W. J. METZGER

File Number 1401-22

Form C24-3171-0

IBM Systems Reference Library

Basic Autocoder 4K for IBM 1401: Specifications

This publication contains language specifications for the Basic Autocoder 4K for IBM 1401 programming system. Also included are machine requirements; explanations of the source program, processor program, and the coding sheet; and information requirements necessary to write a Basic Autocoder 4K statement. The coding sheet used with this Basic Autocoder is the Autocoder Coding Sheet (X24-1350).

Descriptions of Basic Autocoder statements are presented in a special format that describes the operation which the statement performs, shows how the statement is written by the programmer, states the action of the processor program during processing of the symbolic program, describes the effect (if any) of the statement on the object program, and shows an example that uses the statement.

Preface

The IBM 1401 is a stored-program data processing system. It processes card records according to instructions developed by a programmer, then punched in cards, and stored in the computer. To execute the program, the instructions must be presented to the computer in a language it can understand. This language is known as *actual machine language.*

Instructions written in *actual machine language* can vary from one to eight characters in length. The first character is always an operation code identified with a word mark. Other parts of the variable-word-length instruction may consist of one or two operands known as A and B, and in some cases, a modifier known as a d-character. In most cases, operand-A designates the location (the address) of the data to be operated on, and operand-B designates the address of where the operation is to be performed. Addresses of A- and B-operands are always specified in three-character form regardless of the size of the storage unit.

Writing a program in *actual machine language* not only creates problems in assigning data and instructions to storage locations, but it also makes it difficult to write cross references within a program. Furthermore, difficulties can arise when the program is written by a team of programmers, resulting in programs that are difficult to correct and modify.

For these reasons, a symbolic programming system called Basic Autocoder 4K for IBM 1401 has been developed. This system relieves the programmer of the necessity for using actual operation codes and actual-machine-language addresses. Instead, he uses *mnemonic operation codes* that are easier to remember, and *labels* (symbols) of his own choice to designate address locations. For example, a machine-language in-instruction to add an amount in location 1205 to an accumulator in location 1323 would be written A S05 T23. In *symbolic language,* this same instruction can be written A 1205 1323, thus eliminating reference to the core-storage address code chart. If the addresses of the A- and B-operands are unknown to the programmer, he can designate the operands by using labels. The instruction could then be written A AMOUNT ACCUM, or A COST TOTAL.

Because the 1401 cannot execute the program in symbolic language, the program is first translated to actual machine language. A *processor program,* available from IBM, quickly and efficiently translates the source program, which is in symbolic language, to an *object program,* which is in actual machine language. It not only translates, but also determines how much storage is required and takes over the entire task of allocating storage locations to data and instructions. In other words, coding in actual machine language and punching the object program is done automatically. Hence, the name *Autocoder.*

This publication, C24-3171-0, includes the replacement pages issued with Technical Newsletter(s):

Form	Pages	Date
N24-0184	1, 2, 17, 18	2/25/64

Contents

Basic Autocoder 4K for IBM 1401: Specifications

The Basic Autocoder 4K for IBM 1401 simplifies programming for the IBM 1401 Data Processing System. The programming language permits the use of symbolic representations and mnemonic operation codes instead of actual core-storage addresses and operation codes. Provision is also made for defining and allocating areas, and controlling the assembly of a machine-language (object) program.

Note: In this publication, *Basic Autocoder* refers to the programming system *Basic Autocoder 4K for IBM 1401.*

Machine Requirements

The Basic Autocoder language and processor program can be used to produce an object program for any IBM 1401 Data Processing System. However, the machine used to process the source (symbolic) program must have at least:

4,000 positions of core storage
One IBM 1402 Card Read-Punch, Model 1, 2, or 4
One IBM 1403 Printer, Model 1, 2, 4, or 5

The source program written by the user is the input to the processor program which is supplied by IBM. The output from the processor is the object program that is in machine-language form ready for execution.

This publication contains language specifications for the IBM 1401 Basic Autocoder.

Programming with Basic Autocoder

A programmer's job is divided into two phases:

- *Defining* the problem to be solved.
- *Coding* the source program for assembly by the Basic Autocoder processor.

Start by outlining the requirements of the program. An example of such an outline is a block diagram. From this, decide what data, constants, work areas, and instructions are needed to execute the program.

Constants are fixed data whose values do not vary in a program.

Work areas are locations within core storage where data can be manipulated (such as input and output areas, accumulator fields, etc.). Once this has been done, symbols, instead of actual addresses, can be used to refer to areas, data, and instructions.

The IBM 1401 Basic Autocoder is divided into two major categories: the symbolic language used by the programmer to write the source program, and the processor program that translates this symbolic language and assembles a machine-language object program.

Symbolic Language

The symbolic language of the Basic Autocoder includes a set of mnemonic operation codes. They are easier to remember than the machine-language codes because they are usually abbreviations for actual instruction descriptions. For example:

Description	*Mnemonic*	*Machine Language Code*
Multiply	M	@
Clear Word Mark	CW	⌑

Figure 1 shows a list of mnemonic operation codes for the IBM 1401 Basic Autocoder. Also included in the language are mnemonics for statements that define and allocate areas, enter constants, control the area in core storage where the object program will be assigned, etc. These mnemonics have no machine-language equivalent.

The names (symbols) given to data, instructions, and constants are also part of the source program and are usually abbreviations for card fields, record names, and similar items that require frequent reference in the source program.

The Source Program

The source program consists of statements written in symbolic language. These statements contain the information that the processor must have to assemble the object program. This information is divided into three major categories:

- Area definitions (Declarative Operations)
- Instructions (Imperative Operations)
- Processor controls (Processor Control Operations)

Area-Definition Statements

Area-definition statements are used to reserve an area in core storage to store a constant, or to work with data before it is punched or printed. Area-definition statements, in most cases, do not produce instructions to be executed as part of the object program. For these statements the processor program produces cards containing constants and their assigned machine addresses. These cards are a part of the object program and are loaded each time the program is used.

DECLARATIVE OPERATIONS

Mnemonic	Description	Mnemonic	Description
DC	Define Constant	DS	Define Symbol
DCW	Define Constant with Word Mark	EQU	Equate

PROCESSOR CONTROL OPERATIONS

Mnemonic	Description	Mnemonic	Description
END	End	ORG	Origin
		XFR	Transfer

INPUT/OUTPUT OPERATIONS

Mnemonic Op Code	Description	Op Code	A/I-Address	B-Address	d-Character
R	Read Card	1	(XXX)		
RP	Read and Punch	5	(XXX)		
P	Punch	4	(XXX)		
SPF	Start Punch Feed	9			
SRF	Start Read Feed	8			
W	Write a Line	2	(XXX)		
WM	Write Word Marks	2	(XXX)		□
WP	Write and Punch	6	(XXX)		
WR	Write and Read	3	(XXX)		
WRP	Write, Read, and Punch	7	(XXX)		
LU	Load Unit	L	A-Address and/or B-Address and d-character supplied by programmer for I/O devices that do not have special mnemonics		
MU	Move Unit	M			
CU	Control Unit	U			

ARITHMETIC OPERATIONS

A	Add	A	(XXX)	(XXX)	
S	Subtract	S	(XXX)	(XXX)	
ZA	Zero and Add	?	(XXX)	(XXX)	
ZS	Zero and Subtract	!	(XXX)	(XXX)	
*D	Divide	%	XXX	XXX	
*M	Multiply	@	XXX	XXX	

DATA CONTROL OPERATIONS

MCE	Move Characters and Edit	E	XXX	XXX	
MCS	Move Characters and Suppress Zeros	Z	XXX	XXX	
MLC	Move Characters to A- or B-Word Mark	M	(XXX)	(XXX)	
MLCWA	Move Characters and Word Mark from A-Field	L	(XXX)	(XXX)	
MLNS	Move Numerical Portion of Single Character	D	(XXX)	(XXX)	
MLZS	Move Zone Portion of Single Character	Y	(XXX)	(XXX)	
*MRCM	Move Characters to Record Mark or Group Mark — Word Mark	P	XXX	XXX	

