IBM

Reference Manual
IBM 7950 Data Processing System
Assembly Program HAP III

IBM

Reference Manual
IBM 7950 Data Processing System
Assembly Program HAP III

Address comments concerning this manual to:
IBM Corporation
Customer Manuals, Dept. 298
P. O. Box 390
Poughkeepsie, N. Y.

CONTENTS

ASSEMBLY PROGRAM HAP III

HAP III is a system dependent assembly program which is part of the language processor and ultimately assembles all programs to be used within the IBM 7950 Data Processing System. HAP III consists of Autocoder I, HAP II, and the Language Processor Coordinator (LPC) (Figure 1). Autocoder I functions as a macro statement expander, handling the input macro statements from the 7950 machine control program (HMCP) and the 7950 operational system control program (HCP), among others. HAP II provides a one-to-one relationship between input coding and machine language statements, while LPC coordinates the activities of Autocoder I, and HAP II with the 7950 system.

Input to HAP III consists of:

1. Programs written in HAP language, including arithmetic and streaming mode instructions, and various macro statements. These statements are punched on cards and loaded onto 729 tape.

2. HAP language statements produced by the language processor from statements written in ALPHA language.

Output of HAP II consists of:

1. Assembly list tape, which is a 729 tape containing the symbolic output for those programs written in HAP language.

2. Binary output tape, which is a tractor tape containing each assembled program in machine language and in a form ready for insertion into the system library by the library maintenance program.

AUTOCODER I

In HAP III, the Autocoder I portion processes the input statements and passes them on to HAP II. Incoming macro statements are expanded into HAP language calling sequences, and arithmetic and streaming mode instructions, already in HAP II language format, are not directly operated upon. The programmer need take no special note of the existence of Autocoder I; if input statements to HAP III are written in the proper format, their ultimate appearance in machine language will be completely automatic.

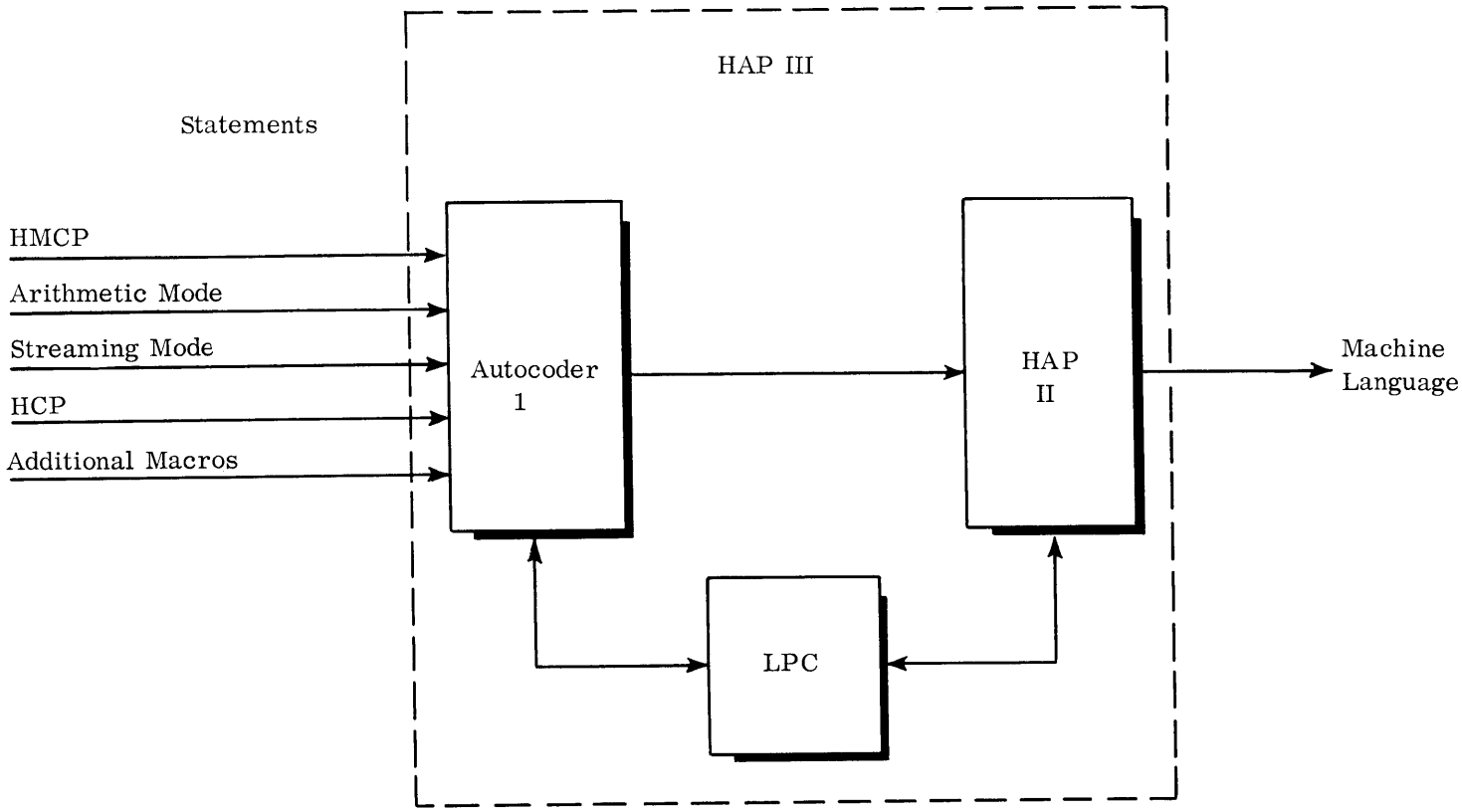


Figure 1. Components of HAP III

HAP III MACRO STATEMENTS

Macro statements have the following general form:

MOP, argument_1, argument_2, etc.

where MOP represents an operation mnemonic prefixed by the letter M and followed by up to seven additional alphabetic characters. Any arguments required, including null arguments, are written in a specified order in the statement field, separated by commas. Each argument can be a literal or a symbolic address plus or minus an absolute increment and modified by an absolute index register. For each symbolic address a maximum of eight alphanumeric characters are permitted, the first of which is alphabetic.

As an example, consider the following sample MADD macro statement:

MADD, A, B, C

With these parameters, the meaning is to load A in the adder, add B and store the result in location C. To accomplish this, Autocoder I will expand the above macro statement into the following calling sequence:

```
  L, A
  +, B
 ST, C
```

The expanded macro, or calling sequence, replaces the original macro in the source program, and is subsequently assembled by HAP II along with the other symbolic statements.

PROGRAMS IN HOPS

The 7950 operational system (HOPS) accommodates an advanced programming language (ALPHA) and a program to handle input, output, and tractor data manipulation (HMCP). Another HOPS program, the 7950 assembly program (HAP), acts as a programming language in itself and as a step in ALPHA compilation. All of these functions are integrated under the 7950 operational system control program (HCP).

The HCP handles the processing of jobs in cycles. A cycle may consist of one or more jobs; a job may consist of one or more steps. Within a cycle, jobs are logically processed in the order submitted. When presented to the HCP, the cycle deck allows completely automatic scheduling and transition from step to step and from job to job for the duration of the cycle.

OPERATIONAL CYCLES

Two distinct modes in which an operational cycle can be run are the HCP production cycle and the HCP debug cycle. Many functions of an operational cycle are common to both modes; however, certain system functions are not needed in a debug cycle while others are needed only in a debug cycle. Those functions not needed in a debug cycle (such as operations on permanent tractor data files, permanent library, and so on) usually are not made available during a debug cycle for system efficiency and for prevention of damage to permanent material during debugging. Those functions needed only in a debug cycle (such as debug dump facilities) are not made available in a production cycle.

Whether a cycle is a debug or a production cycle is determined by the machine operator and depends on a parameter furnished with the job requested. Flow charts of production and debug cycles are given in Figure 2. Note the number of steps within an operational cycle. These steps form the HCP system, together with several other programs and stored procedures.

TYPES OF PROGRAMS AND PROCEDURES

The HCP system treats stored procedures and various programs differently depending on their nature, use, and state of readiness. The broad categories established by the system are:

System Programs: Those programs which make up the HCP. A system program is treated as a problem program during language processing and debugging. It is labeled and treated as a system program during and after replacement in the system library. System programs include System Control, Job Request Analyzer, Language Processor, Program and Procedure Library Maintenance, Program Pre-Execution Supervisor, Tractor Data Load, Tractor Assignment Optimizer, Program Execution Supervisor, Tractor Data Unload, and Tractor Filing System Maintenance.

Problem Programs: Those operational programs executed with the aid and supervision of the system programs, by requesting execution through the job request language. While a generalized file operation (GFO) may be considered an integral part of HOPS, the HCP makes no distinction between a GFO and a special purpose program.

Debug Programs: System or problem programs in the process of being debugged.

Stored Procedures: Job requests having a frequency of use are convenient to retain in a system library. A stored procedure may contain any JRL request, parameter, or file card except a stored procedure request or modification card.

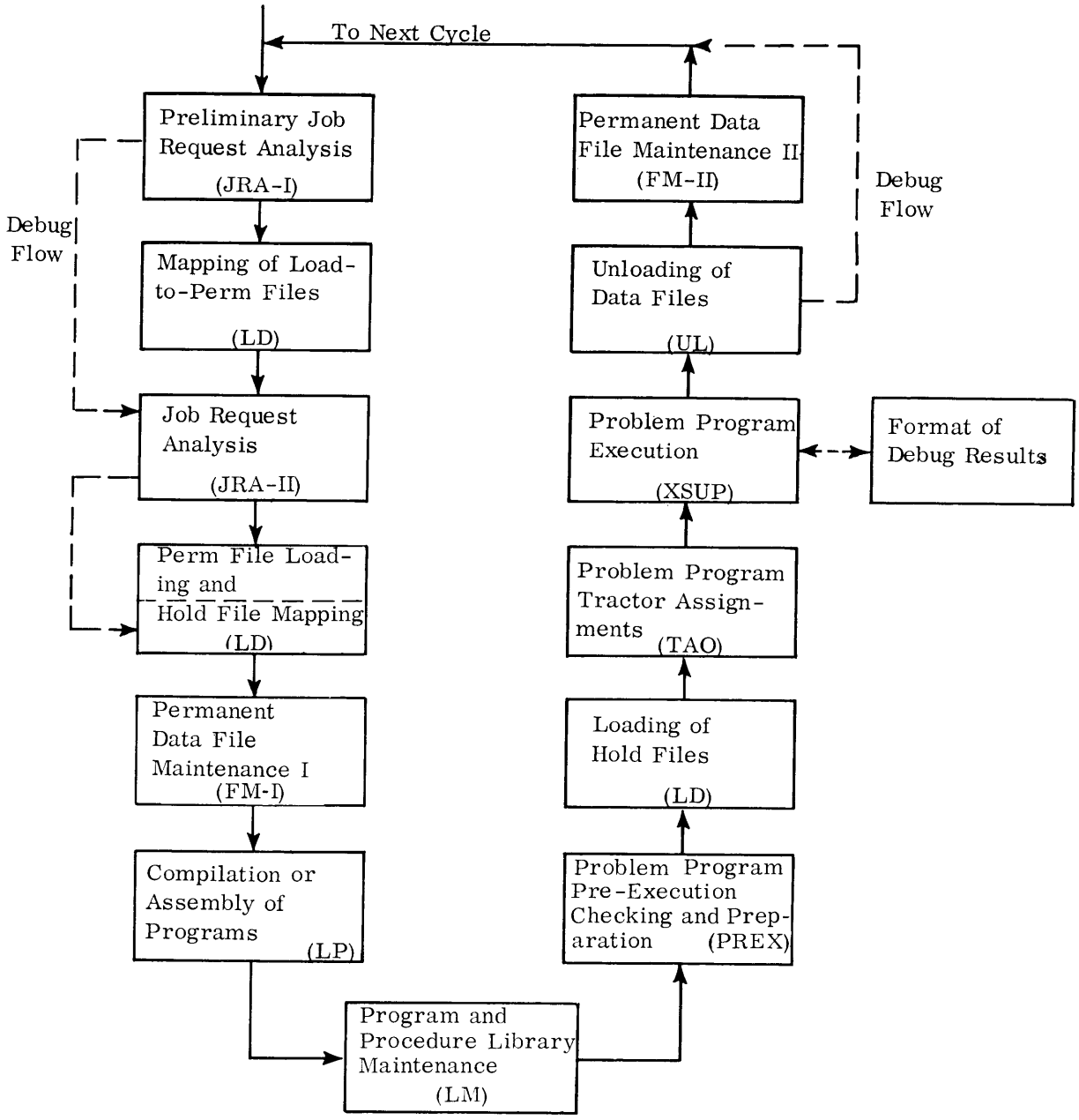


Figure 2. Operational Cycle Flow Chart

THE LANGUAGE PROCESSOR

The HOPS system provides a language processing function for the programmer, enabling him to write his programs on the most applicable level and to use more advanced programming language techniques.

Programs normally are written in symbolic language, then compiled by the language processor before being placed in the system library or executed. The HOPS system also relieves the programmer of much of the burden of data file loading, unloading, bookkeeping, conversion, and hardware assignments. In addition, it contains powerful tools for program debugging and it provides the programmer permanent storage for his program in the system library.

In operation, the compilation or assembly of programs can take place in either a debug or production mode; however, requests for assembly or compilation in a debug mode are valid only in a debug cycle. Programs are compiled according to requests and options in the Internal Request Table, and parameters in the parameter file.

The actual assembly of the program is done by HAP III. Input to the language processor must be in ALPHA or HAP language. ALPHA language programs can be given to HAP III only after being processed by the previous phases of the language processor.

