# 3300
# 3500

## COMPUTER SYSTEMS
## META/MASTER
### GENERAL INFORMATION MANUAL

**Note re scan: Several pages were missing from source material**

[illegible]

# REVISION RECORD

60236000

| REVISION | NOTES |
|---|---|
| 6-28-68 | Original printing |

Additional copies of this manual may be obtained from the nearest Control Data Corporation sales office.

Pub. No. 60236000

©1968 Control Data Corporation
Printed in the United States of America

Address comments concerning this manual to:

Control Data Corporation
Software Documentation
4201 North Lexington Avenue
St. Paul, Minnesota 55112

or use Comment Sheet in the back of this manual.

# PREFACE

Readers unfamiliar with assemblers in general and meta-assemblers in particular may find a small amount of background information helpful. During the relatively short history of computers, computer languages have evolved from machine languages directly interpreted by the computer to symbolic languages more meaningful to the programmer but requiring conversion to the equivalent machine language before execution of a program. The processor that converts the symbolic language is one of two types, an assembler or a compiler. Of the two languages, the assembler language most closely resembles the machine language. It is in this category that we find the 3300/3500 Meta-Assembler (META) although the Meta-Assembler offers features usually found only in more sophisticated compilers.

Early assemblers generated one instruction of machine language code from each symbolic source language instruction. Relocation brought a new dimension to symbolic input and, as assembler output format began to diverge to meet the needs of the relocatable loaders, the role of the assembler increased. At first, however, the ratio of input statements to output instructions was still one-to-one. Then someone questioned this ratio. Why not change the assembler so that a set of code could be called by a single statement? The answer came in the form of pseudo instructions and macro instructions. Macros offer considerable parameterization of code blocks. These instructions are directed more at the assembler than at the machine for which the output is generated. They opened the door for further communications between the programmer and the assembler. But, for a time, conventional assemblers made little use of this promising capability. Assembly essentially remained a farily straightforward conversion of source statements to one or more lines of machine language code. In some assemblers, the programmer could direct the processor to conditionally skip statements or change the format and content of the output listing — but not of the code being generated.

Today, the meta-assembler frees the assembler from its dependence on a machine-oriented input language yet still offers all the features of the conventional assembler.

# CONTENTS

# INTRODUCTION 1

The 3300/3500 Meta-Assembler/MASTER provides users with a versatile, extensive language for directing the generation of object code. The assembler executes on a CONTROL DATA® 3300 Computer System or CONTROL DATA® 3500 Computer System under the supervision of the 3300/3500 MASTER Multiprogramming Executive Operating System.

## FEATURES

Meta-Assembler (META) allows the programmer to select a 3300/3500 relocatable binary output format acceptable for loading and execution under MASTER or as a byte stream. When he chooses to generate a byte stream he is not restricted to a 24-bit object word and can use the Meta-Assembler to generate output for execution on some other computer. Thus, a meta-assembler language is the ideal language in which to code compilers and assemblers or to produce code for an alternate computer system, either real or simulated.

Source statements, called directives, are oriented toward control of the assembler itself and control the meta-assembler much the same as machine language instructions control the computer.

The source program, which consists primarily of directives, can also include definitions of and references to procedures and functions. The definitions are groups of source statements that the assembler interprets each time the procedure or function is referenced. A reference to a procedure definition, because it appears in the command field of a statement, can be likened to a macro call; a reference to a function, because it can be an element in an expression, can be likened to a FORTRAN function reference. There are significant differences, however, and it is better to forget about macros when working with META.

Procedures and functions provide extensive parameterization of source statements. For example, 3300/3500 Meta-Assembler/MASTER includes standard procedures for the 3300/3500 mnemonic language instructions. A source statement, consisting of a mnemonic instruction and parameters, calls one of these procedures. The assembler interprets the procedure which generates the equivalent 3300/3500 relocatable binary object code. Often used or standard procedures definitions can be placed in the library.

META is a self-extending translator. Definition of procedures provides a convenient means for the programmer to expand the META source language or even define a new language (within tho syntax of the Meta-Assembler).

For META, the ratio of lines of input to lines of output code is irrelevant. If conditions warrant, a reference to a procedure may produce no code at all. META generates code only when directed to by the programmer.

META allows the user to define and assign symbols to addresses, to single values, or to sets (lists) of data. An entire set may be referred to by a symbol; each element of a set may be referred to by adding one or more subscripts to the symbol.

META recognizes as operands simple and complex expressions containing any of a set of eighteen operators. Elements of expressions can be symbols, or constants expressed as octal or decimal integers, real (floating-point) values or as BCD or ASCII characters according to convenience.

A unique method of symbol definition allows the value of an expression to be used as a symbol. An operand of a source statement can also be an attribute of an expression, such as its size or type (character, octal, decimal, etc.).

The Meta-Assembler language allows simple, brief notation as well as complex expressions involving nested procedures, functions, and sets.

## OPERATING SYSTEM

META executes under control of the 3300/3500 MASTER Multiprogramming Executive Operating System.

## CONFIGURATION

The requirements for executing META on the 3300 or 3500 are the minimum required for the MASTER multiprogramming system.

## EXECUTION

META is called from the MASTER system library by a META task name card. Parameters on the card define files used during the assembler run, such as the file containing source statements and the files to receive the listable output, the load-and-go output, and the punchable output. The programmer may optionally request that META use files other than the standard job files (INP, OUT, PUN) and the load-and-go file

(LGO) but when he does, the programmer is responsible for allocation of the files and for printing or punching any output. All standard job files are released at the end of the job or after processing by the MASTER postprocessor.

The MASTER executive allocates channels as needed and performs all input/output required during the assembly.

**STANDARD INPUT**

The Meta-Assembler source deck can be on the standard input card reader or on some file, such as a magnetic tape file, specified by the programmer. If it is on the card reader, the MASTER input preprocessor transfers the deck from the standard input card reader onto a mass storage file (INP). The programmer has the option of bypassing this transfer by placing a DIRECT card in front of his deck.

META interprets the source deck statement-by-statement from the file specified.

**PRINTER OUTPUT**

META produces printer output containing a listing of each source statement. List control directives provide the programmer with the option of obtaining a detailed listing as well. Errors detected by the assembler are noted on the listing. This printer output for the Meta-Assembler run is normally accumulated on a mass storage file and automatically printed by a MASTER postprocessor when the run is finished. The programmer may request a simultaneous print through a parameter of a MASTER DIRECT card or may request that the output be placed on some other file (for which printing is not automatic) through a parameter of the META card.

**PUNCH OUTPUT**

Similarly, MASTER accumulates data on a punch file for automatic post-job punching. The programmer may request direct punching or may direct punch output to some other file for which punching is not automatic.

**EXECUTABLE OUTPUT**

Upon programmer request, META allocates the LGO file to receive relocatable binary output acceptable to the 3300/3500 MASTER relocatable

loader. When the assembler has completely processed the source deck, the programmer may call for loading and execution of the object program from the load-and-go file. The MASTER loader links the newly assembled program to any previously assembled programs referred to by the new program. The programmer can designate whether or not the load-and-go file should be executed in spite of errors detected during assembly.

If he desires, the programmer can request binary output in the form of a byte stream. This form of output is not acceptable to the MASTER relocatable loader and is usually intended for further conversion for loading on some other computer.

## MACHINE LANGUAGE INSTRUCTIONS

META includes a set of procedures for interpreting mnemonics for the 3300/3500 machine language instructions and generating equivalent code. While these mnemonics resemble the 3300/3500 COMPASS repertoire, differences in syntax and in notation used for operand fields and modifiers cause incompatibilities between the two languages. In addition, META does not recognize COMPASS macros, pseudo instructions, or numeric operation codes. To cite a difference, the representation of an octal number in the 3300 COMPASS language is a string of octal digits followed by the letter B. The representation of an octal number in the META language is the letter O followed by a string of octal digits enclosed in apostrophes.

# CODING CONVENTIONS 2

## SOURCE STATEMENTS

A source program for the Meta-Assembler is a sequence of statements punched onto 80-column cards, each statement requiring one or more cards. Statements can be written as lines of code on a coding form. A statement begins at character position 1 and may continue through character position 71 of a line.

## CONTINUATION

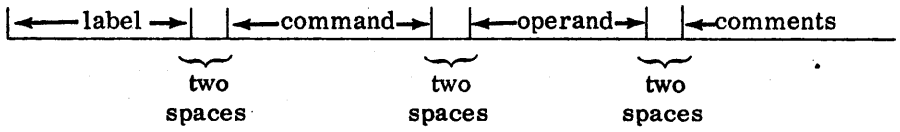
By inserting a semicolon in character position 72 of a line, a programmer notifies the assembler that the next line is a continuation of the statement. Information continues in column 2 of the next line.

## SEQUENCING

META does not examine information in character positions 73-80. Thus, these card columns can be used for card sequencing.

## GENERAL FORMAT

The general form of a statement is:




Each field terminates with two or more spaces.
A statement label (usually optional) begins in the first or second character position of a line and consists of a symbol.

The first nonblank character following the two spaces delimiting the label field begins the command field. This field, which is mandatory, contains a Meta-Assembler directive, a mnemonic language instruction, or a reference to a defined format or procedure.

The first nonblank character following the two spaces delimiting the command field begins either the operand field, which is usually required, or the comment field, which is optional. The contents of the operand field depend on the operand requirements of the command.

An operand field contains one or more expressions, each consisting of one or more symbols and elementary items joined by operators. For directives, the operand field provides information required by the assembler to perform the designated operation. Operands of mnemonic instructions and procedures generally represent addresses, constant values, and evaluatable expressions.

In addition to using the comment field, the user can indicate that all successive characters of a line are comments by beginning a field with an asterisk.

Examples:

The following line contains all four fields.

```
ALPHA   LDA   LØC   LØAD A REGISTER
```

label command operand comments

The following line has a blank label field and does not contain comments.

```
        LDA   LØC
```

command operand

The following line is continued.

```
BETA    NSET  L'DØG','EASY','FØXTRØT',L'KIL
L'Ø','LIMA','MIKE','NANCY','ØSCAR'
```

column 72

The following line contains a command and a comment.

```
        END   *CØMMENTS
```

The following line is a comment line.

```
* THIS IS A COMMENT }
```

## SYMBOLS

Symbols play a vital role in an assembly language. The power, versatility, and flexibility of an assembler relate directly to its symbol handling capability.

A symbol is 1-12 alphabetic characters or numbers. The first character must be alphabetic (A-Z); the symbol represents an address, an arbitrary value, or a list of values (a set). Symbols provide a programmer with a convenient means of referring to these program elements.

