XEROX

Reference Manual

Xerox Business Language

90 10 22C

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

Xerox Business Language

900 Series/9300 Computers

Reference Manual

90 10 22C

October 1967

Price: $2.75

REVISION

This publication, 90 10 22C, is a minor revision of the Xerox Business Language Reference Manual, 90 10 22B. Changes to the previous edition are indicated by a line at the right or left margin of the affected page.

RELATED PUBLICATIONS

Title	Publication No.
Xerox 910 Computer/Reference Manual	90 00 08
Xerox 920 Computer/Reference Manual	90 00 09
Xerox 925 Computer/Reference Manual	90 00 99
Xerox 930 Computer/Reference Manual	90 00 64
Xerox 9300 Computer/Reference Manual	90 00 50
Xerox MONARCH/Reference Manual	90 05 66
Xerox 9300 MONITOR/Reference Manual	90 05 13
Xerox Symbol and Meta-Symbol/Reference Manual	90 05 06
Xerox SORT/MERGE/Reference Manual	90 09 97
Xerox MANAGE/Reference Manual	90 10 46
Xerox Business Programming Systems	66 05 02

CONTENTS

APPENDIX

FIGURES

EXPLANATION OF TERMS

Array	Group of consecutive words.
Assembler	Computer program that prepares a machine language object program from a symbolic language program by substituting machine operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.
Binary-Coded Decimal (BCD)	Decimal number and alphabetic character representation in which each digit or character is represented by a coded combination of 6 binary digits.
Calling sequence	Standardized sequence of instructions appropriate to the calling of a particular subroutine; it usually sets up the input values required by the subroutine, makes provision for reentering the main program when the subroutine is finished, transfers control to it, and finally may specify some action concerning the output values of the subroutines.
Character string	Consecutive set of alphanumeric characters.
Computer word	24 binary digits (bits).
Covered quotient	Number of words required to hold a character string in memory, given by the formula $a+(b-1) \div b$ where a is the number of characters in the string and b is the number of characters in a word.
Decimal digit or alphabetic character	6 bits
Double-precision (double-length)	Number having twice as many digits as are ordinarily used in a given computer.
Field	Successive characters in an assigned area of a record (defined character string) that specify a particular item of information.
File	Sequential set of information units, not necessarily all the same size.
In-line code	Generation of one or more machine language instructions from the applicable source language code which specifies the function desired at that point.
Meta-assembler	A processor whose characteristics supplement those of a conventional assembler, enabling the user to program using a higher-level language than that of the machine itself. Inherent in a meta-assembler is the incorporation of a list structure in the syntax of the source language. The presence of a "Do" verb and a Boolean as well as arithmetic operators in expressions provides the ability to conditionally generate multiple machine-language instructions from a given source-language statement. A valuable feature of meta-assemblers is their ability to provide true program capability among related computers.
Monitor	Executive routine that controls the operation of a complex information-processing system involving one or more computers together with all of the associated software.
Object program	Output of an assemble, or compiler when it has translated the source program to either machine language or intermediate-level assembly language.
Octal digit	3 bits.
Octal word	8 octal digits.
Parameter	Constant or variable used in some calculation; definable characteristic of an item, device, or system; quantity in a subroutine whose value specifies, or partly specifies, the process to be performed.
Record	Group of related items of information organized internally into words, characters, or fields.
Run-time count	Count of number of operand items supplied to a business language instruction at run-time when count is unknown until program is executed.
Shift	Displacement of an ordered set of characters one or more places to the left or right.
Source program	Original program, usually written in a universal symbolic language.
Truncation flag	Flag which is set when value can not be held in available number of bits and truncation to the maximum number of allowable bits occurs.
Word	4 alphanumeric characters.

INTRODUCTION

A comprehensive business programming package, developed by Xerox Data Systems, extends the application of XDS computers to business data processing and management decision-making. This new "software" in conjunction with the extensive scientific library gives XDS users a problem-solving, data-handling capability through a wide range of scientific and business applications. The XDS Business Programming Package† consists of three programming systems: XDS Business Language, SORT/MERGE, and a management information-processing program called MANAGE.

Programming languages like XDS META-SYMBOL, FORTRAN, and ALGOL were not designed for the character manipulation necessary for business data processing. The XDS Business Language is designed specifically to permit XDS binary word computers to be programmed as though they were decimal, character-oriented machines.

XDS Business Language is a procedure-oriented extension of the XDS meta-assembler, META-SYMBOL. The Business Language is free-form, character-oriented, and analogous to the types of assemblers used with character-organized computers.

Availability of this Business Language for binary, fixed-word machines frees the programmer from the tedious tasks of mask selection and loading, extracting and merging, and extensive shift operations.

When called, procedures within the language either generate in-line code or calling sequences that interface with closed subroutines. Which alternative will be taken is a function of: (1) the type of procedure invoked; or (2) the distinct characteristics of the operand of a procedure call. In the latter case, the ability of META-SYMBOL to take conditional action at assembly time allows considerable latitude in generating actual machine code; and this, in turn, produces virtually optimum code. Conventional

†See XDS Business Programming Systems, 66-05-02B

macro-assemblers do not have this capability. Also, the assembler incorporates a unique technique for minimizing temporary storage that is global (common) to all generated in-line code and subroutines. XDS Business Language generates code approaching optimum storage efficiency. The generated code usually has better storage-utilization characteristics than code written by a competent programmer.

XDS META-SYMBOL with Business Language can be used on any XDS 900 Series Computer with 12,288 words of memory, or on an XDS 9300 Computer with 16,384 words of memory; on either system, the complement of peripheral equipment is the same as that required for META-SYMBOL. A typical configuration consists of three magnetic tape units, a card reader/punch, and a line printer.

This manual contains two main parts and an extensive appendix. The first part describes basic elements of the XDS META-SYMBOL Assembly Language; the second describes Business Language instructions. The appendix includes a general programming introduction, an elementary discussion of symbolic coding, the assembly listing format, calling sequence generation, operating information, programming examples, and instruction lists.

The manifold capabilities of the Business Language system will be achieved more readily if the user is also conversant with:

> The computer at his installation (see applicable computer reference manual†).
>
> XDS MONARCH† or XDS 9300 MONITOR† (whichever is applicable to his system).
>
> XDS META-SYMBOL† and its compatible subset, SYMBOL†.

In the body of this manual, the assumption has been made that the reader is familiar with symbolic programming.

†See Related Publications page in this manual.

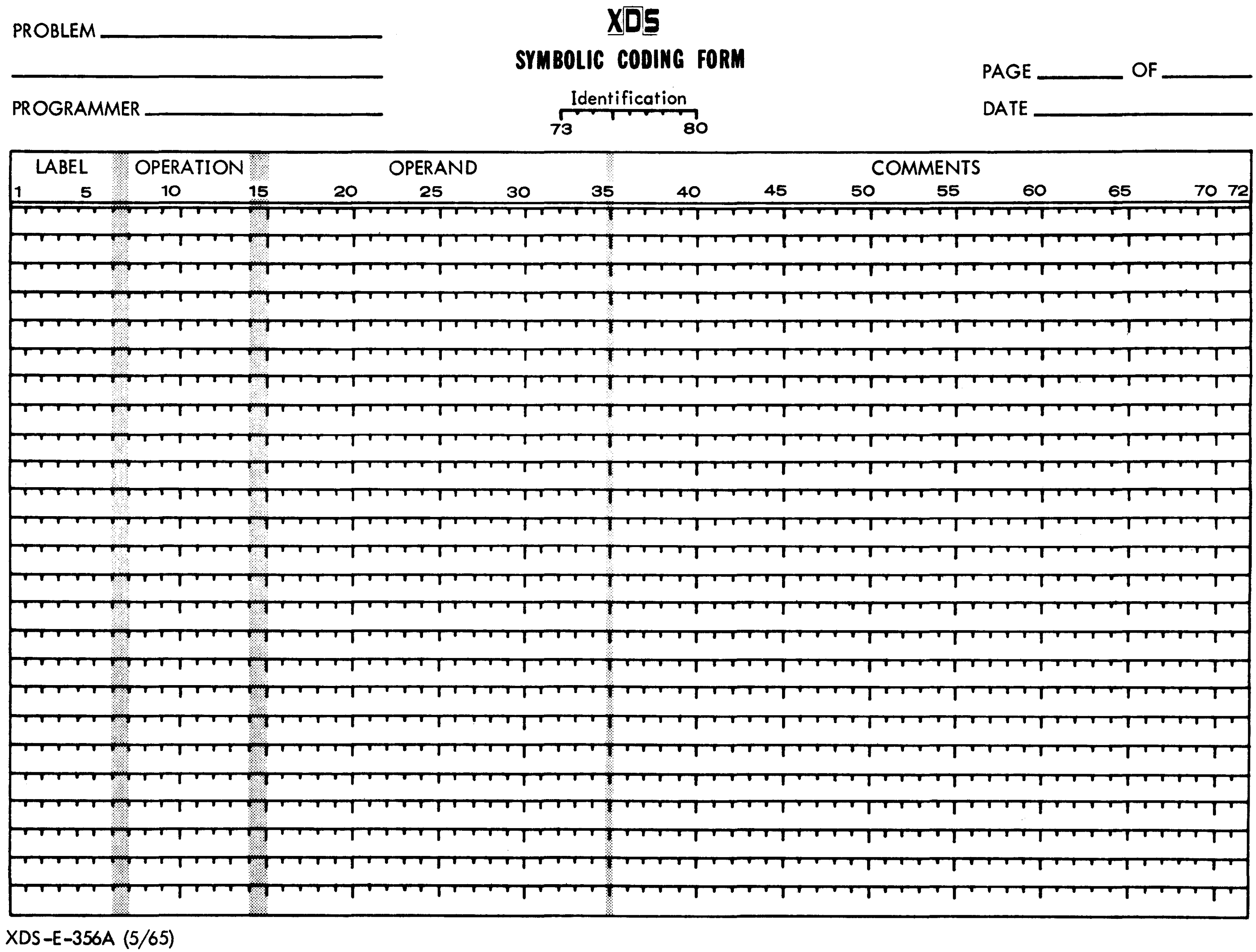

Figure 1. Symbolic Coding Form

2. BASIC ELEMENTS OF THE META-SYMBOL ASSEMBLER

SYNTAX

The syntax of any language is the set of rules governing its sentence (or statement) structure. To use a language, such as an assembly language, one must know its syntax. META-SYMBOL syntax is simpler than that of most assemblers and is more powerful; there are fewer definitions and rules to learn, because each one is more comprehensive. However, to use this language and assembler efficiently, its basic principles must be clearly understood. This section of the manual explains how to use META-SYMBOL to produce efficient business programs.

CHARACTER SET

The "words" used in META-SYMBOL to give instructions to the computer are made up of letters, numbers, and symbols. These characters are the familiar ones of everyday English, but the user will put them together in new and unfamiliar ways while learning this new language. Characters are classified by type as follows:

Alphabetic character: one of the characters A-Z.

Numeric character: one of the characters 0-9.

Alphanumeric character: any character that is either alphabetic or numeric.

Special character: a nonalphanumeric character (e.g., *, $, +).

PROGRAM

META-SYMBOL programs (that is, those written in the META-SYMBOL language) consist of a number of lines of symbolic coding. The coding is symbolic in the respect that each line is only a symbolic representation of the actual numeric instructions that a computer can act on directly. After the program has been written and put onto an input medium such as cards or magnetic tape, the assembler reads the symbolic program and assembles it into a machine language program suitable for being loaded into the computer and operated with no further alteration. (See Figure 2, a sample assembly listing at the end of this section.)

The coding of the program is done on a Symbolic Coding Sheet, a sample of which is shown in Figure 1.

SYMBOLIC LINE

Each line of symbolic code in the original program constitutes a unit record that the assembler processes during an assembly. Usually, each line is punched into one card and the cards are combined to form a symbolic deck for computer input during assembly. META-SYMBOL conveniently allows continuation of a symbolic line onto two or more cards when necessary. A symbolic line is more precisely called a logical record, inasmuch as it may require more than one physical record (two or more cards, for example) to contain it.

A symbolic line consists of four fields; the first three, label, operation, and operand field, are essential elements of assembler instructions and directives; the fourth is a comments field. With the exception of a line consisting entirely of comments, a line must always have something specified in the operation field; the presence of information in the other fields is at the programmer's option.

LABEL FIELD

A label field labels an operation or a value so that it can be symbolically referred to elsewhere. Labeling is accomplished by writing a symbol (defined according to META-SYMBOL rules) in the label field; that is, if the user needs to refer to any instruction or piece of data, he writes a name for it in the referencing operand field and also writes the same name in the label field of the data or instruction to which a reference has been made.

OPERATION FIELD

An operation field contains a mnemonic instruction, a business language instruction, or an assembler directive. A mnemonic instruction produces a single line of object code (i.e., a single operation) merged with information from the operand field. A business language instruction causes the assembler to inspect the parameters in the operand field and conditionally produce zero or more lines of object code. A directive is a pseudo-instruction to the assembler to perform some action at assembly time; it may or may not produce any object code.

OPERAND FIELD

The reference in the operand field to a named, symbolic line or other element of data need not be a simple name. META-SYMBOL allows flexibility in writing compound names. An operand field may contain one or more expressions. Definitions of expressions and expression elements follow.

EXPRESSIONS

An expression is a series of items connected by operators. The processor evaluates expressions by successively combining items, in the manner specified by the connecting operator, and in the order of decreasing operator hierarchy.

ITEMS

An item may be one of the following:

Item	Definition	Example
Symbol	A symbol is a string of alphanumeric characters	ALPHA B1 X1Y
Octal integer	An octal integer is a signed or unsigned string of from 1 to 15 octal digits preceded by a zero.	012 01234567 077777777 -031

Item	Definition	Example
Decimal integer	A decimal integer is a signed or unsigned string of from 1 to 15 decimal digits; the first digit is not zero.	12 1234567 -42
Decimal number	A decimal number is either a decimal integer or string of decimal digits and one or more of the following: decimal point, decimal scale operator, binary scale operator. When an item has a decimal point but has no binary scale operator, the item is of the floating-point mode.	12 0.12 +12.0*+4 (-12.5)*+(-2)*/3
Character data string	A character data string is a string of alphabetic, numeric, and/or special characters enclosed in single quotes.	'B1' 'X1Y' '012'
Current location symbol	The current location symbol represents the current value of the location counter at program execution time.	$
Subexpression	A subexpression is an expression enclosed in parentheses and occurring as part of another expression.[†]	

OPERATORS

An operator may be one of the following:

Operator	Representation	Hierarchy
Conditional		
less than	<	1
equals	=	1
greater than	>	1
Boolean		
sum (OR)	++	2
difference (exclusive OR)	--	2
product (AND)	**	3
Arithmetic		
sum	+	4
difference	-	4
product	*	5
truncated quotient	/	5
covered quotient	//	5
decimal scale	*+	6
binary scale	*/	6

Under hierarchy there are 6 levels, a highest level of 1 and a lowest level of 6; the lowest level is evaluated first.

[†]Examples of the use of parentheses in business instructions are given later.

The covered quotient operator, //, is defined as: a//b = (a+b-1)/b; it is useful in determining the number of memory cells needed to store a characters in a b characters-per-word mode of storage.

The decimal and binary scale operators, *+ and */, respectively, can be used between any two expressions. Where x and y are two expressions:

x *+ y is equivalent to $(x)\cdot(10^y)$
x */ y is equivalent to $(x)\cdot(2^y)$

Note that the nominal binary point of x is to the right of the least significant bit; that is, these operations use integer, not fractional notation.

Actually, */ functions as a logical shift operator, so that ± x */(-y) performs a logical right shift y places. Hence, because of operator precedence, */ functions as an arithmetic operator for ± x */y but not for (-x) */y.

COMMENTS FIELD

A line's comments field may contain comments to annotate the program. The assembler ignores comments, but outputs them on listing.

COMMENTS LINE

Label	Operation	Operand
*THIS IS A COMMENTS LINE		

When an asterisk introduces a symbolic line (i.e., * in column 1), the assembler ignores its contents. Such lines are used to annotate the source program. They appear on the source program output listing produced by the assembler. The comments line may contain a maximum of 64 characters, beginning with the first nonblank following the asterisk. Additional characters are discarded during assembly.

FREE FORM

The assembler provides for free-form symbolic lines; that is, each field need not begin at a prescribed column of the source input record (usually a coding sheet – see Figure 1). Rules for writing such a record are:

The label field begins in column one.

One or more blanks written at the beginning of a line specify no label is desired (i.e., there will be no label field with such a symbolic line).

A blank terminates any field.

Eight or more blanks written following a symbol in either the label or operation field specify that the next field is absent.

When the input record contains 80 columns (i.e., a card), the assembler ignores columns 73-80 and terminates the physical record at column 72.

All lines in Example 1, below, are valid; lines 1, 2, and 4 produce the same result.

Note: These examples are in assembler instruction format.

The format definition given below describes typical machine instructions that may be used along with business language instructions. Detailed information concerning machine instructions for various XDS computers is to be found in their applicable computer reference manuals. For convenience, complete listings of XDS machine instructions are given in the appendixes of this manual.

Label	Operation	Operand
[LABEL]	LDA	[*] E1 [, E2]

In the above example, those items enclosed within brackets are optional in the instruction format. All instructions must have an operation mnemonic and most of them have an operand. Indexing (indicated by E2) and indirect addressing (indicated by *) are optional. The label and comments fields need not be present.

As indicated, a line's operand field may consist of a sequence of expressions. Expressions are represented by the symbols E, E1, E2, . . . throughout the assembler portion of this manual. Additional examples, showing how various operand fields are written, are given with the individual instructions.

SAMPLE ASSEMBLY LISTING

An output listing of a representative (although not typical) program is shown on the following page. A complete description of the output fields on this listing is given in the Appendix.

Example 1.

```
ALPHA   LDA      TEMP                        COMMENT
ALPHA LDA TEMP COMMENT
 STA TEMP COMMENT
ALPHA        LDA         TEMP COMMENT
```

```
                                  1 * READ A CARD: TEST COL 18 FOR 2,4,6,M : IF YES- WRITE ON TP2 BLCK 2
                                  2 *                                                 MOVE COL 18-33 TO TABLE
                                  3 *                                            NO - TYPE OUT CARD
                                  4         EXTEND
                                  5 *
                                  6 OUTAREA DEFAREA  20              DEFINE OUTPUT AREA
   00024                          7 INAREA RES      20               DEFINE INPUT AREA
   00050  00000000                8 FLAG   DATA     0
                                  9 *
*  00051  0 43 0 00000           10 START  REWIND   2
*  00053  0 43 0 00000           11 RDCD   READCD   INAREA
*  00055  0 60 0 00000           12        BLCD     WRAPUP
*  00057  0 43 0 00000           13        BRACNE   TYPCARD,INAREA,18,'2','4','6','M'
*  00065  0 43 0 00000           14        MOVE     INAREA,18,*TABLE,1,16
*
   00072  0 76 0 00146           15        LDA      =4
   00073  0 63 0 00144           16        ADM      TABLE               TRACK NEXT AVAILABLE TABLE ADDR
   00074  0 61 0 00143           17        MIN      SRTCNT              TRACK NO OF ITEMS IN TABLE
   00075  0 60 0 00050           18        SKR      FLAG                TEST BLOCKING FLAG
   00076  0 01 0 00106           19        BRU      SECND               SET UP LOGICAL RCD 2
   00077  0 71 0 00147           20        MOVEWD   INAREA,OUTAREA,20   MOVE 20 WORDS TO LOGICAL RCD 1
   00103  0 61 0 00050           21        MIN      FLAG                FLIP BLOCKING FLAG
   00104  0 61 0 00050           22        MIN      FLAG
   00105  0 01 0 00053           23        BRU      RDCD
*  00106  0 43 0 00000           24 SECND  WRITETP  2,OUTAREA,160       WRITE 2 CARD IMAGES ON TAPE
   00111  0 01 0 00053           25        BRU      RDCD
*  00112  0 43 0 00000           26 TYPCARD  TYPE   INAREA              COL 18 NOT 2,4,6,M
   00114  0 01 0 00053           27          BRU    RDCD
   00115  0 76 0 00150           28 FILLER CLEAR    OUTAREA+20,20       PAD LAST BLOCK
*  00121  0 43 0 00106           29        WRITETP  2,OUTAREA,160       WRITE LAST BLOCK
   00124  0 60 0 00050           30 WRAPUP SKR      FLAG
   00125  0 01 0 00115           31        BRU      FILLER
*  00126  0 43 0 00000           32 ENDMARK WTMARK  2                   WRITE TAPE MARK
*  00130  0 43 0 00051           33        REWIND   2
   00132  0 76 0 00151           34        SORT     TABLE+1,(SRTCNT),4,2,3      SORT 4 WRD ITM ON WRDS 2-4
*
*  00142  0 43 0 00000           35        BRM      M\EXIT
   00143  00000000               36 SRTCNT DATA     0
   00144  00000145               37 TABLE  DATA     $+1
   00145  00 00 00 00            38        DA       2047,0
          00000051               39        END      START
   04144  00000004
   04145  00177754
   04146  00000000
   04147  77777777
   00126                            B\I8
   00130                            B\I9
   00053                            B\I3
   00112                            B\I6
   00055                            B\I23
   00057                            B\S2
   00065                            B\S18
   00071                            B\S12
   00121                            B\I11
   00133                            B\DIR
   00134                            B\SORT
   00142                            M\EXIT
```

Figure 2. Sample Assembly Listing

3. DIRECTIVES

INTRODUCTION

An assembler operates on input data (a source program) to produce output data (machine language object program). Its difference from other programs is that the output data from an assembler generally constitute another program which, when loaded and executed, operates on input data to produce output data. There are two times when the resultant program can be affected logically: at assembly time and at execution time. In the latter case, this is accomplished by input parameters to the program, and in the former case, by input parameters (called directives) to the assembler. Thus, directives are operative at assembly time, whereas instructions are operative at program execution time. The following directives are included in the assembly language:

Assembler Instruction	Data Generation
AORG	DATA
RORG	DED
RES	TEXT
END	Value Declaration
PAGE	EQU

When a user writes a program, he frequently needs to refer, in the operand line, to some name (label) that is or will be defined at a subsequent place or symbolic line. This is called a forward reference; it is not allowed in an assembler directive. However, the forward reference is allowed with all mnemonic machine instructions and business instructions except FIELD. Although FIELD is defined in the Business Language section, a comment on FIELD forward references is in order. The name in the operand field of a FIELD instruction must not be a forward reference, and no field name can appear in any business instruction as a forward reference.

The following examples, showing the functions of directives, include machine instructions. These instructions are for illustrative purposes only; understanding the examples does not depend on knowing the instructions of any particular machine.

AORG AND RORG

Absolute Origin and Relative Origin

Label	Operation	Operand	Comments
[LABEL]	AORG or RORG	E	[PROGRAM ORIGIN]

The origin (E) of a program is the lowest-numbered memory address occupied by (instructions or data of) the program. In other words, it is the nominal beginning of the program.

Generally, it is useful to allow the origin to be relocatable at execution time, so that the program can be executed equally well whether loaded at one location or another.

Relocation of a program to another area of memory is performed automatically at execution time by the loader through actions taken by the assembler. The assembler accomplishes this by producing relocation information together with the binary object program at assembly time. Using this information, the loader (part of the monitor in the XDS MONARCH or MONITOR 9300 systems) performs the relocation when the binary object program is loaded for execution.

In some cases, however, the programmer may desire to control the program origin. For example, all or part of his program might have to occupy fixed memory locations; or in the case of program debugging, it might be easier to relate the contents of memory to an assembly listing if all addresses have absolute values.

The user controls the relocatability of his program through the AORG and RORG directives, for absolute origin and relocatable origin, respectively. See Example 2.

In this example, all addresses except that of I1 are relocatable. Thus, the BRM I1INT is always loaded into location 030, but the contents of its address field, as a relocatable quantity (I1INT), is assigned at loading time. The subroutine I1INT, on the other hand, is completely relocatable, since the loader can override the otherwise automatic loading into location 0200.

Example 2:

```
I1      AORG    030           AT LOCATION 30(OCTAL)
        BRM     I1INT         PLACE LINKAGE TO I1INT
        RORG    0200          I1INT IS TO BE RELOCATABLE
I1INT   HLT
        .
        .
        .
        BRR     I1INT
```

Viewed otherwise, AORG and RORG have the function of resetting the location counter[†]; the symbol I1 has the same value (030) whether it appears on the AORG line or on the following line.

The operand can be a completely general expression and is not restricted to numeric values. Its value must, however, be defined within the program previous to its use in the operand.

RES

Reserve

Label	Operation	Operand	Comments
[LABEL]	RES	E	[RESERVE A BLOCK]

RES is primarily used to reserve and (optionally) label storage areas. See Example 3.

END

End of Program

Label	Operation	Operand	Comments
	END	[E]	[END OF PROGRAM]

END indicates the end of the program to the assembler.

When the END line terminates a program, the operand field may (but need not) be used to specify to the loader the location to which it will transfer after loading the program.

PAGE

Eject Page

Label	Operation	Operand	Comments
	PAGE	[E]	[EJECT PAGE]

When PAGE is encountered, the assembler causes a page-eject to occur on the output listing medium during assembly. The PAGE line is the first line on the new page. Pages may be numbered in the operand field.

[†] The "location counter" is a special memory cell used by the assembler in defining labels at assembly time.

DATA

Generate Data Block

Label	Operation	Operand	Comments
[LABEL]	DATA	E1,[E2, . . . , EN]	[GENERATE DATA BLOCK]

Data permits convenient representation of single-precision data within the symbolic program. Since operands may be general expressions, octal, decimal, binary-coded decimal and symbolic data may all be generated with a single directive. In all cases, the translated expression is right-justified within the computer word; except for negative data, unfilled bit positions always contain zeros.