The output of the language processor is actually HAP III output and appears in two forms: Symbolic output is recorded on a 729 list tape for possible off-line printing, and binary machine language output is recorded on a tractor tape for subsequent inclusion in the library.

SUBROUTINE LINKAGE

Through the use of the appropriate macro statements, the programmer can call into play whatever subroutines are available to him from the Library. These macro statements are described in a later section of the manual.

HAP LANGUAGE

HAP III must be able to process statements calling for arithmetic and streaming modes of system operation, in addition to HMCP input-output operation, data-defining pseudo operations and miscellaneous operations. Several basic formats for these instructions and operations exist in HAP language, and are classified as:

Arithmetic Mode Instructions - with the following formats:

Floating point
Miscellaneous, unconditional branch
Direct index arithmetic
Immediate index arithmetic
Count and branch
Indicator branch
Variable field length arithmetic; connect; convert
Progressive indexing
Swap; transmit full words
Branch on bit
Load value with sum
Branch enabled to streaming
Clear memory block

Streaming Mode Instructions - with the following formats:

Streaming instructions (SBBB, SMER, SSER, SSEL, STIR, SQNL, SILS, SNOP)
Adjustments
Set-up words (ten words)
Indexing

Macro Statements - including HMCP, HCP, and other miscellaneous macros.

INPUT-OUTPUT FORMAT

INPUT FORMAT

Coding Sheet

All HAP III statements use the same coding sheets and card formats. Symbolic machine instructions are written in the statement field of the HAP coding sheet. Symbolic instructions are divided by commas into several fields (operation, mnemonic, data description, address, offset and so on). These major fields may be further divided into subfields or may be modified by expressions contained in parentheses, such as index register specifications, secondary operations in progressive indexing and so on.

The coding sheet makes it easy to write instructions in a neat and orderly fashion. Figure 3 shows how arithmetic mode instructions are entered on a HAP coding sheet. The coding sheet is divided into four fields:

1. Class (first column) identifies continuation cards and comment cards.
2. Name (next eight columns) identifies the statement. May be any programmer symbol.
3. Statement (next 63 columns) expresses an instruction or pseudo instruction.
4. Identification (last eight columns) identifies the card.

Card identification (columns 73-80) is reproduced on the output listing, but does not contribute any information to the assembly program for translating instructions.

Punched Card

The format of the coding sheet is directly related to the format of the symbolic input card; both are divided into the same four fields. The coding sheet is most useful as a document from which the card punch operator can punch the program directly onto the input cards. The first instruction from the coding sheet is shown in Figure 4 as it would be punched on a HAP input card. Note that one line on the coding sheet represents one punched card. Normally, one machine instruction or pseudo operation is written per line.

Comment Mark (')

A comment may follow any instruction. The beginning of a comment is signaled by an apostrophe('); the end is usually indicated by the end of the card. For example:

```
CONV3L(BU,64,8), TABLE          'BEGIN CONVERSION
```

	NAME	
	ANYNAME	L(BU,64,8),DATA(#X3),7
	ADDER	+(BU,8,8),SINES

Figure 3. HAP Coding Sheet

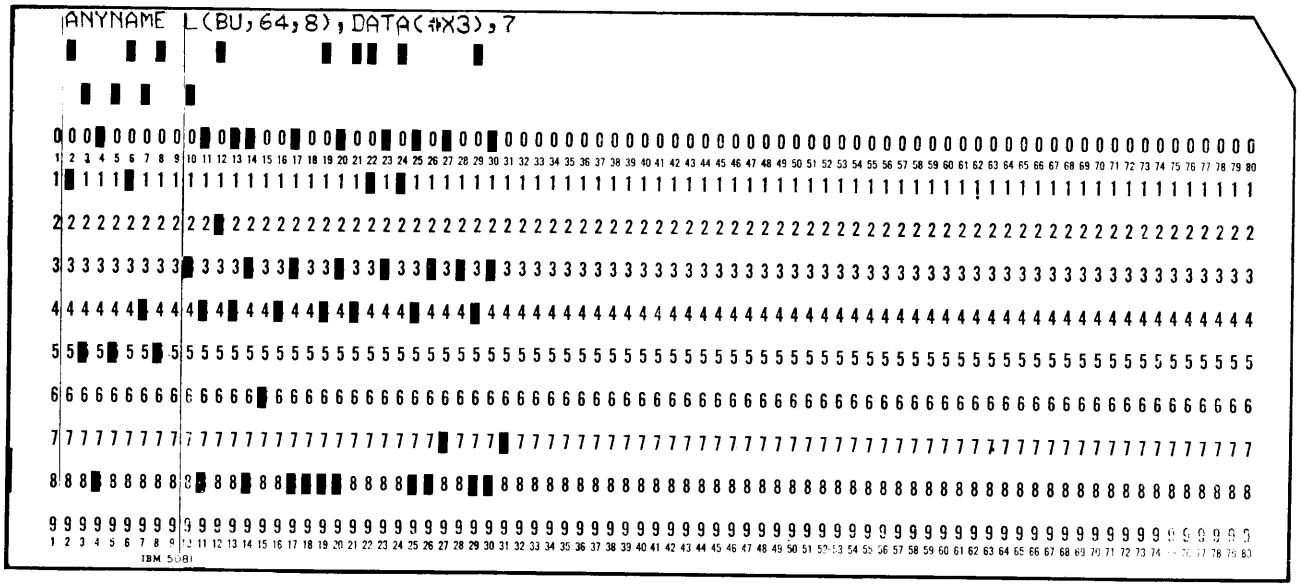


Figure 4. HAP Symbolic Card

Comments are reproduced on the output listing, but do not effect the assembly program in any way. If an apostrophe (') appears anywhere in the name field, the entire card is treated as a comment, reproduced on the listing, but not assembled.

Multiple Statement Mark (?)

Several statements may be written in the statement field of a single symbolic input card with the exception that only one macro statement is allowed per card. Multiple statements are separated by the question mark character (?), implying the end of a statement. Therefore, the question mark can never be used in a comment, except when an apostrophe appears in the name field, signifying that the entire card is to be treated as a comment.

The following three statements can be arranged as shown for entry on a single card.

```
BEGIN L (N), DATA ?+(N), FIRST1 ?-(N), ANGLE
```

The number of instructions written on one line is limited only by the number of columns available in the statement field of the card. If a card has more than one instruction in its statement field, then its name field is associated with the first instruction only. The remaining instructions are treated as if they appeared on separate cards having blank name fields.

Continuation Card Mark (*)

The name field and/or the statement field of a symbolic input card can be continued on subsequent cards by use of a continuation card. A continuation card is identified by an asterisk (*) punched in column 1. In all other respects it is identical to the standard symbolic input card. An example of the use of a continuation card is given on the coding sheet by the following instruction, whose name is REALLONGNAME:

```
REALLONG        L, DATAWORD ?ST, ADDRESS

*  NAME
```

If continuation cards are used to extend the name field, two restrictions apply: the first character of the name must appear in the name field of the first card, and a name must consist of not more than 128 characters. A name, regardless of its length, is always attached to the first statement in the set of cards. Thus, in the example, REALLONGNAME only applies to L, DATAWORD. The next instruction, ST, ADDRESS is nameless.

Card Blocks, Records, and Files

Several terms are commonly used to describe methods of storing data on cards. Definitions of these terms appear below, and will be referred to throughout this manual.

Physical Card Block or Record: The amount of data that can be recorded on one full card.

Logical Card Block or Record: The card space needed for a single HAP language statement or statement name. Several statements may appear on one card through use of the question mark (?), or one statement may require the use of several continuation cards. Thus a logic card block may be formed by one or more physical card blocks; or a physical block may contain one or more logic blocks.

Logic and Physical Card File: All logic records pertaining to one application or activity form a logic file. A logic file has one or more physical files, thus all punched cards pertaining to one activity form a logic file; however, they may be stored in card file drawers, where each drawer is a physical file.

OUTPUT FORMAT

The output of any HAP assembly appears in two forms: a printed listing of the symbolic instructions, and a taped record of the machine language (binary) instructions. If desired, punched cards can also be output from the assembly.

Several types of instructions are classified under the general title of control statements or pseudo operations. Some of these pseudo operations pertain to control of the output format; others define data to the assembly program. In general, pseudo operations may be categorized as follows:

1. Output listing or punching control statements. Although these affect the format of the output, they do not themselves cause any instructions to be compiled.

2. Data defining pseudo operations which appear as data in the output formats.

3. Miscellaneous control statements which allow the programmer to directly control his program and either appear as data or affect the output formats.

Those statements affecting output format will be described in this chapter, other statements will be discussed later in the manual.

OUTPUT LISTING

The output listing produced by HAP contains two types of information. On the right half of the page, each HAP statement is reproduced as it was punched on the symbolic input card. On the left half of the page, the location assigned each statement is displayed in octal, followed by a numeric representation of the compiled information. A sample listing appears in Figure 5.

Available Print Formats

Arithmetic Mode Instruction Formats

The following five print formats are available for representing compiled information numerically:

1. Octal-hex: The octal-hex representation uses two different radices to represent each half-word instruction compiled by HAP. The first 24 bits of the 32-bit half-word are displayed in octal and the last eight bits are given by the two hexadecimal characters. The hexadecimal number system is based on 15, (any number from 0 to 15 in the decimal system can be written as 0 to 9, A, B, C, D, E, or F in the hexadecimal system). Thus, any four-bit binary number can be represented by only one hexadecimal character. For economy, the last eight bits of the 32-bit half-word are represented by only two hexadecimal characters rather than three octal characters. If a full-word instruction has been compiled, two half-word octal-hex expressions are used. Note that a period is supplied on the listing between the sixth and seventh octal characters (the 18th and 19th binary bits) to facilitate reading HAP bit addresses.

2. Floating Point: When a data definition statement with a floating point use mode is specified, the compiled data entry is printed in octal but is separated into the components of the floating point format: exponent, exponent sign, fraction, fraction sign, and data flags (see lines 14 and 15 in Figure 5).

3. Index Word: When the XW pseudo operation is employed to create storage elements in the format of index words, the printed display of the compiled information is clearly divided into four fields comprising the index word: value field plus sign, index flag and two unused bits, count field, and refill field (see line 17 in Figure 5).

4. Octal: Binary signed and unsigned data compiled via a DD statement are printed on the output listing in straight octal format (see lines 19 or 20 in Figure 5).

5. Decimal: A decimal use mode in a DD statement causes the compiled data to be displayed in decimal.

```
                    TIME CLOCK        011000101
                                                                    PAGE
      LOCATION            BINARY OUTPUT          NAME     STATEMENT
000077.00 LOWER MEMORY BOUND
000115.00 UPPER MEMORY BOUND
 1- 000100.00                                                       SLC,64.
 2- 017777.00+        +00000000                     NEXT     SYN,(8)17777.0
 3- 000100.00    000112.16 10                       ABLE     LX,7,INDEX              -LOAD INDEX
 4- 000100.40    000113.20 80 000000.20 50                   L(BU),BAKER
 5- 000101.40    000000.10 87 204000.20 D0          CHARLIE  ST(BU,4)(V+IC),.8($7)
 6- 000102.40    000101.70 40                               BZXCZ,CHARLIE
 7- 000103.00    000107.00 60                               L(N),DOG
 8- 000103.40    000110.00 20                               +(N),DOG+1.
 9- 000104.00    000111.00 E0                               ST(N),FOX
10- 000104.40    000113.34 80 030000.20 50                  L(BU,24),SOME VERY LONG NAME
11- 000105.40    000113.64 80 030000.20 D0                  ST(BU,24),SOME OTHER LONG NAME      -TESTING LONG COMMENTS THAT
12-                                                         WILL CARRY OVER TO THE NEXT LINE
13- 000106.40    017777.10 00                               B,NEXT

14- 000107.00   0007+ 6777700000000000 +000     DOG      DD(N),28671X7,183007S7
15- 000110.00   0022+ 5453370000000000 +TUV
16- 000111.00   000001.00                       FOX      DR(BU),(1)
17- 000112.00   000113.00+ 000 000004 000000    INDEX    XW,ZEBRA,4
18- 000113.0    000000.20                      ZEBRA    DR(BU,4),(4)
19- 000113.20                     5703         BAKER    (8)DD(BU,12),5703
20- 000113.34                  00067777        SOME VERY LONG NAME
21-                                                         DD(BU,24),28671
22- 000113.64   000000.30                      SOME OTHER LONG NAME
23-                                                         DR(BU,24),(1)
24- 000114.14   000100.00                                   END,ABLE
    THIS ASSEMBLY REQUIRED       00000032 SECONDS
```

Figure 5. Sample Listing of Arithmetic Mode Instructions

Streaming Mode Instruction Formats

In addition to the print formats described, several additional formats are used in listing the streaming mode symbolic instructions. The appearance of these instructions is given below, with x's written in place of numbers. Parentheses indicate that the number within is in binary form; if the number is outside of parentheses, it is in octal form. The letter U indicates unused bits; for example, U5 means five unused bit locations, or five zeros. Both streaming and arithmetic mode formats may appear on the same listing page. Streaming mode formats are:

1. Streaming Instructions: Use the following format.

   (xxxx xxxx xxx xx xx xxx x xxx xx xxxx x xxx)

2. Adjustment Instructions:

   xx (xx) xxx xxx xxx

3. BES, CLM Instructions:

   xxxxxx.xx xx
   (octal-hex)

4. Indexing Instruction:

   First word: xxxxxxxx± x(xx xx xx) xxxxxx xx(x xx xx xxx)

   Second word: xxxxxxxx xxx xxxxxx xxxx U2

5. TX, TXO Instructions: Use a format identical to the first word of the indexing instruction. No second word is required.