Examples:

JOE
A3
B7A5

Each symbol used as a label for a source statement is assigned a value or a set of values by META. Assignment depends on whether the symbol is to be assigned a value representing a relocatable address (one that may change when the program is relocated) or is to be assigned a value or set of values indicated by the operand field of the statement.

## SYMBOL DEFINITION

Symbol definition means that META enters the symbol in a table where it maintains the value or set of values assigned to the symbol together with known attributes of the symbol.

## ATTRIBUTES

META permits the programmer to inquire about characteristics of the value, such as its size in words or bytes and its mode of representation (decimal, octal, character, etc.). These attributes can be referred to through a type of symbolic reference known as an attribute function. META arbitrarily assigns a value to each attribute of a symbol. For example, a symbol defining a set element that is six ASCII characters

has a mode attribute of 7 and a size attribute (expressed in characters) of 6. Attributes are discussed again in chapter 4.

**FORWARD REFERENCE**

Generally, META permits a reference to a symbol before it is defined (a forward reference) if the term referred to does not effect location counting. A forward reference to a value subsequently redefined normally yields the last value assigned.

**CURRENT ADDRESS SYMBOL**

The special symbol $ as an address operand represents the current value of the location counter in use for the control section.

**SYMBOL LEVELS**

META recognizes 16 levels of symbol definition (0-15). Symbols defined at a given level are available at the given level and all higher (or inner) levels and cannot be referenced at lower levels.

Symbols made external to the program are defined at level zero.

Symbols defined in the program but outside of procedures or functions are at level one. Symbols defined within procedures or functions (nested) are at level two or higher. For each nesting of the definition, one level is added to the symbol definition.

If a programmer chooses, he can denote by means of a dollar sign ($) immediately following the symbol in the label field that the symbol is to be defined at the next lower level. He can use multiple dollar signs to lower a symbol level by more than one, to a minimum of one level.

**ELEMENTARY ITEMS**

An item in an expression that has a value itself rather than representing a value is a self-defining elementary item. META does not assign a value to an elementary item; it is able to interpret the character string comprising an elementary item without further information. For example, the decimal integer 23 has the value 23. An elementary item has a maximum precision of 48 bits.

META recognizes three types of integer notation, three types of character notation, and a real (floating-point) notation for elementary items.

An elementary item can be used alone or can be combined with symbols, operators, and other elementary items to form an expression.

## INTEGER NOTATION

The three types of notation for integers are decimal, octal, and BCD decimal.

| | |
|---|---|
| Decimal | A string of decimal digits.<br><br>Example:<br><br>429 |
| Octal | The letter O followed by a string of octal digits enclosed in apostrophes.<br><br>Example:<br><br>Ø'2777' |
| BCD Decimal | The letter D followed by a string of not more than eight decimal digits enclosed in apostrophes. This notation provides for 6-bit BCD arithmetic. For a negative value, a minus sign is placed with the rightmost character.<br><br>Examples:<br><br>D'4927'<br>-D'123' |

## CHARACTER NOTATION

The two types of BCD character notation indicate whether the BCD character string is to be stored as 6-bit characters right adjusted with the remainder of the field filled with zeros or whether the character string is stored left adjusted and the field filled with spaces (blanks).

A third type of character notation stores the character string in 8-bit ASCII rather than BCD code. ASCII characters are stored right adjusted. The remainder of the field is zero filled.

Because single apostrophes are delimiters, two apostrophes represent a single apostrophe within the character string. The legal character set

for character strings is given in appendix A. Note that a space is a legal character.

| | |
|---|---|
| BCD<br>Right Adjusted | The letter C followed by a string of not more than eight BCD characters enclosed in apostrophes, or simply a string of BCD characters enclosed in apostrophes. |

Examples:

| | |
|---|---|
| C'AB$5' | (4 characters: AB$5) |
| 'A''^B''' | (5 characters: A'^B') |

| | |
|---|---|
| BCD<br>Left Adjusted | The letter L followed by a string of not more than eight BCD characters enclosed in apostrophes. |

Example:

L'ABC'

| | |
|---|---|
| ASCII | The letter A followed by a string of not more than six ASCII characters enclosed in apostrophes. |

Example:

A'AB$5'

## REAL NOTATION

A real or floating-point number is defined by the appearance of a decimal point somewhere in the value which optionally consists of an integer, a fraction, or an E followed by an exponent (scale factor). It can be in the range $10^{-308}$ to $10^{308}$.

Examples:

| | |
|---|---|
| .35 | (.35) |
| 1.E+2 | (100) |
| 1.14159E-3 | (.00114159) |
| 4.79 | (4.79) |

The value is converted to 3300/3500 48-bit internal floating-point format.

## EXPRESSIONS

An elementary item or a symbol can be used alone to form a simple expression or can be combined with operators to form complex expressions. Operations include addition, subtraction, multiplication, division, binary and decimal scaling, and relational and masking operations. Appendix B summarizes the 18 operators recognized by META. Note that for some operations META recognizes both mnemonic and symbolic operators.

Generally, a single space has no significance in a field and serves only to enhance the readability of an expression. However, when the operator in an expression is mnemonic rather than a symbol, for example, EQ rather than =, META requires spaces as separators.

Expressions can include references to set elements and functions.

The programmer can form subexpressions by using parentheses in the normal role of arithmetic grouping.

The assembler evaluates expressions from left to right performing the operations with lower hierarchies first. It evaluates parenthetical expressions first, expanding them from the inside out.

Examples of expressions:

| | |
|---|---|
| A1 + 10 | Indicates an address 10 words greater than address A1. |
| A*B | Product of the values of A and B. |
| (A<B)++(C<D) | If both inequations are false, zero; otherwise one. |

Associated with each item in an expression is a mode that defines how the value is to be interpreted for an arithmetic operation. Thus, META discerns whether a value is an octal integer, a real or floating-point value, a character string (BCD or ASCII), or a relocatable address. The mode determines whether the assembler will use integer, real, or decimal arithmetic when evaluating the expression. Normally, if all the values are not of the same mode, the assembler uses a value of zero for the expression and flags the error. The permitted mixing of octal integers and real or floating-point values represents an exception to this rule.

## SETS

A set is a list of one or more elements separated by commas. Each list element is an expression, a set name, or a subset (a set enclosed in brackets). A null expression is interpreted as a zero.

Examples:

| | |
|---|---|
| 1.24, 4 | A set of two elements. |
| X+5, [1.25,4] | A set of two elements, the first of which is an expression, the second of which is a set of two elements. |
| X+5, B | A set of two elements, the first of which is an expression, the second of which names a set. |
| 3, 2, 1, | A set of four elements, the fourth of which is zero. |

META includes directives that assign symbolic names to sets so that elements of sets or entire sets can be referred to symbolically. To refer to a defined set, the programmer simply writes the set name. If he wishes to refer to an element, he follows the name with a pair of brackets enclosing one or more expressions separated by commas. The subscript expressions represent the element's ordinal location in the set. From left to right, they represent the level of the element in a set containing subsets.

Example:

The symbol A is defined as the set 5, C, [9, [3, 4]] . The set has three elements. The third element [9, [3, 4]] contains two elements, the second of which also contains two elements [3, 4].

| Reference | Element | Value |
|---|---|---|
| A | All | 5, C, [9, [3, 4]] |
| A[1] | First element of A | 5 |
| A[2] | Second element of A | C |
| A[3] | Third element of A | 9, [3, 4] |
| A[3, 1] | First element of subset of third element of A | 9 |
| A[3, 2] | Second element of subset of third element of A | 3, 4 |
| A[3, 2, 1] | First element of subset of second element of subset of third element of A | 3 |

In the preceding example, if C is a set name for a set consisting of the list elements 7, 8, 6, elements of C could be referred to as follows:

| Reference | Element | Value |
|---|---|---|
| A[2,1]or C [1] | First element of C | 7 |
| A[2,2] or C[2] | Second element of C | 8 |
| A[2,3] or C[3] | Third element of C | 6 |

The Meta-Assembler maintains information about a set and its elements together with the symbol defining the set. The programmer can access this information for use by the assembler through attribute function references. For example, the NUM attribute function supplies the number of elements in the set.

## LITERALS

A literal is an expression preceded by an equal sign. The assembler assigns the value of the expression to a location in a literal table following a control section as determined by a LIT directive.

Examples:

```
=Ø'70707070'
=A + B - $
```

# DIRECTIVES 3

A programmer using the Meta-Assembler directs the assembly of object code by using a set of nearly forty commands called directives. A directive controls the operation of the Meta-Assembler much the same as a machine language instruction directs the computer.

Through directives, a programmer can:

- Define the word size for object code when assembling code for a machine other than the 3300 or 3500.
- Define a symbol and assign a value or set of values to it for subsequent reference by the symbol.
- Specify that a symbol referred to by the program being assembled is defined external to it, perhaps by a program previously assembled or, conversely, that a symbol defined in the program being assembled can be referred to by some other program.
- Conditionally repeat or skip source statements.
- Assign up to 15 relocatable location control counters and one absolute control counter for the Meta-Assembler to use for address assignment.
- Generate code to be loaded and executed on the object computer including the ability to subdivide each word to be generated into fields and assign values to the fields.
- Identify a group of statements as being a procedure and assign it one or more names so that he can subsequently call the procedure by a name and pass it parameters.
- Identify a group of statements as being a function, assign it one or more names, and use a name as a value in an expression such that the value varies according to parameters of the function reference.
- Control the format and content of the listing META produces during assembly.
- Terminate assembly of a subprogram or group of subprograms.

## MACHINE DEFINITION (UNIT)

At the beginning of his source statements, the programmer can issue a directive that causes the assembler to generate words and bytes of a correct size for some computer other than the 3300 or 3500.

Format:

| label (optional) | UNIT | byte size, | word size | comments (optional) |
|---|---|---|---|---|

Byte size defines the number of bits per byte; word size defines the number of bytes per word.

Example:

```
UNIT  6,6          $
```

The object computer word is a 36-bit word comprised of six 6-bit bytes.

## SYMBOL DEFINITION

The directives that assign or reassign values to symbols and sets are EQU, RDEF, NSET, PROC, and FUNC. (PROC and FUNC are discussed later.) In most other directives a symbol in the label field is assigned an address value depending on the counter used for maintaining addresses for the program section.

### EQU

EQU directs META to assign the value and attributes of an expression in the operand field to a symbol in the label field.