In conventional assembly programs, interpretation of the contents of the operand field depends on the contents of the operation field. This restriction does not apply to META-SYMBOL programs, where data unambiguously describe their own item type. See Example 4.

A DATA statement can have a maximum of 72 bytes per statement. A byte is:

- A symbol.
- An octal or decimal integer.
- A scaled decimal integer.
- A character data string.
- Two or more of the four above connected by double operators (++, --, **, //, *+).
- A single arithmetic operator (+, -, *, /).
- A comma (,).
- A blank field (contiguous series of blanks of any length).

A DATA statement can be continued by terminating the current line with one of the separating commas and starting (in any column) the next physical line with the continuing data items; the total number of all columns in the continued lines cannot exceed 56. See Example 5.

Each continuation (trailing comma in a line) causes a blank byte to be generated at the end of the line. The only effect of this is that the maximum number of bytes that may be written in the DATA statement is reduced by 1 for each continuation. The number of bytes shown in Example 5 is 19, Note that a string of alphanumeric information within quotes, as designated by the last item in Example 5, should not exceed four characters (i.e., 24 bits).

Example 3:

```
TABLE   RES    10                          RESERVE 10 LOCATIONS
        RES    PTAB-RTAB                   AREA BETWEEN LABELS PTAB
*                                            AND RTAB
```

Example 4:

Location	Contents	Label	Operation	Operand	Comments
01000		TENS	AORG	01000	
01000	00000010		DATA	010	OCTAL 10
01001	00000012		DATA	10	DECIMAL 10
01002	00000100		DATA	'10'	BCD 10
01003	00001000		DATA	TENS	CURRENT VALUE OF LABEL TENS
01004	02101012		DATA	(TENS + 10)++('A'*0100000)	
01005	00000011		DATA	011, 11, 012121252, 'TENS'	
01006	00000013				
01007	12121252				
01010	63254562				

Example 5:

```
LABEL   OPERATION        OPERAND                                    COMMENTS
LABEL   DATA       24,25,65,57,
 987,90,2,
                  78,'ER'
```

DED

Decimal Double Precision

Label	Operation	Operand	Comments
[LABEL] *	DED	E1 [, E2..., EN]	[GENERATE DP DECIMAL DATA]

DED enables the programmer to represent double-precision decimal data conveniently within a symbolic program. The resultant data will be generated in standard XDS double-precision fixed- or floating-point format according to the mode of the expression(s) in the operand field. In the case of DED, only decimal numbers constitute legitimate expressions. See Example 6.

Because numeric quantities are restricted to 15 digits in length, the use of scientific or floating-point notation is preferable to absolute notation (e.g., 0.0000147235821). When both a binary and a decimal scale factor are desired, the decimal scale factor should be specified first.

TEXT OR BCD

Binary-Coded Character String

Label	Operation	Operand
[LABEL]	TEXT or BCD	E, character string
[LABEL]	TEXT or BCD	<character string>

The programmer often needs to incorporate output messages in binary-coded decimal form within a program. This can be accomplished by subdividing the message into 4-character (24-bit) strings and placing them in the operand field of a DATA directive line. Normally, however, the TEXT statement is used for all textual and heading-type information and messages.

Using the TEXT directive, the programmer places the character string (not enclosed in quotes) in the operand field and specifies the total message length in one of two ways:

1. Precede the character string by a character count, separating the string and the count by a comma. The count can be indicated by writing a number or any general expression, provided that it has previously been defined within the program.

2. Enclose the character string by the characters < and >. (This is the more convenient way, since the character count need not be known in short messages.

Two important rules must be noted:

1. The number of characters that can be written with one TEXT statement is limited to the number of characters

Example 6:

```
LABEL   OPERATION        OPERAND                                    COMMENTS
PI         DED         3.14159265 35               FLOATING
AVO        DED         6.023*+23                   FLOATING
E          DED         2.7182828*/45               FIXED
LIGHT      DED         1.86*+5*/23                 FIXED
```

that can be written in the operand line on which the TEXT statement is written; there can be no continuation.

2. The message is left-justified within the block of computer words allocated to it. Unfilled character positions always contain blanks (060).

The BCD directive is identical to TEXT, except that in the computer words generated blanks are represented by 012 in BCD and 060 in TEXT. This distinction may be ignored when using business input/output instructions.

A usual programming need is to generate 132-character line printer headings that consist of alphanumeric information. The simplest way to generate such a heading is to write three successive TEXT statements whose total character count is 132. Only the first need be labeled to label the entire heading. It is important that each of the first two lines consists of a multiple of four characters; otherwise, undesirable blanks will be intermixed in the message (rule 2 above). See Example 7.

Example 7.

Location	Contents	Line	Label	Operation	Operand
01000		1	MSG	RORG	01000
01000	22232460	2		TEXT	8, BCD INFO
01001	31452646				
01002	22232460	3		TEXT	<BCD INFO>
01003	31452646				
01004	00222324	4		DATA	'BCD', 'INFO'
01005	31452646				
01006	22232460	5		TEXT	4, BCD
01007	60222324	6		TEXT	4, BCD
01010	00222324	7		DATA	'BCD'
01011	22232412	8		BCD	'BCD'

Example 8:

```
XR          EQU        025
ONE         EQU        1
TWO         EQU        ONE+XR                  TWO HAS VALUE 1+025=026
```

Note that lines 2 and 3 in Example 7 result in identical code, whereas lines 5, 6, 7, and 8 do not.

EQU

Equals

Label	Operation	Operand	Comments
LABEL	EQU	E	LABEL COMPULSORY

Since the DATA and TEXT enable the programmer to centralize and label execution-time data specifications, they contribute to both the readability and flexibility of the symbolic program. For the same reasons, it is frequently desirable to specify assembly-time data symbolically, or to use "parametric programming," a technique that is useful whenever a number of symbolic lines are related to one another by their common dependence upon one or more values. Using the parametric approach, the programmer labels the value(s) by an EQU directive and replaces all references to the appropriate value(s) by its (their) symbolic equivalent(s). See Example 8.

Another example of EQU use is equating the index register to a label, like X2, for mnemonic identification:

```
X2     EQU    2
```

Assembler instructions with indexing indicated can, therefore, be written:

```
LABEL  LDA    WORK,X2
```

For the XDS 9300 with three index registers, X1, X2, and X3 can be used.

4. ADDITIONAL PROGRAMMING FEATURES

LITERALS

Label	Operation	Operand
LABEL	OP	= E

Typically, computer instructions operate on variables and constants. When an instruction operates on a variable, it must know the location of the variable because the variable value is not known until the instant of obtaining it. Thus, a variable's location is "important" to the instruction using it. This is not true of constants; since the value of a constant does not change, its value is essential, not its location.

Symbolic programming facilitates the representation of both types of values. Variable operands can be given symbolic names (such as, X, ALPHA, INPUT) and can be referred to by these names throughout the symbolic program. For operating with constants, however, it is generally desirable to refer to the constant by value rather than by name; literals provide this capability.

To use literals, the programmer writes the value of the expression, rather than a name, in the operand field of the symbolic line, and precedes the expression by an equal sign (=). Detecting the leading equal sign, the assembler computes, as usual, the value of the expression that follows, but it then stores this value in a literal table that it constructs following the program. The address portion of the generated instruction is then made to refer to the literal table entry rather than to contain the value of the computed expression. See Example 9. As shown in this example, the processor detects the duplicate equal values (1*8 is equivalent to 010) and enters them once into the literal table.

Example 9:

Location	Contents	Label	Operation	Operand
00144			RORG	100
00144	07600151	TENS	LDA	=010
00145	07600152		LDA	=10
00146	07600153		LDA	='10'
00147	07600154		LDA	=TENS
00150	07600151		LDA	=1*8
			END	
00151	00000010			
00152	00000012			
00153	00000100			
00154	00000144			

RELOCATION

It is usually desirable to assemble a symbolic program without allocating the program to a particular memory area or starting location. When a program can be executed independently of its origin, that is, independently of where it is physically located within the computer, it is called a relocatable program.

All instructions are relocatable unless they have been affected by an AORG directive. All decimal and octal numbers are nonrelocatable (since adding the loading origin to the address portion of an instruction would change a number). Assuming the absence of an AORG directive, all symbols are relocatable that are not equated to a nonrelocatable expression by an EQU directive. As a symbol, $ is always relocatable.

The assembler assigns a code number of 1 to each relocatable item and assigns a code number of 0 to each nonrelocatable item. When an expression consists of at least one relocatable item, the expression is (see Example 10):

> Relocatable if R (the algebraic sum of the relocatable codes) is equal to 1.
>
> Nonrelocatable if R is equal to zero.
>
> Illegal if $0 \neq R \neq 1$ or if the expression involves any operations other than addition and subtraction upon two relocatable items.

The assembler provides relocation information in the text section of the binary output. Detecting a relocation flag for any instruction, the loader adds a bias (the loading origin) to the address portion of the instruction. See the META-SYMBOL Reference Manual for binary card format details.

Example 10:

```
R1      DATA    0
R2      DATA    0
NON     EQU     1
A       EQU     R1+R2                      ILLEGAL
B       EQU     R1-R2                      NON-RELOCATABLE
C       EQU     R1+NON                     RELOCATABLE
D       EQU     R1*NON                     ILLEGAL
E       EQU     R1*R1                      ILLEGAL
        END
```

5. THE BUSINESS LANGUAGE

This section describes the set of "higher-level" instructions that make up the XDS Business Language proper. These instructions provide the business data-processing user with the character-and-word manipulative capability that is a requirement of most business data processing applications. XDS Business Language programs are processed by the XDS META-SYMBOL Assembler which operates under control of MONARCH (for XDS 900 Series Computers) or under MONITOR (for XDS 9300 Computers).

INDEXING AND INDIRECT ADDRESSING IN THE BUSINESS LANGUAGE

In the following description of the elements (parameters) specified in the operand field of business instructions, the main operand (such as the location of a move area) is referred to as Ei. Unless otherwise specified, an element from this class of operands (E1, E2, ...) specifies a location and can be indexed and indirectly addressed. Specification of indexing is different for the XDS 900 Series machines and the XDS 9300 Computer.

Any entry in the operand field of a business instruction can be a general META-SYMBOL expression except:

1. Labels that are indexed, and
2. Those operands expressly prohibited in the instruction description.

XDS 900 SERIES COMPUTERS

Ei specifies a simple address.

```
        MOVEWD     TABL1, TABL2, 3
```

The statement above moves the contents of the three words at TABL1 into the three words at TABL2.

*Ei specifies indirect addressing.

```
        MOVEWD     *VECTOR, TABL2, 3
        .
        .
        .
VECTOR  PZE        TABL1†
```

The above series of instructions perform the same three-word move as the first example. The statement "VECTOR PZE TABL1" is one way to place the numerical address of TABL1 into location VECTOR.

(Ei) specifies indexing, that is, parentheses denote indexing.

Assume that the index register contains a 4.

```
        MOVEWD     (TABLO), TABL2, 3
        .
        .
        .
TABLO   PZE
TABLA   PZE
TABLB   PZE
TABLC   PZE
TABL1   PZE
        .
        .
        .
```

This series of instructions also performs the same three-word move, beginning in location TABL1, that the preceding examples accomplished.

(*Ei) specifies both indirect addressing and indexing.

XDS 9300 COMPUTER

Indirect addressing for the XDS 9300 is specified the same way as for XDS 900 Series Computers, that is, *E1. However, the XDS 9300 has three index registers; (E1, 1), (E1,2) or (E1, 3) specifies the particular index register. Any label equated via EQU to 1, 2, or 3, may be used in place of the index register number.

Assume the same conditions as in the previous examples, with index register number 2 containing 4; the same move is written:

```
        MOVEWD     (TABLO, 2), TABL2, 3
```

or where

```
X2      EQU        2
```

equivalently,

```
        MOVEWD     (TABLO, X2), TABL2, 3
```

Notes: In META-SYMBOL, the program counter is identified by the label $. Contrary to META-SYMBOL usage, this label is not allowed in the operand field of a business instruction.

As a general rule, the contents of the A and B registers are volatile for any call of a business instruction.

Labels (symbols) that begin with "REG" are reserved for system use.

For certain classes of instructions, parentheses enclosing an operand signify other things than indexing. In each case the exception is noted in the description of the instruction.

†PZE means halt.

BUSINESS INSTRUCTIONS

AREA DEFINITIONS

FIELD

Define Data Field

Defines a character string (data field) in memory relative to a defined and labeled memory area.

In a business instruction, the name (label) of a field carries to the assembler all of the information needed to define the first location of the related area, the position of the first character in the defined field, and the number of characters in the field.

Label	Operation	Operand
L1	FIELD	E1, HC1, CC

L1 = Name of the field; it may not be blank.

E1 = First location of the defined memory area; it may be indexed and/or indirectly addressed.

HC1 = High count, the number of the first character in the field, counting to the right from the left-most character in E1.

CC = Character count, the number of successive characters in the defined field.

Names are assigned to memory areas via FIELD. When used in other business instructions, the assembler automatically translates the field name into the starting reference location (E1), the starting high count character position (HC1), and the field length (CC). Using field names saves labor when defining considerable data manipulation in memory. FIELD generates no execution-time code.

Via the RES directive, reserve the following:

```
JONO        RES         50
```

Using FIELD, define a field named JONO10 that is 10 characters long and begins at the 103rd character of JONO.

```
JONO10      FIELD       JONO, 103, 10
```

Note: Field-defined labels can be used only with the following instructions:

MOVE
MOVEIZ
MOVEED
COMPARE
CLEARCH
FILLCH
BLANKCH

DEFAREA (DA)[†]

Define and Reserve Area of Memory

Defines and reserves (similar to RES) an area of memory containing a specified number of words. If a number or character is present following the word count, the words in the indicated memory area are initialized to this specified character (four characters per word) at object program load time.

Label	Operation	Operand
L1	DEFAREA	N
L1	DEFAREA	N, 'CH'

L1 = Any label or blank.

N = Number of words in the defined area.

'CH' = Number or character to which the indicated memory area is to be initialized. If an alphanumeric character is written, it must be enclosed in single quotes (e.g., 'A'). A number need not be enclosed in quotes. If neither a character nor a number is specified, the DA instruction functions like a RES. If the entry is a number, it must be less than or equal to octal 77. Any character, special or alphanumeric, can be written within the quotes.

An "S" flag is generated on the instruction line during assembly, if the number of arguments in the operand is greater than two or less than one. An "E" flag is generated if there is more than one argument and the first argument (N) is greater than 2047_{10}.

DATA TRANSMISSION INSTRUCTIONS

MOVEWD (MVW)[†]

Move Word String

Moves a word or consecutive sequence of words from one memory area to another.

Label	Operation	Operand
L1	MOVEWD	E1, E2, N
L1	MOVEWD	E1, E2, N, X

L1 = Any label or blank.

E1 = Source, the first location of the source memory area; E1 can be indexed, indirectly addressed, or both.

[†] In this manual, alternative and equivalent instruction mnemonics are written in parentheses; they are for the user desiring coding brevity.

E2 = Destination, the first location of the destination memory area; E2 can be indexed, indirectly addressed, or both.

N = Count, the number of words to be moved. If written (N) where N is a user-reserved label, the contents of N at execution time are taken as the word count.

X = User-defined label (or number) whose presence instructs MOVEWD to preserve the contents of the index register(s).

MOVEWD generates in-line code that varies from two to six words; or it generates a four-word calling sequence if an operand is indirectly addressed or indexed. The calling sequence is also generated if a run-time count is requested. If the word count as an execution-time parameter is negative or zero, one word is moved. If the word count as an assembly-time parameter is zero, no word is moved. If the word count is blank, one word is moved. If the word count as an assembly-time parameter is negative, a syntax error is recorded at assembly-time.

MOVEWD disturbs the A and B registers.

MOVE

Move Character String (Field)

Moves a string of characters in consecutive positions from one memory area to another. MOVE need not begin or end the move operation on a word boundary and can move characters between fields, consecutive word sequences, or both.

Label	Operation	Operand
L1	MOVE	E1, HC1, E2, HC2, CC
L1	MOVE	E1, HC1, F2
L1	MOVE	E1, HC1, CC, F2
L1	MOVE	F1, E2, HC2
L1	MOVE	F1, E2, HC2, CC
L1	MOVE	F1, F2

L1 = Any label or blank.

E1 = Source, the first location of the source memory area; E1 can be indexed, indirectly addressed, or both.

HC1 = High count of the source; the number of the first character to be moved, determined by counting from the left in location E1 to, and including, that first character.

E2 = Destination, the first location of the destination memory area; E2 can be indexed and indirectly addressed.

HC2 = High count of the destination; the number for the first character in the destination area.

CC = Character count, the number of characters to be moved.

F1 = Source field, the field F1, defined in the user's program as the source memory area. No indexing or indirect addressing is allowed.

F2 = Destination field, like F1.

MOVE moves a character string defined by a source location, high character position count, and length (number of characters) or by a field definition, into a second memory area that is also defined by a field definition or source location, high count, and length. If a conflict arises between the length of the source area, destination area, and character count, the instruction will move the number of characters equivalent to the minimum value. The string in the destination area may be truncated on the right in such cases, since the move is performed from left to right. If a field length of 10 and a character count of 8 occur, for example, the move is 8 characters only from the left end of the string. MOVE locates the word containing the first character in the memory area by computing Ei + (HCi ÷ 4), where E1 is the first location and HCi is the high count.

When written for indirect addressing, MOVE computes the *Ei and then adds to it the HCi ÷ 4 to find the first word. The high character position count must be no more than 1027; the length must be no more than 256. Exceeding these limits generates a T (trancation) flag during assembly. The calling sequence generated is 5 words in length.

MOVE does not disturb the index register(s).

MOVEIZ

Move Character String with Zero Fill

Performs the same operations as the MOVE instruction, except that all leading blanks in the source area are converted to zeros in the destination area. Once a non-blank character is encountered, MOVEIZ functions identically to MOVE.

MOVEED (EDIT, MCE)

Move and Edit Character String

Performs a MOVE operation on a string of BCD digits, automatically suppressing leading zeros in the destination area and optionally punctuating the string with $, commas, decimal point, and CR or - (minus) for credit. These items must be placed in the operand list in the order given here.

The format is the same as the MOVE directive, with the options added following a comma and enclosed within parentheses.

Label	Operation	Operand
L1	MOVEED	E1, HC1, CC, F2, ('$', 'C', P, 'CR')
L1	MOVEED	F1, F2, (0, 'C', , 'CR')

'$' = Float a leading dollar sign.

'C' = Intersperse a comma at intervals of 3 digits.

P = Integer; place a decimal point P digits to the left of the right-most digit.

'CR' = Place trailing CR symbol in destination area if string of digits being moved is negative; or

'-' = Place trailing - symbol in the destination area if string of digits being moved is negative.

Any, all, or none of the actions can be written in one statement. However, when any option is unused, its place must be marked by a zero or by a comma alone. An option list specifying only the minus would be: (0, 0, 0, '-'). An option list for the $ and the minus for credit would be ('$', 0, 0, '-') or equivalently ('$', , , '-').

Conflicts in length between source area, destination area, and character count are resolved the same as in MOVE.

If no option list is written, the zero suppression is still active.

If the operand is alphabetic, the zone bits are automatically stripped.

DECIMAL ARITHMETIC INSTRUCTIONS

Number Formats

Arithmetic instructions generate decimal-character numbers with a sign and magnitude format of 1) an overpunched (merged) least significant digit for negative numbers, and 2) no indication on positive numbers. Any character other than 1 through 9 is treated as a zero. Any negative zero number is changed to a positive zero, if encountered as data or generated by an intermediate operation.

The least significant digit indicates the sign of a quantity, as follows.

Characters	Card Code	XDS Internal Code	Numeric Value of LSD	Sign of Field
Space, Blank (b)	8-2, Blank	12, 60	0	+
&, +	12	20	0	+
Backspace, ?	12-0	32	0	+
0	0	00	0	+
1, A	1, 12-1	01, 21	1	+
2, B	2, 12-2	02, 22	2	+
3, C	3, 12-3	03, 23	3	+
4, D	4, 12-4	04, 24	4	+
5, E	5, 12-5	05, 25	5	+
6, F	6, 12-6	06, 26	6	+
7, G	7, 12-7	07, 27	7	+
8, H	8, 12-8	10, 30	8	+
9, I	9, 12-9	11, 31	9	+
-	11	40	0	-
Carriage Return, !	11-0	52	0	-
J	11-1	41	1	-
K	11-2	42	2	-
L	11-3	43	3	-
M	11-4	44	4	-
N	11-5	45	5	-
O	11-6	46	6	-
P	11-7	47	7	-
Q	11-8	50	8	-
R	11-9	51	9	-

Note: The characters S through Z are unpredictable.

Examples:

-377_{10}	[030747]	characters
-400_{10}	[040040]	
400_{10}	[040000]	

Upon encountering negative numbers whose least significant digit is octal 52 (equivalent to a punched card zero overpunched with an 11-zone), the arithmetic package replaces the octal 52 code with the octal 40 code.

The arithmetic instructions have the form:

Label	Operation	Operand
L1	OPCODE	E1, LO1, CC1, E2, LO2, CC2
L1	OPCODE	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3

L1 = Any label or blank.

Ei = First location of the memory area containing the ith operand (i = 1, 2, or 3).

LOi = Character position number of the least significant character of the ith operand. LOi must be greater than or equal to one.

CCi = Number of characters in the ith operand. CCi must be greater than or equal to one and less than or equal to 14.

E1, LO1, and CC1 refer to the first operand. E2, LO2, and CC2 refer to the second operand; in the first form above, they also define the area in which the result is placed. E3, LO3, and CC3 define the area in which the result is placed in the second form above. For example, in the add operation DADD A, 2, 2, B, 2, 2 the resulting sum of A and B is placed in the area defined by B, completely replacing the operand previously in B. For the addition DADD A, 2, 2, B, 2, 2, C, 2, 2 the result is placed in the area defined at C; areas A and B are left undisturbed.

A third form of decimal arithmetic instruction is provided for addition and subtraction:

Label	Operation	Operand
L1	OPCODE	E1, LO1, CC1

in which a decimal one is added to or subtracted from the given number and the result is placed back into the given area.

The Ei can be indirectly addressed and/or indexed.

For the XDS 900 Series, the form of the operand signified by Ei is

*Ei	Indirectly Addressed
(Ei)	Indexed
(*Ei)	Both

For the XDS 9300, the form is

*Ei	Indirectly Addressed
(Ei, X)	Indexed, where X = 1, 2, 3, or is some label made equivalent to one of these numbers by an EQU
(*Ei, X)	Both

Arithmetic Overflow

When an arithmetic overflow occurs during a decimal arithmetic operation, the instruction sets an overflow flag. The BAOV business instructions can test this. The flag can be reset only by using another arithmetic instruction. An attempt to divide by 0 or -0 sets the overflow flag and causes no other action. An overflow may also occur during multiply, add, or subtract when the result at run-time is larger than the defined size of the result field.

Note: No FIELD-defined labels can be used with decimal arithmetic instructions.

DADD

Decimal Add

Performs decimal character addition in one of three ways: 1) adds 1 to the decimal character string given, 2) adds the first string to the second and replaces the second with the result, or 3) adds the first string to the second and places the result in a third memory area.

Label	Operation	Operand
L1	DADD	E1, LO1, CC1
L1	DADD	E1, LO1, CC1, E2, LO2, CC2
L1	DADD	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3

DSUB

Decimal Subtract

Performs decimal character subtraction in one of three ways: 1) subtracts 1 from the decimal character string given, 2) subtracts the second string from the first and replaces the second with the result, or 3) subtracts the second string from the first and places the result in a third memory area.

Label	Operation	Operand
L1	DSUB	E1, LO1, CC1
L1	DSUB	E1, LO1, CC1, E2, LO2, CC2
L1	DSUB	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3

DMUL

Decimal Multiply

Performs decimal character multiplication in one of two ways: (1) multiplies the first string by the second and replaces the second with the result, or (2) multiplies the first string by the second and places the result in a third memory area.

Label	Operation	Operand
L1	DMUL	E1, LO1, CC1, E2, LO2, CC2
L1	DMUL	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3

DDIV

Decimal Divide

Performs decimal character division in one of two ways: 1) divides the first string by the second and replaces the second with the result, or 2) divides the first string by the second and places the result in a third memory area.

Label	Operation	Operand
L1	DDIV	E1, LO1, CC1, E2, LO2, CC2
L1	DDIV	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3

Remainder

The remainder from a division operation is found right-justified in a 14-character memory area whose label is B\REG2 (the label's peculiar makeup is due to its being a special system label). The sign is in the least significant digit position (as with any other decimal number), and it is the same as for the quotient.

BAOV

Branch on Arithmetic Overflow

Tests the decimal arithmetic overflow flag and branches to location E1 if it is set true. BAOV does not reset the flag.

Label	Operation	Operand
L1	BAOV	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation.[t] No indexing is allowed.

Decimal Conversion Instructions

BINBCD	BCDBIN
Binary to BCD	BCD to Binary

Label	Operation	Operand
L1	BINBCD	E1, HC1, CC1, E2
L1	BCDBIN	E1, HC1, CC1, E2

E1 = First location of the memory area associated with the decimal character string.

HC1 = Character position of the most significant character of the decimal string.

CC1 = Length of the character string.

E2 = First core location (defined label) of the binary word pair.

[t] See computer reference manual.

BINBCD converts a binary integer to its BCD (decimal) equivalent. It converts the binary words, located by E2, to a BCD character string; the result is placed in the memory area indicated by E1, HC1 with a character string length of CC1.

BCDBIN converts a BCD character string to its binary equivalent. E1, HC1 specify the memory area of the defined character string, with CC1 indicating the number of characters; the result is placed in the area located by E2.