LOGIC OPERATIONS

B	Branch Unconditional	B	XXX		
BAV	Branch on Arithmetic Overflow	B	XXX		Z
*BBE	Branch if Bit Equal	W	(XXX)	(XXX)	d†
BCE	Branch if Character Equal	B	(XXX)	(XXX)	d†
BCV	Branch on Carriage Channel 12	B	XXX		@
BC9	Branch on Carriage Channel 9	B	XXX		9
BE	Branch on Equal Compare (B = A)	B	XXX		S
BH	Branch on High Compare (B > A)	B	XXX		U
BIN	Branch if Indicator On	B	XXX		d†
BL	Branch on Low Compare (B < A)	B	XXX		T
BM	Branch on Minus	V	(XXX)	(XXX)	K
*BSS	Branch if Sense Switch On	B	XXX		A-G†
BU	Branch on Unequal Compare (B ≠ A)	B	XXX		/
BW	Branch on Word Mark	V	(XXX)	(XXX)	1
BWZ	Branch on Word Mark or Zone	V	(XXX)	(XXX)	d†
C	Compare	C	XXX	XXX	

MISCELLANEOUS OPERATIONS

Mnemonic Op Code	Description	Op Code	A/I-Address	B-Address	d-character
CC	Control Carriage	F			d†
CS	Clear Storage	/	(XXX)	(XXX)	
CW	Clear Word Mark	□	(XXX)	(XXX)	
H	Halt	.	(XXX)	(XXX)	
*MA	Modify Address	#	XXX	(XXX)	
NOP	No Operation	N	(XXX)	(XXX)	
*SAR	Store A-Address Register	Q	XXX		
*SBR	Store B-Address Register	H	XXX	(XXX)	
SS	Select Stacker	K			d†
SW	Set Word Mark		(XXX)	(XXX)	

*Special Feature
†d-character must be coded in operand portion of instruction
(XXX) Address not required for some formats of instruction

Figure 1. IBM 1401 Basic Autocoder Mnemonic Operation Codes

For example, a constant card containing the date to be printed on the heading line of each invoice is loaded into core storage. A word mark is placed over the high-order position of the date. The date can then be moved, during object program execution, to a place in the print area in preparation for printing a heading line. To change the date, duplicate all columns in the constant card except the columns that contain the date itself. Then punch the new date into the card and insert it into the program deck in place of the outdated constant card.

Instruction Statements

Most of the statements in the source program are instructions that are used to read in data, process it, and write it out. The processor program translates the statements to machine-language instructions and causes the object program to be punched in cards. The processor automatically generates an additional sequence of instructions (called a *loader*) that loads the object program into the core storage.

Processor-Control Statements

The Basic Autocoder permits a limited amount of programmer control over the assembly process. For example, to locate a program in a particular area of core storage, direct the processor program to start assigning core storage at a specific address by writing an ORG (see *Origin*) processor-control statement. These statements are used by the processor during assembly.

All Basic Autocoder statements must be presented to the processor program according to a specific format. There are also rules and restrictions for writing the information in these statements. These requirements are necessary because the processor needs and can handle only certain kinds of information from each type of Basic Autocoder statement, and it must know where in the statement that information can be found.

The Processor Program

The 1401 processor program analyzes the information it receives when the source program statements are fed into the machine. As each statement is analyzed, the processor program assembles the machine-language instruction, or constant, and punches it into an output card. The punched output cards also contain the loader. Thus, the object program is called *self-loading*.

Coding Sheet

Basic Autocoder statements are written on a coding sheet designed to organize them into the format required by the processor program.

Enter all statements and comments to be included in the source program on this coding sheet. Column numbers on the sheet indicate the card-punching format for all cards in the source deck. Each line of the coding sheet is punched into a separate card.

Figure 2 shows the Autocoder Coding Sheet, Form X24-1350.

Page Number (Columns 1 and 2)

In this field, write the sequence number of the coding sheet. It may contain any valid 1401 characters. These should be used in the low-to-high-order of the 1401 collating sequence. Blank is the lowest character; 9 is the highest.

Line Number (Columns 3-5)

Use this field to indicate the sequence of entries on the page of each coding sheet. The units position of this field may be left blank. It can be used later to indicate the sequence of inserts on a page. The five unnumbered lines at the bottom of the page can contain these inserts.

For example, to make an insert between lines 02 and 03, use number 021. Line numbers on the coding sheet do not have to be consecutive, but the source deck should be in 1401 collating sequence when it is used as input to the processor. Line numbers may contain any valid 1401 characters.

Note that inserts can affect address adjustment. An insert may make it necessary to change the adjustment factor in one or more entries; see *Address Adjustment*.

Label (Columns 6-15)

This field may contain a symbolic label or it may be left blank, but it may not contain an actual or machine address.

A symbolic label may have as many as six alphameric characters, but the first character must be alphabetic. Special characters and blanks must not be used within a label. The label always starts in column 6. In the 1401 Basic Autocoder, columns 12 through 15 are always blanks.

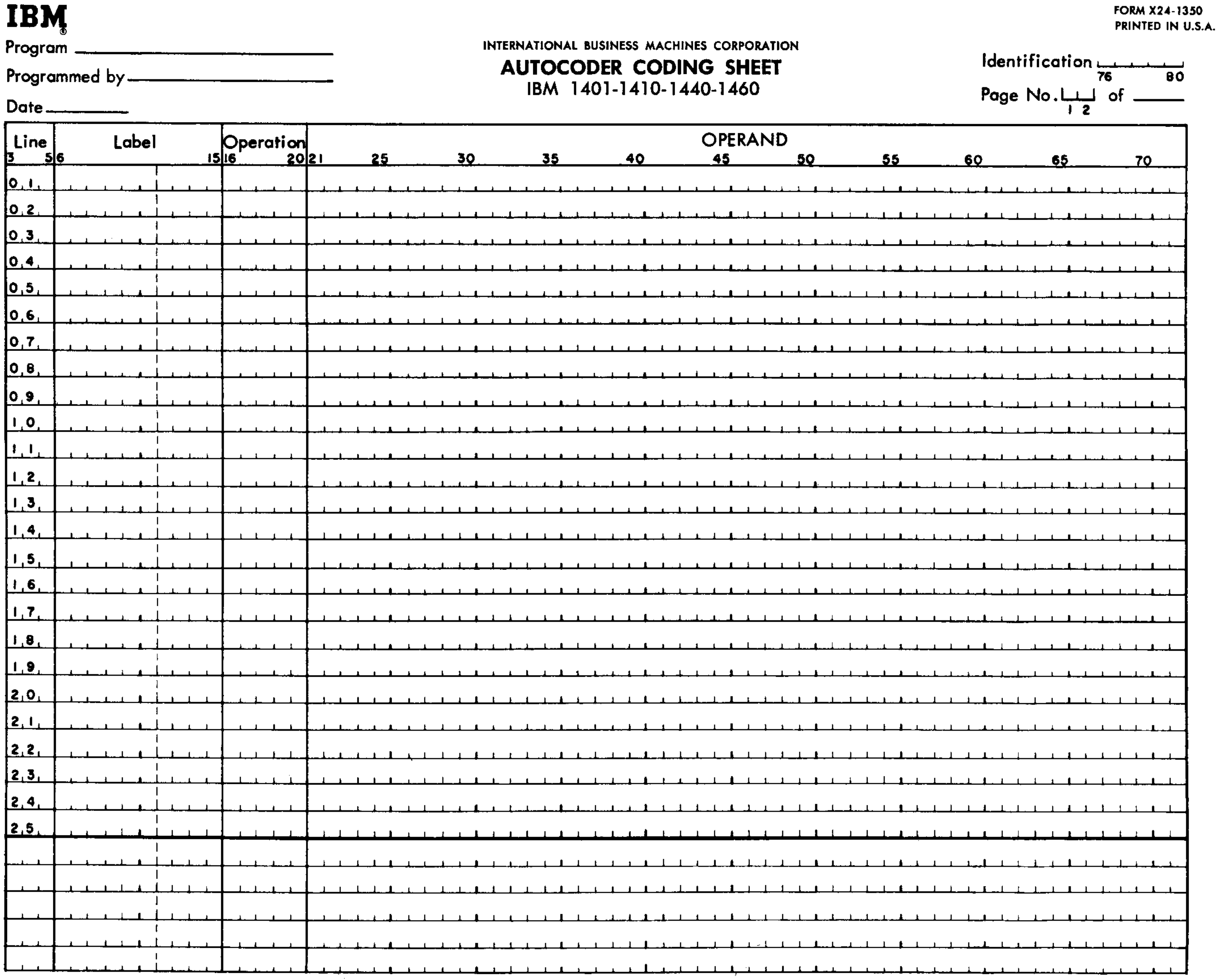
Operation (Columns 16-20)