Format:

| symbol | EQU | expression | comments (optional) |
|---|---|---|---|

Example:

```
ABLE  EQU  BAKER + 6      $
```

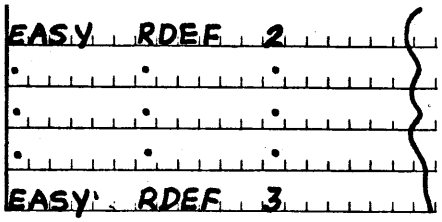
**RDEF**

RDEF directs META to assign the value and attributes of an expression in the operand field to a symbol or set element referred to in the label field. The directive can be the first definition of a symbol or it can be a redefinition of a symbol previously defined by an RDEF or an NSET.

Format:

| symbol | RDEF | expression | comments (optional) |
|---|---|---|---|

Example:




First definition; EASY = 2

Redefinition; EASY = 3

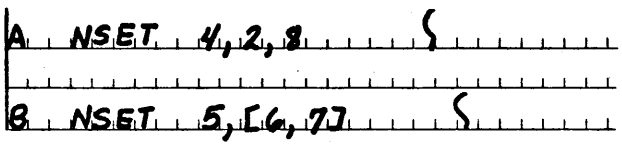
**NSET**

NSET directs META to assign the symbol in the label field as the set name of the list of expressions in the operand field.

Format:

| symbol | NSET | set | comments (optional) |
|---|---|---|---|

Examples:



## PROGRAM LINKAGE (EXT/ENTRY)

Two directives facilitate linkage between independently assembled programs. EXT notifies the assembler of symbols defined by some program external to the source program and referred to by the source program; ENTRY notifies the assembler of symbols that are defined by the source program and can be referred to by some program external to it.

Format:

| label | | | |
|---|---|---|---|
| (optional) | EXT<br>ENTRY | $symbol_1, \ldots, symbol_n$ | comments<br>(optional) |

Examples:

```
EXT   EXPF, LØGF, SIGNF, ABSF, FIXF
ENTRY START, TABLE1, TABLE2
```

## REPEAT AND SKIP

Two directives conditionally repeat (RPT) or skip (GOTO) source statements.

### RPT

A repeat directive (RPT) conditionally repeats source statements a specified number of times. A zero or negative repetition count causes the source statements in the repeat range to be skipped. The repeat range begins with the statement immediately following RPT and ends with a statement specified by an operand of the RPT directive. If no end statement is specified, only the line following the RPT directive is repeated. The Meta-Assembler permits one RPT to lie within the range of another. RPT directives can be nested to a level of six.

Format:

| count symbol | | | |
|---|---|---|---|
| (optional) | RPT | expression, line id | comments<br>(optional) |

The symbol in the label field is optional. If present, it is assigned the value of the repetition count. Its value is incremented with each repetition of the source statements.

Examples:

```
XYZ   RPT   10,JOE                    (
```

Statements through statement labeled JOE are repeated 10 times. XYS has values 1, 2,...,10.

```
C   RPT   A>B,D                       (
```

When A is less than or equal to B, the above repeat acts like a skip.

**GOTO**

A skip directive (GOTO) conditionally skips source statements. The GOTO directive uses the value of an expression in the operand field as an index to a list of line identifiers. For example, if the value of the expression is 3, the third line id identifies the next statement to be interpreted by the assembler. If the value of the expression is negative, zero, or greater than the number of entries in the list, assembly continues at the next statement.

Format:

label (optional) GOTO expression, line $id_1$, line $id_2$,...,line $id_3$ comments (optional)

Example:

```
    GOTO   A-B,ALPHA,BETA,GAMMA        (
```

**LNID**

The LNID directive allows a programmer to define a dummy label for line identification purposes. The label has no value and no entry in the symbol table. It is especially useful for defining the range of an RPT since

the use of normal labels in such instances could result in duplicate symbol definitions.

Format:

| label | LNID | comments (optional) |
|---|---|---|

Example:

```
GAMMA  LNID                (
```

## LOCATION CONTROL

META provides for one absolute and up to 15 relocatable location counters. A program can be assembled into one or more control sections, each with its own counter. A symbol defined in a control section is not unique to it and can be referred to by any control section in the program. Location counters are maintained as byte counts.

The programmer may refer to the current value of the location counter in use by using a dollar sign ($) as an operand. It achieves the same result as if a symbol were placed in the label field and used as an operand of the same statement.

Seven directives assign names and destinations to control sections and values to location counters.

| | |
|---|---|
| SECD | LIT |
| SECP | RES |
| SECA | RESB |
| ORG | |

### SECD

The SECD directive specifies the control section name, chapter number, and maximum length for a labeled common block or a numbered common block.

Format:

| label | SECD | symbol, chapter, size | comments |
|---|---|---|---|
| (optional) | | (all optional) | (optional) |

Symbol defines the common block as labeled, numbered, or blank (zero). The chapter operand indicates whether the common block is to be located in the first or second chapter of a MASTER program task. The size operand provides an estimate of the size of the common block.

Examples:

```
SECD  L,2,200
```

L, a 200-word labeled common block, is assigned to chapter two. A subsequent SECD directive referring to L would not require the chapter and size operands.

```
SECD  25,,100
```

Numbered common block 25 occupies 100 words in chapter one.

**SECP**

The SECP directive specifies a control section name for a relocatable subprogram.

Format:

| label | SECP | symbol | comments |
|---|---|---|---|
| (optional) | | | (optional) |

Example:

```
SECP  PROGA
```

Subprogram is named PROGA.

**SECA**

The SECA directive specifies a control section name for an absolute subprogram. Location of this subprogram is controlled by the absolute location counter.

Format:

| label (optional) | SECA | symbol | comments (optional) |
|---|---|---|---|

Example:

```
SECA  ABS  ASSEMBLE IN ABSOLUTE
```

**ORG**

The ORG directive tells the assembler to switch from the location counter in use to a location counter for some other control section or to change the counter in use to a specified value.

Format:

| label (optional) | ORG | expression | comments (optional) |
|---|---|---|---|

If the expression contains a symbol naming a control section, ORG uses the value of the location counter for that section in place of the symbol.

Example:

```
SECP  ALPHA
.
.
.
SECD  COMM,1,100
.
.
.
ORG   ALPHA
```

Program control section ALPHA.

Chapter one labeled common block COMM.

Resume program control section.

**LIT**

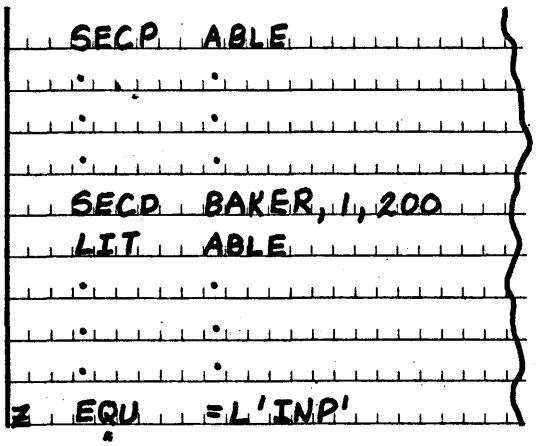
The LIT directive designates the location of literals. The assembler places literals in the control section most recently specified by a LIT directive, regardless of which control section contains the reference. In the absence of a LIT directive, the assembler appends the literals to the first program control section. META makes only one entry for identical literals in any given literal table.

Format:

| label | | | |
|---|---|---|---|
| (optional) | LIT | symbol | comments (optional) |

Symbol names a previously defined control section after which the literals are to be placed.

Example:




Program control section named ABLE.

Labeled common block BAKER. Literals to follow ABLE.

The value of Z is the address of the literal left-adjusted BCD value INP.

**RES/ RESB**

A programmer can change the contents of the current location counter by issuing an RES or RESB directive.

Format:

| label | | | |
|---|---|---|---|
| (optional) | RES RESB | expression | comments (optional) |

The value of the expression can be in words (RES) or bytes (RESB). The label is assigned the value of the current location counter before the addition.

Examples:

```
B   RESB   16
C   RESB   $-B
```

Increment location counter by four words (16 bytes). Increment location counter by 16 more bytes.

In the above example $ is word address and B is a byte address. Evaluation of the expression $ - B causes conversion of $ from a word address to a byte address. Thus, $ = B + 16 bytes, and $ - B becomes (B + 16) - B, or 16 bytes.

## DATA GENERATION

Seven directives generate words or bytes of information to be loaded into the computer at execution time:

| | | |
|---|---|---|
| GEN | TEXT | FORM |
| GEND | TEXTC | |
| GENB | TEXTA | |

### GEN/ GEND/ GENB

The GEN, GEND, and GENB directives generate one word, two words, or one byte, respectively, for each element in the operand field set. A set name may be used in lieu of a set. Subsets are not allowed.

Format:

| label | | | |
|---|---|---|---|
| (optional) | GEN<br>GEND<br>GENB | set | comments<br>(optional) |

Examples:

```
    GEN    11,12,13
```

Generates three object computer words.

```
    GENB    1,63
```

Generates two object computer bytes. The value of an expression must not exceed the number of bits per byte.

```
    GEND    2.7,C'ABCDEFGH'
```

Generates two 2-word (double-precision) elements.

```
A   NSET    1,2,4
    GEN     A
```

Generates three object computer words.

**TEXT/**
**TEXTC/**
**TEXTA**

The TEXT, TEXTC, and TEXTA directives direct the assembler to generate words of BCD text, characters of BCD text, and words of ASCII text, respectively, from a character string delimited by apostrophes given in the operand field.

Format:

| | | | |
|---|---|---|---|
| | TEXT | | |
| label | TEXTC | 'string' | comments |
| (optional) | TEXTA | | (optional) |

Examples:

```
A   TEXTC   ' THIS IS TEXT '
```

Generates a string of 6-bit characters without padding the last word.

```
B  TEXT  ' THIS IS TEXT '  $
```

If the object computer uses 24-bit words, generates four words containing the internal BCD character string THIS IS TEXT left adjusted and the last word filled with blanks.

| ^ | T | H | I | S | ^ | I | S | ^ | T | E | X | T | ^ | ^ | ^ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| word 1 | | | | word 2 | | | | word 3 | | | | word 4 | | | |

```
C  TEXA  'THIS IS ASCII TEXT'  $
```

Generates a string of 8-bit characters and pads the last word with the internal representation of ASCII blanks.

**FORM**

The FORM directive facilitates the generation of data by fields. It does not generate data. It defines a data format from left to right in terms of fields in one or more bytes.