Indexing and indirect addressing for E1 and E2 is as defined in the decimal arithmetic instructions. All binary numbers are considered to be double-precision (double-length) integer values and are held in memory, in the XDS 900 Series, as follows:

Location N	Least significant half
Location N + 1	Most significant half

and for the XDS 9300:

Location N	Most significant half
Location N + 1	Least significant half

BINBCD and BCDBIN also observe these conventions.

Note: No FIELD-defined labels may be used with these instructions.

Number Formats

Number formats are the same as with the decimal arithmetic instructions.

CHARACTER MANIPULATION INSTRUCTIONS

PACK (PACKL)

Pack Left-Justified Character String

Packs a group of characters, contained one per word and left-justified in the word, into an array of words packed four characters per word.

Label	Operation	Operand
L1	PACK	E1, E2, E3, ...

L1 = Any label or blank

E1 = Location of the first word of the resultant array of packed words. It may be indexed and/or indirectly addressed.

E2, E3,... = Locations of the individual character words whose left-justified characters are retrieved and packed. Any of these addresses may be indexed and/or indirectly addressed.

The packing process scans data from left to right, placing the first character into the left-most position of the packing area, the second character into the position adjacent to the first, and so on until all characters have been stored.

If the number of characters specified in the source line is not an even multiple of four, the last word in the packed array will contain trailing zeros. For example, an operand list

E1, E2, E3 ... E8

will cause characters to be packed as shown:

word E1:	E2	E3	E4	E5
E1+1	E6	E7	E8	0

PACK generates a calling sequence whose length is 3+ number of character words specified.

An "S" error flag is generated at assembly time if the number of arguments in the operand is less than two.

UNPACK

Unpack Left-Justified Character String

Unpacks characters, contained four per word in an array of words, into individual words. The character in each word is left-justified.

Label	Operation	Operand
L1	UNPACK	E1, E2, E3, ...

L1 = Any label or blank

E1 = Location of the first word of the array of packed words. It may be indexed and/or indirectly addressed.

E2,E3,... = Locations of the individual words into which the left-justified characters are formed by unpacking. Any of these addresses may be indexed and/or indirectly addressed.

UNPACK unpacks an array of words, packed four characters per word, into individual character words where each contains the character left-justified with trailing blanks (60's). The unpacking operation proceeds from left to right through contiguous packed words. That is, the first character word formed contains the leftmost character of the initial packed word, the second character word contains the second character of the initial packed word, and so on, for as many character words as specified.

UNPACK generates a variable-length calling sequence, the length of which is equal to 3+ the number of character words specified. An exception to this occurs when only one character is unpacked; in this case, four words of in-line code are generated.

An "S" error flag is generated during the assembly if the number of arguments in the operand is less than two.

PACKR

Pack Right-Justified Character String

Packs an array of right-justified single-character words into a four-characters-per-word array of packed words.

Label	Operation	Operand
L1	PACKR	E1, E2, N

L1 = Any label or blank

E1 = Location of the first word of the resultant array of packed words. It may be indexed and/or indirectly addressed.

E2 = Location of the first word of the array of unpacked right-justified single-character words. It may be indexed and/or indirectly addressed.

N = Count of the characters to be packed, i.e., the number of character words in the array at E2. N may be written as a user-reserved label enclosed in parentheses, indicating the symbolic address of a location containing the count (right-justified) at run-time.

PACKR packs the right-justified characters contained in an array of character words into an array of packed words (four characters per word). The packing process is executed from left to right, with the character in the first unpacked character word being placed in the left-most character position of the initial packed word, the character in the second unpacked word being placed in the adjacent character position of the first packed word, and so on. If the number of unpacked character words is not a multiple of four, the last word in the resultant array of packed words will contain trailing blanks (60's).

PACKR generates a four-word calling sequence.

An "S" error flag is generated at assembly time if the number of arguments in the operand is not equal to three.

UNPACKR

Unpack Right-Justified Character String

Unpacks an array of packed words into an array of right-justified, single-character words.

Label	Operation	Operand
L1	UNPACKR	E1, E2, N
L1	UNPACKR	E1, E2, N, 'CH'

L1 = Any label or blank

E1 = Location of the first word of the array of packed words. It may be indexed and/or indirectly addressed.

E2 = Location of the first word of the resultant array of right-justified single-character words formed by the unpacking. It may be indexed and/or indirectly addressed.

N = Count of characters to be unpacked, i.e., the number of character words in the array at E2. N may be written as a user-reserved label in parentheses, indicating the symbolic address of a location containing the count (right-justified) at run-time.

'CH' = Alphanumeric or special character enclosed in single quotes, to be inserted as the leading three characters in all derived unpacked character words. If this argument is not specified, three leading blanks (60's) will be inserted instead.

UNPACKR unpacks an array of packed words (four characters per word) into an array of right-justified unpacked character words, where each derived character word normally contains leading blanks (60's). An optional fill character can be specified, which is inserted instead of blanks (60's). The unpacking process occurs from left to right, with the left-most character in the initial word of the packed array forming the first unpacked character word.

UNPACKR generates a calling sequence of five words.

An "S" error flag is generated at assembly time if the number of arguments in the operand is less than 3 or greater than 4.

DATA TESTING INSTRUCTIONS

COMPARW (CPW)

Compare Word String

Compares one consecutive sequence of memory words to another and sets the high, low or equal flag according to whether the second area is higher, lower, or equal to the first. The comparison is left to right and is performed according to the collating sequence specified by the COLLATE instruction (XDS or BDP). The comparison flag can be tested by the Business Language branch instructions.

Label	Operation	Operand
L1	COMPARW	E1, E2, N

L1 = Any label or blank

E1 = The first location of the first memory area; E1 can be indexed, indirectly addressed, or both.

E2 = First location of the second memory area; E2 can be indexed, indirectly addressed, or both.

N = Count, the number of words to be compared. If written (N) where N is a user-reserved label, the contents of N at execution time are taken as the word count (right-justified).

COMPARW generates a four-word calling sequence. At assembly time, if the word count has been given as zero, no words will be compared and no flags will be set; if the count has been given as blank, one word pair will be compared; if the count has been given as negative, a syntax error indication will be recorded. At execution time if the count is negative or zero, one word pair will be compared.

COMPARW disturbs the A and B registers.

COMPARE

Compare Character String

Compares a consecutive string of characters in one memory area to another, setting the high, low or equal flag according to whether the second area is higher, lower, or equal to the first. The comparison is from left to right and is performed according to the collating sequence specified by COLLATE (XDS or BDP). COMPARE need not begin or end the comparison on a word boundary and thus can compare between fields, consecutive word sequences, or both.

Label	Operation	Operand
L1	COMPARE	E1, HC1, E2, HC2, CC
L1	COMPARE	E1, HC1, F2
L1	COMPARE	E1, HC1, CC, F2
L1	COMPARE	F1, E2, HC2
L1	COMPARE	F1, E2, HC2, CC
L1	COMPARE	F1, F2

L1 = Any label or blank

E1 = First, the first location of the first memory area; E1 can be indexed, indirectly addressed, or both.

HC1' = High count of the first area; the position number of the first character to be compared, as determined by counting from the left in location E1 to, and including, that first character.

E2 = Second, the first location of the second memory area; E2 can be indexed, indirectly addressed, or both.

HC2 = High count of the second area; the position number of the first comparison character in the second area.

CC = Character count, the number of characters to be compared.

F1 = First field; the field F1 defined in the user's program as the first memory area. No indexing or indirect addressing is allowed.

F2 = Second field.

COMPARE compares a character string defined by a first location, high character count, and length (number of characters) or by a field definition with a second field or character string in a second memory area. Comparing from the left, COMPARE sets the high, low or equal flag as appropriate; this flag is testable by the Business Language branch instructions. The number of characters compared will be equal to the smaller of the comparison areas or to the character count, whichever is the lesser. COMPARE locates the word containing the first comparison character by computing $Ei + (HCi \div 4)$. When Ei is indirectly addressed, COMPARE first computes the indirect address then adds it to $HCi \div 4$ to find the first word. The high character position count must be no more than 1027; the length must be no more than 256. Exceeding these limits generates a T (truncation) flag during assembly.

COMPARE does not disturb the index register(s).

PROGRAM BRANCH CONTROL INSTRUCTIONS

BREQ (BE)

Branch on Equal

Branches to the location specified, if the preceding COMPARE instruction set the compare flag to EQUAL.

Label	Operation	Operand
L1	BREQ	E1

L1 = Any label or blank

E1 = Symbolic address of the branch-to location. If written (E1), the branch instruction will be a branch and mark place. No indexing is allowed. Indirect addressing is allowed.

BRNE (BU)

Branch on Not Equal

Branches to the location specified, if the preceding COMPARE instruction set the compare flag to NOT EQUAL.

Label	Operation	Operand
L1	BRNE	E1

L1 = Any label or blank

E1 = Symbolic address of the branch-to location. If written (E1), the branch instruction will be a branch and mark place. No indexing is allowed. Indirect addressing is allowed.

BRHI (BH)

Branch on High

Branches to the location specified, if the preceding COMPARE instruction set the compare flag to HIGH (i.e., the second operand is greater than the first).

Label	Operation	Operand
L1	BRHI	E1

L1 = Any label or blank

E1 = Symbolic address of the branch-to location. If written (E1), the branch instruction will be a branch and mark place. No indexing is allowed. Indirect addressing is allowed.

BRLO (BL)

Branch on Low

Branch to the location specified, if the preceding COMPARE instruction set the compare flag to LOW.

Label	Operation	Operand
L1	BRLO	E1

L1 = Any label or blank

E1 = Symbolic address of the branch-to location. If written (E1), the branch instruction will be a branch and mark place. No indexing is allowed. Indirect addressing is allowed.

BRACNE

Branch on any Character Not Equal

Branches to the location specified, if the character in the specified location and character position is not equal to one of the characters in an indicated list of characters.

Label	Operation	Operand
L1	BRACNE	E1, E2, E3, 'Q1', 'Q2', ...

L1 = Any label or blank

E1 = Symbolic address of the branch-to location on nonequality. If enclosed in parentheses, a mark place and branch (BRM) will be executed. No indexing is allowed. Indirect addressing is allowed.

E2 = Symbolic address of the base location of the character argument to be compared; it may be indexed and/or indirectly addressed.

E3 = Relative character position (offset) from the base location of the character to be compared. If enclosed in parentheses, it indicates the symbolic address of a location containing the offset (right-justified) at run-time.

'Q1','Q2',... = The list of alphanumeric or special characters to which the character argument is compared for equality. Each member of the list must be written within single quotes unless it is numeric. To specify a blank, write ' '.

BRACNE compares a character argument at a given location and offset to a list of characters. The offset is the number of the first character to be compared, determined by counting from the left in location E2 to, and including, the first character. If the character argument in the effective location is not equal to any member of the specified list, control is transferred (BRU) to E1. If E1 is enclosed in parentheses, a BRM to E1 is executed. Normally, BRACNE generates a calling sequence whose length is four plus the covered quotient of the number of members in the compared list ÷ 4. This is conditionally supplemented by a preceding three-word calling sequence if E2 is indexed or indirectly addressed, or if E3 is a run-time parameter (i.e., E3 is enclosed in parentheses).

An "S" error flag is generated at assembly time if the number of arguments in the operand is less than four.

BRACEQ

Branch on any Character Equal

Branches to the location specified if the character in the specified location and character position is equal to one of the characters in an indicated list of characters.

Label	Operation	Operand
L1	BRACEQ	E1, E2, E3, 'Q1', 'Q2', ...

L1 = Any label or blank

E1 = Symbolic address of the branch to location on equality. If enclosed in parentheses, a mark place and branch (BRM) will be executed. No indexing is allowed. Indirect addressing is allowed.

E2 = Symbolic address of the base location of the character argument to be compared; it may be indexed and/or indirectly addressed.

E3 = Relative character position (offset) from the base location of the character to be compared. If enclosed in parentheses, it indicates the symbolic address of a location containing the offset (right-justified) at run-time.

'Q1', 'Q2', ... = The list of alphanumeric or special characters to which the character argument is compared for equality. Each member of the list must be written within single quotes unless it is numeric. To specify a blank, write ' '.

BRACEQ compares a character argument, at given location and offset, to a list of characters. The offset is the number of the first character to be compared, determined by counting from the left in location E2 to, and including the first character. If the character argument in the effective location is equal to any member of the specified list, control is transferred (BRU) to E1. If E1 is enclosed in parentheses, a BRM to E1 is executed. Normally, BRACEQ generates a calling sequence whose length is four plus the covered quotient of the number of members in the compared list ÷ 4. This is conditionally supplemented by a preceding three-word calling sequence, if E2 is indexed or indirectly addressed or if E3 is a run-time parameter (i.e., E3 is enclosed in parentheses).

An "S" error flag is generated at assembly time if the number of arguments in the operand is less than four.

DATA FIELD INITIALIZING INSTRUCTIONS

CLEAR

Clear Word String to Zeros

Clears (to zeros) an indicated area of memory for a specified number of words.

Label	Operation	Operand
L1	CLEAR	E1, N
L1	CLEAR	E1, N, X

L1 = Any label or blank

E1 = Initial location of the memory area to be cleared; it may be indexed or indirectly addressed.

N = Count of the number of memory locations to be cleared; if missing, N is assumed to be one. If enclosed in parentheses, N indicates the symbolic address of a location containing the word count at run-time. No indexing or indirect addressing is allowed.

X = If present, the index register(s) is saved and restored.

Clears an area of memory to zeros. CLEAR generates various combinations of in-line code, from a minimum of two words to a maximum of eleven.

An "S" error flag is generated at assembly time if there are no arguments in the operand.

CLEARCH

Clear Character String to Zeros

Clears to zeros a consecutive sequence of characters in memory.

Label	Operation	Operand
L1	CLEARCH	E1, HC1, CC
L1	CLEARCH	F1

L1 = Any label or blank

E1 = First location of the area to be cleared; E1 can be indexed, indirectly addressed or both.

HC1 = High count; the position number of the first character of the string to be cleared.

CC = Character count, the number of characters in the string.

F1 = User-defined field name indicating the field to be cleared.

CLEARCH selects the first character in the same way as MOVE. HC1 must be no more than 1027; CC must be no more than 256. Exceeding these limits causes a T (truncation) error at assembly time. The calling sequence is nine words in length.

CLEARCH does not disturb the index register(s).

BLANK

Set Word String to Blanks

Clears to blanks (60's) an indicated area of memory for a specified number of words.

Label	Operation	Operand
L1	BLANK	E1, N
L1	BLANK	E1, N, X

L1 = Any label or blank

E1 = Initial location of the memory area to be cleared; it may be indexed or indirectly addressed.

N = Count of the number of memory locations to be cleared (if missing, N is assumed to be one). If enclosed in parentheses, N indicates the symbolic address of a location containing the word count at run-time. No indexing or indirect addressing is allowed.

X = If present, the index register(s) is saved and restored.

Clears an area of memory to blanks. BLANK generates various combinations of in-line code from a minimum of two words to a maximum of eleven.

An "S" error flag is generated at assembly time if there are no arguments in the operand.

BLANKCH

Set Character String to Blanks

Clears to blanks (060's) a consecutive sequence of characters in memory.

Label	Operation	Operand
L1	BLANKCH	E1, HC1, CC
L1	BLANKCH	F1

L1 = Any label or blank

E1 = First location of the area to be cleared; E1 can be indexed, indirectly addressed, or both.

HC1 = High count; the position number of the first character of the string to be cleared.

CC = Character count; the number of characters in the string.

F1 = User-defined field name indicating the field to be cleared.

BLANKCH selects the first character in the same way as MOVE. HC1 must be no more than 1027; CC must be no more than 256. Exceeding these limits causes a T (truncation) error at assembly time. The calling sequence is nine words in length.

BLANKCH does not disturb the index register(s).

FILL

Fill Word String with Character

Fills an indicated area of memory with the character specified for a specified number of words.

Label	Operation	Operand
L1	FILL	E1, N, 'CH'
L1	FILL	E1, N, 'CH', X

L1 = Any label or blank

E1 = Initial location of the memory area to be filled; it may be indexed and/or indirectly addressed.

N = Count of the number of memory locations to be filled. (If missing, N is assumed to be one.) If enclosed in parentheses, it indicates the symbolic address of a location containing the word count at run-time. No indexing or indirect addressing is allowed.

'CH' = Character with which memory is to be filled.

X = If present, it indicates that the index register(s) should be saved and restored.

Fills an area of memory with a specified character, packed four to a word. FILL generates various combinations of in-line code ranging from a minimum of two words to a maximum of eleven.

An "S" error flag is generated at assembly time if the number of arguments in the operand is less than three or if the word count is less than one.

FILLCH

Fill Character String with Character

Fills a consecutive sequence of character positions in memory with a specified character.

Label	Operation	Operand
L1	FILLCH	E1, HC1, CC, 'CH'
L1	FILLCH	F1, 'CH'

L1 = Any label or blank

E1 = First location of the area to be filled; E1 can be indexed, indirectly addressed, or both.

HC1 = High count; the position number of the first character of the string to be filled.

CC = Character count; the number of characters in the string.

F1 = User-defined field name indicating the field to be filled.

'CH' = Alphanumeric or special character with which memory is to be filled.

FILLCH selects the first character in the same way MOVE does. HC1 must be no more than 1027; CC must be no more than 256. Exceeding these limits causes a T (truncation) error at assembly time. The calling sequence is nine words in length.

INTERNAL SORTING INSTRUCTIONS

SORT (SORTDS, SORTBIN, SORTBDS)

Sorts a table of items in memory, into ascending or descending sequence; an item being a specified number of words in length. A logical sort is performed on BCD items; an algebraic sort on binary items.

SORT = Ascending BCD Sort.

SORTDS = Descending BCD Sort.

SORTBIN = Ascending Binary Sort

SORTBDS = Descending Binary Sort

Label	Operation	Operand
L1	SORT	E1, E2, E3, E4, E5
L1	SORT	E1, E2, E3, E4, E5, E6

L1 = Any label or blank

E1 = Location of origin of table to be sorted. It may be indexed and/or indirectly addressed.

E2 = Number of items to be sorted. If enclosed in parentheses, it indicates the symbolic address of a location containing the number of items at run-time. No indexing or indirect addressing is allowed.

E3 = Length of an item (number of words). Parentheses indicate the symbolic address of the location containing the item length at run-time. No indexing or indirect addressing is allowed.

E4 = Relative (high-order) word position of the sort key within the item. Parentheses indicate the symbolic address of the location containing key position at run-time. No indexing or indirect addressing is allowed.

E5 = Length of key (number of words). Parentheses indicate the symbolic address of the location containing key length at run-time. No indexing or indirect addressing is allowed.

E6 (optional) = Location of the origin of the sorted table, if it is to be relocated; it may be indexed and/or indirectly addressed.

SORT performs an internal sort on the table indicated, overlaying the result on the original table unless overridden by the presence of an entry for E6. Comparisons during sorting are influenced by the status of the last COLLATE command executed prior to the SORT. (See COLLATE description in "Special Operations" group of instructions.) If a COLLATE 'BDP' has been given, compares act on a translated key (commercial collating) sequence. It is suggested that COLLATE 'BDP' be given only when absolutely necessary, because sorting is appreciably slower (by a factor of approximately six) if translation is required. SORT generates a six-word calling sequence and also generates in-line code prior to the calling sequence, varying from one to 12 words as a function of the arguments in the operand.

An "S" error flag is generated at assembly time if (a) the number of arguments in the operand is less than five or greater than six, (b) assembly-time item length is greater than 50 words, (c) assembly-time key position or key length is greater than 50 words, or the total size of the table (assembly-time arguments) is greater than 8191 words.

REGISTER SHIFT INSTRUCTIONS

LSHIFT (RSHIFT)

Logical Left Shift AB Register (Logical Right Shift AB Register)

Performs a left (right) logical shift on the eight-character, BCD contents of the A and B registers taken together as one register (the AB register).

Label	Operation	Operand
L1	LSHIFT	N
L1	RSHIFT	N

L1 = Any label or blank

N = Length of shift (in character positions).

Shifts the BCD contents of the AB register N character positions to the left or right. This instruction generated a single, machine-language binary shift instruction. The shift count N can be indexed (N) and/or indirectly addressed *N (900 Series only).

An "S" error flag is generated at assembly time if the number of arguments in the operand is not equal to one. On the XDS 9300 only, it will generate an "S" error flag for any indirect shift attempt.

SPECIAL OPERATIONS

COLLATE

Set Collating Sequence

Indicates that compares made thereafter are dependent on 'XDS' internal collating sequence or on 'BDP' (IBM 1400 Series) collating sequence, depending on the specified operand.

Label	Operation	Operand
L1	COLLATE	'XDS'
L1	COLLATE	'BDP'

L1 = Any label or blank

'XDS' = Compares made are based on the XDS internal collating sequence.

'BDP' = Compares made are based on the commercial collating sequence.

The operand of COLLATE indicates whether ensuing compares will be based on normal XDS internal collating sequence or commercial (IBM 1400 Series) collating sequence. If COLLATE is never given in the source language program, all compares will be based on XDS collating sequence. COLLATE 'BDP' actually initiates a translation of the COMPARE arguments into an inverted character set representative of IBM 1400 Series. The COLLATE instruction can be given repetitively during the flow of a program. If a programmer calls the Business Language "SORT" verb, comparisons made within the SORT will be based on the sequence stated in the last COLLATE executed prior to calling SORT.

COLLATE generates either one or two words of in-line code.

An "S" error flag will be generated at assembly time if the operand is neither 'XDS' or 'BDP'.

MEMORY

Compute Memory Size

Computes the memory size of the machine in which the instruction is executed and place a 2, 4, 6, 8, 12, 16, 20, 24, 28, or 32 in the A register. These digits respectively represent two thousand through 32 thousand words of memory.

Label	Operation	Operand
L1	MEMORY	
L1	MEMORY	E1

L1 = Any label or blank

E1 (optional) = Location in which memory size is to be stored; it can be indirectly addressed (not indexed).

MEMORY computes the memory size of the machine in which it is executed. Memory size is represented by the appropriate number 2, 4, 6, 8, 12, 16, 20, 24, 28, or 32 being placed in the A register. If an operand is present, computed memory size is stored in that location. MEMORY generates either 15 or 16 words of in-line code.

EXTEND

Specify Extended Assembly Mode

Instructs META-SYMBOL to perform an assembly in the extended XDS Business Language mode. It is mandatory that this directive be given at the beginning of any source program that has called any Business Language instruction (PROCedure).

Label	Operation	Operand
L1	EXTEND	

L1 = Any label or blank

No operand is applicable to this directive.

This directive must be present in every Business Language assembly. As a matter of good practice, it should be given as the first statement in any source-language program (i.e., immediately following the △META card). No object code is generated by this directive.

BUSINESS LANGUAGE INPUT/OUTPUT INSTRUCTION ROUTINES

XDS 900 SERIES

Before a Business Language input/output instruction is initiated, the I/O routine checks to see if the buffer is active. If it is active for more than two seconds, the routine halts and displays a NOP in the C register. The address portion of the register contains zero if the buffer being tested is the W buffer, and one if it is the Y buffer.

When the buffer is found not active (i.e., ready), the routine queries Breakpoint 4. If it is set, the routine waits in a short loop, allowing the operator the opportunity to adjust any portion of the physical system such as setting up magnetic tapes. As soon as Breakpoint 4 is reset (or if the routine initially finds it reset), the routine continues I/O processing. Interrupts are never used by or with the I/O package. If any interrupt is enabled during the use of the I/O package, unpredictable results will be obtained. All I/O is performed in the single-word mode of transmission.

When any of the I/O instructions is called within a program, the entire package is loaded into memory. It occupies approximately 713 (decimal) locations.

XDS 9300

All input/output operations and the errors detected during them are handled by the 9300 MONITOR. The Business Language recognizes and handles errors as noted in the following descriptions only when the 'F' option is used by the operator. That is, only when a ΔF typed input instructs MONITOR to return does the awareness of the error get back to the Business Language Input/Output Routine. If the error is passed on to the routine, the channel-error flag (testable by BCER) is set. For operational policy to be decided, a thorough knowledge of the MONITOR 9300 as found in the MONITOR Reference Manual is required.

The more normal indicators, such as end-of-file, end-of-tape, beginning-of-tape, are testable via the I/O branch tests explained later in this section.

WRITETP

Write Magnetic Tape Record (BCD Mode)

Writes a record of specified length, in BCD format, onto tape from a specified memory area.

Label	Operation	Operand
L1	WRITETP	LU, E1, CC

L1 = Any label or blank

LU = Logical tape unit number.

E1 = Symbolic address of the output area; it may be indexed and indirectly addressed.

CC = Number of characters to be written; this parameter cannot be omitted. If CC is a label and is enclosed in parentheses, the binary contents of location CC is the number of words to be written (four-characters-per-word output mode). If CC is a label, is enclosed in parentheses, and is prefixed with an asterisk (*CC), the number of words to be written is the contents of the effective address, where the asterisk in-indicates indirect addressing.

WRITETP writes a record, in BCD format, of a specified number of words (an execution-time parameter) or a specified number of characters (an assembly-time parameter). When writing a number of characters, WRITETP writes an integral number of words which cover the specified number of characters. The calling sequence generated in-line by WRITETP is three words in length. Note: In writing or reading tapes, any 012 code characters (zeros) written will read back as 00.

900 Series Error Checking

At the beginning of WRITETP execution, it resets the channel error, tape mark, beginning-of-tape, end-of-tape and file-protect flags. If the beginning-of-tape marker is sensed as the write operation begins, WRITETP erases a length of tape as a leader before it begins writing the record. WRITETP performs the following sequence of operations in writing tape:

Writes the number of words (or characters //4) specified. If this record is less than four words in length, it is always read back as a noise record and, as such, is lost to the user. If the character count CC is 12 or less, a T (truncation) flag syntax error will be printed on the WRITETP line during program assembly.