6. IOD, IOX Instructions: Fifteen full words for IOD and three full words for IOX, each using the following format:

   xxxxxx.xx xx xxxxxx.xx xx
   Octal Hex Octal Hex

7. SPE, PLE Instructions: Use a format identical to IOD, IOX instructions, except that only two full words are required.

8. TEY Instruction:

   xxxxxx xx (x) xxx± (x) xxxxxxxxxx

9. LIMits

xxxxxx.xx± (xxx) xxxxxx xxxxxx

10. Setup: The setup listing uses a unique format requiring 20 printed lines (Figure 6).

Control Statements and Printed Output

Printing Control Statements

The following control statements directly control the printing of the output listing but do not compile any machine instructions.

1. Print Single-Spaced: PRNS

This control statement causes the assembly listing to be printed with single spacing. Double spacing is the normal printing mode, and is the mode in effect for every assembly except those in which PRNS is specifically written.

2. Print Double-Spaced: PRND

This control statement restores printing to the normal double spacing mode after the use of a PRNS. At the conclusion of each assembly, the mode is automatically reset to double space, so that PRND need only be used if it is desired to change mode from single to double space in the middle of one assembly.

3. No Printing: NOPRNT

This control statement stops printing the output listing until any other printing control statement is encountered in the program, at which time printing is resumed.

4. Suppress Printing of Unused Symbols: SPNUS

This control statement suppresses printing the list of unused symbols that appears at the beginning of the output listing. The list is suppressed for the compilation of the entire program in which the SPNUS appears. Printing the list is not restored until the beginning of the next assembly.

Print ID (PRNID) Statement

In addition to the statements that actually control the manner in which the listing is printed, one other printing statement exists that has a slightly different function in the program. This is the print ID statement, with the following format:

PRNID xxxxxxxxxxxxx...xx

```
WCH(xxxxxxxx) U1 WCN(xx) WOP(xxx) WM(x) WS(x) XCH(xxxxxxxx) XCN(xxx) XOP(xxx) XM(x) XS(x)
YCH(xxxxxxxx) YCN(xxx) YOP(xxx) YM(x) YS(x) ZCH(xxxxxxxx) ZCN(xx) U1 ZOP(xxx) U1 ZS(x)
SSM(xxxx xxxx xxxx xxxx xxxx xxxx xxxx x) U3
SATH xxxxxxxx SASTM(xxxx) DEBUG(xxxx)
SAVAL xxxxxxxx U8
SCVAL xxxxxx U6 SAM(xx) SCM(x) SCSTM(xxxxx)
SCLM xxxxxx U1 FSTIM(xxxxxxxxx) FLIM(xx) FACT(xx)
TBA xxxxxx xx TBAHO(xx) U5 MDM(x)
U1 TAPS(xxxxx) TAPI(xxxxx x) U1 TAPN(xxxxx) U1 TAPJ(xxxxx) U3 TAPM(xxxxx)
U1 TAQS(xxxxx) TAQI(xxxxx x) U1 TAQN(xxxxx) U1 TAQJ(xxxxx) MOD(xxxxxxxx)
TES(xxxxxx) TEI(xxxxxx x) U5 TEN(xxxxxx) TEBM(xxxxxxxx)
TAO xxxxxx xx TAOHO(xx) TEM(xxxxxx)
U32
U4 SSS(xxxx xxxx xxxx xxxx xxxx xxxx xxxx)
PS xxxxxx xx U8
PIX xxxxxx U14
QS xxxxxx xx U8
QIX xxxxxx U14
RS xxxxxx xx U8
RIX xxxxxx U14
```

Figure 6. Setup Listing Format

Normally, PRNID is the first statement to appear in a program. It instructs the assembly program to write immediately the entire contents of the card block on the output tape. PRNID provides for heading the assembly listing with such information as the problem name, programmer, and so on. A typical PRNID statement might be

PRNID, BCD CONVERSION ROUTINE BY JOE SMITH

If a PRNID appears in the middle of a program, it will appear both at the beginning of the listing and at the point where it actually appeared in the code. When several PRNID statements appear in one program, they are listed sequentially in one group at the top of the listing and each one is listed in its appropriate place in the program. The practice of writing all PRNID statements at the beginning of the listing is useful when a program being assembled is composed of many subroutines and each subroutine begins with a PRNID statement. The PRNID's, when they appear at the top of the listing, form an index of the names of the subroutines included in that assembly.

A very long message may be written following a PRNID; if the message overflows the card, continuation cards may be used. An alternate spelling of the mnemonic, PRNID, is also accepted by HAP.

Miscellaneous Control Statements

Control statements that do not cause any binary information to be compiled present certain unique printing formats. For instance, DR compiles no binary information, therefore HAP prints the number of words and bits reserved by the control statement as an octal bit address. SYN, on the other hand, can define a symbol in terms of either an integer value or a HAP bit address value. When the symbol is defined as a bit address, an octal bit address equivalent is printed in the column where the location counter setting is usually displayed. If the symbol is defined as an integer, a straight octal representation of the converted integer is printed where all other compiled statements and data are shown. If the control statement SLC is used, the contents of the location counter resulting from the appearance of the SLC are displayed in the usual column as an octal bit address.

Additional Listing Information

Some additional information is supplied on the listing. For example:

1. Time Clock: The first item to appear on each assembly listing is a binary representation of the internal time clock showing when the assembly began. The time required to complete the assembly is displayed in seconds as the last item printed on the listing. The time clock can be used for identification pruposes.

2. Symbol Lists: Four lists of symbols are supplied by HAP at the start of the listing. The first is a tabulation of those programmer symbols that were not defined by the programmer, along with the definitions supplied by HAP. The second list contains all programmer symbols defined by the programmer but are never referred to or used. The third and fourth lists contain those symbols that are multi-defined with contradictions and pseudo defined.

3. Column Headings: Headings are placed over each column of information to clarify where location, binary output, name, and statement appear.

4. Storage Bounds: Immediately following the column headings, upper and lower storage bounds are printed as octal bit addresses. The boundaries for each program are determined by HAP in this manner: The lower memory bound is the address of the full word in storage immediately preceding the first word used by the program, while the upper memory bound is the address of the full word in storage that immediately follows the last word used by the program.

5. Line Numbers: The printed lines on each page are numbered sequentially, beginning with 1, and the leftmost column on the listing contains these line numbers. Each page also contains a number that appears at the top of the page, just below the time clock display. Thus, HAP can easily refer to any line of printed output by page and line number.

Error Messages

Certain error conditions can be detected by HAP during compilation. (See Section entitled "Error Condition.") At the completion of an assembly, HAP can list error messages by page, line number, and field wherein the error occurred. Since many statements occupy more than one line on the listing (see lines 11 and 12 on the sample listing), an error message references only the first line occupied by the statement's binary output.

Error Flags

Five other error conditions, all caused by incorrect definition of programmer symbols, can be detected by HAP and reported on the output listing by means of error flags. These flags are five or six-character symbols that appear on the listing on the line immediately preceding the first line of the statement that contains the symbol erroneously defined. The five flags and their meanings are:

1. UNDEF: An undefined symbol has been detected. HAP has assigned to this symbol the bit address value equal to the first full word location following the highest full word used by the program in which this symbol appears. If several symbols are undefined, they are assigned sequential full-word locations from this starting point, in the order in which they are encountered by HAP.

2. QUEST: A multidefined symbol has been encountered. However, the definitions are not contradictory, that is, two or more definitions of the same programmer symbol have been found and all definitions assign the identical value to the symbol. This situation occurs in this sequence of instructions:

```
        SLC, 1000.0
SYMBOL  LI, ANOTHERSYMBOL
        +I, STILLANOTHER
SYMBOL  SYN, 1000.0
```

HAP accepts the definition as legal and does assign the specified value. The appearance of the flag warns the programmer of the unnecessary multiple definition.

3. MULTI: This flag signals a more serious case of multiple definition where the definitions are contradictory. If the following two statements were found in a program,

```
A   SYN, 100.0
A   SYN, 100.32
```

the MULTI error flag would appear on the output listing on the line immediately preceding the second SYN statement. When contradictory definitions occur, HAP assigns the first value encountered and discards all subsequent definitions.

4. PSEUDO: Pseudo definitions are often called circular definitions and are best shown by the illustrations below.

```
A   SYN, B
B   SYN, A+5
```

HAP assigns a value of 0 to A and a value of 5 to B.

5. CONTAG: A contagious error occurs wherever a programmer symbol depends on another programmer symbol which has been erroneously defined in one of the four ways described above. In the following case,

```
        SLC, 500.0
A       SYN, 1000.0
A       SYN, 500.0
B       SYN, A
        L(N), B
        +(N), A
```

MULTI flag would appear on the listing on the line immediately preceding the add statement, and the CONTAG flag would be found on the line preceding the load statement. HAP would assign the value 500.0 to A and B.

Printed Line Carry-over

Because HAP has provision for very long programmer symbols and continuation cards, the symbolic listing of the contents of the cards may extend over two or more lines. For example, if the name of the statement is too long to fit in the name column, it extends into the statement column, and the remainder of the statement is printed on the next line, as shown on line 20 and line 21 of the sample listing. Note that even though the statement uses two lines, the compiled binary information is printed on the first line. In another instance, the programmer may use a continuation card to append a very long comment to a statement. An example of a long comment forcing a format change is seen on lines 11 and 12 in the sample listing.

The reverse situation occurs when several D fields are written on one DD card or multiple statements are written on a single card. Then the binary output spreads over two or more lines, while the symbolic duplication of the input card appears on one line. Lines 14 and 15 illustrate a DD with more than one data entry.

BINARY OUTPUT

In addition to a printed listing, binary information is also output in card-image form. All binary output from the 7950 assembly program is written on binary output tape and consists of two files:

1. Binary Output File: containing all information resulting from the actual assembly of the symbol program.

2. Debug Table File: produced only in debug mode of operation (not production mode) and containing all symbols listed in the symbol table for this assembly and the corresponding location for each.

Each of these files is further subdivided into blocks and records which are described as follows:

1. Blocks: Standard sized divisions within files consisting of 50 records, or a total of 750 words. The last block of either the Binary Output File or the Debug Table File is the only block that may be less than standard size.

2. Records: Standard sized divisions within blocks (card image size) consisting of 15 words, or a total of 960 bits.

Figure 7 illustrates how binary output in card-image form may appear on the binary output tape. Four types of column binary card-images may be produced: origin, flow, branch, and PUNFUL.

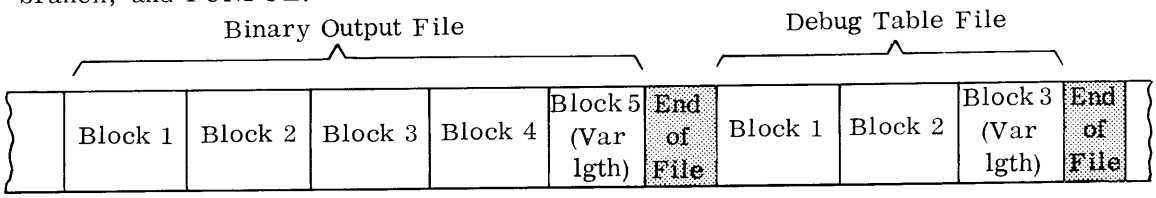


Figure 7. Binary Output Tape Showing Output from HAP

Card Types

Origin Card

Every binary deck to be loaded into the 7950 system via the standard loader program must have an origin card as its first card. The origin card contains an origin address, a checksum, and up to 23 half words of data and/or instructions. The origin address tells the loader where to start loading the half words of data and/or instructions. The origin address is taken from the SLC statement, which is normally the first statement in any program following the identifying statements (PRNID and PUNID).

The complete format of the origin card is shown below. In the convention used to number card columns and rows, the first number specifies the card column (a number ranging from 1 through 80). The second number, separated from the column number by a period, is the row number. The card is considered to be divided into 12 rows with the row nearest the top of the card, 0, and the row nearest the bottom of the card, 11. For example, 10.8 means column 10, row 8.

Card Column and Row	Use
1.0-1.11	Code column (origin card--1.9, 1.10, 1.11 punches)
2.0-2.11	Identification column (binary)
3.0-3.11	Sequence number (binary)
4.0-4.11	Checksum
5.0-5.1	Control bits
5.2-5.11	Primary bit count--number of bits to be loaded
6.0-7.11	24-bit origin address
8.0-9.11	Secondary bit count--number of bits to be skipped or set to zero, as designated by the two control bits
10.0-10.7	Not used
10.8-71.11	Up to 736 information bits
73.0-80.11	Identification (card code)--ignored by the loader

The fields not previously mentioned in the format have the following uses:

1. Code Column: A multiple punch code that tells the loader the type of card being loaded. For an origin card the code is a punch in 1.9, 1.10 and 1.11.

2. Identification Column: Twelve bits of the 36-bit time clock ($TC) indicating the status of the clock at the start of each assembly are punched in column 2 of every binary card produced by HAP to identify the assembly. Column 2 is ignored by the loader.

3. Sequence Number: A binary number computed by HAP to aid the loader in checking the sequence of cards being loaded. The first card in every deck punched by HAP is given the sequence number 1, the second is given sequence number 2, and so on.

4. Checksum: A 12-bit field in which the sum of all bits punched on the card is entered for checking purposes.

5. Primary Bit Count: A 10-bit count telling the loader the number of bits of binary information (columns 10 through 72) that are to be loaded into storage. Any number from 0 to 748 can be specified. Bits not intended to be loaded are ignored by the loader.