Format:

symbol FORM $expression_1$, $expression_2$, . . . , $expression_n$ comments (optional)

The symbol is the name by which the FORM is subsequently referenced. Each expression in the operand field defines a field size in bits. The total number of bits must be a discrete number of bytes.

A programmer generates data specified by a FORM directive by placing the label of the FORM directive in the command field of a source statement and supplying a set of expressions corresponding to the fields in the operand field.

Format:

| label (optional) | form label | set | comments (optional) |
|---|---|---|---|

Examples:

```
ITEM  FORM  6,2,1,15
.     .     .
.     .     .
.     .     .
ITEM  12,3,0,$+2
```

Defines 4 fields in 4 bytes (one word) as FORM named ITEM.

Generates word using FORM named ITEM and values supplied as operands.

```
CHARS  FORM  6,6,6,6
.      .     .
.      .     .
.      .     .
CHARS  C'A',63,15,Ø'12'
```

Defines four 6-bit fields as FORM named CHARS.

Generates 6-bit fields using FORM named CHARS. Octal values of fields are:

21 77 17 12

## PROCEDURE DEFINITION AND USE

A procedure definition consists of lines of source statements beginning with a PROC directive and ending with an ENDS directive. The definition includes at least one NAME directive giving a symbol by which the procedure is subsequently referenced.

When the assembler encounters a procedure definition, it interprets only the NAME directives in the code body. It compresses and stores the remainder of the definition in an assembler table of definitions.

Each time the assembler interprets a reference to the procedure, it temporarily defines as many as three sets and interprets the body of the procedure. Statements within the procedure refer to these sets as though they were defined by NSET directives. The sets allow extensive parameterization and conditional generation of code by a procedure.

The name of the first set is derived from the label field of the PROC directive. The elements are taken from the operand field of the NAME directive identified in the call.

The names of the second and third sets are derived from the command field and operand field of the PROC directive. The elements are supplied in the corresponding fields of the statement containing the procedure reference.

A procedure can contain other procedures, or references to other procedures, nested to a level of 14.

Procedures can be stored on the system library or some other file for later retrieval by a LIBS directive.

The Meta-Assembler includes a standard set of library procedures for the 3300/3500 mnemonic machine instructions (appendix D).

**PROC**

A PROC directive declares the beginning of a procedure and provides for the definition of three sets used in the procedure.

Format:

| $setname_1$ | PROC, $setname_2$ | $setname_3$, expression | comments |
|---|---|---|---|
| (optional) | (optional) | (both optional) | (optional) |

The set names are optional. The expression in the operand field of the PROC directive must be nonzero for a procedure containing a forward reference. It notifies the assembler that assembly of the procedure requires two passes.

Example:

```
X3300    PRØC    PR,1
```

X3300 names a set, the possible elements of which are in the operand fields of the NAME directives; PR names a set, the elements of which are in the operand field of the procedure reference statement.

**NAME**

Each NAME directive in a procedure provides a name (procname) by which the procedure can be referenced and, optionally, supplies a NAME-dependent set of elements. The NAME directive acts as an entry point: only statements following it are interpreted when a procedure is referenced by the procname.

Format:

| procname | NAME | set (optional) | comments (optional) |
|---|---|---|---|

Example:

```
X3300   PROC   PR,1
LDA     NAME   Ø'20'
STA     NAME   Ø'40'
.
.
.
```

Procedure name LDA assigns one-element set with value $20_8$ to set named X3300.

Procedure name STA assigns one-element set with value $40_8$ to set named X3300.

**ENDS**

The statement that terminates a procedure is ENDS.

Format:

| label (optional) | ENDS | comments (optional) |
|---|---|---|

**PROCEDURE REFERENCE**

After defining a procedure, the programmer can refer to it by entering a procname in the command field of a source statement. The procname can be any of the procnames assigned by NAME directives in the definition. The reference optionally provides elements for two set names in corresponding fields of the PROC directive.

Format:

| label (optional) | procname, | $set_1$ (both optional) | $set_2$ | comments (optional) |
|---|---|---|---|---|

After interpretation of a procedure, the assembler interprets the statement following the statement containing the procedure reference.

Example:

```
X3300   PRØC,X   PR,1
LDA     NAME     Ø'20'
STA     NAME     Ø'40'
FI      FØRM     6,3,15
        FI       X3300[1],PR[2],PR[1]
.
.
.
        ENDS
```

Procedure LDA.
Procedure STA.
Form reference

```
KING    LDA      TAG,B3
        STA      BØB,B2
.
.
.
```

Procedure references: each passes PR set elements through operand field and picks up elements of NAME-dependent set X3300. Neither reference passed elements to command field set X.

Code Generated:

| | | |
|---|---|---|
| $20_8$ | (B3) | (TAG) |
| $40_8$ | (B2) | (BOB) |

23 18 17 15 14 00

X3300(1) PR(2) PR(1)

**LIBS**

A programmer can retrieve predefined procedures from a library file or some other file by issuing a LIBS directive. The procedures are stored in the library file by the MASTER library generation program, GLIB.

Format:

label (optional) LIBS L'dsi', $symbol_1, \ldots, symbol_n$ comments (optional)

The symbols are procnames of procedures to be obtained from the 3300/3500 MASTER file identified by its data set identifier, dsi.

Example:

```
LIBS  L'*LIB', ALPHA, BETA, GAMMA
```

Obtain procedure definitions named ALPHA, BETA, and GAMMA from the MASTER system library, *LIB. Note that if all three names are in the same definition, the assembler loads only one definition.

```
LIBS  L'ABC', PSI, ØMICRØN
```

Obtain PSI and OMICRON from file with MASTER data set identifier ABC.

## FUNCTION DEFINITION AND USE

A function definition consists of lines of source statements beginning with a FUNC directive and ending with an ENDS directive. The definition includes at least one NAME directive, giving a symbol by which the function is subsequently referenced.

When the assembler encounters a function definition, it interprets only the NAME directives in the code body. It compresses and stores the remainder of the definition in an assembler table of definitions.

Each time the assembler interprets a reference to the function, it defines up to two sets, interprets the body of the function, and returns information to the statement containing the function reference. The sets allow extensive parameterization and conditional generation of code.

The name of the first set is derived from the label field of the FUNC directive. The elements are taken from the operand field of the NAME directive identified in the reference. The name of the second set is derived from the operand field of the FUNC directive. The elements are supplied as parameters of the function reference enclosed in parentheses.

A function can contain other functions, or references to other functions, nested to a level of 14.

The Meta-Assembler includes an intrinsic set of functions called attribu functions (chapter 4).

After interpretation of a function, the assembler uses the information to complete evaluation of the expression containing the reference.

**FUNC**

A FUNC directive declares the beginning of a function and provides for the definition of two sets used by the function.

Format:

| | | | |
|---|---|---|---|
| $setname_1$ (optional) | FUNC | $setname_2$ (optional) | comments (optional) |

Example:

```
BITTEN  FUNC    NUMB                    5
```

The function refers to sets named BITTEN and NUMB.

**NAME**

Each NAME directive in a function provides a name (funcname) by which the function can be referenced and, optionally, supplies a NAME-dependent set of elements. The NAME directive acts as an entry point; only statements following it are interpreted when a function is referenced by the funcname.

Format:

| | | | |
|---|---|---|---|
| funcname | NAME | set (optional) | comments (optional) |

Example:

```
BITTEN   FUNC   NUMB
LENGTH   NAME   36
WIDTH    NAME   24
HEIGHT   NAME   36
.        .      .
.        .      .
.        .      .
```

Each name assigns a different one-element set to the set named BITTEN.

**ENDS**

The statement that terminates a function is ENDS.

Format:

| label (optional) | ENDS | expression | comments (optional) |
|---|---|---|---|

The expression is either an expression that defines the function value or a set that defines a set of values to be used in the calling statement.

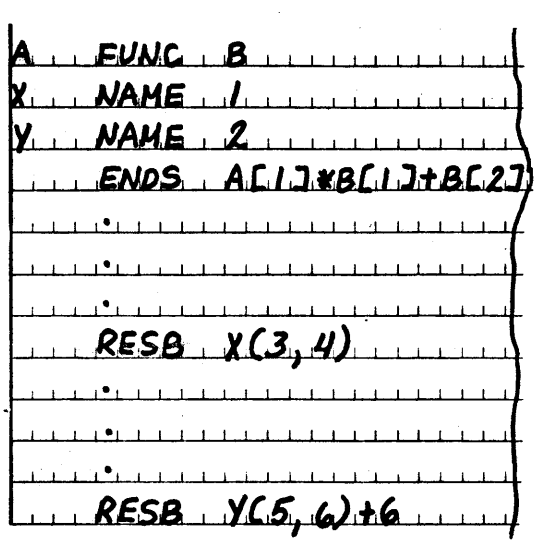
Example:

```
    ENDS    BITTEN[I]*NUMB[I]
```

**FUNCTION REFERENCE**

After defining a function, a programmer can refer to it by using a function name as a symbolic element in an expression. The function name can be any of the funcnames assigned by NAME directives in the definition.

Example:




Function named X.

Function named Y.

Reference to function X; increments location counter by 7.

Reference to function Y; increments location counter by 22.

Note that elements for set B are passed as parameters of the function reference.

## LISTING CONTROL

Seven directives provide programmer control over the format and content of the assembler-generated listing.

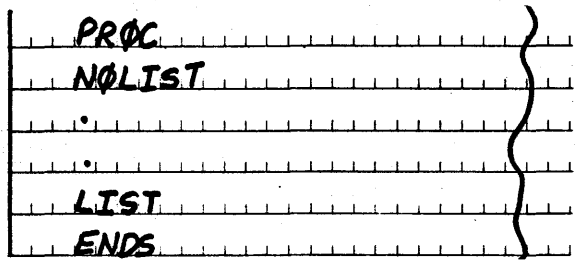
### NOLIST/ LIST

The NOLIST directive causes the assembler to discontinue generation of the output listing, beginning with the NOLIST directive, until it encounters a LIST directive.

Format:

| label | NOLIST | comments |
|---|---|---|
| (optional) | LIST | (optional) |

Example:




**SPACING**

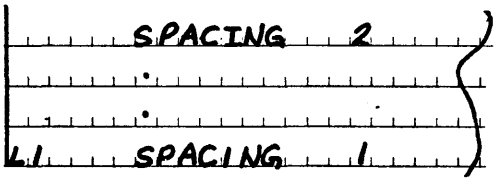
The SPACING directive causes the assembler to use the specified spacing (single, double, or triple) until it encounters another SPACING directive.

Format:

| label (optional) | SPACING | expression | comments (optional) |
|---|---|---|---|

The value of the expression can be one, two, or three.