If a channel error is detected, WRITETP backspaces the length of the record and writes the record again. If another channel error is detected, it backspaces again and erases over the record. This is attempted no more than five times. If failure continues, WRITETP returns to the user with the channel error flag set. The tape is positioned past the bad area that has been erased.

If the end-of-reel is encountered, WRITETP sets the end-of-tape flag and returns to the user.

If a file-protected tape is encountered, WRITETP sets the file-protect flag and returns without writing.

WTPBIN

Write Magnetic Tape Record (Binary Mode)

Writes a record of specified length, in binary format, onto tape from a specified memory area.

Label	Operation	Operand
L1	WTPBIN	LU, E1, CC

L1 = Any label or blank

LU = Logical tape unit number.

E1 = Symbolic address of the output area; it may be indexed or indirectly addressed.

CC = Number of characters to be written, as follows (this parameter cannot be omitted). If CC is a label and is enclosed in parentheses, the binary contents of location CC is the number of words to be written. If CC is a label, is enclosed in parentheses, and is prefixed with an asterisk (*CC), the number of words to be written is the contents of the effective address, where the asterisk indicates indirect addressing. If written alone as a number without parentheses, CC is a character count that is divided by four (covered quotient) to generate the binary record word count.

WTPBIN functions indentically to WRITETP, except that it writes the record in binary and not in BCD. If a character count is specified (assembly-time parameter), WTPBIN forms the word count via the covered quotient CC//4.

Error checking is identical to WRITETP.

READTP

Read Magnetic Tape Record (BCD Mode)

Reads a record from magnetic tape, in BCD format, into a specified memory area, until the number of characters specified has been read into the specified memory area.

Label	Operation	Operand
L1	READTP	LU, E1, CC

L1 = Any label or blank

LU = Logical tape unit number.

E1 = Symbolic address of the input area; it can be indexed and/or indirectly addressed.

CC = Number of characters to be read; if omitted, the entire record is read. If CC is a label and is enclosed in parentheses, the contents of location CC is the number of words to be read (four-characters-per-word input mode). If CC is a label, is enclosed in parentheses, and is prefixed with an asterisk (*CC), the number of words to be read is the contents of the effective address, where the asterisk indicates indirect addressing.

READTP reads an entire record, the specified number of words (an execution-time parameter), or the specified number of characters (an assembly-time parameter). When reading characters, READTP reads an integral number of words containing enough characters to cover the requested amount (CC). The calling sequence generated in-line by READTP is three words in length.

XDS 900 Series Error Checking

At the beginning of READTP execution, it resets the channel error, tape-mark, beginning-of-tape, end-of-tape and file-protect flags. READTP performs the following sequence of operations:

Reads the number of words (characters//4) specified or reads to the end-of-record, whichever occurs first.

Tests for having read four or more words. If not four or more, a test is made for a tape mark (EOF). If the tape mark has been encountered, READTP sets the tape mark flag; and if not end-of-file it increments a noise-record count, reads the next record, and queries four or more words again, and so on. After fifty such 'noise records' are read in a row, the number 02000004 is placed in the C register and the program halts. To clear the count and read the next record via the same READTP instruction, clear the halt. Run-time arguments specifying three words or less will cause noise records and can only be detected as stated above. If the character count CC is 12 or less, a T (truncation) flag will be printed on the assembly line during program assembly.

If four or more words are read into memory, other errors are checked as follows:

If a channel error is detected, READTP attempts to read the record ten times. If unsuccessful, it sets the channel error flag and positions the read head after the bad record.

If the end-of-reel is encountered, READTP sets the end-of-tape flag.

RTPBIN

Read Magnetic Tape Record (Binary Mode)

Reads a record from magnetic tape, in binary format, into a specified memory area; or, reads words from tape until the end-of-record has been read into the specified memory area.

Label	Operation	Operand
L1	RTPBIN	LU, E1, CC

L1 = Any label or blank

LU = Logical tape unit number.

E1 = Symbolic address of the input area; it can be indexed and/or indirectly addressed.

CC = Number of characters to be read, as follows. If omitted, the entire record is read. If CC is a label and is enclosed in parentheses, the binary contents of location CC is the number of words to be read. If CC is a label, is enclosed in parentheses, and is prefixed with an asterisk, the number of words to be read is the contents of the effective address, where the asterisk indicates indirect addressing. If written alone as a number without parentheses, CC is a character count that is divided by four (covered quotient) to generate the record word count.

RTPBIN functions and checks errors identically to READTP, except that it reads in binary rather than BCD format. If a character count is specified (e.g., only a CC), RTPBIN divides the count by four and forms the word count via the covered quotient CC//4.

REWIND

Rewind Magnetic Tape

Rewinds the specified tape to the load point (beginning of tape reflective point).

Label	Operation	Operand
L1	REWIND	LU

L1 = Any label or blank.

LU = Logical tape unit number.

REWIND rewinds the tape to the load point. The calling sequence generated in-line by REWIND is two words in length.

XDS 900 Series Error Checking

At the beginning of REWIND execution, it resets the channel-error, tape-mark, end-of-tape and file-protect flags. After executing the instructions to position the tape at the load point, REWIND sets the beginning-of-tape flag.

WTMARK (WTM)

Write End-of-Tape Mark

Writes an end-of-tape mark.

Label	Operation	Operand
L1	WTMARK	LU

L1 = Any label or blank.

LU = Logical tape unit number.

WTMARK writes an end-of-file record (equivalently, a tape mark). The calling sequence generated in-line by WTMARK is two words in length.

XDS 900 Series Error Checking

At the beginning of WTMARK execution, it resets the channel-error, tape-mark, beginning-of-tape, end-of-tape and file-protect flags. It performs the following operations:

Writes a tape mark.

If a channel error occurs, WTMARK backspaces over the mark, erases forward over the mark record, and writes another mark.

This procedure is continued until a tape mark is successfully written or until the program hangs up because it has written the tape off the end of the reel.

It checks for and sets the flags accordingly for end-of-tape and file-protected tape.

BACKSPACE

Backspace Magnetic Tape

Backspaces N records or files, as specified, on tape unit LU.

Label	Operation	Operand
L1	BACKSPACE	LU, N

L1 = Any label or blank.

LU = Logical tape unit number.

N = Number of records to be backspaced over; if written (N), N specifies the number of files to be backspaced over.

BACKSPACE backspaces tape unit LU over N records, if N is written alone, or N files if N is written within parentheses. To backspace over N files means that BACKSPACE moves the tape backward over N end-of-file marks and then returns forward past the last one, so that reading can begin within the last file passed over. For example, BACKSPACE LU,(1) positions the tape at the beginning of the first record of the current file. The calling sequence generated in-line by BACKSPACE is two words in length.

XDS 900 Series and XDS 9300 Error Checking

BACKSPACE first resets all tape flags (see WRITETP). Once the backspacing operation has begun, it is stopped as follows:

If an end-of-file is encountered during a backspace of N records, BACKSPACE returns control to the user, with the tape-mark flag set and the tape head positioned in front of the end-of-file mark.

If the beginning-of-tape (load point) is encountered during a backspace, the beginning-of-tape flag is set and control returns to the user.

If the required number of records is backspaced over, control returns to the user without setting any flags. If the required number of files is encountered during a file-counting backspace, control returns to the user with only the tape-mark flag set.

SKIPTAPE

Skip Magnetic Tape Records Forward

Skips forward N records or files as specified on tape unit LU.

Label	Operation	Operand
L1	SKIPTAPE	LU, N

L1 = Any label or blank.

LU = Logical tape unit number.

N = Number of records to be skipped over; if written (N), N specifies the number of files to be skipped over.

SKIPTAPE skips tape unit LU over N records if N is written alone, or N files if N is written within parentheses. The calling sequence generated in-line by SKIPTAPE is two words in length.

XDS 900 Series and XDS 9300 Error Checking

SKIPTAPE initially resets all tape flags. Once the skip operation has begun, it stopped as follows:

If an end-of-file is encountered during a skip of N records, SKIPTAPE returns control to the user, with the tape-mark flag set and the tape head positioned in front of the end-of-file mark.

If the end-of-tape (end-of-reel marker) is encountered during a skip, the end-of-tape flag is set and control returns to the user.

If the required number of records is skipped over, control returns to the user without setting any flags. If the required number of files is skipped over, control returns to the user with only the tape-mark flag set.

READCD

Read BCD Card

Reads a card in BCD format into a specified memory area.

Label	Operation	Operand
L1	READCD	E1

L1 = Any label or blank.

E1 = Symbolic address of the input area; if E1 is enclosed in parentheses it indicates indexing is to be performed on the address. Indirect addressing is allowed.

READCD reads one card (80 columns) into a 20-word memory area defined by E1. The calling sequence generated in-line by READCD is two words in length.

XDS 900 Series Error Checking

Initially READCD resets the channel-error flag and the last card flag. A validity check, read check, or any channel error (other than feed check) causes READCD to set the channel-error flag. A feed check causes READCD to halt the program and display 02000003 in the C register; straightening the current card, readying the reader, and clearing the halt causes the program to continue undisturbed. The card hopper is tested empty before the card is read and, if it is empty, READCD sets the last-card flag and executes the next instruction in sequence.

PUNCH

Punch BCD Card

Punches an 80 column card in BCD format from a specified memory area.

Label	Operation	Operand
L1	PUNCH	E1

L1 = Any label or blank.

E1 = Symbolic address of the output area; if E1 is enclosed in parentheses it indicates indexing is to be performed. Indirect addressing is allowed.

PUNCH punches a BCD card, taking the information from a 20-word memory area defined by E1, using either a buffered or unbuffered punch unit. The calling sequence generated in-line by PUNCH is two words in length.

XDS 900 Series Error Checking

PUNCH initially resets the channel-error flag; any detectable channel error causes PUNCH to set the channel-error flag.

TYPEIN

Input from Typewriter

Accepts characters from the typewriter and stores them in a specified memory area, until 80 characters have been received or until a carriage return has been typed.

Label	Operation	Operand
L1	TYPEIN	E1

L1 = Any label or blank.

E1 = Symbolic address of the input area; if E1 is enclosed in parentheses it indicates indexing is to be performed. Indirect addressing is allowed.

TYPEIN lights the typewriter type-in light and waits for the operator to type his message. The routine will accept up to 80 characters or up to a carriage return. If terminated by the carriage return, the carriage-return character (code 052) appears as the last nonblank character in the input area; the rest of the current word and the rest of the 20-word input area is blank (060) filled. On the XDS 9300, the carriage return code is not placed in the input area nor is the rest of the area blanked out. Spaces input from the typewriter are placed into the input area as 060 blanks. Input is packed four characters per word. The calling sequence generated in-line by TYPEIN is two words in length.

XDS 900 Series and XDS 9300 Error Checking

Initially TYPEIN resets the channel-error flag, and any channel error (returned via ΔF on the 9300) causes TYPEIN to set the channel-error flag.

TYPE

Output on Typewriter (XDS 900 Series only)

Types characters on the typewriter, from a specified memory area, until 80 characters have been typed or until a carriage-return character is detected in the memory area.

Label	Operation	Operand
L1	TYPE	E1

L1 = Any label or blank.

E1 = Symbolic address of the output area; it can be indexed and indirectly addressed.

TYPE types characters beginning at location E1, and continues until 80 characters (a 20-word memory buffer) followed by an automatic carriage return, have been typed or until the character to be typed is the carriage-return character. This character is the same as the exclamation point and is written as ! or octal code 52. TYPE converts all 060 code blanks to 012 blanks; 012 codes are not altered. TYPE does not disturb the memory area. Typing is done on the console typewriter (unit 1, channel 0). The calling sequence generated in-line by TYPE is two words in length.

Initially TYPE resets the channel-error flag; and any channel error causes TYPE to set the channel-error flag.

Label	Operation	Operand
MSG	TEXT	< AB CD! >
	⋮	
	TYPE	MSG

The above sequence causes AB CD to be typed, followed by a carriage return.

TYPE

Output on Typewriter (XDS 9300 only)

Types 80 characters on the typewriter, from a specified memory area.

Label	Operation	Operand
L1	TYPE	E1

L1 = Any label or blank.

E1 = Symbolic address of the output area; it can be indexed and/or indirectly addressed.

TYPE types 80 characters, beginning at the first character in location E1. Typing terminates when only trailing blanks remain in the 80-character image. It converts all 060 code blanks to 012 blanks, 012 being the typewriter carriage space. If no carriage return code is encountered at or before the 80th character is typed, TYPE will not automatically return the carriage. Four characters per word are typed out even though only blanks remain in the buffer. This is significant when a carriage return occurs at the end of a line. If the carriage return is other than the fourth character of a word, the following one, two, or three blanks in the word would be typed on the next line. Typing is performed on the console typewriter (unit 1, channel 0). The calling sequence generated in-line by TYPE is two words in length.

PRINT

Print 132-Character Line

Prints one 132-character line on the line printer, with the specified upspacing. Upspacing always occurs prior to the printing of the line.

Label	Operation	Operand
L1	PRINT	E1
L1	PRINT	E1, N
L1	PRINT	E1, 'Q'

L1 = Any label or blank.

E1 = Location of the output area; it can be indexed and indirectly addressed. When no other parameter is present, PRINT upspaces one line.

N = Channel, on the print control tape loop, to which the printer upspaces prior to printing the line. N=0, . . . , 7.

Q = Alphabetic character specifying the number of print lines that the printer upspaces prior to printing the line. Q=-, J, K, L, M, N, O, or P, specifying upspacing of 0, 1, . . . , 7 lines, respectively. The alphabetic character must be written within single quotes.

PRINT always upspaces, as specified, before printing. The characters in the entire 33-word memory area beginning at E1 are printed. The calling sequence generated in-line by PRINT is two words in length.

XDS 900 Series Error Checking

Initially, PRINT resets the print-fault, page-overflow and channel-error flags. PRINT operates in the following manner: (1) if a channel error is encountered during printing, the channel-error flag is set and control returns to the user; (2) if there had been a print fault on the previous printer operation (e.g., printing a line, upspacing, channel skipping, or a restore), PRINT sets the print-fault flag and prints the current line before returning control to the user; (3) if the page-overflow condition (true condition on channel 7 of the print control loop) is found prior to this print operation, the page-overflow flag is set and printing continues.

PRT 120

Print 120-Character Line

Prints one 120-character line on the line printer, with the specified upspacing. Upspacing always occurs prior to the printing of the line.

Label	Operation	Operand
L1	PRT120	E1
L1	PRT120	E1, N
L1	PRT120	E1, 'Q'

L1 = Any label or blank.

E1 = Symbolic address of the output area; it may be indexed and indirectly addressed. When no other parameter is present, PRT120 upspaces one line.

N = Channel, on the print control tape loop, to which the printer upspaces prior to printing the line. N = 0, . . . , 7.

Q = An alphabetic character specifying the number of print lines that the printer upspaces prior to printing the line. Q = -, J, K, L, M, N, O, or P, specifying the upspacing of 0, 1, . . . , 7 lines, respectively. The alphabetic character must be written within single quotes.

PRT120 always upspaces, as specified, before printing. The characters in the entire 30-word memory area beginning at E1 are printed. The calling sequence generated in-line by PRT120 is two words in length.

Error checking is identical with that of PRINT.

UPSPACE

Upspace Line Printer

Upspaces N print lines, where N can range from 0 to 7.

Label	Operation	Operand
L1	UPSPACE	N

L1 = Any label or blank.

N = Number of lines to be upspaced; 0, . . . , 7.

The calling sequence generated in-line by UPSPACE is two words in length.

XDS 900 Series Error Checking

Initially, UPSPACE resets the print-fault, page-overflow, and channel-error flags. It sets page-overflow if channel 7 on the format loop is 'true' before the upspace starts. It sets printer-fault if a print fault had occurred on the previous printer operation.

SKPCHN

Skip to Channel N

Upspaces the printer to channel N on the format control loop.

Label	Operation	Operand
L1	SKPCHN	N

L1 = Any label or blank

N = Number of the channel, on the format control loop, to which the printer is to upspace; N=0, . . . , 7.

The calling sequence generated in-line by SKPCHN is two words in length.

XDS 900 Series Error Checking

Initially, SKPCHN resets the print-fault, page-overflow, and channel-error flags. It sets page-overflow if channel 7 on the format loop is 'true' before the upspace starts. It sets printer-fault if a print fault had occurred on the previous printer operation.

RESTORE

Skip to Channel 1

Upspaces the printer paper to the top of the form (skips to channel 1 on the format control loop).

Label	Operation	Operand
L1	RESTORE	

L1 = Any label or blank.

The calling sequence generated in-line by RESTORE is two words in length.

XDS 900 Series Error Checking

Initially, RESTORE resets the print-fault, page-overflow, and channel-error flags. It sets page-overflow if channel 7 on the format loop is 'true' before the upspace starts. It sets printer-fault if a print fault had occurred on the previous printer operation.

INPUT/OUTPUT BRANCH TESTS

BCER

Branch on Channel Error

Tests the channel-error flag, branches to location E1 if it is set true, and unconditionally resets the flag.

Label	Operation	Operand
L1	BCER	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation. Indirect addressing is allowed but not indexing.

BCER generates two words of in-line code. The branch address cannot be indexed.

BPOV

Branch on Page Overflow XDS 900 Series only)

Tests the page-overflow flag, branches to location E1 if it is set true, and unconditionally resets the flag.

Label	Operation	Operand
L1	BPOV	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation. Indirect addressing is allowed but not indexing.

BPOV generates two words of in-line code. The branch address cannot be indexed. The page overflow flag is set by sensing a punch in channel 7 of the printer format control tape.

BPRF

Branch on Printer Fault

Tests the printer-fault flag, branches to location E1 if it is set true, and unconditionally resets the flag.

Label	Operation	Operand
L1	BPRF	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation. Indirect addressing is allowed but not indexing.

BPRF generates two words of in-line code. The branch address cannot be indexed.

BTMK

Branch on Tape Mark

Tests the tape-mark flag, branches to location E1 if it is set true, and unconditionally resets the flag.

Label	Operation	Operand
L1	BTMK	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation. Indirect addressing is allowed but not indexing.

BTMK generates two words of in-line code. The branch address cannot be indexed.

BBTP

Branch on Beginning of Tape

Tests the beginning-of-tape flag, branches to location E1 if it is set true, and unconditionally resets the flag.

Label	Operation	Operand
L1	BBTP	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation. Indirect addressing is allowed but not indexing.

BBTP generates two words of in-line code. The branch address cannot be indexed.

BETP

Branch on End of Tape

Tests the end-of-tape flag, branches to location E1 if it is set true, and unconditionally resets the flag.

Label	Operation	Operand
L1	BETP	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation. Indirect addressing is allowed but not indexing.

BETP generates two words of in-line code. The branch address cannot be indexed.

BFPT

Branch on File-Protected Tape

Tests the file-protected-tape flag, branches to location E1 if it is set true, and unconditionally resets the flag.

Label	Operation	Operand
L1	BFPT	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation. Indirect addressing is allowed but not indexing.

BFPT generates two words of in-line code. The branch address cannot be indexed.

BLCD

Branch on Last Card

Tests the last-card flag, branches to location E1 if it is set true, and unconditionally resets the flag.

Label	Operation	Operand
L1	BLCD	E1

L1 = Any label or blank.

E1 = Symbolic address of the branch; if E1 is written within parentheses, the branch is a branch and mark operation. Indirect addressing is allowed but not indexing.

BLCD generates two words of in-line code. The branch address cannot be indexed.

APPENDIX

GENERAL PROGRAMMING INTRODUCTION

NUMBER SYSTEMS

The decimal number system is based on powers of ten. The value of a decimal number is the total of each digit, where a digit is 0 through 9 times its corresponding power of ten.

Example: 987654 is the sum of

$$\begin{aligned}
4 \times 10^0 &= 4 \\
5 \times 10^1 &= 50 \\
6 \times 10^2 &= 600 \\
7 \times 10^3 &= 7000 \\
8 \times 10^4 &= 80000 \\
9 \times 10^5 &= \underline{900000} \\
& 987654
\end{aligned}$$

The binary number system is based on powers of two. The value of a binary number is the total of each digit, where a digit is 0 or 1 times its corresponding power of two.

Example: 1010100 is the sum of

$$\begin{aligned}
0 \times 2^0 &= 0 \\
0 \times 2^1 &= 0 \\
1 \times 2^2 &= 4 \\
0 \times 2^3 &= 0 \\
1 \times 2^4 &= 16 \\
0 \times 2^5 &= 0 \\
1 \times 2^6 &= \underline{64} \\
& 84
\end{aligned}$$

Binary numbers are convenient for computer use, since only two digits are involved. These digits can be represented physically as "off" and "on." Thus, in a magnetic core memory, cores magnetized in one direction can be "on", or 1, and cores magnetized in another direction can be "off", or 0.

COMPUTER WORD ORGANIZATION

In the XDS 9300 or 900 Series Computers, a word is defined as 24 core elements, or bits[t]. Each bit can be either a 0 or a 1. Thus, in a 24-bit computer word, the 84 in the above example would be represented as

000000000000000001010100

For humans, reading or writing numbers in 24-bit form is awkward. It is more convenient to represent a 24-bit word in sets of three bits, the highest value of any set of three bits being seven:

$$\begin{aligned}
1 \times 2^0 &= 1 \\
1 \times 2^1 &= 2 \\
1 \times 2^2 &= \underline{4} \\
& 7
\end{aligned}$$

[t]"Bit" is a contraction of "binary digit."

Thus, the 24-bit word for 84 can be represented by eight sets of three bits each; these sets of three bits are called octal digits:

00000124

and octal 124 equals decimal 84 and binary 1010100.

As before, the highest octal digit is 7. To represent a decimal 8, two octal digits are required:

$$\begin{aligned}
0 \times 2^0 &= 0 \\
0 \times 2^1 &= 0 \\
0 \times 2^2 &= 0 \\
1 \times 2^3 &= \underline{8} \\
& 8
\end{aligned}$$

and octal 10 equals decimal 8.

With six bits, numbers from octal 00 to octal 77 are available:

Octal	Decimal	Octal	Decimal
00	0	05	5
01	1	06	6
02	2	07	7
03	3	10	8
04	4	11	9

We have represented all decimal digits. Many pairs of octal digits are still unused. The letters A through Z and a few special characters, such as the dollar sign, are represented by assigning them the same kind of two-digit codes:

21 = A
22 = B
23 = C

and so on, until two-digit combinations are exhausted. (XDS computer reference manuals contain lists of the XDS codes.) This type of representation is called binary-coded decimal, or BCD.

Looking again at the 84 in the above examples, which in sets of three binary digits was equivalent to

00000124_8

we find that, in sets of six binary digits, this combination is equivalent to

0 0 1 D (in XDS internal code)

Note that a BCD number does not "equal" a decimal number as a binary or octal one does.

9 9 9 9	in BCD

is

11111111	in octal

which is

001 001 001 001 001 001 001 001	in binary

or

$$
\begin{aligned}
1 \times 2^{0} &= 1\\
1 \times 2^{3} &= 8\\
1 \times 2^{6} &= 64\\
1 \times 2^{9} &= 512\\
1 \times 2^{12} &= 4096\\
1 \times 2^{15} &= 32768\\
1 \times 2^{18} &= 262144\\
1 \times 2^{21} &= \underline{2097152}\\
& 2396745 \quad \text{in decimal}
\end{aligned}
$$

Binary representation of decimal 9999 is

000 000 000 010 011 100 001 111

$$
\begin{aligned}
1 \times 2^{0} &= 1\\
1 \times 2^{1} &= 2\\
1 \times 2^{2} &= 4\\
1 \times 2^{3} &= 8\\
1 \times 2^{8} &= 256\\
1 \times 2^{9} &= 512\\
1 \times 2^{10} &= 1024\\
1 \times 2^{13} &= \underline{8192}\\
& 9999
\end{aligned}
$$

that is, octal 23417 equals decimal 9999.

Programmers easily write routines to convert BCD to octal or binary, that is, 11111111 to 00023417 or vice-versa. These conversion routines are built into the XDS Business Language Programming System.

BUSINESS LANGUAGE WORD ORGANIZATION

Programmers involved with business data processing work primarily with BCD because most data input is in decimal digits and alphabetic characters. Thus, for this type of programming, the computer word is best considered divided into four sets of six bits each.

The way that four BCD characters are represented in a 24-bit computer word has been shown. Words, obviously, are not always four characters long, nor are numbers necessarily four digits. A man named Smith, for example, might have a salary of 50,000 dollars. Manipulation of this man's name and salary in 24-bit or four-character words becomes complicated by extensions into additional computer words.

XDS Business Language performs the otherwise tedious manipulation of the data, allowing the programmer to handle data as though no such restriction as a four-character word existed. He may work with character strings of any length. Alternatively, he can define character strings as "fields" and thereafter refer to them by name without being concerned with their length in characters.

Suppose that data were entered on a punched card that can contain up to 80 characters in its 80 columns, as follows:

Field	Card Columns	Data
LASTNAME	1 - 26	SMITH,
FIRST	27 - 36	JOHN,
AGE	37 - 38	54
SALARY	39 - 43	50000
LOCATION	44 - 63	WESTERN STEEL
START	64 - 65	53
DEGREE	66 - 69	PHD

When this information on the card is read into the computer, it is entered four characters per word:

Location	Word	Characters
A	62 44 31 63	S M I T
A + 1	30 73 60 60	H ,
A + 2	60 60 60 60	
⋮	⋮	
A + 6	60 60 41 46	J O
A + 7	30 45 73 60	H N ,
A + 8	60 60 60 60	
A + 9	05 04 60 60	5 4 ,
	etc.	

But, as above, the programmer may ignore the computer's word structure and manipulate character strings; moreover, he can define the character strings as fields.

A field definition requires the name of the area in which the data appears, its relative character position, and the number of characters to go into the field. The relative position is the high-order, or left-most, card position of the first character in a field.