6. Secondary Bit Count: A 24-bit count interpreted by the loader in conjunction with the two control bits.

Bit 5.0	Bit 5.1	Meaning of Secondary Bit Count
0	0	Skip n bits before loading card contents
0	1	Skip n bits after loading card contents
1	0	Set n bits to zero before loading card contents
1	1	Set n bits to zero after loading card contents

Bit skipping or zeroing before loading starts at the origin address. Skipping and zeroing after loading starts with the bit location immediately following the last bit loaded from the origin card. Information for skipping or zeroing is determined from the pseudo operations DR and DRZ. If DR has been given, bit skipping is called for, while DRZ specifies setting bits to zero. The setting of control bit 5.1 is determined by the position of the DR or DRZ in the code. When a DR or DRZ immediately follows an SLC, skipping or zeroing information can be placed on the origin card and the proper control bit set before loading the contents of the origin card. The contents are the instructions or data that follow the DR or DRZ in the program. (See flow card description below.)

7. Identification: In this field HAP punches the card code characters specified in the last PUNID statement encountered.

Flow Card

A flow card contains 25 half words of data in column binary form, to be loaded in sequence with the data of the previous card loaded. The format of the flow card is:

Card Column and Row	Use
1.0-1.11	Code column (flow card--1.9 and 1.11 punches)
2.0-2.11	Identification number (binary)
3.0-3.11	Sequence number (binary)
4.0-4.11	Checksum
5.0-5.3	Not presently used
5.4-71.11	25 half words of binary information
73.0-80.11	Identification field (ignored by the loader)

All columns reserved on a flow card for compiled data or instructions must be used. No primary bit count is provided for. All of these columns are read by the loader, and any that contain no punches are interpreted and loaded as zeros. If HAP is constructing a flow card and a DR or DRZ is encountered before the data columns (5.4-7.11) are full, HAP immediately changes the card to an origin card. A primary bit count can now be given so that instructions and data ready to be punched in the card can be loaded, but the remaining blank columns can be ignored. Now a control bit can be set so that the skipping or zeroing is done after the contents of the converted origin card are loaded.

Branch Card

A branch card contains an address to which the loader transfers control, usually the entry address for that portion of the program just loaded. A branch card is produced as a result of HAP encountering an END card or a TLB card. If no address is specified with the pseudo operation, control is transferred to the address given as the origin on the first origin card produced for the subject program.

The format of the branch card is:

Bits Assigned	Use
1.0-1.11	Code column (branch card--1.8, 1.9, 1.11 punches)
2.0-2.11	Identification number (binary)
3.0-3.11	Sequence number (binary)
4.0-4.11	Checksum
5.0-5.11	Not presently used
6.0-7.11	24-bit transfer address

The card before the branch card is often forced to be an origin card. As before with DR or DRZ, if the TLB or END is encountered when the flow card being composed does not have columns 5.4 through 71.11 filled, the flow card is changed to an origin card. The next card will be the branch card.

PUNFUL

A PUNFUL card is a special card requested by the programmer through the PUNFUL statement. The format of the PUNFUL differs from other cards in that all 80 columns of the card are used for column binary data or instructions.

Binary Output File

Each assembled program becomes one physical file on the binary output tape. A program is made up of one or more fixed length blocks each containing a part (or all) of the assembled program in binary card format and each (with the possible exception of the last) containing 50 records (Figure 8). Each card image occupies 15 words and contains or represents a variable number of words of the program depending on the card type (origin, flow, TLB, or branch). The first two card images (origin cards) in the first block are used to construct the Program Name-Limit Table (see Section entitled "Program Name and Limit Statements").

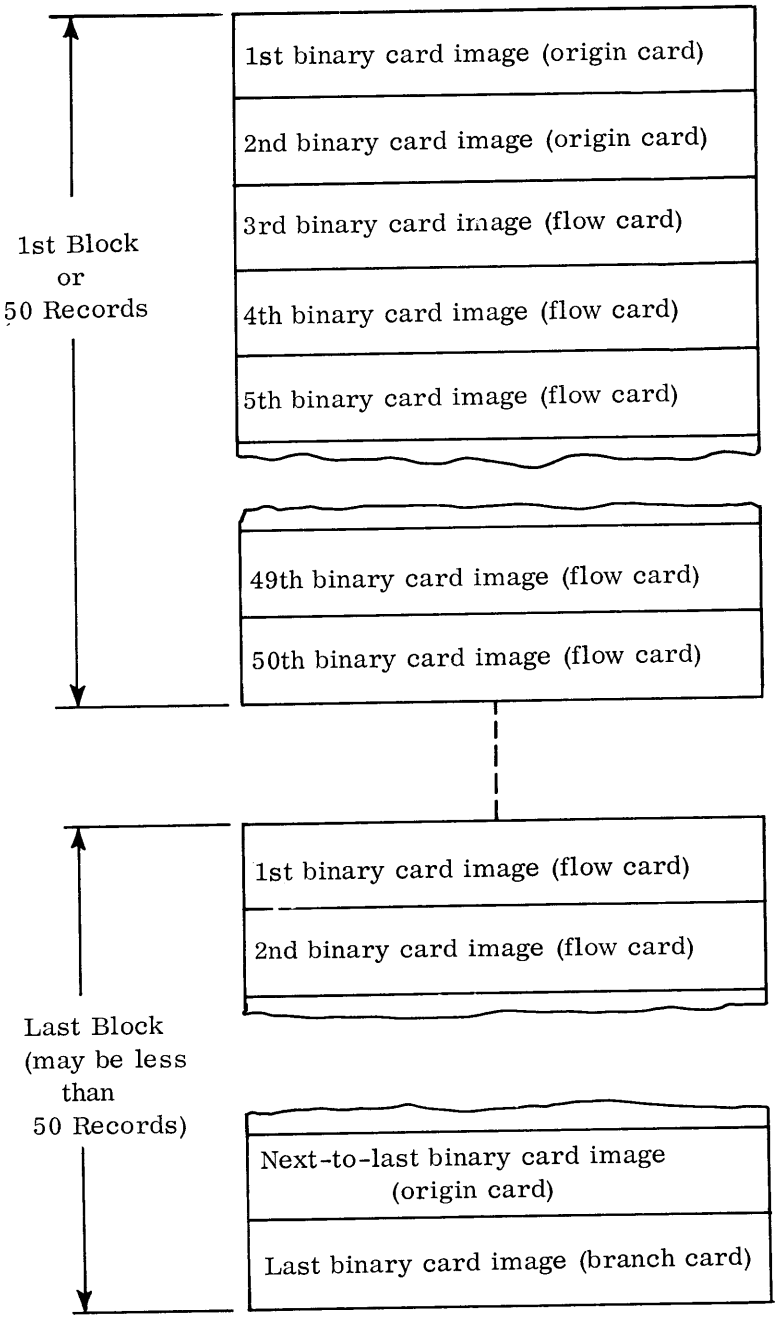


Figure 8. Card-images on Binary Output Tape

Except for the following branch cards, the remaining cards in the binary output file consist of origin and flow cards containing the assembled binary information.

Branch cards tell the program where to go for its next instruction and are produced when either of the following instructions is met:

TLB Instruction: which terminates each segment of the assembly.
END Instruction: which terminates the complete assembly and also writes an EOF on the binary output file.

Debug Table File

Besides the binary output file, an additional file called the debug table file, consisting of N blocks, is produced on the library tape when in debug mode of operation. Each block of this file, except possibly the last, contains 70 records. As an output from the 7950 assembly program, the file contains each symbol listed in the symbol table (in IBM BCD sort sequence) along with the corresponding location for each. The debug table file is accumulated by HAP for use by the HCP.

The card-image sequence consists simply of PUNFUL cards followed by an end of file mark, indicating the logical end of the file.

Card-Image Control Statements

The following control statements are used to control card-image output.

1. Punch Full Cards: PUNFUL

Full cards (80 columns of column binary information) are punched without checksum, first word address, identification, and so on.

2. Punch Normally: PUNNOR

This control statement restores normal punching (72 columns) of origin and flow cards after the use of a PUNFUL.

3. Punch Origin: PUNORG

This control statement causes an origin to be punched in every binary card in the output deck, thus making every binary card produced by HAP an origin card.

4. No Punch: NOPUN

Punching of the binary output deck by HAP can be halted by the use of the NOPUN control statement. Punching remains suppressed until a PUNNOR or PUNFUL control statement is encountered.

5. Punch Cards for Symbols: PUNSYM, A, A', A'', ..., A_n

The A_i are any legal programmer symbols used elsewhere in the program. After the entire binary deck has been punched out, one card in the following format is punched out for each A_i specified.

Card Column	Contents
1	
2-9	Programmer Symbol
10-12	SYN
13-21	dds
22-25	,(8)
26-28	Blank
29	Sign
30-38	Bit Address (xxxxxx.xx)
39-41	Blank
42-44	(8)
45	Integer Sign
46-53	Integer
54-55	Blank
56-60	Index Value (#xx)
61-72	Array Dimensions
73-80	ID specified by the latest PUNID

The SYN cards thus produced permit reassembly of the portion of a program that refers to symbols defined in another portion not being reassembled. The SYN cards are put in the symbol table at reassembly time, and the symbols involved are thereby legally defined.

The format of the card produced by PUNSYM allows for symbols that are defined as bit addresses, integers, or arrays. When a symbol has been defined as an integer somewhere in the program, PUNSYM yields a card that has the integer definition in columns 44-55, and the fields reserved for a bit address definition or an array definition are left blank. Note the presence of the radix specifier which denotes that bit address and integer definitions are always punched in octal. If the symbol is too long to fit in the name field of one card, HAP automatically supplies a continuation card or cards. Similarly, if the array definition is too long to fit on one card, a continuation card is supplied and the definition is continued beginning in column 10.

6. Punch All: PUNALL

This control statement causes HAP to punch a SYN card for every symbol used in the program.

Punch ID (PUNID) Statement

In addition to the above statements that actually control card-image output, one other punching statement exists, which is the punch ID statement, with the following format:

```
PUNID, XXXXXXXX
```

PUNID fulfills the same basic function as PRNID (see the Section entitled "Print ID (PRNID) Statement") except that the identifying information is punched or written on tape on the binary output produced by HAP. The assembly program takes the first eight characters following the comma that terminates the operation field, and inserts them in columns 73-80 of every binary card-image produced as output of that assembly. The following statement

```
PUNID, IBMSINE1
```

causes the characters IBMSINE1 to be inserted in the last eight columns of each binary card-image produced in that assembly.

The identifying characters represented by X's above may be any legal card code characters (except ? and '). Every assembly must contain a PUNID statement or the binary cards will contain no identification other than the time clock setting as described in the Section entitled "Origin Card").

ARITHMETIC MODE INSTRUCTIONS

The arithmetic mode instructions employ a total of 14 symbolic instruction formats, each being a slight variation or expansion of the basic HAP instruction format, which is:

OP, A

where OP stands for operation code, and A stands for storage location address.

The inclusion of index modification of the principal address expands this basic pattern to the arithmetic mode format

OP, A (I)

used in unconditional branches, indicator branches, and miscellaneous instructions. Through the addition of the data description field

OP (dds), A (I)

the format for floating point instructions is obtained. Adding to this format the offset specification and its index modifier

OP (dds), A (I), OF (I')

the variable field length format is developed.

Other changes in the basic format yield the other HAP formats. For example, the insertion of the J field to specify the index register being operated on.

OP, J, A (I)

becomes the basis for the index arithmetic and count and branch formats.

STATEMENT FIELD LAYOUT

The major fields in any HAP format are separated by commas. All possible fields in a certain format need not necessarily be used. For example, an offset need not always be specified for every variable field length (VFL) statement. Therefore, a right-to-left dropout order for major fields has been established; thus, missing fields are compiled by HAP as if they contained zeros and were added at the end of the statement, A missing field always compiled in some standard fashion (in this case zero) is referred to as a null field.

The following example shows the complete right-to-left dropout of fields in a VFL statement and illustrates how the expression of an arithmetic mode statement can vary within the framework of the format for that class of instructions.

```
OP (dds), A ( I ), OF ( I' )
OP (dds), A ( I ), OF
OP (dds), A ( I )
OP (dds), A
```

Note that even when a statement as complex as the VFL is written including only the essential information, the result is a statement that differs very little from the basic HAP instruction format previously illustrated. As shown later, even the (dds) field can almost always be eliminated in the instruction proper.

NULL FIELDS

A major field may be null even if other non-null fields follow. Such is the case if nothing but the comma denoting the field termination is written. Thus, a VFL instruction written with its address and index modifier null but with an offset specification following would appear as:

```
OP (dds),    , OF ( I' )
```

Note that it is only the presence of the comma that indicates the missing address field. If the comma were omitted, HAP would assume that the offset field were null and would actually compile the offset specification as the address expression.

Some of the components of a major field can be made null simply by omission. For example, the offset specification in a VFL statement need not be indexed and can be written

```
OP (dds), A ( I ), OF
```

Similarly, the address expression need not be indexed, and can be written

```
OP (dds), A, OF
```

Obviously, if all the components of a major field are omitted (both offset expression and its index modifier, for example), the field is made null. Normally this is exactly what the programmer desires, but care must be taken if the null field occurs in the middle of the statement. As explained, if the comma denoting the termination of the null field is also missing, the null field is assumed to be missing from the right-hand end of the statement.

MAJOR FIELDS

Three major fields in the HAP arithmetic mode instructions are the operation field, the address field, and the offset field. They are common to most instructions and they illustrate important programming features and facilities of HAP. The operation field, for example, is common to all instructions and the data description, when used, appears as a subfield of the operation field. The address field is also common to all instructions, although it varies considerably in length. The offset field is found only in variable field length instructions, but it is interpreted as an address field of unusual length. It also illustrates some unique methods of index modification.