Example:




Double space listing.

Single space listing.

**EJECT**

An EJECT directive causes the assembler to terminate listing on the current page and resume listing at the top of the following page.

Format:

| label | EJECT | comments |
|---|---|---|
| (optional) | | (optional) |

**TITLE**

The programmer can issue a TITLE directive to eject the current page and begin the listing on the following page with a line of text supplied in the directive.

Format:

| label | TITLE | 'text' | comments |
|---|---|---|---|
| (optional) | | | (optional) |

Text is a character string.

Example:

```
[, TITLE, 'PROCEDURE NAMED ALPHA AND BETA',)
```

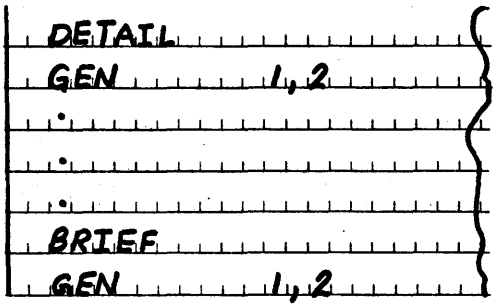
**DETAIL/ BRIEF**

The DETAIL directive causes the assembler to list all lines of generated code and procedure and function expansions in addition to normal list output until it interprets a BRIEF directive. A NOLIST directive takes precedence over a DETAIL directive. If no DETAIL directive is issued, the mode is BRIEF.

Format:

| label | DETAIL | comments |
|---|---|---|
| (optional) | BRIEF | (optional) |

Example:




Begin detailed listing.
Produces two lines of print.

Begin brief listing.
Produces one line of print.

## ASSEMBLY TERMINATION

The programmer can independently assemble more than one subprogram during a Meta-Assembler run. Two directives indicate whether the assembler is to terminate assembly of a subprogram or terminate assembly completely.

### END

The programmer issues an END directive at the end of each subprogram being assembled. The directive optionally specifies a symbolic location at which program execution is to begin.

Format:

| label | | symbol | comments |
|---|---|---|---|
| (optional) | END | (optional) | (optional) |

### FINIS

FINIS causes termination of the assembly process. The directive follows the END directive for the final subprogram.

Format:

| label | | comments |
|---|---|---|
| (optional) | FINIS | (optional) |

An attribute is an inherent characteristic of an expression symbol or set such as its size or arithmetic mode. Seven intrinsic functions defined by META provide the programmer with access to attributes which are returned as values. An attribute function call consists of the function name, used as an element of an expression, followed by an argument enclosed in parentheses.

Ordinarily, a reference to an attribute function occurs in an operand field.

## MDE

The mode attribute function (MDE) returns the mode of the argument as a decimal integer value.

| Mode | Value |
|---|---|
| No value | 0 |
| Integer value | 1 |
| Real value | 2 |
| BCD character string, right adjusted | 3 |
| BCD integer string | 4 |
| BCD character string, left adjusted | 5 |
| ASCII character string | 7 |
| Word address | 9 |
| Byte address | 11 |

Reference format:

```
MDE (expression)
```

Example:

```
MJ  GOTO  MODE(DELTA),B,C,D,E,F,,G,,H,,J
```

For an integer value, assembly continues at B; for a real value, assembly continues at C; etc.

**SZE**

The size of a defined symbol value is accessible through the size attribute function (SZE). This function returns the number of bytes occupied by an address or an integer or real value, or it returns the number of characters in a BCD or ASCII character string.

Reference format:

SZE (expression)

Example:

```
A   EQU   2.4
    RESB  SZE(A)
```

Real value, 2 words (8 bytes).
Reserve 8 bytes.

**REL**

The relocation attribute function (REL) returns the number of the location counter used by the indicated control section.

Reference format:

REL (expression)

Example:

```
CTR  EQU  REL(ALPHA)
```

The value assigned to CTR is a decimal integer designating the number of the location counter (0-15) used by control section ALPHA.

**NUM**

The number of elements function (NUM) returns the number of elements in the named set as an octal integer.

Reference format:

NUM (setname)

Example:

```
SETSIZE  EQU  NUM(BETA)          ;
```

The value assigned to SETSIZE is the number of elements in the set named BETA.

**BYT**

The byte address function (BYT) returns the byte address of the argument expression.

Reference format:

BYT (expression)

Example:

```
A  EQU  BYT($)+1                 ;
```

A is assigned the current value of the location counter currently in use as a byte address plus one.

```
B  EQU  BYT(XRAY+1)              ;
```

**WRD**

The word address function (WRD) returns the word address of the argument expression. A byte address that does not correspond to a word address is flagged as an error.

Reference Format:

WRD (expression)

Example:

```
FF   EQU   WRD($)
```

If the current value of the location counter currently in use is a discrete number of words, the number of words is assigned to FF. Otherwise, an error results.

**SYM**

The symbol attribute function (SYM) permits the value of the argument expression to be used as a symbol. The reference can occur in any field of a statement.

Reference format:

SYM (expression)

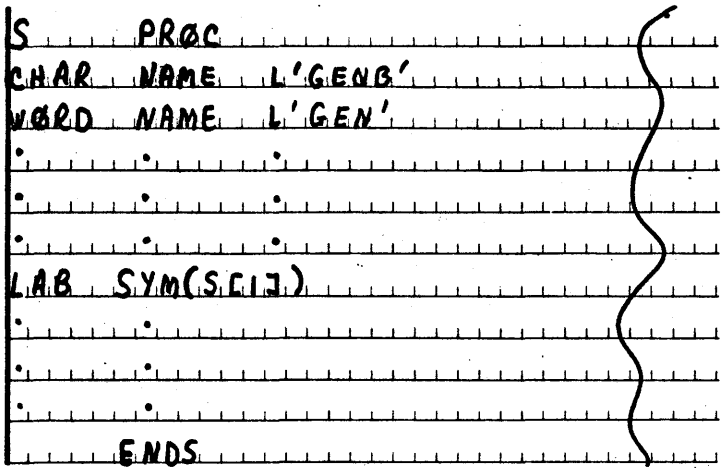
Examples:

```
A      EQU   L'XYZ'
SYM(A) EQU   10
```

The value of A is XYZ. The SYM reference to A causes XYZ to be interpreted as a symbol in the EQU statement. Thus the symbol XYZ is assigned the value 10.

```
SYM(0'21212121') RDEF  1
```

The symbol AAAA is assigned a value of 1.




If the procedure is called by name CHAR, the statement labeled LAB uses GENB to generate bytes; if called by name WORD, the same statement generates words.

# CHARACTER SET



| Type of Character | Printer Graphic | Internal Code Octal | Card Code |
|---|---|---|---|
| Alphabetic | A | 21 | 12,1 |
| | B | 22 | 12,2 |
| | C | 23 | 12,3 |
| | D | 24 | 12,4 |
| | E | 25 | 12,5 |
| | F | 26 | 12,6 |
| | G | 27 | 12,7 |
| | H | 30 | 12,8 |
| | I | 31 | 12,9 |
| | J | 41 | 11,1 |
| | K | 42 | 11,2 |
| | L | 43 | 11,3 |
| | M | 44 | 11,4 |
| | N | 45 | 11,5 |
| | O | 46 | 11,6 |
| | P | 47 | 11,7 |
| | Q | 50 | 11,8 |
| | R | 51 | 11,9 |
| | S | 62 | 0,2 |
| | T | 63 | 0,3 |
| | U | 64 | 0,4 |
| | V | 65 | 0,5 |
| | W | 66 | 0,6 |
| | X | 67 | 0,7 |
| | Y | 70 | 0,8 |
| | Z | 71 | 0,9 |
| Numeric | 0 | 00 | 0 |
| | 1 | 01 | 1 |
| | 2 | 02 | 2 |
| | 3 | 03 | 3 |
| | 4 | 04 | 4 |
| | 5 | 05 | 5 |
| | 6 | 06 | 6 |
| | 7 | 07 | 7 |
| | 8 | 10 | 8 |
| | 9 | 11 | 9 |

| Type of Character | Printer Graphic | | Internal Code Octal | Card Code |
|---|---|---|---|---|
| Blank | blank | | 60 | space |
| Special | + | plus | 20 | 12 |
| | - | minus | 40 | 11 |
| | x | times | 54 | 11,4,8 |
| | / | divide | 61 | 0,1 |
| | = | equals | 13 | 3,8 |
| | < | less than | 32 | 12,0 |
| | > | greater than | 57 | 11,7,8 |
| | . | period | 33 | 12,3,8 |
| | , | comma | 73 | 0,3,8 |
| | ( | left parenthesis | 74 | 0,4,8 |
| | ) | right parenthesis | 34 | 12,4,8 |
| | % | percent | 16 | 6,8 |
| | $ | dollar | 53 | 11,3,8 |
| | ' | apostrophe | 14 | 4,8 |
| | ≤ | less or equal | 15 | 5,8 |
| | ≥ | greater or equal | 35 | 12,5,8 |
| | [ | left bracket | 17 | 7,8 |
| | ] | right bracket | 72 | 0,8,2 |
| | ↑ | decimal exponent | 55 | 11,5,8 |
| | ↓ | binary exponent | 56 | 11,6,8 |
| | ¬ | not | 36 | 12,6,8 |
| | ; | semicolon | 37 | 12,7,8 |
| | → | right arrow | 75 | 0,5,8 |
| | ≡ | identity | 76 | 0,6,8 |
| | : | colon | 12 | 2,8 |
| | ∨ | or | 52 | 11,0 |
| | ∧ | and | 77 | 0,7,8 |

15 ✓

13 ×

# OPERATORS B

| Symbol | Alternate Mnemonic | Meaning |
|---|---|---|
| + | | Unary plus† |
| - | | Unary minus† |
| ↑ | DS | Decimal scaling |
| ↓ | BS | Binary scaling |
| * | | Arithmetic product |
| / | | Arithmetic quotient |
| + | | Arithmetic addition |
| - | | Arithmetic subtraction |
| < | LT | Less than (compare) |
| = | EQ | Equal (compare) |
| ¬= | NE | Not equal (compare) |
| > | GT | Greater than (compare) |
| ≤ | LE | Less than or equal (compare) |
| ≥ | GE | Greater than or equal (compare) |
| ** | AND | Logical product (AND) |
| -- | XOR | Logical difference (exclusive OR) |
| ++ | OR | Logical addition (inclusive OR) |
| = | | Unary equals† (literal) |