A mnemonic operation code is written in the operation field starting in column 16.

Operand (Columns 21-72)

The operand field contains the addresses of the data to be operated upon and the d-character if one is required. The A/I-operand, B-operand, and the d-character must be separated by commas because the coding sheet format is free-form (the operand and d-character fields are not divided into fixed fields). To address-adjust or index an operand, place these codes immediately following the address being modified. Figures 3, 4, and 5 show typical Basic Autocoder statement formats.

Note: Leave columns 61-72 blank.



IBM

INTERNATIONAL BUSINESS MACHINES CORPORATION

AUTOCODER CODING SHEET

IBM 1401-1410-1440-1460

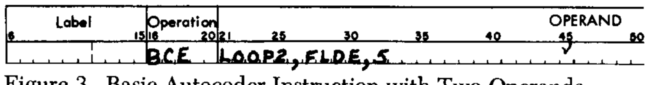
FORM X24-1350
PRINTED IN U.S.A.

Program ______
Programmed by ______
Date ______

Identification ______ 76 80
Page No. ___ 1 2 of ______

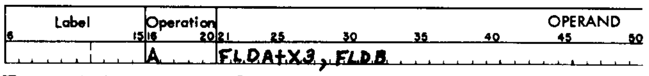
Line 3 5	Label 6 15	Operation 16 20	OPERAND 21 25 30 35 40 45 50 55 60 65 70
0 1			
0 2			
0 3			
0 4			
0 5			
0 6			
0 7			
0 8			
0 9			
1 0			
1 1			
1 2			
1 3			
1 4			
1 5			
1 6			
1 7			
1 8			
1 9			
2 0			
2 1			
2 2			
2 3			
2 4			
2 5			

Figure 2. Autocoder Coding Sheet



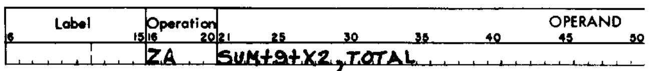
Label 6 15	Operation 16 20	OPERAND 21 25 30 35 40 45 50
	BCE	LOOP2,FLDE,5

Figure 3. Basic Autocoder Instruction with Two Operands and a d-Character



Label 6 15	Operation 16 20	OPERAND 21 25 30 35 40 45 50
	A	FLDA+X3,FLDB

Figure 4. Basic Autocoder Instruction with Indexing



Label 6 15	Operation 16 20	OPERAND 21 25 30 35 40 45 50
	ZA	SUM+9+X2,TOTAL

Figure 5. Basic Autocoder Instruction with Address Adjustment and Indexing

Comments

The programmer can include a remark anywhere in the operand field of a Basic Autocoder statement if he leaves at least two non-significant blanks between it and the operands.

Write a comments line to include a whole line of explanatory information anywhere in the source program. This line can contain comments only and must have an identifying asterisk in column 6. Use columns 7-60 for the comment. The information contained in a comments card appears in the symbolic-program listing produced by the processor during assembly, but it does not affect the object program in any way.

Blank (Columns 73-75)

Leave columns 73-75 blank.

Identification (Columns 76-80)

To identify a program or program overlay, assign it an identification number or description. Punch this identification into each card in the source deck. The processor does not use this field.

Other Coding Sheet Areas

The areas labeled *Program, Programmed by,* and *Date* are for the user's convenience only. Their contents are never punched into the source deck cards.

Information Requirements

Three general kinds of information can be written in a 1401 Basic Autocoder statement: labels, operation codes, and operands.

Labels

Labels are descriptive terms selected to identify a specific area or instruction in a source program statement. A label that suggests the meaning of the area or instruction makes coding easier. It also makes the program more easily understood by others. For example:

Type of Statement	*Meaning*	*Label*
Area Definition	Withholding Tax	WHTAX
Instruction	Update	UPDATE

The processor allocates storage and assigns addresses for all instructions and most area definitions. If the statement has a label, the processor equates the symbolic label to the assigned address referred to in this publication as the *equivalent address.* The equivalent address of the label for an *instruction* is the leftmost (high-order) core-storage position of the area the processor has allocated for it. The equivalent address of the label of an *area-definition statement* is the rightmost (low-order) core-storage position of the area the processor has allocated for the constant or work area. During assembly the processor maintains a table of labels and their equivalent addresses.

If a label appears in any Basic Autocoder statement, it may be written as a symbol in the *operand* portion of another Basic Autocoder statement. Thus, the programmer refers symbolically to the equivalent address of the constant, work area, or instruction. The processor substitutes the equivalent addresses of labels for their corresponding symbols in the symbolic program when it assembles the object program. No two labels used in a source program can be identical.

Operation Codes

All Basic Autocoder statements have operation codes. In imperative instruction statements they are machine-operation codes such as A (Add), S (Subtract), and P (Punch).

In area-definition statements they are commands to the processor to allocate storage such as DCW (Define a Constant with a Word Mark).

In processor control statements they are signals to the processor such as ORG (begin or originate the program) and END (end the program).

Operands

Use the operand portion of a Basic Autocoder statement to specify:

1. For instruction statements: The address of the data to be operated upon or the input/output units to be operated, and the d-character modifier to the operation code.
2. For area-definition statements: The constant or area to be defined, or the address that is to be the equivalent of the label.
3. For processor-control operations: The address to be used with the particular part of the processor program to be affected.

Thus, an operand can designate a core-storage address, an input-output unit, a particular operation to be performed (d-character), or a constant to be defined.

Core-Storage Address Operands

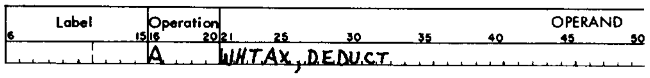
There are five types of core-storage address operands: symbolic, actual, asterisk, blank, and literals.

Symbolic Addresses

A symbolic address can have as many as six letters or digits but no special characters. Special characters have particular meanings when used in the operands of Basic Autocoder statements (Figure 6). If they are used as symbols, processing difficulties and errors can occur. Blanks must not be used within a symbolic address. The first (high-order) character of a symbolic address must be a letter. It refers to an area-definition or instruction statement in the source program whose label is identical to it. (This is the symbol previously described under *Labels*).

For example, if ENTRYA is used as a label for an instruction in the source program, ENTRYA can then be used as the symbolic operand of another instruction that references it, such as B ENTRYA (branch to the instruction whose label is ENTRYA).

Writing a symbolic operand in a statement that precedes the labeled statement in the source program is permitted. See *ORG* and *EQU* for exceptions.