All arithmetic mode instruction fields are unsigned. Any numeric entries that are negative are converted by HAP and expressed as the two's complement of the entry. All numeric entries in the illustrations are assumed to be written in the decimal radix. Entries in other radices are permitted in HAP if the radix is specified in a standard fashion. See Section entitled "Entry Mode".

OPERATION FIELD

All 7950 instructions may be expressed by the use of mnemonics in the operation field. For programming, mnemonics are desirable because they make instructions brief, easy to remember, and easy to recognize.

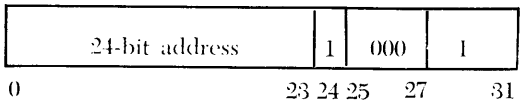
A complete list of HAP III mnemonics is given in Appendix A. Note the following rules for choice of mnemonics. First, the mnemonic should be as brief as possible and still unambiguously identify the instruction. Second, standard symbols are used for arithmetic operations: + for add, - for subtract, * for multiply and / for divide. Third, the receiving register (the register that receives the result of the operation) in arithmetic operations is indicated by the letter to the left of the arithmetic symbol. In cases where the result is in the accumulator, the accumulator is assumed but not mentioned in the mnemonic. For example, + is the mnemonic for straight add where the result is left in the accumulator, M+ is the mnemonic for add to memory and V+ means add to value. Fourth, certain basic operations may be altered to invoke immediate addressing by adding the suffix I as in V+I, add immediate to value.

Null Operation Code Field

A null operation code field occurs if the first character in a statement is a comma, as:

```
, EXIT (I)
```

HAP treats a null operation field as a special case; it compiles the statement as a half word with a 24-bit address field, the 25th bit set to 1 and all the rest of the bits set to 0, thus:



When compiled, this half word appears to be the first half of a full word instruction because of the one-bit in bit 24 and the zeros following. This can be helpful to the programmer if, for example, it is desired to load the compiled address field into the value field of an index register; then load value effective (LVE), which is indexable, can be used. This instruction examines the half word to determine the class of instructions to which it belongs and since the half word resembles the first half of a full word instruction, LVE loads all 24 bits of the address field. If load value (LV) is used, 25 bits will be loaded (24 bits plus sign) and a one in the 25th bit position makes the value appear negative. Therefore, use caution when creating value fields in storage by means of statements with null operation fields.

Operation Subfields

At least two types of subfields may be attached to an operation field: a secondary operation subfield and a data description subfield.

A secondary operation is enclosed in parentheses and follows the primary operation mnemonic. It is commonly used as a subfield of the operation field in progressive indexing with VFL instructions.

The data description subfield appearing in the operation field of certain instruction formats is symbolized by the letter dds enclosed in parentheses.

Data Description (dds)

The data description subfield is required only by the floating point and VFL formats. It appears within parentheses immediately following the operation mnemonic except in progressive indexing, where it may precede or follow the secondary operation. The dds describes three operation code specifications: use mode (M), field length (FL), and byte size (BS), appearing in the dds parentheses in the same order, separated by commas, thus:

(M, FL, BS)

In floating point instructions, the data description tells whether the instruction calls for normalized or unnormalized operations. Only the use mode specification is requested, field length and byte size are not required. In VFL statements, the data description specifies signed or unsigned binary or decimal operations. It also describes the field length and byte size of the data to be used. One additional mode, the properties mode, may appear in either type instruction (as explained later). Field length and byte size are not appropriate with the P mode.

HAP provides seven mnemonics to designate a use mode:

N	Normalized Floating Point
U	Unnormalized Floating Point
B	Binary (Signed)
BU	Binary Unsigned
D	Decimal (Signed)
DU	Decimal Unsigned
P	Properties Mode

The field length and byte size specifications are normally numeric entries, but they may be symbolized by the programmer, provided that the symbols are correctly defined elsewhere in the program.

A typical floating point instruction with data description is:

```
L (N), SINEX
```

The data description (N) indicates that a normalized floating point data word located at SINEX is to be operated upon. In the following VFL instruction

```
L (BU, 30, 6), ADJUST
```

the data description describes the data at symbolic location ADJUST as binary unsigned, 30 bits in length, composed of six-bit bytes. Note that in cases where the operation mnemonic is the same for VFL and floating point instructions, it is the data description that tells HAP to which class the operation belongs and, hence, which operation code to compile.

Data Entry or Data Reservation Statements

A data description given with any of the four data entry or data reservation statements (Data Definition (DD), Data Definition Immediate (DDI), Data Reservation (DR), and Synonym (SYN)) is attached to the symbol in the name field of that statement, and is automatically invoked whenever that symbol appears in the principal address field of an arithmetic mode instruction (see Section entitled "Data Definition"). Thus, it is normally unnecessary to write a data description in arithmetic mode instructions. When several symbols are joined arithmetically in an address field, the data properties of the last one written down are invoked for the statement.

When the data description is written as a subfield in the operation field of a machine instruction, it overrules any other data description derived from a symbol in the address field for only that statement.

P Use Mode

A dds containing the properties mode (P) symbol is written:

(P, DATA)

which specifies that the data description associated with the symbol DATA is to be invoked as if it had been written out explicitly in this instruction. Thus, in an instruction, the properties mode invokes a data description that overrules any data description implied by a symbol in the principal address field.

Null dds Fields

Within a data description field, the usual right-to-left dropout order holds except that the mode field can never be null, so that a data description may appear in any of the following four forms:

(M, FL, BS)	
(M, FL)	Byte size is null
(M, , BS)	Field length is null
(M)	Field length and byte size are null

If the field length is null, a field length of zero (see 7030 Reference Manual; effectively 64 except in the case of VFL immediate where it is 24) is compiled. However, if the byte size is null, the byte size compiled by HAP is a function of the mode specified.

Mode	Standard Byte Size
D or DU	4
B	1
BU	8
N or U	Fixed format of 64 bits; field length and byte size not appropriate.

Error Conditions

Four error conditions arising from discrepancies between operation and data description are possible. If HAP encounters any of these, the indicated action is taken and an error message is printed on the output listing.

1. A programmer error can cause a data description and an operation to be inconsistent; for example, the operation mnemonic specifies a floating point operation and the use mode in the data description is binary unsigned. In this case, the operation overrules.

2. No data description is available, either from the symbolic address or an explicit data description field. If the operation symbol can stand for either VFL or floating point operations (+, -, *, /), the operation is compiled as a VFL operation with the data description (BU, 64, 8).

3. No data description is available and the operation mnemonic can stand for a VFL operation only (M + 1, for example). The statement is then assigned a data description (BU, 64, 8). If the operation is clearly VFL immediate, then (BU, 24, 8) is assigned.

4. No data description is available and the operation mnemonic can be only a floating point operation (-A, or *NA). The operation is assembled as normalized floating point, except for the case of E + I (Add Immediate to Exponent) and its modified forms, where unnormalized is assumed.

ADDRESS FIELD

The maximum core storage capacity of the 7950 computer is 262,144 words (each word 64 bits in length) or 2^{18} distinct locations. Hence, 18 binary bits can unambiguously specify any word in 7950 memory.

Any single bit in core storage can occupy one of 64 positions within a word. Conventionally, bit positions in a word are numbered from 0 (the leftmost bit position) to 63 (the rightmost bit position); therefore, six bits are sufficient to specify any bit position within a 7950 word.

Then, 18 + 6 = 24 binary bits are adequate to address a single bit anywhere in 7950 core storage; the first 18 bits specify a full word and the last six bits specify a bit position within that word. Such a 24-bit binary address, when appearing in the address field of a statement, is known as a standard binary bit address, commonly abbreviated to "bit address."

General Addressing Rules

In 7950 programming, the address of an instruction need specify only the leading bit of the operand since the field length of the operand is always known. If the operand is an instruction, the operation code determines whether the field length is one-half or full word. If the operand is data, the data description gives the field length; for example, floating point data always occupy a full word, while the field length of VFL information is specified explicitly in the dds.

Instruction or Data Operand Addressing

Certain rules for the location of data or instructions further simplify the addressing of operands. To address an index word, which always begins at a full word, only 18 bits are required. A 19-bit standard binary bit address is adequate to address any instruction, since instructions can only be located to begin at half or full words. Other examples are:

1. A VFL operand may begin anywhere in core storage; therefore, a 24-bit standard binary bit address is required.

2. A floating point operand must begin at a full word; therefore, this location can be specified in 18 bits.

3. I-O control words must begin at a full word, which again can be specified by 18 bits.

4. Index arithmetic operands can begin at either half or full word locations and 19 bits are sufficient to address either of these locations.

Instruction Address Fields

A floating point instruction needs only to address floating point data; hence the size of the address field of a floating point instruction is limited to 18 bits. A VFL instruction must be capable of addressing any field, or any bit, in storage. Its format, therefore, provides for a 24-bit address field. In general, HAP instruction formats are designed to provide the largest address field demanded by the operations of a particular class. When an instruction does not require a 24-bit address field, a smaller one is provided, allowing the bits not used as part of the address field to be efficiently used in other fields of that instruction. This leads to the variations in format already shown.

Address Field Entries

It is difficult for the programmer to write 24-bit, or even 19 or 18-bit binary addresses in his program. Instead of a binary address, HAP permits the programmer a choice of entries in address fields such as:

1. HAP bit address
2. Integer
3. Programmer symbol
4. System symbol

Address fields are unsigned fields. When a negative quantity is expressed in an unsigned field, the two's complement of the quantity is computed and compiled by HAP.

HAP Bit Address

A HAP bit address provides a simple means of writing a standard binary bit address. Thus, the programmer writes two integers separated by a period, such as:

124.32

The integer to the left of the period specifies the word address portion, while the integer to the right of the period specifies the bit position within that word. If the example appeared in the address field of a VFL instruction, HAP would interpret it as "location 124, bit position 32 - the first bit of the second half word." Note that the period is definitely not a decimal point. This can be proven by the following illustration, true only for bit address notation.

777.1 = 777.01

A HAP bit address is translated by HAP and compiled as a 24-bit binary integer. The period that separates the two integers always lines up between bit positions 17 and 18. If the address field of the instruction is 24 bits long, the binary integer is placed in that field. If the address field is smaller than 24 bits, the 24-bit standard binary address must be truncated before it is inserted. For a 19-bit address field, the rightmost five bits are dropped; for an 18-bit address field, the rightmost six bits are dropped. The sample HAP bit address, 124.32, would yield the proper meaning when inserted in a 19 or a 24-bit address field, but would be truncated to 124.0 for an 18-bit field.

The only restriction on the size of a HAP bit address is that it must be able to be expressed in 24 binary bits. If a HAP bit address is symbolized by A.B, then

$$64A + B < 2^{24}$$

The following three examples are all acceptable HAP bit address representations of the same address.

505.17 = 500.337 = 0.32337

Integer Addresses

An integer, written without a period, may also be used to specify an address. HAP translates an integer into a standard binary bit address, which depends on the environment in which the integer is found. The operation determines the environment by the length of the address field. The integer specified is converted to binary and inserted in the address field with the unit bar placed in the rightmost bit position of the field.

An integer can be interpreted by the programmer to count in the units that are specified by the length of the address field. A 24-bit address field specifies bits; an integer in this field counts bits. A 19-bit field specifies half words; an integer here counts half words. An 18-bit field specifies full words; an integer here counts full words.

Consider the following instruction.

```
C + I, #3, 13
```

The environment is determined by the operation C + I (Add Immediate to Count). This instruction has an 18-bit address field, so an integer is inserted with its unit in bit 17. This is equivalent to

```
C + I, #3, 13.0
```

and the integer can be considered to count full words. However, the same integer in the following instruction

```
V + I, #3, 13
```

has a different meaning. Here the V + I instruction has a 19-bit address, and the integer inserted in this field is equivalent to 13 half words or location 6, bit position 32. This is the same as writing

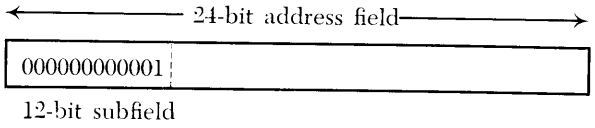
```
V + I, #3, 6.32
```

Advantages of the Use of Integer Addresses

The use of an integer to express an address requires special care on the part of the programmer since the size of the address field determines the interpretation of the integer. However, the integer is often the most desirable form of address specification, and simpler to use than a HAP bit address. One such case is immediate addressing.

```
LI (BU, 12, 8), 1
```

The load immediate instruction specifies, through its data description, a 12-bit address field. The integer address, in this case 1, is inserted as an integer in this 12-bit field. Thus, the instruction is compiled:



The same load immediate instruction could be written with a HAP bit address specification as follows:

```
LI (BU, 12, 8), 64.0
```

The two statements are equivalent, but the one written with the integer address is more desirable because it is simpler to code and, when the statement is reviewed at a later date, its original meaning is easier to recognize.

Programmer Symbols

A programmer symbol can be any sequence of 128 or fewer alphabetic and numeric characters that conform to the following conditions:

1. It contains only alphameric characters. This example

THISISALONGNAME2SHOWTHATHAPNAMESCANBESENTENCES

is a proper programmer symbol. This example

A *B

is not a proper symbol.

2. The first character is specifically alphabetic; that is,

A123456

is acceptable, but

6ALPHABET

is not correct.

3. The programmer symbol appears in the name field of a HAP statement at some point in the program, at which time it is defined and assigned a value that is either a standard binary bit address or an integer.

```
BEGIN     L (BU, 8, 8), A123456
```

The symbol BEGIN is assigned a standard binary bit address equal to the value of the location counter within HAP at the time this load instruction is encountered in the code. The HAP location counter always contains a 24-bit standard binary bit address.

