attribute 32
   Symbolic operands 34

S' (see scaling attribute)
Sample program 132
Scale modifier
   Fixed-point constants 45
   Floating-point constants 45
Scaling attribute
   Decimal fields 75
   Defined 74
   Examples of 75,76
   Fixed-point fields 74
   Floating-point fields 75
   Notation 73
   Restrictions on use 75
   Symbols 74
   Use of 75
Self-defining terms 17
   (see also specific terms)
Sequence checking 54
Sequence symbols
   AGO instruction 84
   AIF instruction 84
   ANOP instruction 85
   How to write 77
   Invalid examples of 77
   Macro instruction 77
   Use of 77
   Valid examples of 77
Set symbols
   Assigning values to 72
   Defining 72
   Symbolic parameters, constrasted 72

Use 72
(see also local SET symbols, global SET symbols, and subscripted SET symbols)
SET variable 92
SETA instruction
Examples of 78,79
Format of 77
Operand field of 77
Evaluation procedure 77
Operators allowed 77
Parenthesized terms 78
Terms allowed 77
Valid examples of 77
Operand sublist 78
Example 79
SETA symbol
AIF instruction 78
Arithmetic relations 82
Assigning values to 72
Defining 72
SETA instruction 78
SETB instruction 78
SETC instruction 82
Using 78
SETB instruction
Example of 83
Format of 82
Logical expression in 82
Arithmetic relations 82
Blanks in 82
Character relations 82
Evaluation of 83
Operators allowed 82
Terms allowed 82
Operand field of 82
Invalid examples of 82
Valid examples of 82
SETB symbol
AIF instruction 83
Assigning values to 72
Defining 72
SETA instruction 83
SETB instruction 83
SETC instruction 83
Using 83
SETC instruction
Character expressions in 79
Ampersands 80
Periods 79
Quotation marks 79
Concatenation in
Character expressions 79,80
Substring notations 81
Examples of 79-82
Format of 79
Operand field of 79
Substring notations in 80
Arithmetic expressions in 80
Character expressions in 80
Invalid examples of 80
Valid examples of 80
Type attribute in 79
Example of 79
SETC symbol
Assigning values to 72
Defining 72
SETA instruction 82
Using 80
Severity code in MNOTE instruction 88
SI machine-instruction format 32
Address specification 33
Length attribute 32
Symbolic operands 34
SPACE instruction 52
SS machine-instruction format 32
Address specification 33
Length attribute 32
Length field 33
Symbolic operands 34
Standard value
Attributes of 97
Keyword prototype statement 96
START instruction
Positioning of 27
Unnamed control sections 28
Statements 13,14
Boundaries 13
Examples 14
Macro-instruction 69
Prototype 64
Summary of 128
Storage, defining (see DS instruction)
Sublist (see Operand sublist)
Subscripted SET symbols
Defining 92
Examples 93
Dimension of 92
How to write 92
Invalid examples of 92
Subscript of 92
Using 93
Examples 93
Valid examples of 93
Substring notation
Arithmetic expressions in 80
Character expression in 80
How to write 80
Invalid example of 81
SETB instruction 82
SETC instruction 81
Valid examples of 81
Symbol definition, EQU instruction for 17
Symbolic linkages 29
Symbolic operand formats 34
Symbolic parameter
Comments field 65
Concatenation of 65
Defined 64
How to write 65
Invalid examples of 65
Model statements 64
Prototype statement 63
Replaced by 65
Valid example of 65
Symbols
Defining 15
Length attributes 32
Referring to 20
Length, maximum 15
Previously defined 17
Restrictions 17
Value attributes 32
System macro-instructions defined 61
System variable symbols
Assigned values by assembler 93
Defined 93