Label (6–15)	Operation (16–20)	OPERAND (21–50)
	A	WHTAX,DEDUCT

Figure 6. Symbolic Addresses

Actual Addresses

The programmer may use an actual address as an operand in any Basic Autocoder statement. This address is a one- to five-digit number within the range 0 to 15999, representing a 1401 core-storage position. The programmer may use an actual address to assign a particular storage area instead of letting the processor assign it.

For example, to cause a word mark to be set in location 001 during execution of the object program, write in the symbolic program the instruction shown in Figure 7. Note that it is not necessary to write high-order zeros in an actual address written in Basic Autocoder.

Label 6–15	Operation 16–20	OPERAND 21 25 30 35 40 45 50
	SW	1

Figure 7. Actual Addresses

Asterisk Addresses

Writing an asterisk address operand in a Basic Autocoder statement directs the processor to assign an address. The processor will assign an address equivalent to the rightmost position the instruction (or last position assigned) is to occupy in core storage after the object program is loaded. See *EQU* and *ORG.*

Figure 8 shows a Basic Autocoder statement with an asterisk operand. Assume that during assembly the processor assigned the address 906 to the high-order position of this instruction. Because the instruction has seven characters, its low-order position will occupy core-storage location 912 in the object machine. The processor substitutes this address in the A-operand of the statement shown in Figure 8 and assembles it $\underline{M}$ 912 285. Thus, the machine-language instruction appears in core storage as shown in Figure 9 after the object program is loaded.

Blank Addresses

Blank addresses are valid in statements where no operand is needed or when useful addresses are supplied by the chaining method.

Label 6–15	Operation 16–20	OPERAND 21 25 30 35 40 45 50
	MLC	*,285

Figure 8. Asterisk Operand

Character	$\underline{M}$	9	1	2	2	8	5
Core-Storage Location	906	907	908	909	910	911	912

Figure 9. Instruction in Object-Core Storage

Literals

The 1401 Basic Autocoder can process three kinds of literals:

- Numeric Literals
- Alphameric Literals
- Address-Constant Literals

A literal refers to actual data to be operated upon by a particular instruction in the object program. It is possible to refer to this data by writing a literal operand in the instruction that uses it. For all literal operands the processor produces a constant that is loaded (with a word mark in the high-order position) as part of the object program.

The processor assigns a storage area for the constant and inserts the equivalent address of the constant wherever the literal operand appears in the symbolic program. Thus, the literal operand is the symbol that refers to the low-order position of the stored constant. The programmer may address-adjust and/or index a literal. See *Indexing* and *Address Adjustment.*

Figure 10 shows literal operands and the constants produced for them.

Type of Literal	Literal Operand	Stored Constant
Numeric	+10	$\underline{1}$?
Alphameric	@ TODAY@	$\underline{T}$ODAY
Address Constant	+CASH	$\underline{X}$XX (Equivalent Address of Cash)

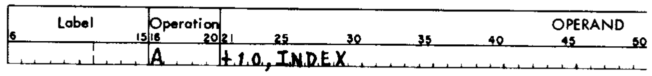
Figure 10. Literals

Numeric Literals. The specifications for writing a numeric literal (a number or digit) are:

1. Put a plus or minus sign ahead of the literal to tell the processor which sign is to be placed over the literal when it is loaded into the object machine prior to program execution. The maximum length of a numeric literal is five characters and a plus or minus sign. *Note:* To store an unsigned number, use an alphameric literal because an unsigned number would be confused with an actual address.

2. The processor assigns a storage area for a literal only once per program no matter how many times the same literal appears in the symbolic program.

Figure 11 shows how a numeric literal can be coded in a Basic Autocoder imperative instruction. Assume that the literal (+10) is assigned storage locations 584 and 585 and INDEX is assigned an equivalent address of 682. The symbolic instruction causes the processor to produce a machine-language instruction ($\underline{A}$ 585 682) that adds +10 to the contents of INDEX when the instruction is executed in the object program.



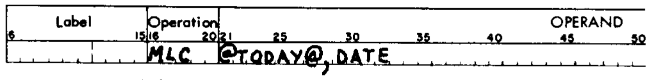
Label	Operation	OPERAND
	A	+1.0, INDEX

Figure 11. Numeric Literal

Alphameric Literals. The specifications for writing an alphameric literal are:

1. Write an @ symbol preceding and following the literal. The literal may contain *any* alphabetic, numeric, or special character in the 1401 character set, including the @ symbol itself. However, the maximum number of characters between @ signs is five. All characters appearing between the @ symbols are assumed by the processor to be part of the literal. One alphameric literal may appear in a symbolic program line.
2. The processor will assign to the literal an area only once per program.

Figure 12 shows how to use an alphameric literal in an imperative instruction. Assume that during assembly the literal TODAY is assigned a storage area whose equivalent address is 906 and DATE is assigned 230. For the statement shown in Figure 12 the processor produces a machine-language instruction (M̲ 906 230) which moves the literal TODAY to DATE.



Label	Operation	OPERAND
	MLC	@TODAY@, DATE

Figure 12. Alphameric Literal

Address-Constant Literals. With Basic Autocoder it is possible to define the 3-character machine address (the equivalent address) assigned to a label. The loading routine will load the 3-character address constant into core storage at program-load time. The programmer may refer to the area where the address is located by writing an address-constant literal in his source program.

To code an address-constant literal, write in the operand field the symbol whose equivalent address is needed. Precede the symbol with a plus sign. This symbol may have a maximum number of six characters. It must also appear elsewhere in the program and must have a corresponding label.

When the processor encounters an address-constant literal, it:

1. Assigns a 3-position area in the object machine that will contain the equivalent address of the symbol at execution time.
2. Makes the address of the 3-position area equivalent to the symbol preceded by a plus sign. For example, if CASH is the symbol whose address is needed as the address-constant literal, +CASH is the symbol that refers to the address of the equivalent address of CASH.

Figure 13 shows two address-constant literals (+CASH and +CHECKS) used in a source program. It also shows the entries the processor makes in the object program and the results when the instructions are executed in the object program. The programmer did not know which addresses would be assigned to CASH and CHECKS when he wrote the source program statements. He did, however, write two instructions (A and C) that move these addresses into instruction B (ENTRY1). The address-constant literals (+CASH and +CHECKS) cause the processor to substitute the equivalent addresses of these constants in instructions A and C during assembly. They also cause the loader to store the machine addresses of CASH and CHECKS into the object machine.

Indexing

If an object machine has the indexing and store address register special feature, the programmer specifies that an operand is to be indexed by following it with a plus sign and X1, X2, or X3, which represent index locations 1, 2, and 3, respectively.

X1, X2, and X3 are handled as actual locations and do not need definition. When the processor encounters an indexed operand, it puts tag bits over the tens position of the 3-character machine address assigned to the operand as shown in Figure 14.

In the example shown in Figure 15, assume that at object-program execution time the contents of an area labeled TOTAL are to be moved to a location whose address is equal to the sum of the equivalent address of ACCUM plus the contents of index location 2. Thus, if the actual instruction for the statement shown in Figure 15 is M̲ A01, 1MO, the contents of the TOTAL area, whose address is A01, are moved to the area whose address is equal to the sum of the equivalent address of ACCUM (140) and the contents of index location 2 at program-execution time. The M̲ in the tens position of the B-address is a 4-punch with an 11-zone punch. The zone punch specifies index location 2.

Note: Because X1, X2, and X3 are reserved by the Basic Autocoder program for index register reference, these names must not appear as labels in source-program statements even when the object machine does not contain the indexing and store address register special feature.