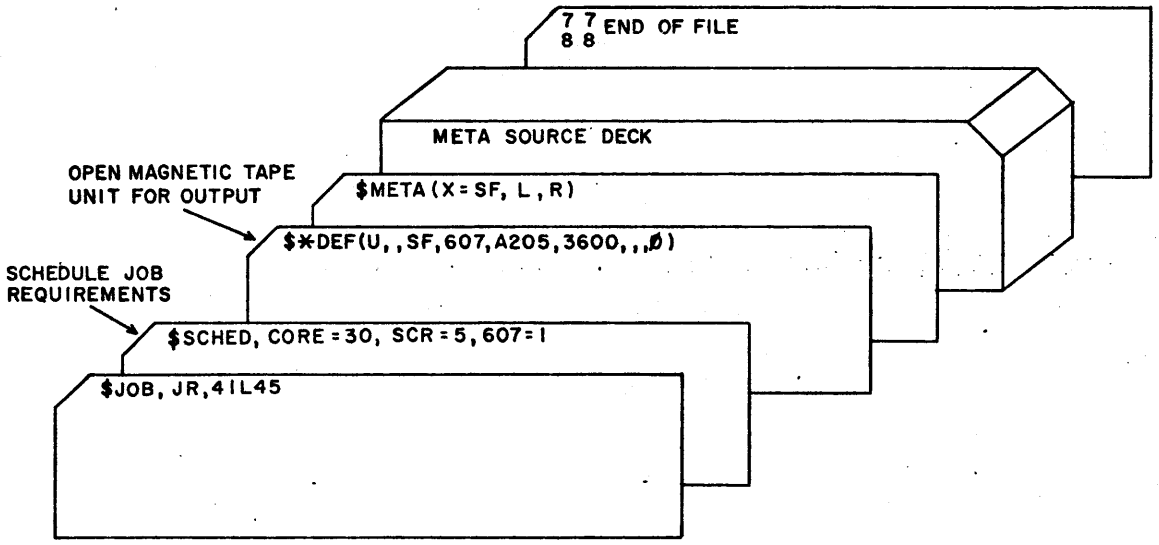
Examples:

A>B has value 1 if the value of A is greater than the value of B; otherwise it has value 0. Zero is greater than -0.
A GT B is an alternate representation of the above. A single blank separates 'A' and "GT" and 'B".

† A unary operator has only one operand. It is most commonly used as the sign of a simple expression.

# SAMPLE PROGRAM C

This sample program defines procedures for 3600 mnemonic instructions LDA, NOP, RTJ, and UJP and illustrates how a program can be coded using the 3600 mnemonics. Because assembly is for a machine other than the 3300 or 3500, output must be written on a permanent file. In this example, a magnetic tape file is used.



```
        UNIT    24,2                    DEFINE 48-BIT WORD FOR CDC 3600
F       FORM    6,3,15                  DEFINE BASIC INSTR FORMAT
N       PROC    R
LDA     NAME    0'12'                   FUNCTION CODE FOR LDA IS 12
NOP     NAME    0'50'                   FUNCTION CODE FOR NOP IS 50
        F       N[1],R[2],WRD(R[1])     GENERATE LDA OR NOP INSTR
        ENDS
N       PROC    R
RTJ     NAME    0'75'                   FUNCTION CODE FOR RTJ IS 75
A       RESB    0                       A = CURRENT BYTE ADDR
        GOTO    A AND 1,B
        NOP                             GENERATE NOP IN UPPER HALF WORD
B       F       N[1],4,WRD(R[1])        GENERATE RTJ INSTR.
        ENDS
N       PROC    R
UJP     NAME    0'75'                   FUNCTION CODE FOR UJP IS 75
        F       N[1],0,WRD(R[1])        GENERATE UJP INSTR.
        ENDS
START   UJP     0                       LOC 00000
        LDA     START
        RTJ     SUBR                    LOC 00001 WITH NOP IN UPPER HALF
        UJP     START                   LOC 00002
LOC     RES     1                       LOC 00003
SUBR    UJP     0                       LOC 00004
        LDA     LOC
        UJP     SUBR                    LOC 00005
        END     START
```

# 3300/3500 MNEMONIC INSTRUCTIONS D

A 3300/3500 META mnemonic instruction is a procedure reference in which the label field optionally contains a symbolic address, the command field contains a mnemonic instruction and modifiers, and the operand field contains operands that depend on the mnemonic.

Mnemonic instructions can be used in a program that does not contain a UNIT directive.

Abbreviated descriptions of the mnemonic language given here subscribe to the following conventions:

| | |
|---|---|
| ( )⟶( ) | Indicates that the contents of one (or two consecutive) register, operand, or address field is replaced by the contents of another (or two consecutive) register, operand, or address field. For example, (M)⟶(A) means "replace contents of A register with contents of M operand field." |
| ∧ | Indicates the logical sum (AND) |
| ∨ | Indicates a logical OR |

| Term | Meaning |
|---|---|
| A | 24-bit A register |
| b | Index register designator 1 to 3 |
| B | Index register defined by $B^b$ |
| $B_m$ | Index register flag, M = m+($B_m$) for these instructions only |
| $B_r$ | Index register flag; if $B_r$ = 1 or 3, R = r+ ($B^1$); if $B_r$ = 2, R = r+ ($B^2$); if $B_r$ = 0, R = r |
| $B_s$ | Index register flag; if $B_s$ = 1 or 3, S = s+ ($B^1$); if $B_s$ = 2, S = s+ ($B^2$); if $B_s$ = 0, S = s |
| c | $00\text{-}77_8$ BCD code of search character |
| cm | 8-bit channel mask |
| E | 48 (52)-bit E register |
| $E_\ell$ | Lower half of 48-bit E register (bits 23-00) |
| $E_u$ | Upper half of 48-bit E register (bits 47-24) |
| i | Increment or decrement, 0 to 7 |

| Term | Meaning |
|---|---|
| k | Shift count |
| $\ell$ | Field length of block, $0\text{-}177_8$ |
| $\ell_r$ | Number of characters in field R |
| $\ell_s$ | Number of characters in field S |
| m | 15-bit word address, first operand or jump address |
| M | Actual operand or jump address as modified; M = m+ ($B^b$) |
| n | Same as m, second operand address |
| p | 15 (or 17)-bit P register |
| Q | 24-bit Q register |
| r | 17-bit character address |
| R | Actual character address as modified; R = r+ ($B^b$) |
| s | Same as r, second operand address |
| S | Same as R, second operand address; S = s+ ($B^b$) |
| v | 6-bit address in register file |
| sc | Scan character |
| w | Page index file address |
| x | Connect code or interrupt mask |
| y | 15-bit operand |

| Modifiers | Meaning |
|---|---|
| A | Conversion |
| B | Backward read or write |
| C | Evaluate address expression modulo $2^{17}-1$ |
| dc | Delimiting character |
| EQ | Equal |
| GE | Greater than or equal |
| H | Half assembly or disassembly |
| I | Indirect addressing |

| Modifiers | Meaning |
|---|---|
| INT | Interrupt on completion |
| N | No assembly or disassembly |
| NC | No conversion |
| NE | Not equal |
| S | Instruction modifier denoting sign extension: S present, sign extended; S omitted, no sign extension |

| Command Field | Operand Field | Operation |
|---|---|---|
| ACI | | $(A_{00-02}) \rightarrow$ channel index register |
| ADA, I | m, b | $(A)+(M) \rightarrow (A)$ |
| ADAQ, I | m, b | $(A, Q)+(M, M+1) \rightarrow (A, Q)$ |
| ADM | $r, B_r, \ell_r, s, B_s, \ell_s$ | Add field R to field S $\rightarrow$ field S |
| AEU | | $(A) \rightarrow (E_U)$ |
| AIA | b | $(A)+(B^b) \rightarrow (A)$, sign of $(B^b)$ is extended prior to addition |
| AIS | | $(A_{00-02}) \rightarrow$ instruction state register |
| ANA | y | $y \wedge (A) \rightarrow (A)$ |
| ANA, S | y | $y \wedge (A) \rightarrow (A)$, sign of y extended |
| ANI | y | No operation |
| ANI | y, b | $y \wedge (B^b) \rightarrow (B^b)$ |
| ANQ | y | $y \wedge (Q) \rightarrow (Q)$ |
| ANQ, S | y | $y \wedge (Q) \rightarrow (Q)$, sign of y extended |
| AOS | | $(A_{00-02}) \rightarrow$ operand state register |
| APF | w, 2 | $(A_{00-11}) \rightarrow$ page file |
| AQA | | $(A)+(Q) \rightarrow (A)$ |

| Command Field | Operand Field | Operation |
|---|---|---|
| AQE | | $(A, Q) \longrightarrow (E_U, E_L)$ |
| AQJ, EQ | m | If (A) = (Q), RNI m, otherwise, RNI P+1 |
| AQJ, GE | m | If (A) $\geq$ (Q) RNI m, otherwise, RNI P+1 |
| AQJ, LT | m | If (A) $<$ (Q), RNI m, otherwise, RNI P+1 |
| AQJ, NE | m | If (A) $\neq$ (Q), RNI m, otherwise, RNI P+1 |
| ASE | y | If y = $(A_{00-14})$, RNI P+2, otherwise, RNI P+1 |
| ASE, S | y | If y = $(A_{00-14})$, RNI P+2, otherwise, RNI P+1, sign of y is extended |
| ASG | y | If (A) $\geq$ y, RNI P+2, otherwise, RNI P+1 |
| ASG, S | y | If (A) $\geq$ y, RNI P+2, otherwise, RNI P+1, sign of y is extended |
| ATD | $m, B_m, \ell_m, s, B_s$ | Translate American Standard Code field M $\longrightarrow$ BCD character field S |
| ATD, dc | $m, B_m, \ell_m, s, B_s$ | Translate American Standard Code field M $\longrightarrow$ BCD character field S with delimiting character possibility |
| AZJ, EQ | m | If (A) = 0, RNI m, otherwise, RNI P+1 |
| AZJ, GE | m | If (A) $\geq$ 0, RNI m, otherwise, RNI P+1 |
| AZJ, LT | m | If (A) $<$ 0, RNI m, otherwise, RNI P+1 |
| AZJ, NE | m | If (A) $\neq$ 0, RNI m, otherwise, RNI P+1 |
| CIA | | 0 $\longrightarrow$ (A), then channel index register $\longrightarrow$ $(A_{00-02})$ |