In the following case

```
EIGHT     SYN, 8
```

the symbol EIGHT is assigned the value of the integer 8 through use of the synonym control statement.

Symbols that name instructions are automatically assigned data descriptions by HAP and are given a field length equal to the length of the particular instruction named (that is, either 32 or 64 bits), a byte size of 8, and a use mode of binary unsigned (BU).

A programmer symbolized field may contain programmer symbols or system symbols. Of the fields shown in the instruction formats previously illustrated, all may contain programmer symbols except the operation field and the mode field of a data description.

An integer in a programmer symbolized field is always converted to binary, and is limited in length to the length of the field in which it is to be inserted. An integer that cannot be expressed in 24 binary bits cannot be symbolized.

System Symbols

System symbols have values that are fixed in the compiler. They are identified in programmer symbolized fields by the appearance of the special prefix character #, (which, as one of the nonalphameric characters, can never appear in a programmer symbol), followed by seven or fewer alphabetic or numeric characters. System symbols may appear in arithmetic expressions in programmer symbolized fields where, in cases where restrictions apply, they can be considered the same as numeric entries because their values are immediately available to the compiler.

All system symbols that represent the addresses of special registers in storage, such as #AOC (the all ones counter) or special bits in storage, such as #LC (the lost carry indicator) are bit addresses. All others are real numbers.

The # character alone acts as a special system symbol that provides a standardized substitute for a name for the current statement. Effectively, the character # is a bit address which, when it appears in a statement, functions as if it had been defined by being written in the name field of that statement. If the instruction actually compiles space in the program, the # symbol represents the value of the location counter at the time the instruction is encountered by the compiler. The appearance of the # in the following example:

```
B, #-2.
```

means "Branch to the instruction which begins two full words before this branch instruction" or:

```
B, #+.32
```

in which the meaning is "Branch to the next instruction," effectively, a "no operation."

Another special use of the # character is to prefix any operation code in this manner: #OP. This directs the compiler to suppress any error indications that arise in connection with the compilation of this statement.

System Symbol Groups

HAP assigns a dds to every system symbol. The system symbols used for arithmetic mode instructions can be classified in the following groups:

1. Index Register Symbols: The system symbols #0 through #15 or #X0 through #X15 represent index registers 0 through 15, addresses 16.0 through 31.0 in 7950 storage. The advantage of using a system symbol is that HAP always compiles the correct value, regardless of the size of the field in which the symbol is written. Therefore, in the instruction

```
*+ (N), ARLE(#5)
```

the index specification involving the system symbol #5 directs HAP to compile correctly the binary integer 5 in the 4-bit index subfield. In a similar fashion, HAP correctly interprets the system symbol when used as an address as in

```
ST (BU), #X5
```

and compiles the standard binary bit address 21.0 in the address field of the store instruction.

2. Special Register Symbols: The names of all the special registers used in the arithmetic mode portion of the 7950 computer are listed below, along with the system symbol for addressing each register and the bit address assigned to each system symbol. HAP also assigns a data description to each symbol with a use mode of binary unsigned (BU), a byte size of 8, and a field length equal to the length of the register. When a system symbol for a special register appears in the principal address field of a VFL instruction, no data description need be written out explicitly in that instruction.

Name	Mnemonic	Bit Address	Length
Word number zero	#Z	0.0	64
Interval timer	#IT	1.0	19
Time clock	#TC	1.28	36
Interruption address	#IA	2.0	18
Upper boundary	#UB	3.0	18
Lower boundary	#LB	3.32	18
Boundary control	#BC	3.57	1
Maintenance bits	#MB	4.32	64
Channel address	#CA	5.12	7
Other CPU	#CPU	6.0	19
Left zeros count	#LZC	7.17	7
All ones count	#AOC	7.44	7
Left half of accumulator	#L	8.0	64
Right half of accumulator	#R	9.0	64
Sign byte	#SB	10.0	8
Indicator register	#IND	11.0	64
Mask	#MASK	12.20	64
Remainder register	#RM	13.0	64
Factor register	#FT	14.0	64
Transit register	#TR	15.0	64

The use of the system symbol for the indicator register is illustrated as:

```
L, #IND
```

No data description need be written explicitly in the load instruction because the dds (BU, 64, 8) has been attached to the system symbol. This instruction is thus compiled by HAP to mean, "Load the contents of the entire 64-bit indicator register into the right half of the accumulator at zero offset."

3. Indicator Bit Symbols: The complete list of the system symbols for the indicator bits are listed in Appendix B. Each system symbol, when prefaced with # and placed in a programmer symbolized field, represents the correct bit position in word 11 of the indicator named.

The system symbols for the indicator bits are also used as part of the mnemonic for the branch on indicator instruction. In this usage, however, the # is not required. The mnemonic for this instruction is composed of the B (representing branch) followed by the system symbol for the indicator being interrogated without the # sign. Thus, BXH is the operation mnemonic for branch on index high, and BXVGZ is the operation mnemonic for branch on index value greater than zero.

All system symbols in classes 1, 2, and 3 are bit addresses and are assigned standard data descriptions with mode BU, byte size 8, and a field length equal to the length of the particular register or bit.

4. Symbols for Mathematical Constants: Five mathematical constants, useful in many scientific and engineering problems, can be represented by system symbols. These system symbols and their values are:

Symbol	Mathematical Constant
#E	e
#M	$\log_{10} e$
#N	$\log_e 2$
#PI	π
#INF	∞ (infinity)

These five symbols may only be used in a data field of a data definition (DD) statement where normalized floating point (N) has been specified in the use mode field of the dds. The following data definition

```
CONSTANT    DD(N),  #PI
```

assigned the floating point equivalent of the quantity π to the symbol CONSTANT. When CONSTANT is used in the address of a 7950 instruction such as

```
+,  CONSTANT
```

the normalized floating point data description is invoked and the full word floating point equivalent of π is added into the accumulator.

Index Modification of Address Fields

Index modification of an address field is performed in standard fashion. The index register to be used is specified as a subfield of the address field. The index subfield is a four-bit field, enclosed in parentheses, immediately following the address expression. HAP bit addresses, system symbols, and programmer symbols that are defined as bit addresses are all proper entries in an index field.

In the case of a bit address entry, the period is assumed to occur at the right end of the field. Thus, when converted to binary, the rightmost six bits of the entry are truncated, as are the leftmost 14 bits.

System symbols are the simplest to use, acting as if a bit address had been entered. All of the following entries in the index subfield of an address mean the same:

$$4.32 = 4.0 = \#4 = 20.0 = 52.0$$

and all are translated by HAP to mean index register 4.

If an integer is written in the index field, the meaning is entirely different. The integer tells HAP that the symbol in the address field proper had been defined as an array and the integer is addressing an element in that array (see data reservation statement description).

In the case of progressive indexing in a VFL instruction, it is the index register specified within the address field that is stepped by the immediate address.

OFFSET FIELD

Offset fields are similar in content to address fields; HAP bit addresses, integers, system symbols, and programmer symbols are all acceptable entries in an offset field.

Integers with Offset Field

The most common entry for an offset specification is an integer. An integer specifies a count of the number of bits to offset a field from the right end of the accumulator. An offset field has a fixed length of seven bits. An integer entry is converted to a 24-bit binary integer by HAP and the rightmost seven bits are placed in the offset field. If a programmer writes the statement

```
L (BU,  64),  PAYROLLDEDUCTION, 5
```

HAP assembles the instruction to mean load the 64-bit quantity found at symbolic location PAYROLLDEDUCTION into the accumulator offset five bits from the right end. Since the offset field can contain a maximum of seven binary bits, the programmer can specify any offset from 0 to 127. When specifying offsets of greater than 64 bits or one full word,

it may be more convenient to begin counting bits from the left end of the double length accumulator. This can be done easily by using negative offsets. The offset field is unsigned, hence HAP translates any negative entry to the two's complement. The 128 bits of the accumulator, proceeding from left to right, are referred to by the offsets 127, 126, ... 0 or, alternatively, -1, -2, -3,... -128.

Programmer Symbols in the Offset Field

If an offset specification is a parameter in a program that may vary from time to time, it is helpful to use a programmer symbol in place of an integer.

```
INDENT    SYN, 4
              (intervening instructions)
              ST (BU, 24), WORD1,        INDENT
              + (BU, 24), WORD2,         INDENT
              L (BU, 24), WORDSUM,       INDENT
```

The programmer symbol INDENT in the example above, can be defined as an integer early in the program (in this case by the synonym statement). If the programmer changes the SYN card that defines INDENT to

```
INDENT    SYN, 5
```

and reassembles, all offsets specified by this particular symbol are changed in value to 5 as well.

HAP Bit Addresses in the Offset Field

Exercise care when using HAP bit addresses instead of integers in offset fields because the length of the offset field is fixed, so an integer always has the same meaning, but the meaning of a bit address is not immediately clear from its appearance in the instruction. Also, a bit address is not the natural means of expressing an offset, and it complicates the specification unnecessarily. A HAP bit address here is converted to a 24-bit binary integer and the rightmost seven bits are inserted in the offset field while the leftmost 17 bits are truncated. Any HAP bit address expression that specifies an address above 1.63 will overflow the offset field when converted to binary and only the rightmost seven bits will participate.

System Symbols in the Offset Field

The use of a system symbol to specify an offset is unlikely, but permissible. As previously stated, a system symbol is equivalent to a numeric entry; therefore, specifying an offset by means of a system symbol defined as a bit address, such as #IT in this example:

```
+ (BU, 32), THISIS, #IT
```

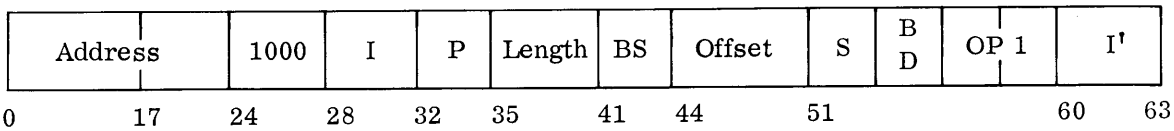
is the same as writing 1.0 or specifying an offset of 64 bits.

Index Modification of Offset Fields

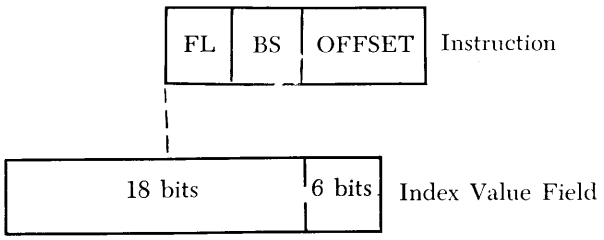
Index register specification is treated in the same way as an index modifier in an address field, except that the modification can affect the field length and byte size as well as the offset. The HAP instruction format for VFL statements including the following data description

$$OP\ (M,\ FL,\ BS),\ A_{24}(I),\ OF,\ (I')$$

does not indicate the relationship between field length, byte size, and offset. The internal VFL instruction format



into which a HAP VFL instruction is translated, does show that the offset field is adjacent to the field length and byte size fields. The index modifier in the second half word treats all three fields together as one 16-bit field. For the modification process, the two fields are aligned as follows:



If the magnitude of the contents of the value field of the index register does not exceed 2^6, only the offset field can be modified. If the value field does exceed 2^6, the byte size may be affected (by a carry, for example). The diagram above shows how larger value fields modify byte size and field length. This 7950 feature provides very flexible and elaborate indexing of certain VFL instruction fields.

ARITHMETIC MODE INSTRUCTION FORMATS

The format of the symbolic instruction varies with the class of arithmetic mode instruction to which it belongs. Thirteen symbolic instruction formats are given below. Note that certain of the instructions use numeric subscripts on the A, B, or OF fields. These subscripts refer to the number of bits needed to correctly address the instruction. The field A_{18}, for example, means that the instruction addressed is a full-word instruction. Similarly, A_{19} or B_{19} stands for a half-word instruction, while A_{24} or B_{24} means that any bit in the 64-bit word may be addressed. An offset field, such as OF_7, must be addressed by a seven-bit field.

1. Floating Point

OP (dds), A_{18} (I)

Example: ST (U), BUCKET(#2)

This instruction says "Store the contents of the accumulator as an unnormalized floating point number in the storage location symbolized by BUCKET modified by index register 2."

2. Miscellaneous, Unconditional Branch, SIC

OP, A_{19} (I)

Example: B, START(#X12)

This instruction means, "Branch to, or transfer control to, the instruction whose location is symbolized by START modified by index register 12."

3. Direct Index Arithmetic

OP, J, A_{19} (I) or OP, J, A_{19} (I)

Example: LX, #3, XWORD(#6)

This instruction, when executed, tells the computer, "Load index register 3 with the contents of the word found at the location symbolized by XWORD modified by index register 6."

4. Immediate Index Arithmetic

OP, J, A_{19} or OP, J, A_{18}

Example: V + I, #10, 1024

The meaning of this instruction is "Add the address of this instruction to the value field of index register 10."

5. Count and Branch

OP, J, B_{19} (K)

Example: CB, #8, BEGIN(#1)

This instruction directs the computer to "Subtract one from the count field of index register 8, then test the count field. If it is not zero, branch to the location specified by the symbolic location BEGIN modified by index register 1. If the count field is zero, do not branch but proceed to the next instruction in sequence."

6. Indicator Branch

$$OP,\ B_{19}\ (K)$$

Example: BZM, ERROR(#7)

This instruction, whose operation code mnemonic is partially constructed from the name of the indicator says "Branch to the instruction located at the location symbolized by ERROR modified by index register 7 if the zero multiply indicator is on. If it is not on, proceed to the next instruction in sequence."