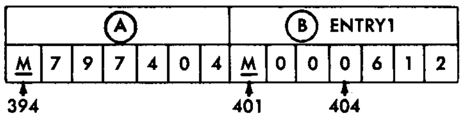
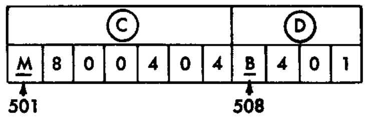
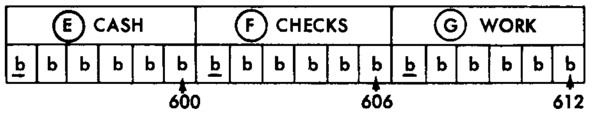
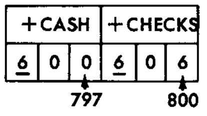
Address Adjustment

If address adjustment is specified in the operand fields of Basic Autocoder statements, it is not necessary to devise so many labels for a source program.

To do this, write a number preceded by a plus or minus sign immediately following the address in the operand field. The processor then develops an address equal to the address (actual, asterisk, literal, or symbolic) in the operand field, plus or minus the adjustment factor, and inserts it into the object-program

SOURCE PROGRAM STATEMENTS

```
Line  Label       Operation  OPERAND
3  5  6        15 16    20  21   25   30   35   40   45
0.1
0.2
0.3
0.4               MLC        +CASH, ENTRY1+3           A
0.5   ENTRY1      MLC        0,WORK                    B
0.6
0.7
0.8
0.9               MLC        +CHECKS,ENTRY1+3          C
1.0               B          ENTRY1                    D
1.1
1.2
1.3
1.4   CASH        DCW        @     @                   E
1.5   CHECKS      DCW        @     @                   F
1.6   WORK        DCW        @     @                   G
1.7
1.8
1.9
2.0
2.1
2.2
2.3
2.4
2.5
```

TYPE	Object program in core storage after it has been loaded into the object machine.
INSTRUCTIONS	 A B ENTRY1 M 7 9 7 4 0 4 M 0 0 0 6 1 2 394 401 404 
AREAS	 E CASH F CHECKS G WORK b b b b b b b b b b b b b b b b b b 600 606 612
ADD. CON. LITERALS	

SYMBOLS	EQUIVALENT ADDRESSES
ENTRY1	401
CASH	600
CHECKS	606
WORK	612
+CASH	797
+CHECKS	800

NOTE: Assume that before step A is executed, data will be moved into the CASH, CHECKS and WORK fields.

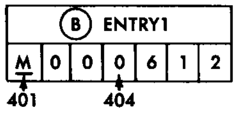
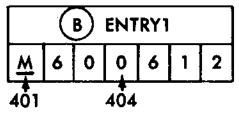
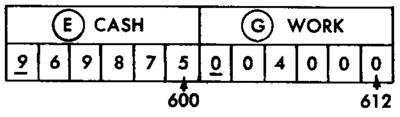
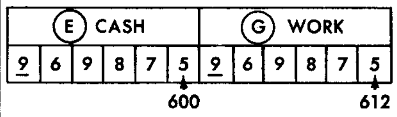
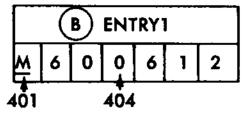
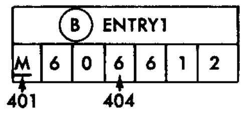
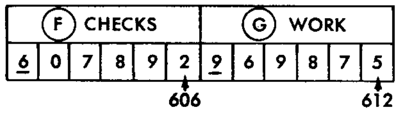
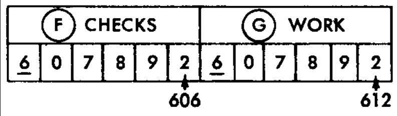
PROGRAM STEP EXECUTED	OPERATION	CORE STORAGE BEFORE OPERATION	CORE STORAGE AFTER OPERATION
A	The address of CASH is moved to the A-address of B (ENTRY 1+3). B is thus modified.		
B	The contents of CASH are moved to WORK.		
C	The address of CHECKS is moved to the A-address of B (ENTRY 1+3). B is again modified.		
D	Program branches back to execute B.	NO CHANGE	NO CHANGE
B	The contents of CHECKS are moved to WORK.		

Figure 13. Address Constant Literals

Index Location	Core-Storage Locations	3-character Machine Address	Zone Punch	Tag bits in tens position of 3-character machine address
1	087-089	089	ZERO	A-bit, No B-bit
2	092-094	094	ELEVEN	B-bit, No A-bit
3	097-099	099	TWELVE	A-bit, B-bit

Figure 14. Index Locations and Associated Tag Bits

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	MLC	TOTAL,ACCUM+X2

Figure 15. Basic Autocoder Instruction with Symbolic Addresses and Indexing

statement in place of the address adjusted operand. In the example shown in Figure 16 the first statement has an address adjusted operand.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	SBR	LAST+3
	.	
	.	
	.	
LAST	B	0

Figure 16. Address Adjustment

Assume that the statement whose label is LAST is assigned storage locations 404 through 407. The equivalent address of the label LAST is then 404, which is the position that the B operation code of the branch instruction will occupy in core storage when the object program is loaded.

The processor substitutes the address of LAST +3 (407) in place of the symbolic address-adjusted operand (LAST +3) when the object program is assembled:

$\underline{\text{H}}$ 407 . . . . B 000.

When the object program is executed, the contents of the B-address register are transferred to positions 405-407, so that I-address of the branch instruction contains whatever was in the B-address register before the SBR instruction was encountered ($\underline{\text{B}}$xxx).

The first statement in Figure 17 is an instruction that adds a literal (+100) to SUM. The processor allocates a 3-position area in core storage to store this literal. Assume that the equivalent address of this literal is 698 and SUM has an equivalent address of 805. Later in the source program the same literal appears with address adjustment. Because the literal has been previously assigned an area whose address is 698, the address-adjusted literal +100-2 refers to 698-2 or 696. Thus, the assembled instruction, $\underline{\text{A}}$ 696 805, will add 1 into SUM when it is executed in the object program, because storage location 696 contains the 1 portion of the literal +100.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	A	+100,SUM
	.	
	.	
	.	
	A	+100-2,SUM

Figure 17. Address-Adjusted Literal

The adjustment factor can be any number within the limits of its effect on the core storage available in the object machine.

Constant Operands

Constant operands are defined by area-definition statements. See *DCW* and *DC*. The processor assigns an area in core storage in which the constant is stored at object-program load time.

Statement Descriptions

In this publication the Basic Autocoder statement descriptions are presented in a format that:

1. Describes the operation which the statement performs.
2. Shows how the statement is written by the programmer.
3. States the actions of the processor program during processing of the symbolic program.
4. Describes the effect, if any, of the statement on the object program.
5. Shows an example that uses the statement.

Declarative Operations

As discussed previously, the 1401 Basic Autocoder permits writing literals to store constants. In addition, special declarative operations may be used to reserve work areas and store constants. The four declarative operations are:

Op Code	*Purpose*
DCW	Define Constant with Word Mark
DC	Define Constant (no Word Mark)
DS	Define Symbol
EQU	Equate

DCW — Define Constant with Word Mark

General Description. Use a DCW statement to enter a numeric, alphameric, blank, or address constant into core storage at object-program load time.

The programmer.

1. Writes DCW in the operation field.
2. May write a symbolic label in the label field. He can refer to the constant by writing this symbol in

the operand portion of instructions elsewhere in the program. The equivalent address of the label is the address of the low-order position of the constant in the object machine.
3. Writes the constant in the operand field beginning in column 21.

The processor.
1. Allocates a field in core storage that will be used at object-program load time to store the actual constant.
2. Inserts the equivalent address of the label in place of the symbol, whenever it appears in the operand field of another source-program entry.

Result. The constant with a high-order word mark is loaded with the object program.

Numeric Constants

A plus or minus sign may be written preceding a number. A plus sign causes the processor to store the constant with A- and B-bits over the units position; a minus sign causes the processor to store a B-bit over the units position. If a numeric constant is unsigned, it will be stored as an unsigned field.