| Command Field | Operand Field | Operation |
|---|---|---|
| CILO | cm | Lockout external interrupt on masked channels, cm, until channel is not busy |
| CINS | ch | Interrupt mask and internal status $\longrightarrow$ (A) |
| CLCA | cm | Clear the specified channel, but not external equipment |
| CMP | r, $B_r$, $\ell_r$, s, $B_s$, $\ell_s$ | Compare field R to field S, exit upon encountering $\neq$ characters |
| CMP, dc | r, $B_r$, s, $B_s$, $\ell_s$ | Compare field R to field C, exit upon encountering $\neq$ characters; delimiting character possibility |
| CON | x, ch | If channel ch is busy, reject instruction, RNI P+1. If channel ch is not busy, send 12-bit connect code (x) on channel ch with connect enable, RNI P+2 |
| COPY | ch | External status code from I/O channel ch $\longrightarrow$ ($A_{00-11}$), (interrupt mask register) $\longrightarrow$ ($A_{12-23}$), RNI P+1 |
| CPR, I | m, b | (M) > (A), RNI P+1<br>(Q) > (M), RNI P+2<br>(A) $\geq$ (M) $\geq$ (Q), RNI P+3<br>} (A) and (Q) are unchanged |
| CTI | | Set console typewriter input<br>Set console typewriter output<br>} Beginning character address must be present in location 23 of register file and last character +1 must be present in location 33 of the file |
| CTO | | |

| Command Field | Operand Field | Operation |
|---|---|---|
| CVBD | $m, B_n, n, B_n$ | Convert binary field M to BCD ⟶ field N |
| CVDB | $r, B_r, \ell_r$, $m, B_m$ | Convert BCD field R to binary ⟶ field M |
| DINT | | Disable interrupt control |
| DTA | $r, B_r, \ell_r$, $m, B_m$ | Translate BCD field R to American Standard Code ⟶ field M |
| DTA, dc | $r, B_r, \ell_r$, $m, B_m$ | Translate BCD field R to American Standard Code ⟶ field M; delimiting character possibility |
| DVA, I | m, b | (A, Q)/(M) ⟶ (A), remainder ⟶ (Q) |
| DVAQ, I | m, b | (A, Q, E)/(M, M+1) ⟶ (A, Q), remainder with sign extended ⟶ (E) |
| EAQ | | $(E_U, E_L)$ ⟶ (A, Q) |
| ECHA | r | 0 ⟶ (A), then r ⟶ $(A_{00-16})$ |
| ECHA, S | r | 0 ⟶ (A), then r ⟶ $(A_{00-16})$, sign extended |
| EDIT | $r, B_r, \ell_r$, $s, B_s, \ell_s$ | Field R ⟶ field S with COBOL type of editing specified by picture previously stored in field S |
| EINT | | Interrupt control enabled; allows one more instruction to be executed before interrupt |
| ELQ | | $(E_L)$ ⟶ (Q) |
| ENA | y | 0 ⟶ (A), then y ⟶ $(A_{00-14})$ |
| ENA, S | y | 0 ⟶ (A), then y ⟶ $(A_{00-14})$, sign extended |
| ENI | y | No operation |

| Command Field | Operand Field | Operation |
|---|---|---|
| ENI | y, b | $0 \rightarrow (B^b)$, then $y \rightarrow (B^b)$ |
| ENQ | y | $0 \rightarrow (Q)$, then $y \rightarrow (Q_{00-14})$ |
| ENQ, S | y | $0 \rightarrow (Q)$, then $y \rightarrow (Q_{00-14})$, sign extended |
| EUA | | $(E_U) \rightarrow (A)$ |
| EXS | x, ch | Sense external status; if 1 bits occur on status lines in any of the same positions as 1 bits in the mask, RNI P+1; if no comparison, RNI P+2 |
| FAD, I | m, b | Floating-point addition of (M, M+1) to (A, Q) $\rightarrow$ (A, Q) |
| FDV, I | m, b | Floating-point division of (A, Q) by (M, M+1) $\rightarrow$ (A, Q); remainder with sign extended $\rightarrow$ (E) |
| FMU, I | m, b | Floating-point multiplication of (A, Q) and (M, M+1) $\rightarrow$ (A, Q) |
| FRMT | $r, B_r, \ell_r, s, B_s, \ell_s$ | Move field R $\rightarrow$ field S; replace leading zeros with blanks; insert a comma after every three characters moved; insert a decimal point in third lowest order position in S field |
| FSB, I | m, b | Floating-point subtraction of (M, M+1) from (A, Q) $\rightarrow$ (A, Q) |
| HLT | m | Unconditional stop, RNI m upon restarting |
| IAI | b | $(A)+(B^b) \rightarrow (B^b)$, sign of $B^b$ is extended prior to addition |
| IAPR | | Interrupt associated processor |
| IJD | m | No operation |
| IJD | m, b | If $(B^b) = 0$, RNI P+1; if $(B^b) \neq 0$, $(B^b) - 1 \rightarrow (B^b)$, RNI m |

| Command Field | Operand Field | Operation |
|---|---|---|
| IJI | m | No operation |
| IJI | m,b | If $(B^b) - 0$, RNI P+1; if $(B^b) \neq 0$, $(B^b) + 1 \rightarrow (B^b)$, RNI m |
| INA | y | Increase (A) by y |
| INA,S | y | Increase (A) by y, sign of y is extended |
| INAC,INT | ch | (A) is cleared and a 6-bit character is transferred from a peripheral device to the lower 6 bits of A |
| INAW,INT | ch | (A) is cleared and a 12- or 24-bit word is read from a peripheral device into the lower 12 bits or all of A (word size depends on I/O channel) |
| INCL | x | Interrupt faults defined by x are cleared |
| INI | y | No operation |
| INI | y,b | Increase $(B^b)$ by y, signs of y and $B^b$ extended |
| INPC,INT, B,H | ch,r,s | A 6- or 12-bit character is read from a peripheral device and stored in memory at a given location |
| ISG | y,b | If $(B^b) \geq y$, RNI P+2, otherwise, RNI P+1 |
| ISI | y,b | If $(B^b) = y$, clear $B^b$ and RNI P+2; if $(B^b) \neq y$, $(B^b) + 1 \rightarrow (B^b)$, RNI P+1 |
| JAA | | Last executed jump address $\rightarrow (A_{00-14})$ |
| JMP,HI | m | Jump if BDP condition register $> 0$ or + |
| JMP,LOW | m | Jump if BDP condition register $> 0$ or - |

| Command Field | Operand Field | Operation |
|---|---|---|
| JMP, ZRO | m | Jump if BDP condition register = 0 |
| LACH | r, 1 | $0 \rightarrow (A)$, $(R) \rightarrow (A_{00-05})$ |
| LBR | m | Load BDP conditions with the contents of m |
| LCA, I | m, b | $\overline{(M)} \rightarrow (A)$ |
| LCAQ, I | m, b | $\overline{(M)} \rightarrow (A)$, $\overline{(M+1)} \rightarrow (Q)$ |
| LDA, I | m, b | $(M) \rightarrow (A)$ |
| LDAQ, I | m, b | $(M) \rightarrow (A)$, $(M+1) \rightarrow (Q)$ |
| LDI, I | m, b | $(M_{00-14}) \rightarrow (B^b)$ |
| LDL, I | m, b | $(M) \wedge (Q) \rightarrow (A)$ |
| LDQ, I | m, b | $(M) \rightarrow (Q)$ |
| LPA, I | m, b | $(M) \wedge (A) \rightarrow (A)$ |
| LQCH | r, 2 | $0 \rightarrow (Q)$, $(R) \rightarrow (Q_{00-05})$ |
| MEQ | m, i | $(B^1) - i \rightarrow (B^1)$; if $(B^1)$ negative, RNI P+1; if $(B^1)$ positive, test (A) = (Q) $\wedge$ (M); if true, RNI P+2, if false, repeat sequence |
| INPW, INT, B, N | ch, m, n | Word address is placed in bits 00-14; 12- or 24-bit words are read from a peripheral device and stored in memory |
| INQ | y | Increase (Q) by y |
| INQ, S | y | Increase (Q) by y, sign of y extended |
| INS | x, ch | Sense internal status; if 1 bits occur on status lines in any of the same positions as 1 bits in the mask, RNI P+1; if no comparison, RNI P+2 |

| Command Field | Operand Field | Operation |
|---|---|---|
| INTS | x, ch | Sense for interrupt condition; if 1 bits occur simultaneously in interrupt lines and in the interrupt mask, RNI P+1; if not, RNI P+2 |
| IOCL | x | Clears I/O channel or search/ move control as defined by bits 00-07, 08, and 11 of x |
| ISA | | $0 \longrightarrow (A)$, instruction state register $\longrightarrow (A_{00-02})$ |
| ISD | y | If y = 0, RNI P+2; if $y \neq 0$, RNI P+1 |
| ISD | y, b | If $(B^b) = y$, clear $B^b$ and RNI P+2; if $(B^b) \neq y$, $(B^b) - 1 \longrightarrow (B^b)$, RNI P+1 |
| ISE | y | If y - 0, RNI P+2, otherwise, RNI P+1 |
| ISE | y, b | If $y = (B^b)$, RNI P+2, otherwise, RNI P+1 |
| ISG | y | If $y \geq 0$, RNI P+2, otherwise, RNI P+1 |
| MOVE, INT | $\ell$, r, s | Move $\ell$ characters from r to s; $0 \leq \ell \leq 127_{10}$ |
| MTH | m, i | $(B^2) - i \longrightarrow (B^2)$, if $(B^2)$ negative, RNI P+1; if $(B^2)$ positive, test $(A) \geq (Q) \wedge (M)$; if true, RNI P+2; if false, repeat sequence |
| MUA, I | m, b | $(A) * (M) \longrightarrow (Q, A)$ |
| MUAQ, I | m, b | $(A, Q) * (M, M+1) \longrightarrow (A, Q, E)$ |
| MVBF | r, $B_r$, $\ell_r$, s, $B_s$, $\ell_s$ | Move characters from field R $\longrightarrow$ field S; if field S > field R, blank fill |
| MVE | r, $B_r$, $\ell_r$, s, $B_s$, $\ell_s$ | Move characters from field R $\longrightarrow$ field S according to parameters |
| MVE, dc | r, $B_r$, s, $B_s$, $\ell_s$ | Move characters from field R $\longrightarrow$ field S; delimiting character possibility |