7. VFL Arithmetic, Connect, Convert

$$OP\ (dds),\ A_{24}\ (I),\ OF_{7}\ (I')$$

Example: M + (BU, 24, 8), DUMMY(#9), 6 (#4)

This variable field length operation says "A 24-bit unsigned field composed of eight-bit bytes is found offset from the right end of the accumulator by an amount equal to six bits modified by index register 4. Take this field and add it to the field of the same length that is found beginning at location DUMMY modified by index register 9 in storage."

8. Progressive Indexing

$$OP_{1}\ (OP_{2})\ (dds),\ A_{24}\ (I),\ OF_{7}\ (I')$$

Example: ST (V + I) (BU, 24, 8), 30 (#8), 2 (#14)

This VFL instruction with progressive indexing reads "An unsigned 24-bit field composed of eight-bit bytes is found offset from the right end of the accumulator by 2 modified by index register 14. Store this field in the storage location specified by the value field of index register 8. Then increment the value field of index register 8 by 30 bits and proceed to the next instruction in sequence."

9. Swap, Transmit Full Words

$$OP,\ J,\ A_{18}\ (I),\ A'_{18}\ (I')$$

Example: T, #2, TABLE1(#3), TABLE2(#4)

This transmit instruction means "Transmit the number of full words specified by the count field of index register 2 from the storage area beginning at location TABLE1 modified by index register 3 to the storage area beginning at TABLE2 modified by index register 4."

10. Branch on Bit

OP, A_{24} (I), B_{19} (K)

Example: BB, ONEBIT(#5), FIXUP(#9)

This instruction is interpreted to mean, "If the bit in storage whose location is ONEBIT modified by index register 5 is on, branch to the instruction at location FIXUP modified by index register 9. If this bit is not on, proceed to the next instruction in sequence."

11. Load Value with Sum

LVS, J, X_1, X_2, X_3, ...

Example: LV5, #3, #5, #6, #7, #8

This instruction reads, "Add together the value fields of index registers 5, 6, 7, and 8 and store the sum in the value field of index register 3."

12. Branch Enabled to Streaming

OP, A_{18} (I)

Example: BES(I), BEGIN (#X12)

This instruction is used to initiate the streaming mode of operation. The meaning of the example is: "Branch to or transfer control to the streaming instruction whose location is symbolized by BEGIN modified by index register 12."

13. Clear Memory Block

CLM (size), A_{18} (I)

Example: CLM (S), JOHN

The instruction reads: "Clear small memory block starting at the address symbolized by JOHN." The actual number of memory cells cleared and the location of the block depends on the physical configuration of the memory boxes.

ADDRESS ARITHMETIC

It is often convenient for a programmer to write an address expression of two or more symbols, integers, bit addresses, and so on. Relative addressing offers a good example of the need for these expressions. For example, the appearance of a # in an address field has been shown to have the meaning "the location of this same instruction." To refer to a location exactly two full words beyond the location of the instruction containing the #, it is convenient to write the address expression

+ 2.0

rather than to assign a programming symbol to this location and address the location by symbolic name. In another instance, assume a table is known to begin at symbolic location DATA and to be 20 full words in length. Then, by the use of relative addressing, the full word immediately following the last word in the table can be addressed by the expression DATA + 20.0.

HAP offers provisions for the performance of address arithmetic. Virtually any mixture of HAP bit addresses, integers, programmer symbols, and system symbols can be combined by addition, subtraction, multiplication, and division to form a single 24-bit standard binary bit address. Thereafter the truncation (if necessary) and insertion of the bit address into the appropriate address field is completely standard.

Symbols for addition, subtraction, multiplication, and division are standard, (+, -, *, and /, respectively). Addition and subtraction are the most common arithmetic operations and, when like quantities are involved, the procedure is completely straightforward.

Addition and Subtraction of Addresses

When two HAP bit addresses are to be added together, the points are lined up and the two quantities are added. If two integers are to be added, the units positions are lined up before the addition is performed. In either of these cases, subtraction is analogous to addition.

Thus, the address expression in this instruction

```
L (BU), 8. + 64.0 + 12.3
```

is treated as

```
   8.0
  64.0
+ 12.3
 ------
  84.3 = actual instruction address
```

In the case of integer expressions, such as

```
LI (BU, 18, 8), 8 + 2 + 13 + 1
```

addition takes place

```
   8
   2
  13
+  1
 ---
  24 = actual instruction address
```

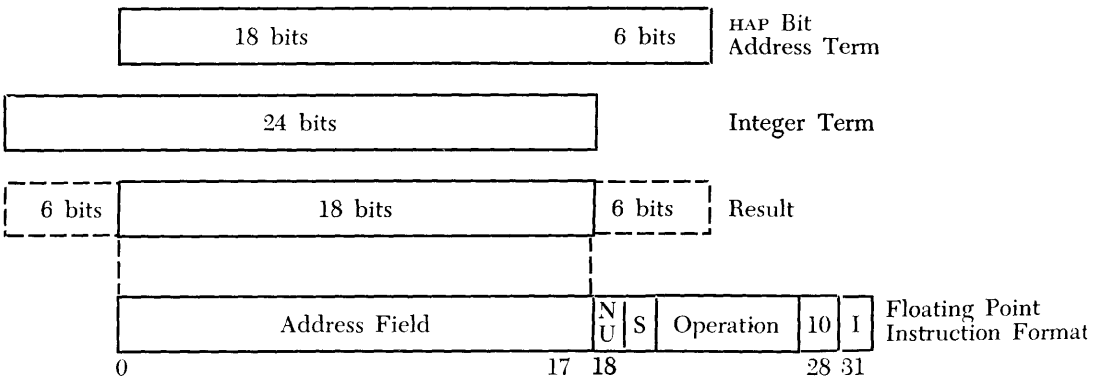
The sequence of steps HAP executes to perform addition and subtraction of like quantities in an address field is:

1. Converts each quantity to a 24-bit binary integer.
2. Quantities are aligned with respect to each other.
3. Numbers are assumed to be signed. Addition is algebraic.
4. The result is complemented if necessary. (Address fields are unsigned.) If the field is signed, such as an XW or VF, the sign bit is inserted in the correct bit and no complementation occurs.
5. The result is truncated, if necessary, to fit the particular address field.
6. The result is inserted into the correct position in the instruction.

When unlike quantities are added or subtracted, the sequence executed by HAP is the same with the exception of a slight modification in step 2. If integers and bit addresses are mixed, a certain amount of shifting, determined by the environment, must be performed before addition takes place. For example, in the floating point instruction

+ (N), 64.0 + 20

the address field is 18 bits in length. The rule for positioning bit addresses is clear: the point must always line up between the 18th and 19th bits in the address field. Earlier it was explained that an integer is right justified in a field; here the units position falls in the 18th bit. Thus, the two numbers are aligned as:



Arithmetic in any Programmer Symbolized Field

Although discussion has been limited to address fields, actually, all previous statements apply to any field where arithmetic is permitted, that is, any programmer symbolized field. Three restrictions must be observed.

1. No arithmetic may appear in the operation code part of the operation field, the mode subfield of the data description, or any entry mode. All of these fields are reserved for designations whose meanings to HAP are absolute and may not be symbolized.

2. No arithmetic may appear in the name field, which is reserved entirely for the definition of symbols. Only one symbol per statement is allowed.

3. The I or K fields must contain at least one HAP bit address term.

Address Arithmetic with Unlike Quantities

Addition and subtraction of unlike quantities require a complete set of rules for shifting and truncation. The two basic concepts involved are:

1. Where a bit address has meaning, the point is positioned between the 18th and 19th bits of the field. If a bit address has no meaning, the entire 24-bit quantity is treated as an integer and right justified in the field. Index fields are an exception.

2. An integer is always treated as an integer in the environment that is the size of the particular field. The integer is right justified so that its units position is aligned with the units position of the field.

Although the following diagrams show the final sum truncated to the appropriate length, the bits are not actually discarded unless they fall outside the address field of the instruction. Some operations do not use all of the space available in their address fields (transmit and input-output select, for example), and in these cases bits may be placed in the unused portions.

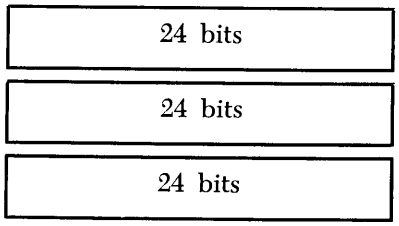
An error indication is given if nonzero bits are discarded when truncation occurs, except in the case of index fields where a 1 bit in the fifth position from the right (the 16 position) is discarded without error indication.

Truncation occurs for particular fields in the following manner:

1. A_{24} Bit Address

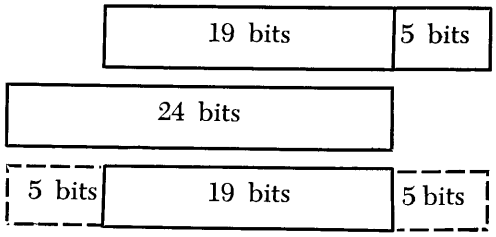
   Rule: No truncation

   Note: An integer in a 24-bit field counts bits



2. A_{19} Half-Word Address

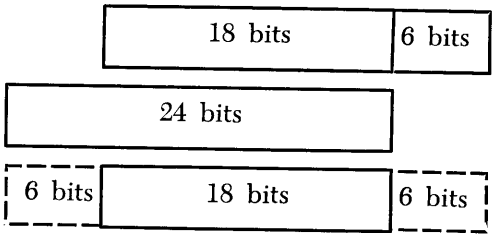
   Rule: Leftmost 5 bits and rightmost 5 bits are truncated from sum



   Note: An integer in a 19-bit field counts half-words

3. A_{18} Full-Word Address

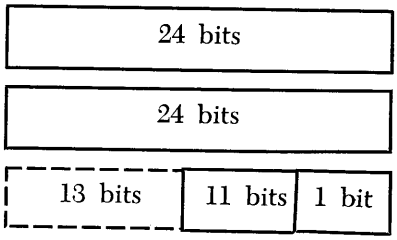
   Rule: Leftmost 6 and rightmost 6 bits are truncated from the sum



   Note: An integer in an 18-bit field counts full words or unit address, control operation, control word address, and so on, in right I-O address.

4. $A_{11\pm}$ Signed 11-Bit Address

   Rule: Leftmost 13 bits are truncated from the sum. Rightmost 11 bits plus sign are placed in leftmost 12 bits of address field of shift and Add Immediate to Exponent instructions

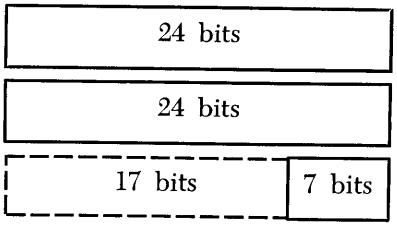


   Note: Integer counts number of bits in shift or number of bits to be added to exponent of floating point word.

5. OF_7 Offset

   Rule: Leftmost 17 bits of sum are truncated

   Note: Integers count number of bits of offset

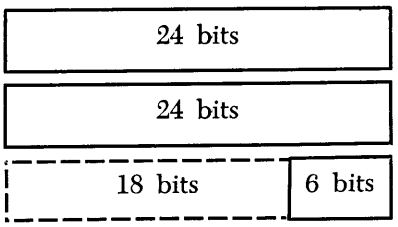


   Bit address 1.32 = .96 = integer 96

6. FL_6 Field Length

Rule: Leftmost 18 bits of sum are truncated

Note: Integers count length of field in bits



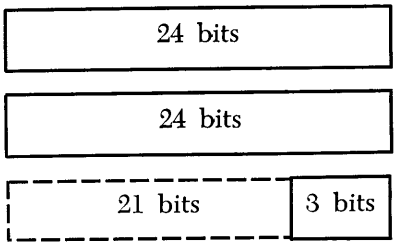
Bit address 1.0 = .64 = 0 not error marked

7. BS_3 Byte Size

Rule: Leftmost 21 bits of sum are truncated

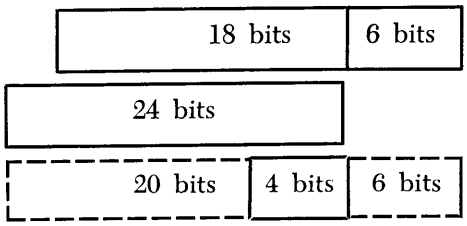
Note: Integers count byte size in bits

.8 = 8 = 0 not error marked



8. I, J 4-Bit Index Fields

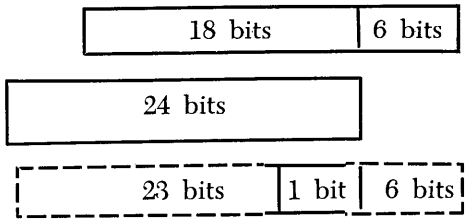
Rule: Leftmost 20 bits and rightmost 6 bits of sum are truncated



Note: Integers represent index register number. A "1" in the bit position immediately to the left of the final sum field is discarded with no error indication.

9. K Single Bit Index Field

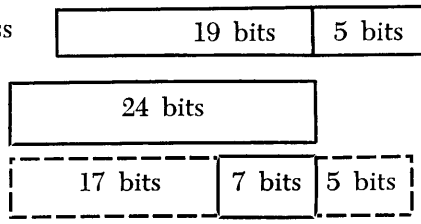
Rule: Leftmost 23 bits and rightmost 6 bits of sum are truncated



Note: Integers specify either index register 0 or index register 1. A "1" in the bit position that corresponds to "16" in the sum is discarded with no error indication.