If the card had been read into an area called CARDIN, the following fields could be defined:

```
LASTNAME  FIELD  CARDIN, 1,26
FIRST     FIELD  CARDIN, 27,10
AGE       FIELD  CARDIN, 37,2
SALARY    FIELD  CARDIN, 39,5
LOCATION  FIELD  CARDIN, 44,20
START     FIELD  CARDIN, 64,2
DEGREE    FIELD  CARDIN, 66,4
```

Now the data can be manipulated without the user being concerned with its orientation in memory. For example, he can refer to the field SALARY and be assured of picking up all five digits, no matter how they "spread" across word boundaries.

The programmer need not handle data in terms of "fields". He can instead treat the data as character strings without defining them as fields. For example, suppose that new information was to be added to the data read from the card in the example above. Assume that another card is read into core memory beginning at a location called CARD2. The first six characters on that card contain the name of the university from which Smith received his degree.

The user can insert this data into the CARDIN area without defining a field for it. He can move six characters, starting at position 1 in CARD2, to the area starting at position 70 in CARDIN, with the following instruction:

```
MOVE   CARD2, 1, CARDIN, 70, 6
```

In XDS Business Language arithmetic operations, the low-order, or right-most character position is to be specified. In the example above, the low-order position of the salary is 43.

An "array" is a group of consecutive words. To the Business Language programmer, this means a set of four-character-per-word words. The words, in this example, containing the 69 characters read in from the card above, constitute an array.

Two more definitions are in order: "record" and "file". There are both physical and logical records and files.

A physical record is that which is physically limited. A punched card, for instance, is a physical record. A portion of magnetic tape written between "gaps" is a physical record.

A physical file on magnetic tape may be indicated by a tape mark that can be sensed with an end-of-file test on the tape drive.

A programmer makes his own logical record or logical file. Considering a metal cabinet of student's folders, for example, it is apparent that record and file assume various meanings. On the one hand, the entire cabinet may be a file; each drawer may be a file; a set of six folders may be a file; or each folder may be a file.

One sheet of paper in a folder may be a record, or a whole folder may be a record, or a set of four folders may be a record.

Generally, logical records are sets of items and logical files are sets of logical records.

SYMBOLIC CODING

Digital computer programming may be an avocation with some readers; others may have entered this field only recently from some other discipline. If their experience has been mainly with systems like FORTRAN or COBOL, they may be unaware of the advantages of using an assembly system such as META-SYMBOL.

This discussion is directed to these individuals. It covers briefly the background information needed to understand basic symbolic programming, and in so doing, explains some of the main features of the XDS META-SYMBOL Assembler and its programming language. XDS Business Language is founded on META-SYMBOL. Business Language instructions are actually pseudo-instructions that are directed to META-SYMBOL; the assembler transforms these into in-line code and subroutine calls in the same efficient manner as it processes standard assembly language programs. Therefore, the Business Language programmer should be conversant with META-SYMBOL. For example, (1) symbolic coding may be intermixed with Business Language coding, (2) constants and other values are generated and memory areas reserved with META-SYMBOL directives, and (3) symbolic addresses or labels in a Business Language program are handled as they are in standard META-SYMBOL coding.

The basic instruction word in the XDS 900 Series Computers and the XDS 9300 is 24 bits. Each bit has a meaning to the central processor that interprets it. For example, the binary number

010 111 110 000 011 100 101 110

would be decoded by the central processor as an instruction to add the contents of the index register to the address given (i.e., 000 011 100 101 110) and move the contents of that memory location to the A register.

Programs conceivably might be coded in this absolute binary form, and would be the "purest" kind of coding.

Octal coding, while still a form of pure machine language, makes a concession to symbolism by grouping the binary digits in threes to obtain octal code (a concession brought about by the desire to make absolute binary coding a little more intelligible). That is,

010 111 110 000 011 100 101 110

becomes

$2\ 7\ 6\ 0\ 3\ 4\ 5\ 6_8$

Programs may be written using sequences of such instructions but it is difficult unless the programmer remembers the meaning of each numeric operation code, and can interpret other digits of the word readily. Let us investigate the individual elements in the octal instruction word by way of learning a more convenient method of writing instructions. In the instruction

2 76 03456

the 2_8 represents 010_2, indicating in that position that the index register is to be used in executing this instruction.

76_8 represents $111\ 110_2$, the operation code that commands the central processor to load the contents of the "effective memory location" into the A register.

03456_8 represents $000\ 011\ 100\ 101\ 110_2$, the base address that will be modified by adding the contents of the index register (because of the 2_8) to create the effective address. At this point we have determined that the instruction has specified three elements: an operation (76_8), a tag (2_8), and a base address (03456_8).

Note that 2_8, 76_8, and 03456_8 are merely octal symbols representing the true binary code the computer understands.

If we look at the octal instruction 2 76 03456 once more, we may note that 76_8 is an invariant; that is, its meaning to the computer to load the A register is fixed. Also, 2_8 fixes the bit that indicates indexing. The programmer may use these numbers only if he desires to perform the loading and indexing operation. If he wishes not to index, he can only not use the index digit; if he wishes the computer to perform another operation, he must use an operation code other than 76_8. However, the use of the address 03456_8 is an arbitrary choice of the programmer. He might have chosen any address, provided the data he wished to transfer to the A register were at that address.

Since the operation code function is fixed, a mnemonic code may be substituted for the binary computer code to make recognition quicker. Thus,

> $111\ 110_2$ becomes 76_8 becomes LDA (LOAD A REGISTER FROM MEMORY) and
>
> $101\ 101_2$ becomes 55_8 becomes ADD (ADD MEMORY TO A REGISTER)

and so on.

Whether we write $101\ 101_2$, 55_8, or ADD, the function performed by the computer is the same. If we write ADD, the assembler will translate that to $101\ 101_2$ for input to the computer.

Therefore, programs may now be written:

Memory Location	Instruction
1000	0 LDX 01015
1001	2 LDA 02345
1002	2 ADD 02772
1003	0 BRX 01002
1004	0 SKA 01016
1005	0 BRU 01012
1006	0 CNA 00000
⋮	
1015	77777774
1016	40000000

A problem arises that if we wish to insert an instruction, say after 1001, many of the following instructions would require their operand addresses to be changed, and each would have to be recorded. If we wished to place the program in another portion of memory, say at 5000, every address would have to be modified.

We can gain much flexibility by using symbolic addresses rather than absolute octal addresses. Remember 03456_8 is just a symbolic representation for the cell addressed as $000\ 011\ 100\ 101\ 110_2$ by the central processor. META-SYMBOL permits us to go one step further and give a cell a name without considering where it will be located at the time the program is executed. (We will be able to arbitrarily designate this location later if we wish.) Since a computer follows instructions serially unless a branch or skip in-instruction is given, we need be concerned only with the positions of instructions relative to each other. For example, in this sequence of instructions,

Memory Location	Instruction
01000	0 LDA 02020
01001	2 ADD 02050
01002	0 SKE 02077
01003	0 BRU 01152

we are concerned only that a cell (which we have named 02020) contains the information we need to load into the A register; we do not really need to fix its position at absolute location 02020. Similarly, the BRU to 01152 means only that we wish to branch to a sequence of instructions in another portion of memory. Because of the absolute nature of octal addressing, we needed to choose a location arbitrarily, namely 01152. However, we may find later that this is an unsuitable location, in which case we will have to change 01152 to some other location, and recode (rewrite the addresses) of all instructions moved from 01152 to the new location.

The solution to difficulties of this kind is to name all cells symbolically with alphanumeric location names rather than absolute octal location names. After the program is written, we (or the assembler) will arbitrarily assign addresses to these alphanumeric location names (labels) in order to preserve the sequential nature of the instructions. The preceding example might then be coded:

Memory Location	Instruction
01000	0 LDA TABLE
01001	2 ADD CONST
01002	0 SKE M1
01003	0 BRU NEWLOC

We know that TABLE is somewhere in memory. When TABLE is given an arbitrary location designation (say 04020),

0 LDA TABLE

will become

0 LDA 04020

and eventually

0 76 04020

when the operation code 76 is substituted for LDA.

Similarly, CONST, M1, and NEWLOC are alphanumeric designators (labels) for cells that will be assigned numeric locations later.

At this point, we are still tied to the arbitrary designation of cells 01000, 01001, etc., to contain these instructions. Instead of the octal designation 1000, let us give it an alphanumeric designation with the label BEGIN. We (i.e., the assembler) will assign a location to BEGIN later.

Our program now appears, as follows:

Memory Location	Instruction
BEGIN	0 LDA TABLE
	2 ADD CONST
	0 SKE M1
	0 BRU NEWLOC

Notice that BEGIN labels only the first instruction of the block. It is unnecessary to label the ADD, SKE, and BRU instructions since the computer's operation is serial, and it is assumed that whatever location is designated for BEGIN, following instructions will have successive locations. (We can refer to them later in the program, if we choose, as BEGIN+1, BEGIN+2, and so on.)

If we later assign 05000 to be the location that BEGIN labels, ADD will be at 05001, SKE will be at 05002, etc. Our program might now take on the appearance:

Location (LABEL)	Instruction
BEGIN	0 LDA TABLE
	2 ADD CONST
	0 SKE M1
	0 BRU NEWLOC
	:
	(sequence of instructions)
	:
TABLE	00000001
CONST	77777776
M1	77777777
	:
NEWLOC	0 CNA
	etc.

Note that the sequential quality of the original program is retained, but that now we can insert as many instructions (cards) between M1 and NEWLOC as we wish without changing the meaning of NEWLOC as a sequential designator for a new block of instructions in memory. When this symbolic program is assembled into a working program, the assembler will substitute numeric quantities such that no matter where in memory NEWLOC may be, BRU NEWLOC will cause a branch to that location in memory. This property is called relocatability of the program in memory.

Note that we now call the location field the LABEL field, since we are using symbolic alphanumeric labels instead of octal addresses to represent locations. One more modification: Since there is no index on

```
0 LDA  TABLE
```

we can write it

```
  LDA  TABLE
```

But

```
2 ADD  CONST
```

we can write

```
  ADD  CONST,2
```

The assembler will set the required index bit. If indirect addressing is required, we can write

```
ADD    *CONST
```

If both indirect addressing and indexing are desired, we can write

```
ADD    *CONST,2
```

We can arbitrarily locate a block of instructions at any location in memory by using the ORG directive. Thus,

```
        AORG   01000
BEGIN   LDA    TABLE
        ADD    CONST,2
```

will cause BEGIN to be given the value 01000, and all other labels in that block to be numbered accordingly. At load time the block of instructions with starting label BEGIN will be located at 01000.

It is alo permissible to write

```
BEGIN   LDA    TABLE
        etc.
```

without the AORG, and later to designate by a parameter given to the loader where the block of instructions starting with BEGIN is to be located (relocated). The loader is a routine in the MONARCH/META-SYMBOL system that loads assembled programs into memory for execution.

META-SYMBOL provides some other convenient features. If we write

```
CONST  DATA   01000
```

we are instructing the assembler to construct a DATA word, 00001000, that will be put into the cell designated CONST. Therefore,

```
LDA    CONST
```

will result in the quantity 00001000 being transferred from its memory location to the A register at execution time.

If we say

```
CONST  EQU    01000
```

we are informing the assembler that CONST is to have the value 01000. Since CONST is a LABEL (alphanumeric designator for a cell), we wish the cell to be designated as 01000. That is, CONST must be cell 01000. Then

```
LDA    CONST
```

would cause the contents of cell CONST (cell 01000) to be loaded into the A register.

To show it another way, if

```
CONST  EQU    01000
```

and

```
CONST  DATA   05000
```

Then cell 01000 would contain 00005000, and

```
LDA    CONST
```

would result in the quantity 00005000 being loaded into the A register. DATA puts a quantity into a cell. EQU fixes the location of a cell.

For instance,

```
INT    EQU    032
INT    BRM    INTSUB
```

would cause a BRM INTSUB to be placed in location 32, for processing an I2 interrupt. If INTSUB is located at 07000, the instruction 0 43 07000 (BRM INTSUB) will be placed in location 032 at load time.

The following would have the same result:

```
AORG   032
BRM    INTSUB
```

Labels must contain alphabetic or numeric characters only. No special characters (*, (, +, etc.) may be used. See the SYMBOL/META-SYMBOL Manual (XDS Publication 90 05 06) for a description of labels.

In summary, the assembler arbitrarily assigns values to labels according to: their relative position in the sequence of instructions and the arbitrary assignment of location numbers by AORG or EQU directives. Therefore, a label is <u>defined</u> (assigned a numerical value by the assembler) by one of two methods: (1) appearing in the label field at some point in the program to establish a value according to its relative position in the sequence, (2) using EQU to establish a definite value for the label. Each label referenced in the operand field of an instruction, as in

```
LDA    XRAY
ADD    M1
```

must be <u>defined</u> at some point in the program. For example:

```
XRAY   EQU    01121
XRAY   DATA   00050
```

defines XRAY as a cell at location 01121 containing the value 00000050.

Finally, remember that a label is unique; that is, it can be the designator for only one memory location. Example:

```
QR      DATA
        :
        .
        LDA     QR
        STA     QR
QR      PZE
```

is illegal, since the assembler does not know which QR is really being referred to in the LDA and STA instructions. Therefore, a label must be defined only once in any program.

Reference should be made to the sample program shown assembled at the end of this section. The first two columns on the sample show the assembled, machine language equivalent of the symbolic code written by the programmer. The third through sixth columns show this symbolic code. The programmer has punched this information one line to a card.

Note that all machine code is in octal form. Since the machine code contains references to other memory locations, they must be octal as well. Thus, sequential locations in memory are numbered octally. One implication of this is that certain conventions must be observed in setting up data words: the assembler must be told whether they are octal or decimal or BCD values.

A zero in front of a number indicates that it is an octal number; absence of a zero indicates that it is decimal and will be converted to octal by the assembler; single quotes (and other conventions) signal that the word is BCD and is not to be converted but coded.

The first directive, AORG 0200, says that the program is to origin or begin at the 200th (octal) word in memory.

Labels start in column 1 on the cards. If an asterisk appears in card column 1, the card is a comment card and is ignored by the assembler except for being printed on the program listing. Also ignored by the assembler are the comments shown in the sixth column of the listing (unless this information, beginning in card column 36, happens to be a continuation of an operand field.)

Column 1 of the listing shows the memory location of the word in column 2. This word is generated from the information in columns 3, 4, and 5; it is the octal number that will actually reside in the computer's memory when the program is executed.

The program is executed sequentially unless told to change direction by a branch instruction. The only instructions used in the example are LDA (16), STA (76), HLT (00) and BRU (01). Their functions are described later.

In memory locations 200-215 is the actual program to be executed. Memory locations 216 through 226 contain reserved areas or data. They are never "executed" since the BRU instruction causes a branch out of the sequential order before they are reached.

The END directive tells the assembler that everything required for the assembly has been transmitted to it.

In memory locations 227-231 are words generated by literals, that is, items in the fifth column preceded by an = sign. These are explained in a later paragraph.

Note in column 5 an asterisk preceding a symbolic address. This specifies indirect addressing, which also is explained later.

Usually, an instruction like LDA (which means load the A register of the computer) loads A with the contents of the memory location specified.

In the same way an instruction like STA (which means store the A register) stores A in the location specified.

When LDA CONS is executed, the contents of CONS are placed in the A register. The next instruction is then executed. The STA LOC1 causes the contents of the A register to be stored in LOC1. Thus, in the example, 00001000 was put into location 217. It is important to remember that nothing has changed in location 200, 201, or 216. The contents of memory are affected only by a storing operation.

Placing an asterisk in front of an address notifies the assembler that it must set a signal to be recognized at execution time. The asterisk causes the "indirect address" bit to be set in the word. At execution time, this alerts the computer to use indirect addressing. If STA *LOC1 is used instead of STA LOC1, the computer is instructed to go to LOC1 to find the address into which to store A. Since LOC1 contains 00001000, the A register will be stored in location 1000 instead of in 217. Indirect addressing can be continued at as many levels as desired (see computer reference manual), but this example indicates how it works.

The instructions are executed one after another and the program continues until it reaches the HLT instruction. At this point, the computer halts. When the operator resets it to "RUN", the next instruction in sequence is executed. In the example, the next instruction is a branch back to the beginning of the program. The program will go to START, that is, it will re-execute.

Literals are a coding convenience. They save the programmer from setting up DATA words. The equal sign in the address portion of the instruction tells the assembler to handle the data following the equal sign as though it had been defined with a DATA directive.

The assembler finds the first location available at the end of the program and generates the data word. The reference to the word generated is then set up. In the instruction LDA=1, a word containing 00000001 is put into the first available memory location at the end of the program, that is, into 227, and the instruction generated is "load A from 227." When the instruction is executed, the contents of 227, that is 00000001, is placed in A.

In LDA =LOC1, the programmer wishes to put the address of LOC1 in the A register. The assembler generates the address in the next available location, 230, as though it had been defined as DATA LOC1. Thus, the instruction generated is "load A with the contents of 230." that is, 00000217.

In LDA ='ENDS', the word is generated as though it had been defined as DATA 'ENDS'.

The first part of this manual presents more information on the use of literals and the generation of constants with DATA, BCD, and TEXT. The few examples given here show some of their uses.

Note that blanks are coded as 060's or 012's, depending on the directive given. Punched cards, magnetic tape, and the printer use the 060 code for blanks, but the typewriter uses the 012 code.

One final note on the preceding discussion should be given. In an attempt to acquaint the reader quickly with the basic ideas of symbolic programs, many unequivocal statements have been made. There are exceptions to these rules. Familiarity with the assembler and the experience of trying variations will broaden the programmer's repertoire of "exceptions" allowed by the assembler. Part of the challenge of using an assembler such as META-SYMBOL is its exceptional flexibility, which allows the programmer to investigate increasingly sophisticated uses.

```
                    *THIS PROGRAM ILLUSTRATES THE ASSEMBLING
                    *OF A SYMBOLIC PROGRAM.
                    *
                          AORG  0200
200   01600216      START LDA   CONS      MOVE ONE DATA WORD
201   07600217            STA   LOC1
202   01600227            LDA   =1
203   07602000            STA   LOC2
204   01600230            LDA   =LOC1
205   47600216            STA   *CONS
206   07600220            STA   LOC1+1
207   01600231            LDA   ='ENDS'
210   07600012            STA   10
211   07600060            STA   060
212   01600221            LDA   FIN
213   07600216            STA   $+3
214   00000000            HLT
215   00100200            BRU   START
                    *
216   00001000      CONS  DATA  01000
217                 LOC1  RES   2
221   60622462      FIN   TEXT  4, SDS
                    LOC2  EQU   02000     USED ONLY BY ASSEMBLER
222   62246260            TEXT  4, SDS
223   00622462            DATA  'SDS'
224   60622462            DATA  ' SDS'
225   62246212            BCD   4, SDS
226   12622462            BCD   4,  SDS
                          END
227   00000001
230   00000217
231   25452462
```

When this program is executed, the following words in core memory will be changed by the store instructions:

Location	Contains
217	00001000
2000	00000001
1000	00000217
220	00000217
12	25452462
60	25452462
216	60622462

This program was assembled on the XDS 9300 Computer. Some machine language octal formats are different for XDS 900 Series Computers.

ASSEMBLY LISTING FORMAT

XDS 9300

The elements of the octal listing output are as follows, except as given within the particular description:

I	=	Indirect address
X	=	Indexing
OP	=	Operation code
ADD	=	Address
DDDD	=	Eight octal digits

An all-zero octal digit pattern is shown under each element, to indicate its form. The number of binary digits equivalent to the octal representation is given on the third line.

Standard Instructions

I	X	OP	ADD
0	0	00	00000
1	2	6	15

Shift Instructions

I	X	OP	SHIFT NO
0	0	0000	000
1	2	12	9

Flag Set/Test

0	OP	SELECT BITS
0	000	0000
3	9	12

COPY Instructions

1.

0	OP	N	BYTES	REG
0	00	0	000	00
3	6	1	8	6

2.

X	OP	BYTES	REG
0	00	0	0000
3	6	3	12

DATA Instruction

DDDD
00000000
24

DED Instruction

DDDDDDDD
0000000000000000
48

XDS 900 Series

The elements of the octal listing are defined the same way as for the XDS 9300.

Standard Instructions

X	OP	I	ADD
0	00	0	00000
3	6	1	14

Shift Instructions

1.

X	OP	I	ADD
0	00	0	00000
3	6	1	14

2.

X	OP	ADD
0	0000	000
3	12	9

Register Change Instructions

X	OP	ADD
0	00	00000
3	6	15

COPY Instruction

X	OP	BYTES	REG
0	00	00	0000
3	6	5	10

DATA Instruction

DDDD
00000000
24

DED Instruction

DDDDDDDD
0000000000000000
48

PZE Instruction

1.

X	00	I	ADD
0	00	0	00000
3	6	1	14

2.

X	00	ADD
0	00	00000
3	6	15

CALLING SEQUENCE GENERATION FOR BUSINESS INSTRUCTIONS

Business Instruction	Word	Calling Sequence	Description
MOVEWD (MVW)	1	BRM B\S1	Link to subroutine "B\S 1"
	2	Address of source array	
	3	Address of destination array	
	4	Word count of number of words moved	

Comments: The above calling sequence is generated only when either

a. A source or destination address is indirectly addressed and the word count is greater than one.

or b. A source or destination address is indexed and the word count is greater than three.

or c. A source or destination address is indexed and the word count is greater than two, with the operand indicating that the index register should be saved.

or d. The word count is a "run-time" variable.

Note: In this case, an LDA/STA is generated preceding the calling sequence.

In all other cases, an LDX/LDA/STA/BRX loop is generated (with conditional saving and restoring of the index register ... STX ... LDX) or a sequence of from one to three successive LDA/STAs is generated.

Business Instruction	Word	Calling Sequence	Description
BRACEQ (BRACNE)	1	BRM B\S8	Linkage to offset routine
	2	Base location for referencing character argument	
	3	Character offset, or location of same (with forced indirect bit) relative to word 2	

Note: The above three words are conditionally generated, as described below.

Business Instruction	Word	Calling Sequence	Description
	4(1)	BRM B\S2	Linkage to BRACEQ/BRACNE routine
	5(2)	Computed location of character argument (assembly time), or zero	
	6(3)	Shift necessary to left justify character argument or zero	
	7(4)	Count of packed words containing character list	
	8(5)	Packed word 1	First four characters in list
	⋮	⋮	
	8(n)	Packed word N	
	n+1	BRU(BRM) to branch address of BRACNE or BRU $+2 if BRACEQ	(Only generated if BRACNE) (Only generated if BRACEQ)
	n+2	BRU(BRM) to branch address for BRACEQ	(Only generated if BRACEQ)
		Continuation of source program	

Comments: The linkage to "B\S8" (words 1-3 above) is only generated when

a. The character offset is a "run-time" variable.

or b. The base location is indexed or indirectly addressed.

In such a case, words 5 and 6 will be zero prior to object program execution.

Business Instruction	Word	Calling Sequence	Description
UNPACK (UNPACKL)	1	BRM B\S3	Subroutine linkage
	2	Number of characters to be unpacked	
	3	Address of packed array	

Business Instruction	Word	Calling Sequence	Description
UNPACK (UNPACKL) (cont.)	4	Address of unpacked character word	
	⋮	⋮	
		Address of last unpacked character word	

Comments: There is one exception to the above calling sequence generation; namely, when only one unpacked character word address is specified. In this case, four in-line instructions are generated: LDA/ETR/MRG/STA.

Business Instruction	Word	Calling Sequence	Description
PACK (PACKL)	1	BRM B\S4	Subroutine linkage
	2	Number of characters to be packed	
	3	Address of packed array	
	4	Address of first unpacked character word	
	⋮	⋮	
		Address of last unpacked character word	
UNPACKR	1	BRM B\S9	Subroutine linkage
	2	Right-justified "fill character" (060 if not specified)	
	3	Address of packed array	
	4	Address of unpacked array	
	5	Number of characters to be unpacked	
PACKR	1	BRM B\S10	Subroutine linkage
	2	Address of packed array	
	3	Address of unpacked array	
	4	Number of characters to be packed	
COMPARW (CPW)	1	BRM B\S6	Subroutine linkage
	2	Number of words to be compared	
	3	Address of first array to be compared	
	4	Address of second array to be compared	

Comments: If the number of words to be compared is a "run-time" variable, the above calling sequence is preceded by an LDA/STA.

COLLATE

Note: With 'XDS' as the operand, an LDA/STA is generated. With 'BDP' as an operand, a MIN (MPO) is generated.

BREQ (BE)

Note: This generates an LDA/SKA/BRU followed by a BRU (BRM if the operand is in parentheses).

BRNE (BU)

Note: This generates an LDA/SKA followed by a BRU (BRM if the operand is in parentheses).

BRHI (BH)

Note: This generates an LDA/SKG followed by a BRU (BRM if the operand is in parentheses).

BRLO (BL)

Note: This generates an SKN/BRU followed by a BRU (BRM if the operand is in parentheses).

Business Instruction	Word	Calling Sequence	Description
BAOV			

Note: This generates an SKN followed by a BRU (BRM if the operand is in parentheses).

Business Instruction	Word	Calling Sequence	Description
MOVE	1	BRM B\S18	Linkage to character setup routine
	2	Word offset - 1 of sending (8 bits), word offset - 1 of receiving (8 bits), character - 1 (8 bits)	
	3	Opcode = relative byte position (0-3), address = address of sending	
	4	Opcode = relative byte position (0-3) address = address of receiving	
	5	BRM B\S12	Linkage to character move routine

Comments: The above is the normal form of the calling sequence generated; but, if the nature of the operand is such that it is word-oriented, a MOVEWD will be generated (see MOVEWD generation).