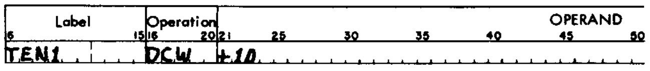
The first blank column in the operand field indicates that the preceding position contains the last digit in the constant.

A constant may be as large as 39 digits with a sign, or 40 digits with no sign.

Examples. Figures 18, 19, and 20 show the three types of numeric constants that can be defined in DCW statements. The labels TEN1, TEN2, and TEN3 identify the constants. Thus, they can be used as symbols to cause the equivalent addresses of +10, −10, and 10 to be inserted into the object program whenever TEN1, TEN2, and TEN3 appear in the operand fields of entries in the source program.

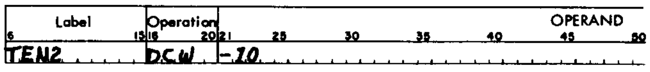
Alphameric Constants

Place an @ symbol before and after the constant. As with alphameric literals, blanks and the @ symbol may appear between these @ symbols, but the @ symbol must not appear in a comment in the same line as the constant.



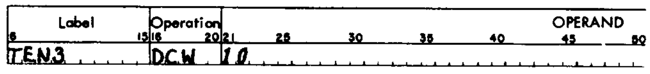
Label	Operation	OPERAND
TEN1	DCW	+10

Figure 18. Numeric Constant with a Plus Value



Label	Operation	OPERAND
TEN2	DCW	-10

Figure 19. Numeric Constant with a Minus Value

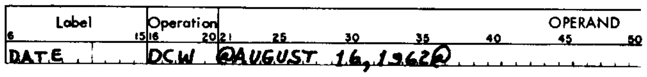


Label	Operation	OPERAND
TEN3	DCW	10

Figure 20. Unsigned Numeric Constant

An alphameric constant may contain as many as 38 valid 1401 characters.

Example. Figure 21 shows how to define the alphameric constant AUGUST 16, 1962 in a DCW statement. The processor will insert the equivalent address of the constant into the object-program instruction wherever DATE appears in the operand of another symbolic-program entry.



Label	Operation	OPERAND
DATE	DCW	@AUGUST 16, 1962@

Figure 21. Alphameric Constant

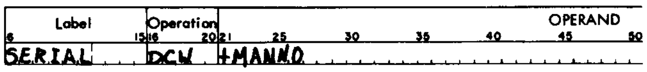
Address Constants

Write a symbol that may be preceded by a plus sign in the operand field.

The address constant is the three-character machine-language address of the field whose associated label appears in the operand.

Address constants may be address-adjusted and indexed. The address adjustment and indexing refer to the address constant itself rather than to the address of the location of the address constant. For example, if CASH is the symbolic address of a field, the equivalent address of CASH is indexed or address-adjusted rather than the equivalent address of +CASH.

Example. Figure 22 shows how an address constant (the equivalent address of MANNO) may be defined by a DCW statement. The address of the address constant (the address of the equivalent address of MANNO) will be inserted into an object-program instruction wherever SERIAL appears as the operand of another symbolic-program entry. Thus +MANNO is the symbolic address of the field that contains the equivalent address of MANNO.



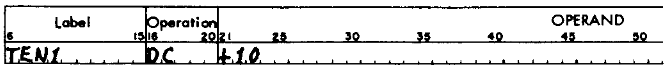
Label	Operation	OPERAND
SERIAL	DCW	+MANNO

Figure 22. Address Constant Defined by a DCW Statement

DC — Define Constant (No Word Mark)

General Description. To load a constant without a word mark, write a DC statement like a DCW statement. The DC operation code is used in the operation field.

Example. Figure 23 shows TEN1 defined as a constant without a word mark.



Label	Operation	OPERAND
TEN1	DC	+10

Figure 23. Constant Defined in a DC Statement

DS — Define Symbol

General Description. Use a DS statement to label and skip over an area of core storage. The bypassed area is undisturbed during the loading process. Thus, any information that was in storage before loading begins will still be there after the object program has been loaded.

The programmer.

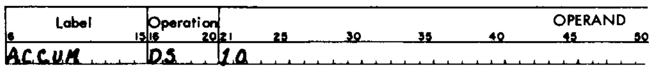
1. Writes DS in the operation field.
2. May write a symbolic address in the label field.
3. Writes a number in the operand field that tells the processor how many positions of storage to bypass.

The processor.

1. Assigns an equivalent address to the label. This address is the 3-character machine address of the low-order position of the bypassed area.
2. Inserts this address wherever the symbol in the label field appears in the operand field of another program entry.

Result. The positions included in the bypassed area remain undisturbed during object-program loading.

Example. Figure 24 shows how to direct the processor to bypass a 10-position core-storage area. Assume that the last core-storage position the processor allocated before it encountered the DS statement was 940. The equivalent address of ACCUM is 950, the address of low-order position of the core-storage area bypassed by the DS statement. Wherever ACCUM is written in the operand field of another source-program entry, 950 will be inserted into its place.



Label	Operation	OPERAND
ACCUM	DS	10

Figure 24. DS Statement

EQU — Equate

General Description. Use an EQU statement to assign a symbolic label to an actual, asterisk, or symbolic address. More than one symbol may be assigned to represent the same storage location.

The programmer.

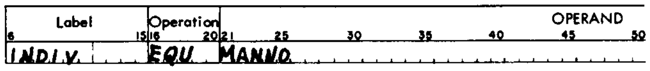
1. Writes EQU in the operation field.
2. Writes a symbolic address of an operand in the label field.
3. Writes an actual, asterisk, or symbolic address in the operand field, which may be address-adjusted but not indexed.

The processor.

1. Assigns to the label of an EQU statement the same 3-character machine address that is assigned to the symbol in the operand field (with the appropriate alteration if address adjustment is indicated).
2. Inserts this equivalent address wherever the symbol in the label field of the EQU statement appears as the operand of another source-program entry.

Result. Either the symbol in the label field or the symbol in the operand field of the EQU statement can be used to refer to the same core-storage location.

Examples. Figure 25 shows how to assign another label (INDIV) to a location that was previously labeled MANNO. The EQU statement causes the processor to assign the same equivalent address (1976) to INDIV that it previously assigned to MANNO. Now, whenever either MANNO or INDIV appears in the operand of another source-program entry, the processor will replace the symbol with 1976.

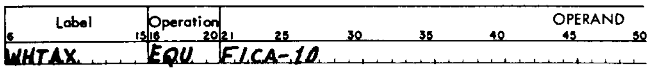


Label	Operation	OPERAND
INDIV	EQU	MANNO

Figure 25. Equating Two Symbolic Addresses

Note: If a symbolic address is used in the operand field of an EQU statement, its corresponding label must be defined ahead of the EQU statement in the source program.

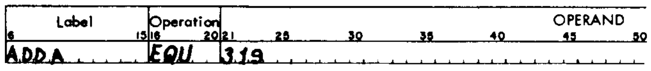
Figure 26 shows a statement equating the equivalent address of FICA-10 to WHTAX. Assume that the processor assigned FICA an equivalent address of 890. WHTAX will be assigned an equivalent address of 880, which is also equal to FICA-10. WHTAX now refers to a field whose units position is 880.



Label	Operation	OPERAND
WHTAX	EQU	FICA-10

Figure 26. Equating a Symbolic Address to an Address-Adjusted Symbolic Address

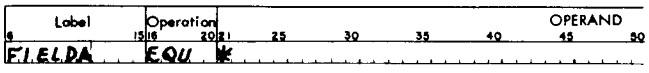
Figure 27 shows how to equate a label to an actual address. Assume that a certain field will be in a storage location whose units position is known to be at actual address 319. The programmer wishes to refer to this field as ADDA, but it has not been labeled elsewhere in the program. To equate the symbolic address ADDA to 319, write the statement shown in Figure 27. Thus 319 becomes the equivalent address of ADDA.