| Command Field | Operand Field | Operation |
|---|---|---|
| MVZF | $r, B_r, \ell_r$, $s, B_s, \ell_s$ | Move characters from field R → field S; if field S > field R, zero fill |
| MVZS | $r, B_r, \ell_r$, $s, B_s, \ell_s$ | Move characters from field R → field S; suppress leading zeros |
| MVZS, dc | $r, B_r, s$, $B_s, \ell_s$ | Move characters from field R → field S; suppress leading zeros; delimiting character possibility |
| NOP | | No operation (COMPASS assembled NOP) |
| OSA | | 0 → (A); operand state register → $(A_{00-02})$ |
| OTAC, INT | ch | Character from $(A_{00-05})$ is sent to peripheral device, (A) retained |
| OTAW, INT | ch | Transfers $(A_{00-11})$ or $(A_{00-23})$, depending on type of I/O channel, to a peripheral device |
| OUTC, INT, B, H | ch, r, s | Storage words assembled into 6- or 12-bit characters and sent to a peripheral device |
| OUTW, INT, B, N | ch, m, n | Transfer 12- or 24-bit words from storage to a peripheral device |
| PAK | $r, B_r, \ell_r$, $m, B_m$ | Convert and pack a 6-bit numeric BCD field R to a 4-bit numeric BCD field and store the result in field M |
| PAUS | x | Sense busy lines; if 1 appears on a line corresponding to 1 bits in x, do not advance P; if P is inhibited for longer than 40 ms, read reject instruction from P+1; if no comparison, RNI P+2 |
| PFA | w, 2 | 0 → (A), then page index file → $(A_{00-11})$ |
| PRP | x | Same as PAUS, except real-time clock cannot increment during the pause. |

| Command Field | Operand Field | Operation |
|---|---|---|
| QEL | | $(Q) \longrightarrow (E_L)$ |
| QSE | y | If y = $(Q_{00-14})$, RNI P+2, otherwise, RNI P+1 |
| QSE, S | y | If y - (Q), RNI P+2, otherwise, RNI P+1, sign of y is extended |
| QSG | y | If $(Q_{00-14}) \geq y$, RNI P+2, otherwise, RNI P+1 |
| QSG, S | y | If $(Q) \geq y$, RNI P+2, otherwise, RNI P+1, sign of y is extended |
| RAD, I | m, b | $(M)+(A) \longrightarrow (M)$ |
| RCR | | Subcondition register $\longrightarrow$ condition register |
| RIS | | Relocate to instruction state |
| ROS | | Relocate to operand state |
| RTJ | m | $(P)+1 \longrightarrow (m_{00-14})$, RNI m+1 |
| SACH | r, 2 | $(A_{00-05}) \longrightarrow (R)$ |
| SBA, I | m, b | $(A) - (M) \longrightarrow (A)$ |
| SBAQ, I | m, b | $(A, Q) - (M, M+1) \longrightarrow (A, Q)$ |
| SBCD | | Set BCD fault logic |
| SBJP | | Transfer system from monitor state to program state when next jump occurs |
| SBM | $r, B_r, \ell_r$, $s, B_s, \ell_s$ | Subtract field R from field S $\longrightarrow$ field S |
| SBR | m | Store BDP conditions in m |
| SCA, I | m, b | Where (M) contains a 1 bit, complement the corresponding bit in (A) |
| SCAN, LR, EQ, dc | $r, B_r, \ell_r$, sc | Scan field R from left to right, stop on = condition; delimiting character possibility |

| Command Field | Operand Field | Operation |
|---|---|---|
| SCAN, LR, NE, dc | $r, B_r, \ell_r, sc$ | Scan field R from left to right, stop on $\neq$ condition; delimiting character possibility |
| SCAN, RL, EQ, dc | $r, B_r, \ell_r, sc$ | Scan field R from right to left, stop on = condition; delimiting character possibility |
| SCAN, RL, NE, dc | $r, B_r, \ell_r, sc$ | Scan field R from right to left, stop on $\neq$ condition; delimiting character possibility |
| SCAN, LR, EQ | $r, B_r\ \ell_r, sc$ | Scan field R from left to right, stop on = condition |
| SCAN, LR, NE | $r, B_r, \ell_r, sc$ | Scan field R from left to right, stop on $\neq$ condition |
| SCAN, RL, EQ | $r, B_r, \ell_r, sc$ | Scan field R from right to left, stop on = condition |
| SCAN, RL, NE | $r, B_r, \ell_r, sc$ | Scan field R from right to left, stop on $\neq$ condition |
| SCAQ | k, b | Shift (A, Q) left end around until upper 2 bits of A are unequal; residue K = k- shift count; if b = 1, 2, or 3, $K \longrightarrow (B^b)$; if b = 0, K is discarded |
| SCHA, I | m, b | $(A_{00-16}) \longrightarrow (M_{00-16})$ |
| SCIM, I | x | Selectively clear interrupt mask register for each 1 bit in x; corresponding bit in the mask register is set to 0 |
| SDL | | Upon next LDA instruction:<br>1. $(M) \longrightarrow (A)$<br>2. $77777777 \longrightarrow (M)$ |
| SEL | x, ch | If channel ch is busy, read reject instruction from P+1; if not busy, send a 12-bit function code on channel ch with a function enable, RNI P+2 |
| SFPF | | Set floating-point fault logic |

| Command Field | Operand Field | Operation |
|---|---|---|
| SHA | k, b | Shift (A); shift count K=k + ($B^b$) (signs of k and $B^b$ extended); if bit 23 of K=1, shift right; complement of lower 6 bits equals shift magnitude; if bit 23 of K=0, shift left; lower 6 bits equal shift magnitude; left shifts end around; right shifts end off |
| SHAQ | k, b | Shift (A, Q) as one register; shift count K=k + ($B^b$) (signs of k and $B^b$ extended); if bit 23 of K=1, shift right and complement of lower 6 bits equals shift magnitude; if bit 23 of K = 0, shift left and lower 6 bits equal shift magnitude; left shifts end around; right shifts end off |
| SHQ | k, b | Shift (Q); shift count K=k + ($B^b$) (signs of k and $B^b$ extended); if bit 23 of K = 1, shift right, complement of lower 6 bits equals shift magnitude; if bit 23 of K = 0, shift left, lower 6 bits equal shift magnitude; left shifts end around; right shifts end off |
| SJ1 | m | If SELECT JUMP 1 is set, jump to m |
| SJ2 | m | If SELECT JUMP 2 is set, jump to m |
| SJ3 | m | If SELECT JUMP 3 is set, jump to m |
| SJ4 | m | If SELECT JUMP 4 is set, jump to m |
| SJ5 | m | If SELECT JUMP 5 is set, jump to m |
| SJ6 | m | If SELECT JUMP 6 is set, jump to m |
| SLS | | Program stops if selective stop switch is on; upon restarting RNI P+1 |

| Command Field | Operand Field | Operation |
|---|---|---|
| SQCH | r,1 | $(Q_{00-05}) \rightarrow (R)$ |
| SRA | | $0 \rightarrow (A)$; subcondition register $\rightarrow (A_{00-02})$ |
| SRCE, INT | c, r, s | Search for equality of character c in list beginning at r until an equal character is found, or until character at s is reached; $0 \leq c \leq 63_{10}$ |
| SRCN, INT | c, r, s | Inequality search; same as SRCE |
| SSA, I | m, b | Where (M) contains a 1 bit, set the corresponding bit in A to 1 |
| SSH | m | Test sign of (m), shift (m) left one place, end around and replace in storage; negative sign, RNI P+2, otherwise RNI P+1 |
| SSIM | x | Selectively set interrupt mask register for each 1 bit in x; corresponding bit in the mask register is set to 1 |
| STA, I | m, b | $(A) \rightarrow (M)$ |
| STAQ, I | m, b | $(A, Q) \rightarrow (M, M+1)$ |
| STI, I | m, b | $(B^b) \rightarrow (M_{00-14})$ |
| STQ, I | m, b | $(Q) \rightarrow (M)$ |
| SWA, I | m, b | $(A_{00-14}) \rightarrow (M_{00-14})$ |
| TAI | b | $(A_{00-14}) \rightarrow (B^b)$; becomes a no-operation instruction if b = 0 |
| TAM | v | $(A) \rightarrow (v)$ |
| TIA | b | $0 \rightarrow (A)$, $(B^b) \rightarrow (A_{00-14})$; if b = 0, $0 \rightarrow (A)$ |
| TIM | v, b | $(B^b) \rightarrow (v_{00-14})$ |
| TMA | v | $(v) \rightarrow (A)$ |

| Command Field | Operand Field | Operation |
|---|---|---|
| TMAV | | Initiate memory request; if reply occurs within 5 usec, RNI P+2; if not RNI P+1; storage address is $(B^b)$ with (operand state register) or zero appended |
| TMI | v, b | $(v_{00-14}) \longrightarrow B^b$ |
| TMQ | v | $(v) \longrightarrow (Q)$ |
| TQM | v | $(Q) \longrightarrow (v)$ |
| TST | r, $B_r$, $\ell_r$ | Test field R; -, 0, or + |
| UCS | | Unconditional stop |
| UJP, I | m | Unconditional jump to M |
| UPAK | m, $B_m$, $\ell_s$ $B_s$, $\ell_s$ | Unpack 4-bit BCD field M into 6-bit BCD field S |
| XOA | y | y V (A) $\longrightarrow$ (A) |
| XOA, S | y | y V (A) $\longrightarrow$ (A), sign of y is extended |
| XOI | y | No-operation |
| XOI | y, b | y V $(B^b) \longrightarrow (B^b)$ |
| XOQ | y | y V (Q) $\longrightarrow$ (Q) |
| XOQ, S | y | y V (Q) $\longrightarrow$ (Q), sign of y extended |
| ZADM | r, $B_r$, $\ell_r$, s, $B_s$, $\ell_s$ | Clear field S; field R $\longrightarrow$ field S, right justify |

CONTROL DATA

## COMMENT AND EVALUATION SHEET

3300/3500 META/MASTER

General Information Manual

Pub. No. 6023600 June 1968

THIS FORM IS NOT INTENDED TO BE USED AS AN ORDER BLANK. YOUR EVALUATION OF THIS MANUAL WILL BE WELCOMED BY CONTROL DATA CORPORATION. ANY ERRORS, SUGGESTED ADDITIONS OR DELETIONS, OR GENERAL COMMENTS MAY BE MADE BELOW. PLEASE INCLUDE PAGE NUMBER REFERENCE.

FROM NAME: _____

BUSINESS ADDRESS: _____

_____

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE STAPLE

FOLD FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

**BUSINESS REPLY MAIL**

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

**CONTROL DATA CORPORATION**
Software Documentation
4201 North Lexington Avenue
St. Paul, Minnesota 55112

ID248

FOLD FOLD

STAPLE STAPLE