Business Instruction	Word	Calling Sequence	Description
MOVEIZ	1	BRM B\S18	Linkage to character setup routine
	2	Word offset - 1 of sending (8 bits), word offset - 1 of receiving (8 bits), character - 1 (8 bits)	
	3	Opcode = relative byte position (0-3), address = address of sending	
	4	Opcode = relative byte position (0-3), address = address of receiving	
	5	BRM B\S13	Linkage to MOVEIZ routine
MOVEED (MCE, EDIT)	1	BRM B\S18	Linkage to character setup routine
	2	Word offset - 1 of sending (8 bits), word offset - 1 of receiving (8 bits), character - 1 (8 bits)	
	3	Opcode = relative byte position (0-3), Address = address of sending	
	4	Opcode = relative byte position (0-3), address = address of receiving	
	5	BRM B\S14	Linkage to MOVEED routine
	6	Optional punctuation parameters (see below)	

Comments: The representation of the punctuation in word 6 is as follows: a one at bit 5 indicates trailing minus sign; a one at bit 6 indicates trailing "CR"; a one at bit 7 indicates comma insertion; a one at bit 8 indicates floating dollar sign; the decimal scaling factor is in bits 16-23.

Business Instruction	Word	Calling Sequence	Description
COMPARE	1	BRM B\S18	Linkage to character setup routine
	2	Word offset - 1 of first operand (8 bits), word offset - 1 of second operand (8 bits), character count - 1 (8 bits)	
	3	Opcode = relative byte position (0-3), address = first operand address	
	4	Opcode = relative byte position (0-3), address = second operand address	
	5	BRM B\S11	Linkage to COMPARE routine

Comments: If the routine of the operand is a word-oriented operation, a COMPARW will be generated (see COMPARW). Character count - 1 in the right-most 8 bits of word 2 is the shorter, if two counts are specified.

Business Instruction	Word	Calling Sequence	Description
CLEARCH	1	BRM B\S18	Linkage to character setup routine
	2	Zero (8 bits), word offset - 1 of operand (8 bits), character count - 1 (8 bits)	
	3	PZE $	
	4	Opcode = relative byte position (0-3) of operand, address = address of operand	
	5	LDB	
	6	BRM	
	7	SKR	
	8	BRU	
	9	LDX	

Comments: If the operand is a word-oriented operation, a FILL will be generated (see FILL generation).

FILLCH

Note: Generation here is the same as that of CLEARCH, except that the address of the LDB is representative of a literal reference to the fill character specified.

Comments: If the operand is a word-oriented operation, a FILL will be generated (see FILL generation).

BLANKCH

Note: Generation here is the same as that of CLEARCH, with the exception that the address of the LDB is to a different literal.

Comments: If the operand is a word-oriented operation, a FILL will be generated (see FILL generation).

FILL, CLEAR, BLANK

Note: These generate various combinations of in-line code, ranging from 2 to 11 words in length. The generated code represents either an STA loop (the A register containing zero, 060's, or the FILL character) or successive STAs. Additional code generation is brought about by the conditional specification of the index register being saved, the word count being a "run-time" variable, and/or the operand being indexed or indirectly addressed.

SORT (SORTDS, SORTBIN, SORTBDS)

Note: Any of the various SORT calls causes the generation of various combinations of in-line code prior to a six-word calling sequence to the SORT routine. For descending sorts, an MIN (MPO for the XDS 9300) is the first word generated; for ascending sorts an LDA/STA. Following this, for binary sorts, an LDA/STA is generated. Continuing, if relocation of the sorted table is specified, conditional code is generated as follows: if the item length and number of items are assembly-time variables, a load/store loop is generated ... STX/LDX/LDA/STA/BRX/LDX. If either item length of the number of items is a "run-time" variable, code generation is LDA/MUL/RSH/STB followed by BRM B\S1/address of unsorted table/address of relocated sorted table/computed total number of words in table (i.e., the four-word calling sequence to the MOVEWD routine is generated). Notice that the unsorted table is relocated prior to the actual sorting, leaving the unsorted table undisturbed. Finally, the SORT calling sequence is generated and appears as:

	1	BRM B\SORT	Linkage to SORT routine
	2	Address of the table to be sorted	
	3	Number of items (or address of same, with forced indirect bit)	
	4	Item length (or address of same)	
	5	Relative key position (or address of same, with forced indirect bit)	
	6	Key length (or address of same)	

Business Instruction	Word	Calling Sequence	Description
READTP	1	BRM B\I10	Linkage to subroutine
	2	BCD indicator, logical unit, address of tape image (see below)	
	3	Word count	

Comments: Entries in word 2 consist of

a. A zero at bit 5, indicating the BCD mode.

b. Bits 6-8 indicate the logical unit.

c. Address, index, and indirect bits (per XDS 900 Series or XDS 9300) representing the address of the first word of the tape image.

Note: The calling sequence is preceded by an LDA/STA if a "run-time" count is specified.

RTPBIN

Note: Code generation is identical to READTP, except that a one at bit 5 in word 2 indicates the binary mode.

WRITETP, WTPBIN	1	BRM B\I11	Linkage to subroutine
	2	BCD indicator, logical unit, address of tape image (see READTP generation)	
	3	Word count	

Comment: As in READTP and RTPBIN, the calling sequence can be preceded by an LDA/STA if a "run-time" count is specified.

BACKSPACE, SKIPTAPE	1	BRM B\I7	Linkage to subroutine
	2	(see below for bit representation)	

Note: The contents of word 2 are represented as:

a. Bit 3 = 1 indicates that files are to be backspaced or skipped.
= 0 indicates that records are to be backspaced or skipped.

b. Bit 4 = 1 indicates backspacing.
= 0 indicates skipping (forward).

c. Bits 6-8 indicate the logical unit.

d. Bits 9-23 indicate the number of records or files to be backspaced or skipped.

PRINT, PRT120	1	BRM B\I1	Linkage to subroutine
	2	(See below for bit representation)	

Note: The contents of word 2 are represented as:

a. Bit 3 = 1 indicates that a 120-character line image is to be printed (PRT120).
= 0 indicates that a 132-character line image is to be printed (PRINT).

b. Bits 4-6 indicate the number of lines to be spaced or the format channel specified.

c. Bit 8 = 1 indicates vertical spacing by a specified number of lines.
= 0 indicates skipping to the format channel specified.

d. Address, index, and indirect bits indicate the location of the line image.

UPSPACE, SKPCHN, RESTORE	1	BRM B\I2	Linkage to subroutine
	2	(See below for bit representation)	

Note: The contents of word 2 are represented as

a. Bit 8 = 1 indicates upspacing by the specified number of lines.
= 0 indicates skipping to the format channel specified.

Business Instruction	Word	Calling Sequence	Description
UPSPACE, SKPCHN, RESTORE (cont.)			

b. Bits 4-6 indicate the number of lines to be upspaced or the format channel to which a skip is to be made.

READCD	1	BRM B\I3	Linkage to subroutine
	2	Address of card image to be read	
PUNCH	1	BRM B\I4	Linkage to subroutine
	2	Address of card image to be punched	
TYPEIN	1	BRM B\I5	Linkage to subroutine
	2	Address of image to be typed in	
TYPE	1	BRM B\I6	Linkage to subroutine
	2	Address of image to be typed out	
REWIND	1	BRM B\I9	Linkage to subroutine
	2	Bits 6-8 indicate logical unit	
WTMARK, WTM	1	BRM B\I8	Linkage to subroutine
	2	Bits 6-8 indicate logical unit	

BPRF, BPOV, BLCD, BTMK,
BBTP, BETP, BFPT, BCER

Note: All of these generate an SKR followed by a BRU or BRM to the conditional branch address.

CALLING SEQUENCE GENERATION FOR DECIMAL ARITHMETIC

Business Instruction		Word	Calling Sequence
DADD or DSUB	E1, LO1, CC1	1	CLR (for DADD) or LDA = -1 (for DSUB)
		2	BRM B\S23
		3	PZE E1
		4	Sign = 1, bits 1-11 = CC1, bits 12-23 = LO1
DADD or DSUB	E1, LO1, CC1, E2, LO2, CC2	1	CLR (for DADD) or LDA = -1 (for DSUB)
		2	BRM B\S23
		3	PZE E1
		4	Sign = 0, bits 1-11 = CC1, bits 12-23 = LO1
		5	PZE E2
		6	Sign = 1, bits 1-11 = CC2, bits 12-23 = LO2
DADD or DSUB	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3	1	CLR (for DADD) or LDA = -1 (for DSUB)
		2	BRM B\S23
		3	PZE E1
		4	Sign = 0, bits 1-11 = CC1, bits 12-23 = LO1
		5	PZE E2
		6	Sign = 0, bits 1-11 = CC2, bits 12-23 = LO2
		7	PZE E3
		8	Sign = 0, bits 1-11 = CC3, bits 12-23 = LO3
DMUL or DDIV	E1, LO1, CC1, E2, LO2, CC2	1	BRM B\S21 (for DMUL) or BRM B\S22 (for DDIV)
		2	PZE E1
		3	Sign = 0, bits 1-11 = CC1, bits 12-23 = LO1
		4	PZE E2
		5	Sign = 1, bits 1-11 = CC2, bits 12-23 = LO2
DMUL or DDIV	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3	1	BRM B\S21 (for DMUL) or BRM B\S22 (for DDIV)
		2	PZE E1
		3	Sign = 0, bits 1-11 = CC1, bits 12-23 = LO1
		4	PZE E2
		5	Sign = 0, bits 1-11 = CC2, bits 12-23 = LO2
		6	PZE E3
		7	Sign = 0, bits 1-11 = CC3, bits 12-23 = LO3
BCDBIN	E1, HC1, CC1, E2	1	BRM B\S32
		2	PZE E1
		3	Bits 0-11 = CC1, bits 12-23 = HC1
		4	PZE E2
BINBCD	E1, HC1, CC1, E2	1	BRM B\S31
		2	PZE E1
		3	Bits 0-11 = CC1, Bits 12-23 = HC1 + CC1 - 1
		4	PZE E2

BUSINESS LANGUAGE ASSEMBLIES

The input deck for Business Language assemblies may be divided into three portions: (1) control cards preceding the program to be assembled, (2) the program to be assembled, and (3) control cards following the program.

The XDS MONARCH Reference Manual or the XDS MONITOR Reference Manual contains a detailed description of card functions. The following material outlines some working possibilities:

CONTROL CARDS PRECEDING THE PROGRAM

A. △JOB – must be on XDS 9300; may be on XDS 900 Series.

B. △ASSIGN – may be one or a series of cards to assign all necessary I/O devices. See applicable computer reference manual for a list of possible I/O devices.

XDS 900 Series

Must ASSIGN X1=MT1W, S=MT0W

ASSIGN necessary options

Possible assignments to check – these are also some of the parameters for the META card (see METAB for the respective computers, below):

SI	Symbolic Input
EI	Encoded Input
EO	Encoded Output
BO	Binary Output
LO	Listing Output

A standard ASSIGN card might be:

△ASSIGN S=MT0W, X1=MT1W, X2=MT2W, SI=CR1W, EI=CR1W, EO=CP1W, BO=CP1W, LO=LP1W.

XDS 9300

These standard assignments are already contained within MONITOR:

EI, SI, EO, BO on cards, X1 to MT1A, X2 to MT2A, GO to MT2A, LO to LP1A

Only deviations from these need be assigned.

C. △METAB910 for object output to execute on XDS 910
△METAB920 for object output to execute on XDS 920
△METAB93H for object output to execute on XDS 9300

Parameters on these cards will be:

SI	If symbolic deck or symbolic correction to encoded deck
EI	Encoded Input
EO	Encoded Output
BO	Binary Output
GO	(XDS 9300 only – for assemble, load, and execute – GO is MT2A unless otherwise assigned)
LO	Listing Output
CONC	Concordance listing
SET	Must be included as a parameter on any METABxxx card

EXAMPLES

XDS 900 Series

△METAB920 SI, EI, EO, LO, BO, CONC, SET

Accepts SI and EI, produces EO, BO (for XDS 920), and LO with CONC listing.

XDS 9300

△METAB93H EI, BO, LO, GO, CONC, SET

Accepts EI, produces BO, LO with CONC, and creates a GO tape that may be loaded and executed.

PROGRAM DECKS

Symbolic deck or symbolic corrections to encoded deck. Must be terminated by △EOF card or EOF condition on SI medium.

Encoded deck – the END card of this deck terminates. (Do not use △EOF here.)

Note: An EXTEND card must be present as the first card to be assembled.

CONTROL CARDS FOLLOWING THE PROGRAM

XDS 900 Series

△ENDJOB resets processor error switch set by assembly errors. Processor error switch must be reset for loading to take place. The following is optional for assemble, load, and execute:

△ASSIGN BI = (as determined by user)
△REWIND MTNW (if BI=MTNW)
△LOAD Address, $\begin{cases}\text{GO}\\\text{TGO}\\\text{STOP}\\\text{TSTP}\end{cases}$

Even if program is absolute (AORG), a load address must be given to permit loading of any necessary relocatable POPs or subroutines from the library.

GO, TGO, STOP, TSTP all result in: (1) loading of programs from BI until an END with transfer address is encountered, (2) a search of the library for needed subroutines or POPs, and (3) a branch to indicated transfer address (not to load address).

GO — process is continuous, no halts; map of unsatisfied references.

TGO — GO and MAP of external definitions and unsatisfied references.

STOP — stops at each END record (for loading of several paper tapes, etc.)

TSTP — STOP with MAP. Set BPT1 for MAP on LP.

If BI=MTNW, then ΔLOAD Address, TGO, results in continuous loading and MAP.

A typical input deck for the XDS 900 Series would consist of these cards (see Figure 3):

ΔJOB (optional)
ΔASSIGN
ΔMETABxxx

optional $\begin{cases}\text{SI corrections} \\ \Delta\text{EOF}\end{cases}$ or Symbolic source ΔEOF
encoded

ΔENDJOB
ΔREWIND (if BI on MT)

ΔLOAD Address, $\begin{cases}\text{GO} \\ \text{STOP} \\ \text{TGO} \\ \text{TSTP}\end{cases}$

<u>XDS 9300</u>

ΔLOAD $\begin{cases}\text{X} \\ \text{XM} \\ \text{XR}\end{cases}$, MAP.

MAP — gives full map of memory allocation
X — execute if no errors
XM — execute if minor errors
XR — execute regardless of severity of errors

Results in rewind of and load from GO tape, search of library for necessary routines, and execution. If input is not to be from GO tape, see XDS MONITOR Reference Manual for details.

A variety of debug features (SNAPS, PATCHES, DUMPS) are available; see XDS MONITOR Reference Manual.

A typical input deck for the XDS 9300 would consist of the following (see Figure 4):

ΔJOB
ΔASSIGN (only if parameters not on standard assignments)
ΔMETABxxx (GO if load and execute desired)

optional $\begin{cases}\text{SI corrections} \\ \Delta\text{EOF}\end{cases}$ or Symbolic source ΔEOF
encoded

ΔLOAD $\begin{cases}\text{X} \\ \text{XR} \\ \text{XM}\end{cases}$, MAP from GO and executes. Prints map

ΔDUMP, etc., for debug (optional)
ΔFIN end of run
or
ΔJOB for next job in batch processing

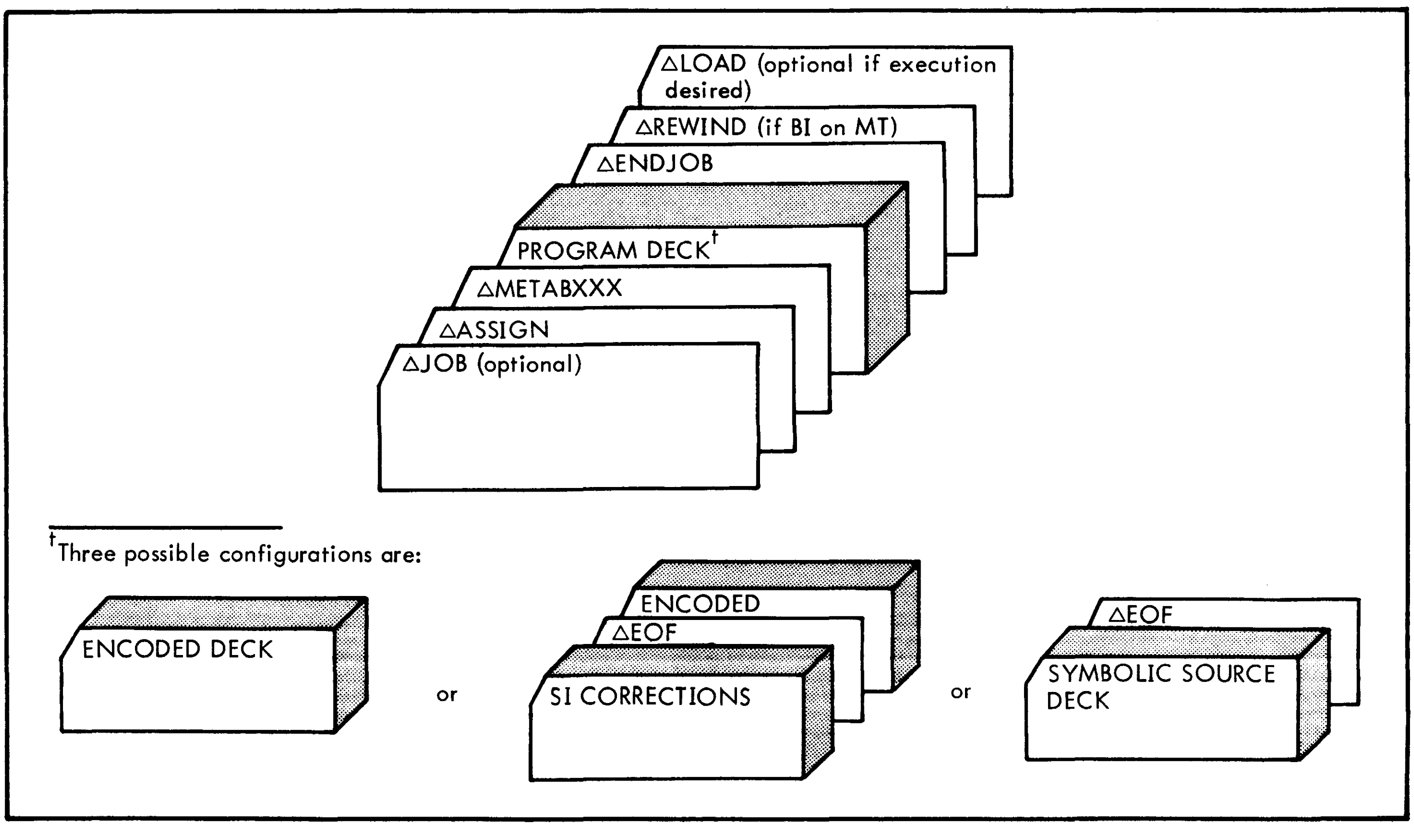

Figure 3. Sample Input Deck for XDS 900 Series

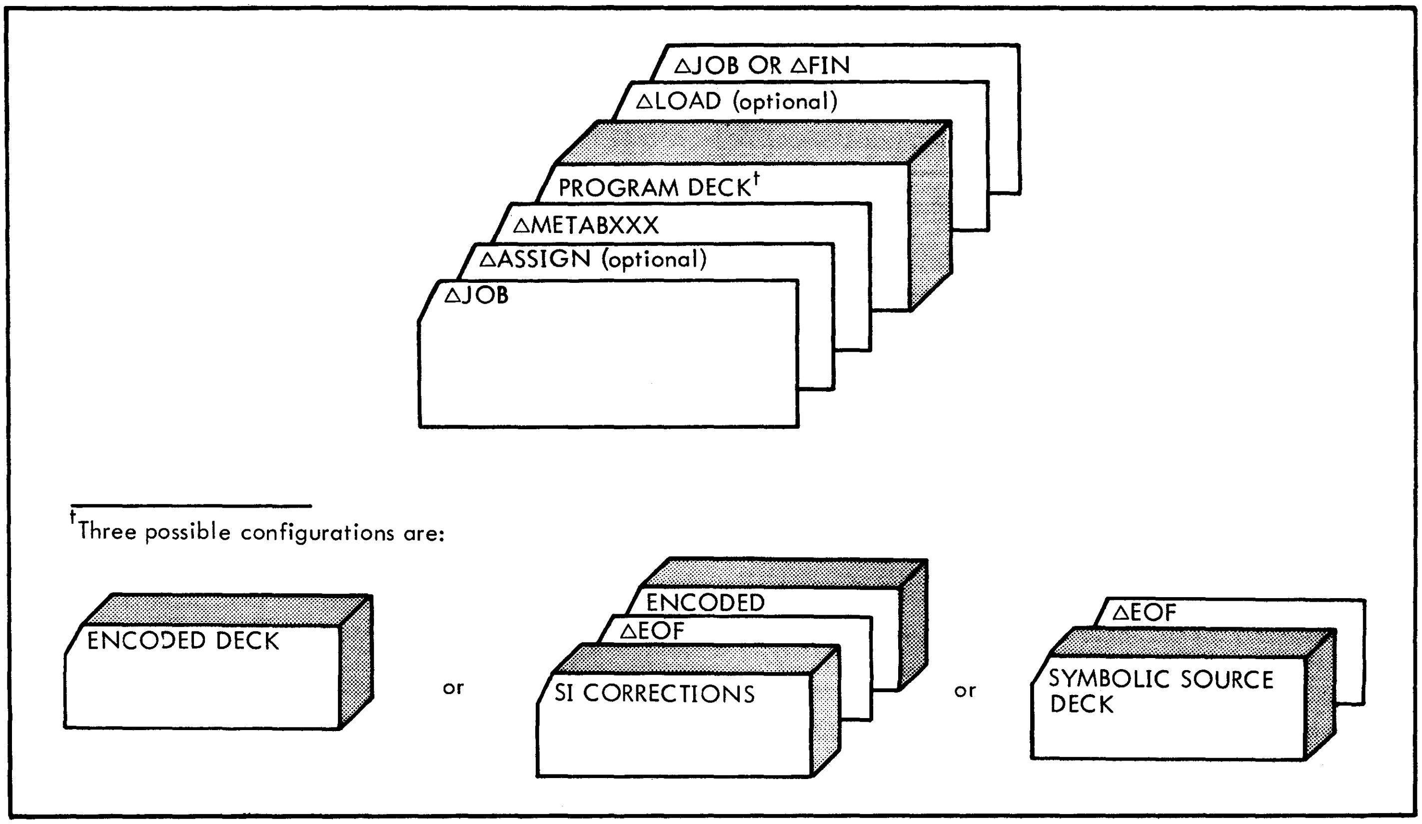

Figure 4. Sample Input Deck for XDS 9300 Computer

MAKING SYMBOLIC CHANGES TO ENCODED PROGRAMS

Symbolic changes are made by a series of insertions and deletions represented by specially formatted records, usually symbolic cards. The encoded program is interpreted as a series of logical lines, as indicated by the line numbers given on the program assembly listing. Note that, via the continuation feature, two or more cards may be considered as one logical line.

SYNTAX OF INSERT, REPLACE, AND DELETE

Insert S1, ..., SN following line α.

```
+α
 S1
 S2
 .
 .
 .
 SN
```

Delete lines α through β inclusively.

```
+α,β
```

Note that, if $\alpha = \beta$, one line is deleted.

Replace lines α through β inclusively with S1, S2, ..., SN.

```
+α,β
 S1
 S2
 .
 .
 .
 SN
```

Notes:

1. The + must be in Column 1.
2. α and β are decimal integers.
3. The first space terminates the scan of the + card.
4. SK (K=1, N) is a symbolic card for assembly.
5. To insert before the first line, use:

```
+0
 S1
 S2
 .
 .
 .
 SN
```

DECK STRUCTURE FOR SYMBOLIC CHANGES TO ENCODED DECK

ΔMETAXXXX SI, EI, etc.

+... } symbolic correction cards (the first card following the ΔMETA must be a +... card).

ΔEOF
} encoded deck

Note: It is impossible to "merge" or "juxtapose" encoded decks.

ASSEMBLY - TIME OPERATIONS

△METABXXX

The assembler to be used for a particular machine is selected via the "△METABXXX" card (statement). The XXX refers to the selected machine: XDS 910, 920, 930, 925, or 93H for the XDS 9300.

The parameters used on this card are the usual ones, with the addition of "SET". SET must be one of the parameters on every Business Language assembly. With the pertinent card equipment assignments, a typical "△METAXXX" assembly request for the XDS 920 Computer might be:

△METAB920 EI, SI, LO, BO, SET.

which directs the MONARCH Monitor to select the XDS 920 Business Language Assembler and perform an assembly from an encoded program deck (EI), with updating via source instruction information (SI); output is to be a program listing on the printer (LO), and encoded deck (EO), and a binary deck (BO).

EXTEND

Though not a control message in the MONITOR/MONARCH sense, this statement (written in the operation code field) must always appear at the beginning of all Business Language assemblies. It initiates the extension of META-SYMBOL into the Business Language, as it pertains to global literals.

Assembly Sample

```
△ASSIGN    S=MT0, X1=MT1, X2=MT2, SI=CR, EO=CP,
           BO=CP, LO=LP.
△META920   SI, EO, BO, LO, SET.
     EXTEND
     .
     .
     Source Program Cards
     .
     .
     END
```

INHS

INHS, Inhibit Suppress, is an optional directive to META-SYMBOL. It inhibits the suppression (normal condition) of generated code produced by Business Language instructions (PROCedures) in an assembly listing. Normally, the only octal code listing is the first word generated (adjacent to the Business Language instruction call). However, if the octal expansion of Business Language PROCedures is needed, the programmer may call the INHS directive.

EXECUTION-TIME OPERATIONS

XDS 900 SERIES; OPERATING UNDER MONARCH

The XDS MONARCH Monitor System operates input/output devices through use of a unit assignment table. That, is before any program can be assembled/compiled/loaded for execution under MONARCH, the user must tell the system which peripherals are available and what the communicating labels for these peripherals are. The △ASSIGN card (statement) is used to set up the table.