Label	Operation	OPERAND
ADDA	EQU	319

Figure 27. Equating a Symbolic Address to an Actual Address

Figure 28 shows how to assign a label to an asterisk address operand in an EQU statement. The * refers to the low-order position of the instruction with which it is associated. Assume that this address is 698. FIELDA has an equivalent address of 698.



Label	Operation	OPERAND
FIELDA	EQU	*

Figure 28. EQU Statement with an * Operand

Imperative Operations

General Description. These imperative operations include all the machine instructions in the 1401 instruction set. They are the symbolic instructions for the commands to be executed by the object computer. A source program will probably contain more of these imperative instructions than any other type of Basic Autocoder statement.

Although the 1401 Basic Autocoder processor can assemble instructions with all the imperative mnemonic operation codes listed in Figure 1, the programmer must remember the particular features and devices that will be included in the object machine for which he is writing the program.

The programmer.

1. Writes the mnemonic operation code for the instruction in the operation field.

2. If the instruction is to be referred to, the programmer can label such an instruction by writing a symbol in the label field. The label will have an equivalent address which is the storage location that will hold the operation code of the associated instruction when the object program is loaded.

Thus, the symbol in the label field can be used as the I-address of a branch instruction elsewhere in the program (see Figure 31).

3. Writes the A/I- or B-operand (see *Operands*) for the data, devices, constants, or instructions in the operand field. Literals may also be written in the operand field (see *Literals*). The first operand will be used as the A- or I-address of the imperative instructions.

If the instruction also requires a B-address, a comma must follow the first operand and its address adjustment and/or indexing codes (if any). Then the operand for the B-address is written. If the instruction requires that the programmer specify the d-character, a comma must precede the machine-language d-character. The d-character is always at the immediate right of the operands.

Note: Several mnemonic operation codes have been developed that cause the d-character to be supplied automatically by the processor. However, some operation codes (for example, BIN) have so many valid d-characters that it is impractical to provide a separate mnemonic for each. For these operation codes, the programmer must supply the d-character, as mentioned previously. In the listing of mnemonic operation codes for imperative instructions (Figure 1), all mnemonics that require a d-character in the operand field are indicated by a dagger (†).

The processor.

1. Substitutes the machine-language operation code for the mnemonic in the operation field.

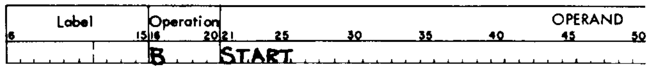
2. Substitutes the 3-character equivalent address of the symbols written in the operand field to indicate the A/I- or B-address of the instructions.

If address-adjustment or indexing codes are written with these operands, the appropriate alteration will be made for these addresses. Tag bits will be inserted into the tens position of indexed operands. Address-adjusted operands will be modified by adding or subtracting the adjustment factor. The processor will supply the d-character for unique mnemonics, or place in the instruction the d-character from the operand field of the Basic Autocoder statement if the programmer has supplied it.

3. Assigns the machine-language instruction in an area of core storage in the object machine. The address of this area is the storage location which the operation code will occupy when it is loaded into the object machine. This is the equivalent address of the label in the source program statement.

Result. The instruction is placed in the object-program deck. The load routine causes a word mark to appear in the operation-code position of the instruction in the object machine.

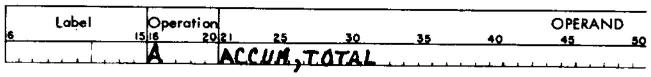
Examples. Figure 29 shows an imperative instruction with an I-operand. When the instruction is executed, a branch to the instruction whose label is START will occur. Assume that START has an equivalent address of 360. The instruction will be assembled $\underline{B}$ 360.



Label	Operation	OPERAND
	B	START

Figure 29. Unconditional Branch

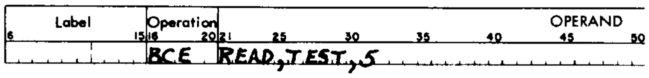
Figure 30 shows an imperative instruction with A- and B-operands. This instruction, when executed, causes the contents of ACCUM to be added to the contents of TOTAL. Assume that the equivalent addresses of ACCUM and TOTAL are 495 and 520, respectively. The processor will assemble the machine-language instruction $\underline{A}$ 495 520.



Label	Operation	OPERAND
	A	ACCUM,TOTAL

Figure 30. ADD Instruction

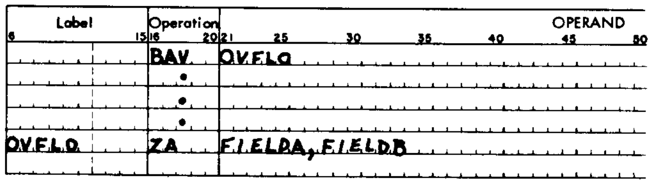
Figure 31 shows an imperative instruction with I- and B-operands and a mnemonic (BCE) which requires that the programmer supply the d-character (5) in the operand. When this instruction is executed in the object program, a branch to the instruction whose label is READ will occur if the location labeled TEST contains a 5. Assume that the equivalent address of READ is 596 and TEST is 782. The assembled instruction will be B 596 782 5.



Label	Operation	OPERAND
	BCE	READ,TEST,5

Figure 31. Branch If Character Equal

Figure 32 shows an imperative instruction with a unique mnemonic (BAV). The processor supplies the d-character (Z) for this instruction when it is assembled. Assume that OVFLO is assigned an equivalent address of 896. When the program is executed, the first instruction will cause a branch to OVFLO if an arithmetic overflow occurs. The assembled instruction is B 896 Z.



Label	Operation	OPERAND
	BAV	OVFLO
	•	
	•	
	•	
OVFLO	ZA	FIELDA,FIELDB

Figure 32. Branch If Arithmetic Overflow

Processor Control Operations

These are the Basic Autocoder statements that permit the programmer to exercise some control over the assembly process:

Operation Code	*Purpose*
ORG	Origin Assembly
XFR	Transfer
END	End Assembly

ORG — Origin

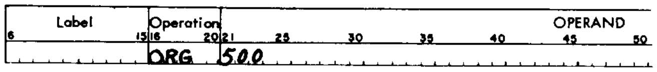
General Description. Use an origin card to tell the processor the address at which to begin allocating storage for the program or a particular part of the program. An ORG statement may be included anywhere in the source program.

The programmer.
1. Writes ORG in the operation field.
2. Writes a symbolic, actual, or asterisk address in the operand field. This address indicates the next storage location to be assigned by the processor. Symbolic or asterisk addresses may have address adjustment. An operand in an ORG statement may not be indexed.
3. If a symbolic address is used in the operand field of an ORG statement, its corresponding label must have been defined previously.

The processor.
1. Assigns addresses to instructions, constants, and work areas beginning at the address specified in the operand field of an ORG statement.
2. If no ORG statement precedes the first entry in the symbolic program, the processor automatically begins allocating storage locations starting at address 334.
3. If the processor encounters an ORG statement anywhere in the symbolic program, it begins allocating storage for subsequent entries beginning at the address specified in the operand field of the new ORG statement.

Result. The programmer can choose the area(s) of core storage where the object program will be located.

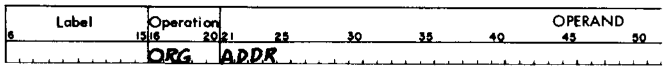
Examples. Figure 33 shows an ORG statement with an actual address. The processor will assign storage to the first symbolic-program entry following this ORG statement with storage location 500 as a reference point. This means that if the first entry following the ORG statement is an instruction; the Op-code position of that instruction will be 500. If the first entry is a 5-character DCW, it will be assigned address 504.



Label	Operation	OPERAND
	ORG	500

Figure 33. ORG Statement with an Actual Address

Figure 34 shows an ORG statement with a symbolic address. The processor will begin assigning addresses with the actual address assigned to ADDR.

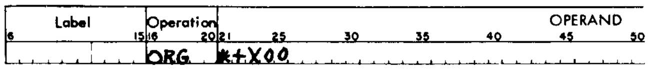


Label	Operation	OPERAND
	ORG	ADDR

Figure 34. ORG Statement with a Symbolic Address

When the processor encounters the statement shown in Figure 35, it will begin assigning addresses to subsequent entries in the source program at the next available storage location whose address is a multiple of 100. For example, if the last address assigned was 525, the next instruction (if the entry is an instruction) will have an address of 600.

Note: +X00 is permitted as a character-adjustment factor only when it is used with an *, and it may be used only in an ORG statement.



Label	Operation	OPERAND
	ORG	*+X00

Figure 35. Adjustment to the Next Available Century Block of Storage

XFR — Transfer

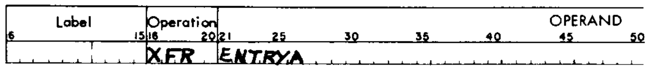
General Description. An XFR statement interrupts the object-program loading process temporarily so that the part of the program that has already been loaded can be executed.

The programmer.
1. Writes XFR in the operation field.
2. Writes an actual or symbolic address in the operand field. This must be the same symbol as the label used for the first instruction to be executed after the loading process has been halted.

The processor. Assembles an unconditional branch instruction. The I-address of this instruction is the equivalent address of the first instruction to be executed after the loading process has been halted. This instruction does not become part of the object program. However, it is used for the loading routine to signify the halt and to transfer machine-instruction execution to the object program.

Note: To continue the loading process after the desired part of the object program has been executed, the programmer must provide re-entry to the load routine.

Example. Figure 36 shows how an XFR statement can be coded. When the loader encounters the branch instruction produced from this statement, the loading process stops and a branch occurs to the instruction whose label is ENTRYA.



Label	Operation	OPERAND
	XFR	ENTRYA

Figure 36. XFR Statement

END — End

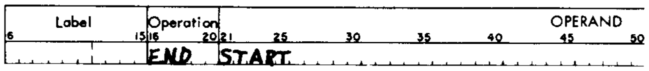
General Description. The END statement signals the processor that all of the symbolic-program entries have been read. This card, which is always the last card in the source program deck, provides the processor with the information necessary to produce a branch instruction that causes a transfer to the first instruction to be executed after the object program has been loaded.

The programmer.
1. Writes END in the operation field.
2. Writes an actual or symbolic address in the operand field. This must be the same symbol as the label used for the first instruction to be executed after the loading process has been completed.

The processor.
1. Assembles an unconditional branch instruction. The I-address of this instruction is the equivalent address of the first instruction to be executed after the loading process has been completed. This instruction does not become part of the object program. However, it is used by the loading routine to transfer machine-instruction execution to the object program.
2. Causes literals that have previously been encountered to be included at this point in the object program.

Result. Object-program execution begins automatically after loading.

Example. Figure 37 shows an END statement.



Label	Operation	OPERAND
	END	START

Figure 37. END Statement

Programming Considerations

Reducing the number of generated symbols required to complete an assembly reduces the size of the address table that must be in storage at load time. It also minimizes the number of generated symbols that appear in the printed listing.

The following suggestions for reducing the size of the address table are offered:

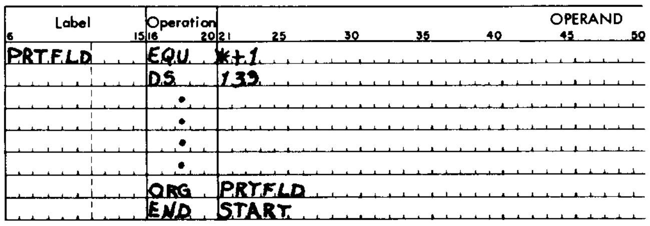
1. Place constants and work areas at the beginning of the program whenever it is possible. If a constant is defined before its symbol is encountered in the operand of a source program statement, the address of that constant is available. Thus, the processor does not have to generate a □nn symbol.
2. Minimize the use of literals. The processor must generate a □nn symbol for each literal encountered.
3. Use an * symbol with address adjustment (instead of a unique label) wherever a few characters separate instructions involving a forward reference.
4. Place subroutines near the beginning of the program whenever possible. If this is done, the processor will have the addresses of the subroutines available when it encounters instructions that refer to them.

Programmer Control over Placement of Table of Addresses

As described previously, the processor allocates storage for a table of addresses for generated symbols. If literals are used in the source program, this table immediately follows the last literal. If literals were not used, the table immediately follows the last instruction or constant encountered before the END statement. Literals and the table of addresses, if present, are loaded before any of the source program statement cards.

If literals are *not* used in the source program, the programmer may instruct the processor to put the table of addresses in an area he chooses. This is accomplished by putting an ORG card containing the address of this area ahead of the END card in the source program. With this facility, the programmer may overlay the table in the object program area. Thus, no additional core storage is needed for the table. Note that the table of addresses must be shorter than the area into which it is being loaded.

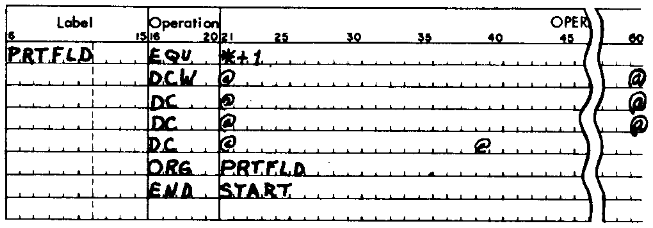
For example, assume that the programmer has used a DS statement to reserve an input/output area. Because nothing is loaded in an object program for a DS statement, this space can be used to store the table during the loading of the object program. Figure 38 shows an example in which PRTFLD is the area the programmer has selected to store the table. The table of addresses will begin at the address the processor assigned to PRTFLD.



```
Label      Operation  OPERAND
6       15 16     20  21   25   30   35   40   45   50
PRTFLD     EQU        *+1
           DS         133
           .
           .
           .
           .
           ORG        PRTFLD
           END        START
```

Figure 38. Table Overlay Using DS

Figure 39 shows another technique for overlaying the table of addresses. In this example the 133 character PRTFLD area (defined by DC and DCW statements) must be placed at the end of the source program deck immediately followed by the ORG PRTFLD and END statements. References to it will require a generated symbol, because the label appears after the symbolic operands in the instructions ahead of the EQU statement that uses it.

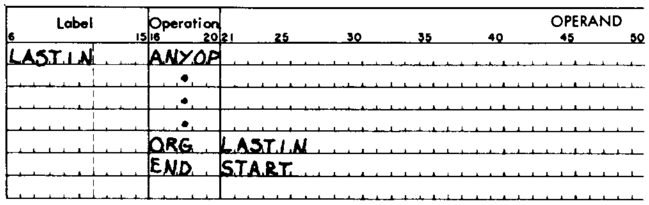


```
Label      Operation  OPER
6       15 16     20  21   25   30   35   40   45     60
PRTFLD     EQU        *+1
           DCW        @                                @
           DC         @                                @
           DC         @                                @
           DC         @                 @
           ORG        PRTFLD
           END        START
```

Figure 39. Table Overlay Using DCW and DC

Note: If the programmer wishes to put PRTFLD into a specific area of core storage, he may do this by placing an ORG statement immediately before PRTFLD.

Figure 40 is an example in which the table is overlaid by the last instructions in the program. When this method is used, the instructions that overlay the table must not contain any generated symbols.



```
Label      Operation  OPERAND
6       15 16     20  21   25   30   35   40   45   50
LASTIN     ANYOP
           .
           .
           .
           ORG        LASTIN
           END        START
```

Figure 40. Table Overlay Using Instructions

631007MTC

Index

IBM 1401 Printed in U.S.A. C24-3171-0

IBM®
International Business Machines Corporation
Data Processing Division