In particular, when loading a program for execution that uses the Business Language, special peripheral assignments must be made. Only those items from the following list that will be used during the execution need be assigned:

```
LLP   Line printer
LCR   Card reader
LCP   Card punch
L1    Magnetic tape unit 1
L2    Magnetic tape unit 2
.     .
.     .
L7    Magnetic tape unit 7
```

Any other peripheral that is used, such as the paper tape reader, need not be assigned; the standard assignments for loading MONARCH take care of the rest. When multiple units, such as three card readers, are available, consult the XDS MONARCH Reference Manual on the ASSIGN statement.

Assuming the use of a card reader, line printer, and magnetic tapes 1 and 2, the ASSIGN card (statement) looks like:

△ASSIGN LCR=CR, LLP=LP, L1=MT1, L2=MT2.

Typically, the cards for loading a program in binary form on cards, with the above peripherals being used, are as follows:

```
△ASSIGN LCR=CR, LLP=LP, L1=MT1, L2=MT2
△LOAD    0200, GO.
```

MONARCH assigns the peripherals and then transfers control to the program, as loaded in location 200 (octal).

XDS 9300; OPERATING UNDER MONITOR 9300

The same assignments and load cards are used with the XDS 9300 Computer.

ASSEMBLY ERROR FLAGS AND ERROR MESSAGE CODES

ERROR FLAGS

D Duplicate definition of level-1 symbol

E 1. Expression error

2. Directive syntax error

   Examples (not exhaustive)

   a. TEXT (if first symbol is value and second symbol not comma)

   b. END (external reference in end line)

3. Procedure syntax error

   Examples (not exhaustive)

   a. LDX, BRX, STX (no index field given)

   b. shifts (use of indirect addressing)

   c. COPY (various syntax errors)

F Illegal forward reference in directive

G Generative code in function

I Unknown opcode

L Illegal label (special characters)

M Improper use of SBRK or DISP (see the XDS META-SYMBOL Reference Manual)

N Missing END line

P Exceeding maximum parenthesis nesting level

R Primitive relocation error

S Business Language syntax error

T 1. Truncation (attempt to use form reference line to insert a value exceeding the capacity of the specified field)

2. Request COPY not available in hardware

U Undefined symbol used in manner that does not allow possibility of external reference

Notes:

1. Error and "MARK" (S) flags generated within PROCs (PROCedures) may appear in three possible places. All Business Language instructions are PROCs.

   a. On the call line, if generated during pass 1 of a two-pass PROC.

   b. On the next generated line.

   c. On a blank line following the PROC if no generative line follows the error.

2. Labels appearing on PROC reference lines are not defined until the end of the PROC. This is necessary to mechanize the lone $ feature. Therefore, if such a label is doubly defined, the D flag will come out on a blank line following the PROC.

   Machine instructions (LDA, etc.) are procedures within the assembler itself.

ASSEMBLER ERROR MESSAGE CODES

The standard abort message is "META-SYMBOL ERROR XX", where XX has the following meanings:

XX	Interpretation
01	Insufficient space to complete encoding of input
02	Corrections to encoded deck but encoded input file is empty
03	End-of-file detected while reading encoded input
04	Insufficient space to complete preassembly operations
05	Insufficient space to complete the assembly
06	Data error; META-SYMBOL does not recognize the data as anything meaningful
07	Requested output on a device that is not available
08	Corrections out of sequence
09	End-of-file detected by ENCODER when trying to read intermediate output tape X1
10	METAXXXX error; the XXXX name not recognized in the system
11	Byte larger than dictionary (bad encoded deck)
12	Not ENCODED deck
13	Checksum error reading system tape
14	Preassembler overflow (ETAB)
15	Not used
16	Data error causing META-SYMBOL to attempt to process procedure sample beyond end of table
24	Shrink overflow

Errors 05, 06, and 16 are accompanied by a printout that shows the value of certain internal parameters at the time of the abort:

LINE NUMBER	BREAK	
BREAK1	SMPWRD	
LOCATION COUNTER	LTBE	SECOND PASS ONLY
UPPER	LTBL	SECOND PASS ONLY
LOWER		

The last six messages are useful in determining the nature of assembler overflow.

For unfamiliar terms not explained in this manual, see the XDS META-SYMBOL Reference Manual.

I/O ERROR MESSAGES AND HALTS

When an I/O error is detected, a simple message is typed and the computer halts. The message consists of a two-letter indication of the type of error and a two-digit indication of the I/O device. The letter indicators are defined below; the two-digit number is the unit address number used in EOM selects (see applicable computer reference manual). The action taken if the halt is cleared depends on the type of error and the device involved. There are three types of error.

XDS 900 SERIES COMPUTERS

Buffer Error (BE)

Examples:

BE11 buffer error while reading magnetic tape 1.
BE42 buffer error while writing magnetic tape 2.

Action upon clearing the halt:

Magnetic tape input – since ten attempts are made to read the record before the halt occurs, continuing causes META-SYMBOL to accept the bad record.

Paper tape or card input – try again.

Magnetic tape output – try again.

Output other than magnetic tape – continues.

Checksum Error (CS)

Examples:

CS06 checksum error card reader.
CS11 checksum error reading magnetic tape 1.

Action upon clearing the halt:

Accepts bad record.

Write Error (FP) (XDS 900 Series Only)

Example:

FP42 magnetic tape 2 file-protected.

Action upon clearing the halt:

Checks again.

XDS 9300 COMPUTER

When an I/O error is detected on the XDS 9300, a message is typed, and control is returned to MONITOR. The message will be

! META ERROR α c

where α is a letter E (Encoder), P (Preassembler), or A (Assembler) indicating which overlay segment of the assembler was last loaded; and, c identifies the type of error:

c	Interpretation
IOC	Checksum error (irrecoverable)
IOE	Buffer error
IOA	Abnormal return

EXTERNAL DEFINITIONS AND REFERENCES[†]

A most useful META-SYMBOL Assembler feature allows separate assembly of independent programs. This feature not only permits reference by name to standard library programs, such as subroutines, but also allows large programs not using Business Language to be segmented without, in either case, shifting the burden of memory allocation to the programmer. Therefore, economies result both in reduced assemblies and debugging.

Symbolic interprogram communication is by means of external labels. Most labels are internal (or local) labels, defined internally to a program. Accordingly, the assembler recognizes a symbolic reference in the operand field of a line only when the symbol is defined elsewhere in the program by its appearance in the line's label field. When a symbolic reference cannot be satisfied within a program, all references to the symbol are called external references; that is, the symbol is assumed to be defined within some context external to the program in which the reference occurs.

The counterpart of the external reference is the external definition; a symbolic definition is made external by preceding it with a dollar sign ($). The programmer may establish an external definition either on the line that defines the symbol or on a subsequent line. In the latter case, the entire line is an external definition line; more than one symbol can be defined as external by listing them following the first symbol. Although additional dollar signs are not required, commas must separate one symbol from another. External references may appear only in the address field of a mnemonic instruction or FORM reference line. External definitions and references are restricted to eight characters. Relative external references (e.g., Symbol ± n) are not permitted.

The three programs shown in Example 11 are assembled separately and use external references for intercommunication. The program, MAIN, via two subroutines, DOUBLE and STORE, loads the number in NUMBER, doubles it, adds a constant to it, and stores it in location BIGGER. The constant, CONST, is defined as 05000_8.

If a user has not yet determined which labels to make external, he can write one or more external definition lines at the end of the program, and then reassemble. As stated previously, more than one such label can be written on one external definition line. However, the external definition line must be written someplace in the program after the symbol has been used in the label field (i.e., has been defined). See Example 12.

As indicated previously, program segmentation may be useful for maintenance or debugging. Segmenting can also be used to facilitate assembly of programs containing many symbols. For especially large programs, the numbers of symbols used

Example 11:

```
LABEL    OPERATION     OPERAND             COMMENTS
BEGIN    LDA           NUMBER              BEGINNING OF MAIN
         BRM           DOUBLE              ENTER DOUBLING SUBROUTINE
         ADD           CONST
         BRM           STORE               TO STORE SUBROUTINE
         HLT                               HALT PROGRAM
$BIGGER  RES           1
CONST    DATA          05000
$NUMBER  DATA
         END           BEGIN               END OF PROGRAM

$DOUBLE  DATA          0                   ENTRY FOR DOUBLE
         ADD           NUMBER              (A)+NUMBER INTO A
         BRR           DOUBLE              RETURN
         END

$STORE   DATA          0                   ENTRY FOR STORE
         STA           BIGGER
         BRR           STORE
         END
```

[†]See also "External Label References and Definitions" in the Appendix and "The Loading Process, External Label References and Definitions" in Section 3 of the XDS MONARCH Reference Manual (90 05 66).

Example 12:

```
LABEL  OPERATION      OPERAND                      COMMENTS
1    5    10    15    20    25    30    35    40    45    50    55    60
START  LDA    X
       BRM    SIN                  GØ TØ SIN RØUTINE
       STA    SINX
       HLT
SINX   DATA   0                    RESERVE A LØCATIØN
X      RES    1                    RESERVE A LØCATIØN FØR X
       END    START

SIN    DATA   0                    ENTRY TØ SIN RØUTINE
       .
       .
       .
       BRR    SIN
CØS    DATA   0                    ENTRY TØ CØS RØUTINE
       .
       .
       .
       BRR    CØS
$SIN,CØS                           DEFINE AS EXTERNAL
       END
```

may overflow the capacity of the assembler's symbol table. In this event, segmentation can be accomplished manually, as follows:

> Divide the program into as many physical segments as desired. Assemble each of these segments separately.
>
> Prepare a set of external definition lines from the external reference lists output at the end of each assembly listing. Relative external references are eliminated.
>
> Duplicate the set of external definition lines for each program segment and include it before each END card.
>
> Reassemble the program segments. The loader can now fulfill all external references.

To communicate external definition and reference information to the loader, the assembler outputs the former prior to the binary output (called "text") and the latter following the text. The external definition table consists of the alphanumeric symbols accompanied by their (relocatable or not) binary values. Each entry in the external reference table consists of the alphanumeric symbol accompanied by the (relocatable or not) binary address of the last location in which the external reference occurred. The address portion of that location points (contains the address of), in turn, to the last previous location where an external reference was made to the same symbol. The chain terminates when the address portion of an instruction contains 0. See Example 13.

Example 13:

Location	Contents	Label	Operation	Operand
00100			ORG	0100
00100	07600000	START	LDA	X
00101	07500000		LDB	Y
00102	03500101		STA	Y
00103	03600100		STB	X
00104	07600103		LDA	X
00105	06400104		MUL	X
00106	03500107		STA	XSQ
00107	00000000	XSQ	DATA	0
			END	START
00105	X			
00102	Y			

BUSINESS LANGUAGE PROGRAMMING EXAMPLES

The following examples illustrate the use of XDS Business Language instructions.

Example A

```
 LABEL    OPERATION        OPERAND                                  COMMENTS
1   5        10     15     20    25    30    35    40    45    50    55    60
         WRITETP TAPE2,ØUTPUT,200
```

A block of information 200 characters long is written on tape unit 2 from the memory area "OUTPUT".

Example B

```
 LABEL    OPERATION        OPERAND                                  COMMENTS
1   5        10     15     20    25    30    35    40    45    50    55    60
         READTP  TAPE1,INPUT
```

A block of information (interrecord gap to interrecord gap) is read from tape unit 1 into the memory area "INPUT".

Example C

```
 LABEL    OPERATION        OPERAND                                  COMMENTS
1   5        10     15     20    25    30    35    40    45    50    55    60
         READTP  TAPE3,CHANGE,80
```

A block of information is read from tape unit 3 into the memory area "CHANGE". Blocks longer than 80 characters are truncated to 80.

Example D

```
 LABEL    OPERATION        OPERAND                                  COMMENTS
1   5        10     15     20    25    30    35    40    45    50    55    60
         SØRT    TABLEUN,18,4,1,2
```

The 18-word table of four word items whose address is "TABLEUN" is sorted and the sorted table is placed in memory in the same locations. The key is two words long and its first word is the first word in an item.

Example E

```
 LABEL    OPERATION        OPERAND                                  COMMENTS
1   5        10     15     20    25    30    35    40    45    50    55    60
         BRACNE  ERRØR,INPUT,18,'E','N','G','L','Z'
```

Where the operation "BRACNE" denotes "Branch on Any Character Not Equal", this PROCedure causes a transfer to "ERROR", if the 18th character in the "INPUT" area is not equal to E, N, G, L, or Z. Otherwise, control is transferred to the next statement in sequence.

Example F

```
 LABEL    OPERATION        OPERAND                                  COMMENTS
1   5        10     15     20    25    30    35    40    45    50    55    60
         MØVE    INPUT,8,ØUTPUT,14,9
```

The nine characters, starting at the eighth character position of the memory area "INPUT", are moved to positions 14 through 22 of the area "OUTPUT".

Example G

```
LABEL   OPERATION   OPERAND                                  COMMENTS
RATE    FIELD       WORK,9,4
```

The four characters, beginning in the ninth character position of "WORK", are defined as a field. The name or label "RATE" is assigned to that field.

Example H

```
LABEL   OPERATION   OPERAND                                  COMMENTS
        MOVE        INPUT,20,RATE
```

A four-character field, starting at the 20th character of the memory area "INPUT", is moved to the field named "RATE", which begins at the ninth character position of the area "WORK" and is four characters long.

Example I

```
LABEL   OPERATION   OPERAND                                  COMMENTS
        COMPARE     VALUE1,VALUE2
        BRHI        HIGHVAL
```

These two PROCedures compare two fields identified as "VALUE1" and "VALUE2" and branch to a location labeled "HIGHVAL" if "VALUE2" is higher than "VALUE1".

Example J

Problem: An existing data file contains logical records of three characters each, blocked by a factor of twenty. The record, in memory at INPUT AREA, must be internally sorted before processing.

Method: Realign logical record internally to four characters each (one per word).

```
LABEL   OPERATION   OPERAND                       COMMENTS
        BLANK       SORTAREA,20
        LDA         =4                            5*THROUGH LOOP
        STA         LOOPCOUNTER                   WILL MOVE 20 FIELDS
        LDA         =INPUTAREA                    INITIALIZE MOVEBASE
        STA         MOVEBASE
        LDX         =0                            INITIALIZE INDEX
LOOP    MOVE        *MOVEBASE,1,(SORTAREA),1,3
        MOVE        *MOVEBASE,4,(SORTAREA+1),1,3
        MOVE        *MOVEBASE,7,(SORTAREA+2),1,3
        MOVE        *MOVEBASE,10,(SORTAREA+3),1,3
        EAX         4                             STEP INDEX
        LDA         =3                            STEP MOVEBASE
        ADM         MOVEBASE                      BY THREE WORDS
        SKR         LOOPCOUNTER                   TEST LOOP PASSES
        BRU         LOOP
        SORT        SORTAREA,20,1,1,3
```

Example K

Note that the previous example has a logical requirement for two index registers, one to modify INPUTAREA, the other to modify SORTAREA. But only one hardware index register is available in the XDS 900 Series Computers. Indirect addressing is easily used to gain the effect of the other index register. For example, the same loop could be written:

```
LABEL    OPERATION    OPERAND                                          COMMENTS
         BLANK        SORTAREA,20
         LDA          =4
         STA          LOOPCOUNTER
         LDA          =SORTAREA
         STA          SORTBASE
         LDX          =0
LOOP     MOVE         (INPUTAREA),1,*SORTBASE,1,3
         MOVE         (INPUTAREA),4,*SORTBASE,5,3
         MOVE         (INPUTAREA),7,*SORTBASE,9,3
         MOVE         (INPUTAREA),10,*SORTBASE,13,3
         EAX          3
         LDA          =4
         ADM          SORTBASE
         SKR          LOOPCOUNTER
         BRU          LOOP
         SORT         SORTAREA,20,1,1,3
```

These examples assume that the labels MOVEBASE (SORTBASE), INPUTAREA, SORTAREA, and LOOPCOUNTER have been defined elsewhere in the program.

Example L

When processing data whose format is a variable at run-time, one cannot specify the high-order character position required for such operations as MOVE. Such data is best scanned and processed after being unpacked to one character per word by the UNPACKR operation.

Assume a data card has been read into CARDAREA containing personnel names and account numbers in the following format: first name first, at least one space (blank), last name, comma, account number. For example:

SAMUEL GREENFIELD, 90060-01

The names and account numbers are to be built up in a table for subsequent processing. Assume a maximum of twenty characters each for the first name and the last name, and twelve characters for the account number. Thus, the internal logical record is fifty-two characters (thirteen words) long.

The code that might be used for this example is given on the following page.

```
LABEL   OPERATION   OPERAND                                COMMENTS
        READCD      CARDAREA
        UNPACKR     CARDAREA,SCANAREA,80   UNPACK 80 COLUMNS
        LDA         =SCANAREA               INITIALIZE
        STA         COLUMNCOUNT             POSITION
*       SCAN TO END OF FIRST NAME
*       CURRENT PLACE IN DATA TABLE IS ALREADY SET UP.
LOOP1   BRACEQ      NEXT,*COLUMNCOUNT,4,060
        MIN         COLUMNCOUNT
        BRU         LOOP1
NEXT    LDA         COLUMNCOUNT
        SUB         =SCANAREA
        STA         FIRSTNAMECOUNT
        PACKR       *DATATABLE,SCANAREA,(FIRSTNAMECOUNT)
        LDA         =5                      STEP DATA TABLE
        ADM         DATATABLE               POSITION 20 CHARACTERS
LOOP2   BRACNE      NEXT2,*COLUMNCOUNT,4,060 SKIP EXTRA
        MIN         COLUMNCOUNT             SPACES PRECEDING
        BRU         LOOP2                   LAST NAME
NEXT2   MOVEWD      COLUMNCOUNT,STARTOFLASTNAME
LOOP3   BRACEQ      NEXT3,*COLUMNCOUNT,4,073 SCAN TO COMMA
        MIN         COLUMNCOUNT
        BRU         LOOP3
NEXT3   LDA         COLUMNCOUNT
        SUB         STARTOFLASTNAME
        STA         LASTNAMECOUNT
        PACKR       *DATATABLE,*STARTOFLASTNAME,(LASTNAMECOUNT)
        LDA         =5                      STEP DATA TABLE
        ADM         DATATABLE               BY TWENTY CHARACTERS
        MIN         COLUMNCOUNT
        PACKR       *DATATABLE,*COLUMNCOUNT,3 MOVE ACCOUNT
*                                           NUMBER
        LDA         =3                      STEP DATA TABLE
        ADM         DATATABLE               BY TWELVE CHARACTERS
```

Note that the above code assumes the actual data table has been blanked prior to the loop so that all names will be appropriately filled out to the assumed length of twenty characters. The labels CARDAREA, DATATABLE, SCANAREA, COLUMNCOUNT, FIRSTNAMECOUNT, STARTOFLASTNAME, and LASTNAMECOUNT must be defined elsewhere in the program.

BPOV for the XDS 9300 Computer

BPOV was not implemented for the XDS 9300 because it can not be done in a completely generalized way. In the first place, MONITOR prints its own heading line at the top of each page and, secondly, any output on the printer by any file name will reduce the line count remaining. If one knows that only one file is going to the line printer, the following PROCedure definitions may be useful even if they are not completely general. See Example M on the following page.

Example M

```
LABEL    OPERATION    OPERAND                         COMMENTS
P        PROC
BPOV     NAME
         SKR          LINECOUNT                  REDUCE LINES REMAINING
         BRU          P(1)                       THERE IS STILL ROOM.
         END                                     LAST LINE HAS BEEN PRINTED

P        PROC
NEXTPAGE NAME
         RESTORE
         PRINT        HEADING
         UPSPACE      N
         LDA          =LINES-1
         STA          LINECOUNT
         END

P        PROC
SETLINES NAME
         BRM          M\MARG
         PZE          P(1)+1
         STZ          LINECOUNT
         END
```

One uses SETLINES at the beginning of the executable program. The number of lines (P(1)) is the user's number of lines including his own heading line and the number of lines (N) upspaced after the heading in the NEXTPAGE PROCedure. LINECOUNT is a counter cell defined by the user. NEXTPAGE restores the page, printing the MONITOR's title line, the user's title (33 words beginning at HEADING, where HEADING is defined by the user), upspaces the margin desired before the regular information, and sets up the counter used by the BPOV PROCedure. LINES is the number of actual information lines per page, excluding headings. By using BPOV prior to each PRINT, the programmer is sure that his title will appear at the top of each page. Note that additional features could be added to the NEXTPAGE PROCedure, including an UPSPACE between the MONITOR title and user title, additional subheadings, etc. The extra lines must be reflected in the SETLINES PROCedure. See Example N below.

This discussion points out an important feature of XDS Business Language. The programmer can use the full META-SYMBOL PROCedure power to define his special combinations of Business Language Procedures, creating a still higher-level language (NEXTPAGE does more than RESTORE). Because of this inherent power, the XDS 9300 user can often implement for himself a BPOV that will satisfy his requirements.

Example N

```
LABEL    OPERATION    OPERAND                         COMMENTS
         SETLINES 35
         .
         .
         .
         BPOV         NEXT
         NEXTPAGE
NEXT     PRINT        INFORMATIONLINE
```

XDS 920/930 INSTRUCTION LIST

Mnemonic	Instruction Code	Name
LOAD/STORE		
LDA	76	LOAD A
STA	35	STORE A
LDB	75	LOAD B
STB	36	STORE B
LDX	71	LOAD INDEX
STX	37	STORE INDEX
EAX	77	COPY EFFECTIVE ADDRESS INTO INDEX
XMA	62	EXCHANGE M AND A
ARITHMETIC		
ADD	55	ADD M TO A
ADC	57	ADD WITH CARRY
ADM	63	ADD A TO M
MIN	61	MEMORY INCREMENT
SUB	54	SUBTRACT M FROM A
SUC	56	SUBTRACT WITH CARRY
MUL	64	MULTIPLY
DIV	65	DIVIDE
LOGICAL		
ETR	14	EXTRACT
MRG	16	MERGE
EOR	17	EXCLUSIVE OR
REGISTER CHANGE		
RCH, COPY	46	REGISTER CHANGE
CLA	0 46 00001	CLEAR A
CLB	0 46 00002	CLEAR B
CLR	0 46 00003	CLEAR AB
CAB	0 46 00004	COPY A INTO B
CBA	0 46 00010	COPY B INTO A
XAB	0 46 00014	EXCHANGE A AND B
BAC	0 46 00012	COPY B INTO A, CLEAR B
ABC	0 46 00005	COPY A INTO B, CLEAR A
CXA	0 46 00200	COPY INDEX INTO A
CAX	0 46 00400	COPY A INTO INDEX
XXA	0 46 00600	EXCHANGE INDEX AND A
CBX	0 46 00020	COPY B INTO INDEX
CXB	0 46 00040	COPY INDEX INTO B
XXB	0 46 00060	EXCHANGE INDEX AND B
STE	0 46 00122	STORE EXPONENT
LDE	0 46 00140	LOAD EXPONENT
XEE	0 46 00160	EXCHANGE EXPONENTS
CNA	0 46 01000	COPY NEGATIVE INTO A
BRANCH		
BRU	01	BRANCH UNCONDITIONALLY
BRX	41	INCREMENT INDEX AND BRANCH
BRM	43	MARK PLACE AND BRANCH
BRR	51	RETURN BRANCH

Mnemonic	Instruction Code	Name
TEST/SKIP		
SKS	40	SKIP IF SIGNAL NOT SET
SKE	50	SKIP IF A EQUALS M
SKG	73	SKIP IF A GREATER THAN M
SKR	60	REDUCE M, SKIP IF NEGATIVE
SKM	70	SKIP IF A = M ON B MASK
SKN	53	SKIP IF M NEGATIVE
SKA	72	SKIP IF M AND A DO NOT COMPARE ONES
SKB	52	SKIP IF M AND B DO NOT COMPARE ONES
SKD	74	DIFFERENCE EXPONENTS AND SKIP
SHIFT		
RSH	0 66 000XX	RIGHT SHIFT AB
RCY	0 66 200XX	RIGHT CYCLE AB
LSH	0 67 000XX	LEFT SHIFT AB
LCY	0 67 200XX	LEFT CYCLE AB
NOD	0 67 100XX	NORMALIZE AND DECREMENT X
CONTROL		
HLT, PZE	00	HALT
NOP	20	NO OPERATION
EXU	23	EXECUTE
BREAKPOINT TESTS (Breakpoints specified as expression list in operand field)		
BPT	0 40 20XX0	BREAKPOINT TEST
OVERFLOW (No operand)		
OVT	0 40 20001	OVERFLOW INDICATOR TEST AND RESET
ROV	0 02 20001	RESET OVERFLOW
REO	0 02 20010	RECORD EXPONENT OVERFLOW (930 only)
INTERRUPT (No operand)		
EIR	0 02 20002	ENABLE INTERRUPT SYSTEM
DIR	0 02 20004	DISABLE INTERRUPT SYSTEM
IET	0 40 20004	INTERRUPT ENABLED TEST
IDT	0 40 20002	INTERRUPT DISABLED TEST
AIR	0 02 20020	ARM INTERRUPTS
CHANNEL CONTROL (Channel designated by expression in operand field)		
ALC	X 0X 50X00	ALERT CHANNEL (930 only)
DSC	X 0X 00X00	DISCONNECT CHANNEL
ASC	X 0X 12X00	ALERT TO STORE ADDRESS IN CHANNEL (930 only)
TOP	X 0X 14X00	TERMINATE OUTPUT ON CHANNEL

Mnemonic	Instruction Code	Name
CHANNEL TESTS (930 only - Channel designated by expression in operand field)		
CAT	X 40 X4X00	CHANNEL ACTIVE TEST
CET	X 40 X1X00	CHANNEL ERROR TEST
CZT	X 40 X2X00	CHANNEL ZERO COUNT TEST
CIT	X 40 X0X00	CHANNEL INTER-RECORD TEST
INPUT/OUTPUT		
MIW	12	M INTO W BUFFER WHEN READY
WIM	32	W BUFFER INTO M WHEN READY
MIY	10	M INTO Y BUFFER WHEN READY
YIM	30	Y BUFFER INTO M WHEN READY
BRTW, BRTY	0 40 2X000	BUFFER READY TEST
BETW, BETY	0 40 200X0	BUFFER ERROR TEST
POT	13	PARALLEL OUTPUT
PIN	33	PARALLEL INPUT
EOM	02	ENERGIZE OUTPUT M
EOD	06	ENERGIZE OUTPUT TO DIRECT ACCESS CHANNEL (930 only)

XDS 910/925 INSTRUCTION LIST

Nonstandard instruction configurations are indicated in parentheses beside the instruction class affected; for examples, see BREAKPOINT TESTS, OVERFLOW.

Mnemonic	Instruction Code	Name
LOAD/STORE		
LDA	76	LOAD A
STA	35	STORE A
LDB	75	LOAD B
STB	36	STORE B
LDX	71	LOAD INDEX
STX	37	STORE INDEX
EAX	77	COPY EFFECTIVE ADDRESS INTO INDEX
ARITHMETIC		
ADD	55	ADD M TO A
MIN	61	MEMORY INCREMENT
SUB	54	SUBTRACT M FROM A
MDE	60	MEMORY DECREMENT
MUS	64	MULTIPLY STEP
DIS	65	DIVIDE STEP
LOGICAL		
ETR	14	EXTRACT
MRG	16	MERGE
EOR	17	EXCLUSIVE OR
REGISTER CHANGE		
RCH, COPY	46	REGISTER CHANGE
XAB	0 46 00000	EXCHANGE A AND B
BAC	0 46 10000	COPY B INTO A, CLEAR B
ABC	0 46 20000	COPY A INTO B, CLEAR A
CLR	0 46 30000	CLEAR A, B
BRANCH		
BRU	01	BRANCH UNCONDITIONALLY
BRX	41	INCREMENT INDEX AND BRANCH
BRM	43	MARK PLACE AND BRANCH
BRR	51	RETURN BRANCH
TEST/SKIP		
SKS	40	SKIP IF SIGNAL NOT SET
SKG	73	SKIP IF A GREATER THAN M
SKN	53	SKIP IF M NEGATIVE
SKA	72	SKIP IF M AND A DO NOT COMPARE ONES
SKM	70	SKIP IF A = M ON B MASK

Mnemonic	Instruction Code	Name
SHIFT		
RSH	0 66 000XX	RIGHT SHIFT AB
RCY	0 66 200XX	RIGHT CYCLE AB
LSH	0 67 000XX	LEFT SHIFT AB
LCY	0 67 200XX	LEFT CYCLE AB
NOD	0 67 100XX	NORMALIZE AND DECREMENT X
CONTROL		
HLT, PZE	00	HALT
NOP	20	NO OPERATION
EXU	23	EXECUTE
BREAKPOINT TESTS (Breakpoints specified as expression list in operand field)		
BPT	0 40 20XX0	BREAKPOINT TEST
OVERFLOW (No operand)		
OVT	0 40 20001	OVERFLOW INDICATOR TEST AND RESET
ROV	0 02 20001	RESET OVERFLOW
INTERRUPT (No operand)		
EIR	0 02 20002	ENABLE INTERRUPT SYSTEM
DIR	0 02 20004	DISABLE INTERRUPT SYSTEM
IET	0 40 20004	INTERRUPT ENABLED TEST
IDT	0 40 20002	INTERRUPT DISABLED TEST
AIR	0 02 20020	ARM INTERRUPT
CHANNEL CONTROL (Channel designated by expression in operand field)		
ALC	X 0X 50X00	ALERT CHANNEL (925 only)
DSC	X 0X 00X00	DISCONNECT CHANNEL
ASC	X 0X 12X00	ALERT TO STORE ADDRESS IN CHANNEL (925 only)
TOP	X 0X 14X00	TERMINATE OUTPUT ON CHANNEL
CHANNEL TESTS (925 only - Channel designated by expression in operand field)		
CAT	X 40 X4X00	CHANNEL ACTIVE TEST
CET	X 40 X1X00	CHANNEL ERROR TEST
CZT	X 40 X2X00	CHANNEL ZERO COUNT TEST
CIT	X 40 X0X00	CHANNEL INTER-RECORD TEST
INPUT/OUTPUT		
MIW	12	M INTO W BUFFER WHEN READY
WIM	32	W BUFFER INTO M WHEN READY
MIY	10	M INTO Y BUFFER WHEN READY
YIM	30	Y BUFFER INTO M WHEN READY
BRTW, BRTY	0 40 2X000	BUFFER READY TEST
BETW, BETY	0 40 200X0	BUFFER ERROR TEST
POT	13	PARALLEL OUTPUT
PIN	33	PARALLEL INPUT
BPO	11	BLOCK PARALLEL OUTPUT (925 only)
BPI	31	BLOCK PARALLEL INPUT (925 only)
EOM	02	ENERGIZE OUTPUT M
EOD	06	ENERGIZE OUTPUT TO DIRECT ACCESS CHANNEL (925 only)

XDS 9300 INSTRUCTION LIST

Mnemonic	Instruction Code	Name
LOAD/STORE		
LDA	16	LOAD A
STA	76	STORE A
LDB	14	LOAD B
STB	74	STORE B
LDX	X - 17	LOAD INDEX
STX	X - 77	STORE INDEX
STZ	0 - 77	STORE ZERO
LDP, LDF	26	LOAD DOUBLE PRECISION (FLOATING)
STD, STF	75	STORE DOUBLE PRECISION (FLOATING)
XMA	36	EXCHANGE M AND A
XMB	34	EXCHANGE M AND B
XMX	X - 37	EXCHANGE MEMORY AND INDEX
LDS	06	LOAD SELECTIVE
STS	70	STORE SELECTIVE
EAX	15	COPY EFFECTIVE ADDRESS INTO INDEX REGISTER 1
ARITHMETIC		
ADD	05	ADD M TO A
DPA	25	DOUBLE PRECISION ADD
SUB	04	SUBTRACT
DPS	24	DOUBLE PRECISION SUBTRACT
MPO	71	MEMORY PLUS ONE
MPT	72	MEMORY PLUS TWO
MUL	63	MULTIPLY
DIV	62	DIVIDE
ADM	35	ADD A TO M
TMU	61	TWIN MULTIPLY
DPN	27	DOUBLE PRECISION NEGATE
FLOATING-POINT		
FLA	65	FLOATING ADD
FLS	64	FLOATING SUBTRACT
FLM	67	FLOATING MULTIPLY
FLD	66	FLOATING DIVIDE
LOGICAL		
ETR	11	EXTRACT
MRG	13	MERGE
EOR	12	EXCLUSIVE OR
REGISTER CHANGE		
Mode I		
RCH, COPY	0 40 XXXXX	
Mode II		
RCH, COPY	X 40 XXXXX	
Mode III		
AXB	4X 40 XXXXX	ADDRESS TO INDEX BASE

Mnemonic	Instruction Code	Name
BRANCH		
BRU	01	BRANCH UNCONDITIONALLY
BRX	X - 57	INCREMENT INDEX AND BRANCH
BRC	0 - 57	BRANCH AND CLEAR INTERRUPT
BRM	03	MARK PLACE AND BRANCH
BMA	43	BRANCH AND MARK PLACE OF ARGUMENT ADDRESS
BRR	41	RETURN ADDRESS
TEST/SKIP		
SKE	45	SKIP IF A EQUALS M
SKU	47	SKIP IF A UNEQUAL TO M
SKG	46	SKIP IF A GREATER THAN M
SKL	44	SKIP IF A LESS THAN OR EQUAL TO M
SKR	73	REDUCE M, SKIP IF NEGATIVE
SKM	55	SKIP IF A = M ON B MASK
SKN	53	SKIP IF M NEGATIVE
SKA	54	SKIP IF M AND A DO NOT COMPARE ONES
SKB	52	SKIP IF M AND B DO COMPARE ONES
SKP	51	SKIP IF BIT SUM EVEN
SKS	20	SKIP IF SIGNAL NOT SET
SKF	50	SKIP IF FLOATING EXPONENT IN B $\geq$ M
SKQ	56	SKIP IF MASKED QUANTITY IN A GREATER THAN M
SHIFT		
SHIFT	60	SHIFT (Used in conjunction with indirect addressing)
ARSA	60-20	ARITHMETIC RIGHT SHIFT A
ARSB	60-10	ARITHMETIC RIGHT SHIFT B
ARSD	60-00	ARITHMETIC RIGHT SHIFT DOUBLE
ARST	60-30	ARITHMETIC RIGHT SHIFT TWIN (A AND B)
LRSA	60-21	LOGICAL RIGHT SHIFT A
LRSB	60-11	LOGICAL RIGHT SHIFT B
LRSD	60-01	LOGICAL RIGHT SHIFT DOUBLE
LRST	60-31	LOGICAL RIGHT SHIFT TWIN (A AND B)
CRSA	60-22	CIRCULAR RIGHT SHIFT A
CRSB	60-12	CIRCULAR RIGHT SHIFT B
CRSD	60-02	CIRCULAR RIGHT SHIFT DOUBLE
CRST	60-32	CIRCULAR RIGHT SHIFT TWIN (A AND B)
ALSA	60-24	ARITHMETIC LEFT SHIFT A
ALSB	60-14	ARITHMETIC LEFT SHIFT B
ALSD	60-04	ARITHMETIC LEFT SHIFT DOUBLE
ALST	60-34	ARITHMETIC LEFT SHIFT TWIN (A AND B)
LLSA	60-25	LOGICAL LEFT SHIFT A
LLSB	60-15	LOGICAL LEFT SHIFT B
LLSD	60-05	LOGICAL LEFT SHIFT DOUBLE
LLST	60-35	LOGICAL LEFT SHIFT TWIN (A AND B)
CLSA	60-26	CIRCULAR LEFT SHIFT A
CLSB	60-16	CIRCULAR LEFT SHIFT B
CLSD	60-06	CIRCULAR LEFT SHIFT DOUBLE
CLST	60-36	CIRCULAR LEFT SHIFT TWIN (A AND B)
NORA	60-60	NORMALIZE A
NORD	60-40	NORMALIZE DOUBLE

Mnemonic	Instruction Code	Name

FLAG REGISTER (Single operand expression)

FLAG	22	FLAG
FIRS	22-0	FLAG INDICATOR RESET/SET
FSTR	22-1	FLAG INDICATOR SET TEST/RESET
FRTS	22-2	FLAG INDICATOR RESET TEST/SET
FRST	22-3	FLAG INDICATOR RESET/SET TEST
SWT	22-4	SENSE SWITCH TEST

CONTROL

HLT, PZE	00	HALT
NOP	10	NO OPERATION
EXU	21	EXECUTE
INT	07	LOAD OP CODE INTO INDEX 2, SKIP ON BIT 1
REP	23	REPEAT INSTRUCTION IN M

INTERRUPTS (No operand)

EIR	0 02 20002	ENABLE INTERRUPT SYSTEM
DIR	0 02 20004	DISABLE INTERRUPT SYSTEM
AIR	0 02 20020	ARM INTERRUPTS
IET	0 20 20004	INTERRUPT ENABLED TEST
IDT	0 20 20002	INTERRUPT DISABLED TEST

CHANNEL CONTROL (Channel designated by expression in operand field)

DSC	X X2 00X00	DISCONNECT CHANNEL
ALC	X X2 50X00	ALERT CHANNEL
ASC	X X2 12X00	ALERT TO STORE ADDRESS IN CHANNEL
TOP	X X2 14X00	TERMINATE OUTPUT ON CHANNEL

CHANNEL TEST (Channel designated by expression in operand field)

CAT	X 20 X4X00	CHANNEL ACTIVE TEST
CET	X 20 X1X00	CHANNEL ERROR TEST
CIT	X 20 X0X00	CHANNEL INTER-RECORD TEST
CZT	X 20 X2X00	CHANNEL ZERO COUNT TEST

INPUT/OUTPUT

EOM	02	ENERGIZE OUTPUT M
EOD	42	ENERGIZE OUTPUT TO DIRECT ACCESS CHANNEL
PIN	33	PARALLEL INPUT
POT	31	PARALLEL OUTPUT
MIA	30	MEMORY INTO CHANNEL A BUFFER
AIM	32	CHANNEL A BUFFER INTO MEMORY

XDS 9300 REGISTER CHANGE INSTRUCTION (040)

This instruction has three main functions:

Interchange and/or modify information between selected bytes of A and B.

Interchange and/or modify information among selected bytes of A, B, and the index registers.

Load the address portion of a selected index register from the address portion of the instruction.

In modes 1 and 2, the address portion of the instruction serves to extend the operand code; each address bit has a particular significance during instruction decoding and execution. In mode 3, however, the interpretation of the address portion is conventional; the 15-bit value defines an operand. Therefore, in mode 3, the instruction is programmed following the mnemonic, AXB, by an expression in the operand field. The assembler inserts the value of the expression in the instruction's 15-bit address portion.

When programmed in mode 1 or 2, the instruction may be given one of two mnemonics: RCH or COPY. The assembler processes the operand field of RCH in the conventional manner, inserting the evaluated operand field expression into the instruction's 15-bit address portion. In general, the expression is an octal number representing the bit pattern that specifies the function to be performed. This implies the programmer's detailed knowledge of the instruction.

The operand field of COPY is interpreted differently. The field consists of a byte selection "mask" followed by one or more grouped expression lists that describe the desired operations (s). The programmer need not be concerned with operand specification via bit patterns. See Example 14.

Unless a merge is specified, the assembler automatically sets the "clear" bit. Thus, the second line causes the generation of 0 40 37703.

Label	Operation	Operand
LABEL	COPY	E, (E11, . . . , E1N), (E21, . . . E2N), . . . , (EM1, . . . , EMN)

Since parenthetical notation is used in the operand field, parentheses are not used to denote "optional." As usual, the label is optional and may or may not be external. The first operand and all successive operand lists are also optional.

Rules:

1. The byte selection mask, if present, is the first expression to appear in the operand field. It is not enclosed within parentheses. In the absence of this expression, the assembler assumes the mask 077777777 to be implicitly specified. Actually, the assembler cannot insert the mask directly into the byte-selection position of the instruction, since the 24-bit value must be mapped into three or eight bits. However, it is convenient to think of the mask in this manner. Since the mask may be an expression, it need not always be written as an octal number. See Example 15.

Unless the programmer indicates that the specified index register be cleared (in a mode 2 register change), the assembler automatically sets one of the bits, 12, 13, or 14, to prevent the register from being cleared.

Example 14.

```
LABEL      OPERATION      OPERAND                     COMMENTS
           COPY    (0,(A,B))           CLEAR A AND B
           COPY    (A,B)               COPY A INTO B
           COPY    (A,B),(B,A)         EXCHANGE A AND B
           COPY    077,(A,B,B)         MERGE THE LOW ORDER SIX
*                                       BITS OF A AND B IN B.
```

Example 15.

```
LABEL      OPERATION      OPERAND                     COMMENTS
EXP        EQU     0777
HI3        EQU     070000000
           COPY    EXP,(B,1),(0,B)     ADDRESS FROM B INTO X1,
*                                       CLEAR B ADDRESS
           COPY    HI3,(A,B)           TAG FIELD FROM A INTO B
```

2. Following the mask, one or more parenthetical expression lists appear, separated by commas. Within a list, two or more expressions (or expression groups) appear. The first of these specifies the source of information flow, and the last specifies the destination. In the case of three or more successive expressions, an OR is implied. Thus, COPY operations are specified by ordered groupings of values. The following definitions relate the value of an expression to the 24-bit source value/register or destination register. Where octal registers are not involved (0 and -1), it is convenient to imagine the existence of two fictitious registers always containing all zeros and all ones, respectively.

Value		Meaning
-5	-(A)	The 2's complement of (A)[†]
-4	1 - (A)	The 1's complement of (A)
-3	1 - (B)	The 1's complement of (B)
-1	-1	
0	0	
1	(X1)	
2	(X2)	
3	(X3)	
4	(B)	
5	(A)	

Therefore, to refer to the registers mnemonically, the programmer must precede his program by equality directives such as:

```
   A  EQU   5
   B  EQU   4
  X2  EQU   2
  IA  EQU  -4
  IB  EQU  -3
ONES  EQU  -1
```

Examples:

Mnemonic Notation	Absolute	Interpretation
COPY (A, B), (B, A)	COPY (5,4), (4, 5)	Exchange A and B
COPY (IA, B), (0, A) COPY (1-A, B), (0, A)	COPY (-4,4), (0, 5)	Copy inverse of A into B and clear A
COPY 070, (ONES, B) COPY 070, (-1, B)	COPY 070, (-1, 4)	Form mask in B_{18-20}

Thus, the programmer can specify any legitimate register change without having to write the necessary bit pattern explicitly and without being restricted to a preselected set of mnemonic opcodes. Also, the assembler diagnoses the variable field for legitimacy.

[†]() denotes contents of.

XDS 920/930 REGISTER CHANGE INSTRUCTION (046)

The XDS 920/930 Register Change Instruction has some, but not all, of the capabilities of its XDS 9300 Computer counterpart. The differences are:

The XDS 920/930 RCH does not provide for byte selection except for selecting the low-order nine bits.

The XDS 920/930 Computers include only one index register.

There is no capability for copying (or merging) the 1's complement of one register into another.

Format:

Label	Operation	Operand
LABEL	COPY or COPYE	(E11, . . . , E1N), (E21, . . . E2N), . . . , (EM1, . . . , EMN)

As before, the label is optional and may or may not be external. All expression lists are optional. The mnemonic COPY implies that operands are whole-word registers; the mnemonic COPYE causes the exponent portion (the low-order nine bits) only to be affected.

COPY(E) operations are specified by ordering groupings of values. The following definitions relate the value of an expression to the 24-bit source value/register or destination register.

Value		Meaning
-5	-(A)	The 2's complement of A
0	0	A register containing all 0's
2	2	The index register
4	(B)	The contents of B
5	(A)	The contents of A

Examples:

Mnemonic Notation	Absolute	Interpretation
COPY (A, B), (B, A)	COPY (5, 4), (4, 5)	Exchange A and B
COPYE (B, X), (0, B)	COPYE (4, 2), (0, 4)	Extend exponent to X, Clear B
COPY (A, B, X)	COPY (5, 4, 2)	Merge A and B to X

BUSINESS INSTRUCTION LIST

Name	Operation	Operand	Page
Area Definitions			
Define Data Field	FIELD	E1, HC1, CC	13
Define and Reserve Area of Memory	DEFAREA (DA)	N	13
	DEFAREA	N, 'CH'	
Data Transmission			
Move Word String	MOVEWD (MVW)	E1, E2, N	13
	MOVEWD	E1, E2, N, X	
Move Character String	MOVE	E1, HC1, E2, HC2, CC	14
	MOVE	E1, HC1, F2	
	MOVE	E1, HC1, CC, F2	
	MOVE	F1, E2, HC2	
	MOVE	F1, E2, HC2, CC	
	MOVE	F1, F2	
Move Character String with Zero Fill	MOVEIZ	'Same as MOVE'	14
Move and Edit Character String	MOVEED (EDIT, MCE)	'Same as MOVE', ('$', 'C', P, '-' or 'CR')	14
Decimal Arithmetic			
Decimal Add	DADD	E1, LO1, CC1	16
	DADD	E1, LO1, CC1, E2, LO2, CC2	
	DADD	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3	
Decimal Subtract	DSUB	E1, LO1, CC1	16
	DSUB	E1, LO1, CC1, E2, LO2, CC2	
	DSUB	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3	
Decimal Multiply	DMUL	E1, LO1, CC1, E2, LO2, CC2	16
	DMUL	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3	
Decimal Divide	DDIV	E1, LO1, CC1, E2, LO2, CC2	16
	DDIV	E1, LO1, CC1, E2, LO2, CC2, E3, LO3, CC3	
Decimal Conversion			
Binary to BCD	BINBCD	E1, HC1, CC1, E2	16
BCD to Binary	BCDBIN	E1, HC1, CC1, E2	16
Data Testing			
Compare Word String	COMPARW (CPW)	E1, E2, N	18
Compare Character String	COMPARE	E1, HC1, E2, HC2, CC	18
	COMPARE	E1, HC1, F2	
	COMPARE	E1, HC1, CC, F2	
	COMPARE	F1, E2, HC2	
	COMPARE	F1, E2, HC2, CC	
	COMPARE	F1, F2	
Program Branch Control			
Branch on Equal	BREQ (BE)	E1	19
Branch on Not Equal	BRNE (BU)	E1	19
Branch on High	BRHI (BH)	E1	19
Branch on Low	BRLO (BL)	E1	19
Branch on any Character Not Equal	BRACNE	E1, E2, E3, 'Q1', 'Q2', . . .	20
Branch on any Character Equal	BRACEQ	E1, E2, E3, 'Q1', 'Q2', . . .	20

Name	Operation	Operand	Page
Character Manipulation			
Pack Left-justified Character String	PACK (PACKL)	E1, E2, . . .	17
Unpack Left-justified Character String	UNPACK (UNPACKL)	E1, E2, . . .	17
Pack Right-justified Character String	PACKR	E1, E2, N	17
Unpack Right-justified Character String	UNPACKR	E1, E2, N	18
	UNPACKR	E1, E2, N, 'CH'	
Data Field Initializing			
Clear Word String to Zeros	CLEAR	E1, N	20
	CLEAR	E1, N, X	
Clear Character String to Zeros	CLEARCH	E1, HC1, CC	21
	CLEARCH	F1	
Set Word String to Blanks	BLANK	E1, N	21
	BLANK	E1, N, X	
Set Character String to Blanks	BLANKCH	E1, HC1, CC	21
	BLANKCH	F1	
Fill Word String with Character	FILL	E1, N, 'CH'	21
	FILL	E1, N, 'CH', X	
Fill Character String with Character	FILLCH	E1, HC1, CC, 'CH'	22
	FILLCH	F1, 'CH'	
Internal Sorting			
Ascending BCD Sort	SORT	E1, E2, E3, E4, E5	22
	SORT	E1, E2, E3, E4, E5, E6	
Descending BCD Sort	SORTDS	E1, E2, E3, E4, E5	22
	SORTDS	E1, E2, E3, E4, E5, E6	
Ascending Binary Sort	SORTBIN	E1, E2, E3, E4, E5	22
	SORTBIN	E1, E2, E3, E4, E5, E6	
Descending Binary Sort	SORTBDS	E1, E2, E3, E4, E5	22
	SORTBDS	E1, E2, E3, E4, E5, E6	
Register Shifting			
Logical Left Shift AB Register	LSHIFT	N	23
Logical Right Shift AB Register	RSHIFT	N	23
Special Operations			
Set Collating Sequence	COLLATE	'SDS' or 'BDP'	23
Compute Memory Size	MEMORY		23
	MEMORY	E1	
Specify Extended Assembly Mode	EXTEND		23
Overflow Test			
Branch on Arithmetic Overflow	BAOV	E1	16
Magnetic Tape			
Write Magnetic Tape Record (BCD Mode)	WRITETP	LU, E1, CC	24
Write Magnetic Tape Record (Binary Mode)	WTPBIN	LU, E1, CC	24
Read Magnetic Tape Record (BCD Mode)	READTP	LU, E1, CC	25
Read Magnetic Tape Record (Binary Mode)	RTPBIN	LU, E1, CC	25
Rewind Magnetic Tape	REWIND	LU	25
Write End of Tape Mark	WTMARK (WTM)	LU	26
Backspace Magnetic Tape	BACKSPACE	LU, N	26
Skip Magnetic Tape Records Forward	SKIPTAPE	LU, N	26

Name	Operation	Operand	Page
Punched Card			
Read BCD Card	READCD	E1	27
Punch BCD Card	PUNCH	E1	27
Typewriter			
Input from Typewriter	TYPEIN	E1	27
Output on Typewriter	TYPE	E1	27, 28
Line Printer			
Print 132-Character Line	PRINT	E1	28
	PRINT	E1, N	
	PRINT	E1, 'Q'	
Print 120-Character Line	PRT120	E1	28
	PRT120	E1, N	
	PRT120	E1, 'Q'	
Upspace Line Printer	UPSPACE	N	29
Skip to Channel N	SKPCHN	N	29
Skip to Channel 1	RESTORE		29
Input/Output Branch Tests			
Branch on Channel Error	BCER	E1	29
Branch on Page Overflow (XDS 900 Series only)	BPOV	E1	29
Branch on Printer Fault	BPRF	E1	30
Branch on Tape Mark	BTMK	E1	30
Branch on Beginning of Tape	BBTP	E1	30
Branch on End of Tape	BETP	E1	30
Branch on File-protected Tape	BFPT	E1	30
Branch on Last Card	BLCD	E1	30

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245

XEROX

Reader Comment Form

We would appreciate your comments and suggestions for improving this publication.

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Sales
☐ Reference ☐ Maintaining ☐ Operating

Is the material presented effectively?

☐ Fully Covered ☐ Well Illustrated ☐ Well Organized ☐ Clear

What is your overall rating of this publication?

☐ Very Good ☐ Fair ☐ Very Poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, Please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your Name & Return Address

2190(12/72)

Thank You For Your Interest. (fold & fasten as shown on back, no postage needed if mailed in U.S.A.)

Staple Staple

Fold

First Class
Permit No. 229
El Segundo,
California

BUSINESS REPLY MAIL

No postage stamp necessary if mailed in the United States

Postage will be paid by

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245

Attn: Programming Publications

Fold