10. A_7 I-O Left Effective Address

Rule: Leftmost 17 and rightmost 5 bits are truncated from sum



Note: Integers specify channel address

Immediate Operation Address Arithmetic

One special condition exists with immediate operation address fields. In this case, the treatment of a mixed expression consisting of both integers and bit addresses differs from the general rules stated previously. The treatment of integers is straightforward and the result justified on the left before insertion in the field (see DDI). If two or more bit address terms are being combined, the arithmetic is performed as usual but no left justification is done. The field length in the dds is ignored and the point is lined up between the 18th and 19th bits as in any other field. However, when integer and bit address terms are to be combined, all terms are considered to be bit addresses; they are aligned accordingly and the result is inserted as a bit address. The following immediate operation

```
LI (BU, 24, 8), 2 + 2.2 + 6
```

is treated by HAP as if it had been written

```
LI (BU, 24, 8), .2 + 2.2 + .6
```

Programmer and System Symbols

Programmer symbols, defined elsewhere in the code as integers or HAP bit addresses, may participate in the address arithmetic and no restrictions other than those already outlined need be observed. System symbols defined as bit addresses may also be used. Therefore,

```
ANKLEBONE SYN, 20.2
FOOTBONE SYN, 1
    L (BU), FOOTBONE + ANKLEBONE - 2 + 888.08
```

is permissible, while

```
CIRCLE ST (BU), CIRCLE - #PI
```

is not allowed, because #PI must be normalized floating point, not binary unsigned.

Number of Terms

There is no limitation on the number of terms that may appear in an arithmetic expression. Continuation cards must be used if the expression exceeds the space available on the symbolic card.

Multiplication and Division of Addresses

Arithmetic expressions involving multiplication and division are handled differently by HAP. Here the assembly program recognizes that certain combinations such as two or more integers, or integers and bit addresses can have meaningful results; while multiplying or dividing two or more bit addresses has little meaning, so that, although such operations are not prohibited, arbitrary rules are imposed on the arithmetic.

In multiplication and division, the basic precept is that both bit address terms and integer terms are treated as 24-bit integers and the bit address point is forgotten once the conversion to binary is accomplished. This means that the address expression

```
2.0 * 2
```

is the same as writing

```
128 * 2
```

and no shifting is done.

The two numbers are simply assumed to be integers, are aligned with respect to each other, and are multiplied or divided on this basis. The result is also treated as an integer, that is, it is right justified in the field in which it is being inserted. If the field is smaller than 24 bits in length, all truncation occurs on the left.

The sequence that HAP follows to multiply or divide an address expression which is a mixture of bit addresses and integers is

1. Converts all terms to 24-bit binary integers.
2. Assumes all terms are signed integers and multiplies or divides as requested.
3. Complements the result if necessary.
4. Truncates the result on the left, if necessary, to fit the particular field.
5. Inserts the result in the field as an integer, that is, right justifies it in the field.

Illustrations

An illustration of multiplication in an address field shows how three different expressions using the same numbers will produce three different results. In the first case

```
CM1010 (BU), 2 * 2 (#X7)
```

multiplication of two integers proceeds as would be expected and the arithmetic is:

```
   2
X  2
---
   4
```

If, however, in the second case the instruction had been written

```
CM1010 (BU), 2.0 * 2
```

the multiplication would now be

```
   128
X    2
-----
 256
```

In a third case, this address expression

```
CM1010 (CU), 2.0 * 2.0
```

is multiplied by HAP as

```
   128
X  128
-----
16384
```

The HAP bit address, when converted to 24-bit binary integer form, specifies an integral number of bits. The 24-bit representation of any integer is also an integral number of bits. The arithmetic results, therefore, are also treated as an integral number of bits. In case one, the answer is four bits as one would expect when multiplying two bits by two bits. In case two, the answer is 256 bits or four words, from multiplying two words by two. However, case three presents a multiplication of two-bit addresses wherein the results can only be arbitrarily defined; in this example, 16384 bits or 256 full words.

Interpretation of Results

The result of multiplication or division can be interpreted by HAP as a bit address. If the expression is enclosed in parentheses and followed by a period, the result is treated as a standard binary bit address, that is, it is appended by six zeros and inserted in the address field with the period lined up between the 18th and 19th bits. Truncation, if required, will be performed in the manner specified for bit addresses. To illustrate, the address expression in this instruction

```
M + (BU), 200 * 50
```

yields a result of 10000 which, when inserted in this address field as an integer, would count bits. If the expression had been written

```
M + (BU), (200 * 50).
```

the result 10000 would now be treated as a bit address, or 10000.0, which would count full words.

Two other alternatives are possible. The instruction could be written

```
M + (BU), 200.0 * 50
```

where the result is 640,000 which is treated as an integer and inserted in the field when compiled to yield an integral bit count. Again, by use of the special notation

```
M + (BU), (200.0 * 50.0).
```

bit address characteristics are attached to the integer result, yielding an address of 640,000.0.

An expression composed of all four types of arithmetic operations is permissible, such as

```
SRD (BU), 200 + 70.0 - 600 * 2 / 4
```

In this expression, HAP performs the arithmetic operations in the following order: multiplication, division, addition, and subtraction. The treatment of each term is in accordance with the rules described previously.

ARITHMETIC DATA OR CONTROL STATEMENTS

Data or control statements are operations created by HAP to provide a simplified means of performing some special functions that are required in writing most programs. For a complete description of this type of statement, see the Section entitled "Control Statements." Certain data or control statements are used almost exclusively in the arithmetic mode of operation and will therefore be discussed here. These statements include index word (XW), value field (VF), count field (CF), refill field (RF), link (LINK), and indicator mask (INDMK).

XW - Index Word

```
XW, VALUE, COUNT, REFILL, FLAG
```

The location counter is rounded to the next full word if it is not already at a full-word address. The contents of the four fields following the operation are compiled in an index word format. The quantity represented by the symbol VALUE is compiled in bits 0-24 of the full word compiled. COUNT is compiled in bits 28-45 of this word and REFILL is compiled in bits 46-63. FLAG denotes the index word field composed of bits 25, 26, and 27. An expression in the flag field of an XW statement is therefore evaluated modulo 2^3.

If the following statement were encountered by HAP in a program

XW, 1001.50, TOTAL, XWORD2, 4

a full word would be compiled in the format of an index word with 1001.50 in the value field, the quantity symbolized by the programmer symbol TOTAL in the count field, and the quantity symbolized by XWORD2 in the refill field, all converted to binary. The 4 is interpreted as the octal integer 4 in the three-bit flag field, which turns on the index flag bit in the index word compiled.

Note: Bit 24, the 25th bit in the word compiled, is assumed to be the sign bit for the value field. All the other fields are unsigned; a negative sign is interpreted in two's complement form in the usual way.

VF - Value Field

VF, VALUE

The location counter is rounded to the nearest half word if it is not already at a half word address. The quantity symbolized by VALUE is compiled in bits 0-24 of the next half word (24 bits plus sign). The location counter stands at bit 25 at the end of the operation.

CF - Count Field

CF, COUNT

The location counter is rounded to the next half word if necessary. The quantity symbolized by COUNT is compiled as an 18-bit integer in bits 0-17. The location counter stands at bit 18 at the end of the operation.

RF - Refill Field

RF, REFILL

This statement is treated exactly as CF, except the word refill should be substituted for the word count.

Note: The four operations just defined are given data descriptions by the compiler; therefore, they cannot be written by the programmer. Specifically, the index words or elements created by these orders have had the following data descriptions affixed automatically, and cannot be overrruled in the statement:

Operation	Data Description
XW	(BU)
VF	(B, 25)
CF or RF	(BU, 18)

LINK - Link

LINK

The LINK statement provides the programmer with a shorthand notation usually used as the beginning of an entry or linkage into a subroutine. At the point in the code where the LINK is encountered, HAP substitutes the operation

LVI, #15, # + 2

which uses index register 15 to store the instruction counter value of the return instruction and has become the standard entry mechanism. LINK must be followed by a branch instruction to complete the entry sequence.

INDMK - Indicator Mask

INDMK, A, A', A'', ...A_n

The A_i are programmer symbols, system symbols, bit addresses, or integers to specify any of the indicators in word 11. This control statement causes the location counter to be rounded to a full word and a 64-bit word is then constructed with 1's in the positions corresponding to the indicators named. A null field (the absence of any A_i) is compiled as zero. The bit corresponding to indicator zero, machine check (#MK), is turned on in the word compiled.

STREAMING MODE INSTRUCTIONS

Streaming mode instructions may be separated into four categories: streaming instructions, adjustments, setup, and indexing. The complete set of streaming mode instructions is given in the following pages.

In the formats shown here, that part of the instruction in capital letters (not underlined) is a nonprogrammer symbol or system symbol and must be written by the programmer exactly as shown. Fields named in lower case letters must be filled with the proper codes, as explained in the notes following the instructions. Fields underlined are to be filled by programmer symbols; that is, any symbolic name chosen by the programmer. Certain fields require a numeric entry whose radix, if not specified, is assumed to be binary or octal rather than decimal, which is the normal case. Note is made of these fields wherever they occur. Fields left blank or omitted are set as indicated in the notes. Commas and parentheses must be used as indicated.

Streaming Instructions

SBBB (data gates), luop, gs, TA (mode and cell size, parallel-serial, replace base address, demand parallel synchrony), STOP (stimulus), SETUP (name)

SMER (up-down, internal-external, simple-offset), gs, STOP (stimulus)

SSER (store data-store address, ordered-random, up-down, simple-offset, search condition), gs, STOP (stimulus)

SSEL (least-greatest, simple-offset), gs, STOP (stimulus)

STIR (replace-take, instruction-data control, up-down, simple-offset), gs, STOP (stimulus)

SQNL (data gates), gs, TA (mode and cell size, parallel-serial, replace base address, demand parallel synchrony), STOP (stimulus), SETUP (name)

SILS (load-store), gs, TA (mode and cell size, replace base address), STOP (stimulus), SETUP (name)

SNOP

Adjustments

ADJ (stimulus, tag), reaction 1, reaction 2, reaction 3

Setup

SETUP	
W	(character, connection, mode, span), action
X	(character, connection, mode, span), action
Y	(character, connection, mode, span), action
Z	(character, connection, mode, span), action
	Note: character must be a numeric entry and is assumed binary.
SC	(limit, value), stim, action
SA	(mode, threshold, value), stim
LU	(modulus, group size), luop
F	(stim), limit, action
TA	(mode and cell size, parallel-serial, replace base address, demand parallel synchrony)
	Note: TAA may be used in place of TA.
TBA	(address, TBAHO, MDM)
TAP	(TPS, TPI, TPN, TPJ)
TAQ	(TQS, TQI, TQN, TQJ), TAO, TAOHO, TPM
	Note: TBAHO and TAOHO code is 00, 01, 10, 11.
TE	(TEI, TEN, TEBM, TES, TEM)
	Note: TEBM must be a numeric entry and is assumed binary.
SSM	(mask)
DEBUG	(mask)
PAD	(starting data address, initial index table address)
QAD	(starting data address, initial index table address)
RAD	(starting data address, initial index table address)
SETEND	

Indexing

PX, QX, RX	(mode, EC/CC, FL, FF, SR, BL, runout or R control, TRU or TRL, FS), increment, total count, byte mask/ branch address, reset address, current count
PXO, QXO, RXO	(same as above), increment offset, total count offset, byte mask/branch address, offset, RBL, reset address, current count
	Note: byte mask must be a numeric entry and is assumed binary.
TX	(FL), byte size in increment, count of bytes in record
TXO	(FL), byte size in increment, count of bytes in control field, offset of control field, RBL.
TEY	(TEC/TCC, TQ/TR), address, offset signed, TEN count, data
	Note: data must be an octal entry, with no radix specified.

BASIC CONCEPTS

The general format for the streaming mode instructions is similar to that used for arithmetic mode instructions, but certain fields must be identified by the programmer. In general, a complete streaming instruction must give the operation, the data gating, the operation of the various units in the streaming pipeline (that is, the logic unit, counter, accumulator, table address assembler, and so on), and a stop code, This must be followed by five half words of adjustments (stimuli and reactions). Also, the programmer must specify the contents of a setup area of at least ten full words, to be transmitted into the 7950 registers before initiating a streaming operation. Further, he must provide at least two full words of indexing information for each of the three streaming units to be used in the operation. And, for one operation, he must provide a table whose entries have a special format. A method provided to do this is covered in the following sections.

Definitions

In general, the terms defined previously under "Arithmetic Mode Instruction," will be used with no change in the streaming mode instructions. New terms and special meanings assigned to familiar terms are defined below:

Entry Mode: A method for describing the form in which data appears; either alphabetic or, if numeric, either decimal, binary, octal, and so on.

Right-to-Left Dropout: As indicated previously, right-to-left dropout means that missing fields are assumed to have been dropped from the right end of the statement and their place taken by zeros. A major field may be null even if other non-null fields follow, as long as the comma denoting the field termination is written.

Statement Fields and Subfields: Statements of the streaming mode are, like arithmetic mode, separated by commas. Most streaming mode statements are composed of a primary operation field and one or more secondary fields. Either of these primary or secondary fields may be further subdivided into one or more subfields. A subfield is made up of one or more subfield elements. The first subfield is always enclosed in parentheses. Actually, multiple subfields only occur in the setup instruction, in which each of the possible statement entries is considered to be one field of a single setup instruction. Ten full words are therefore possible, with the mnemonic SETUP considered as the primary operation field, and the mnemonic SETEND terminating the instruction. Refer to Figure 9 for the use of the terms just described.

General Rules for Streaming Mode

1. If all elements of a primary operation subfield are null, the subfield may be omitted, but the primary operation mnemonic and its following parenthese must still be retained.

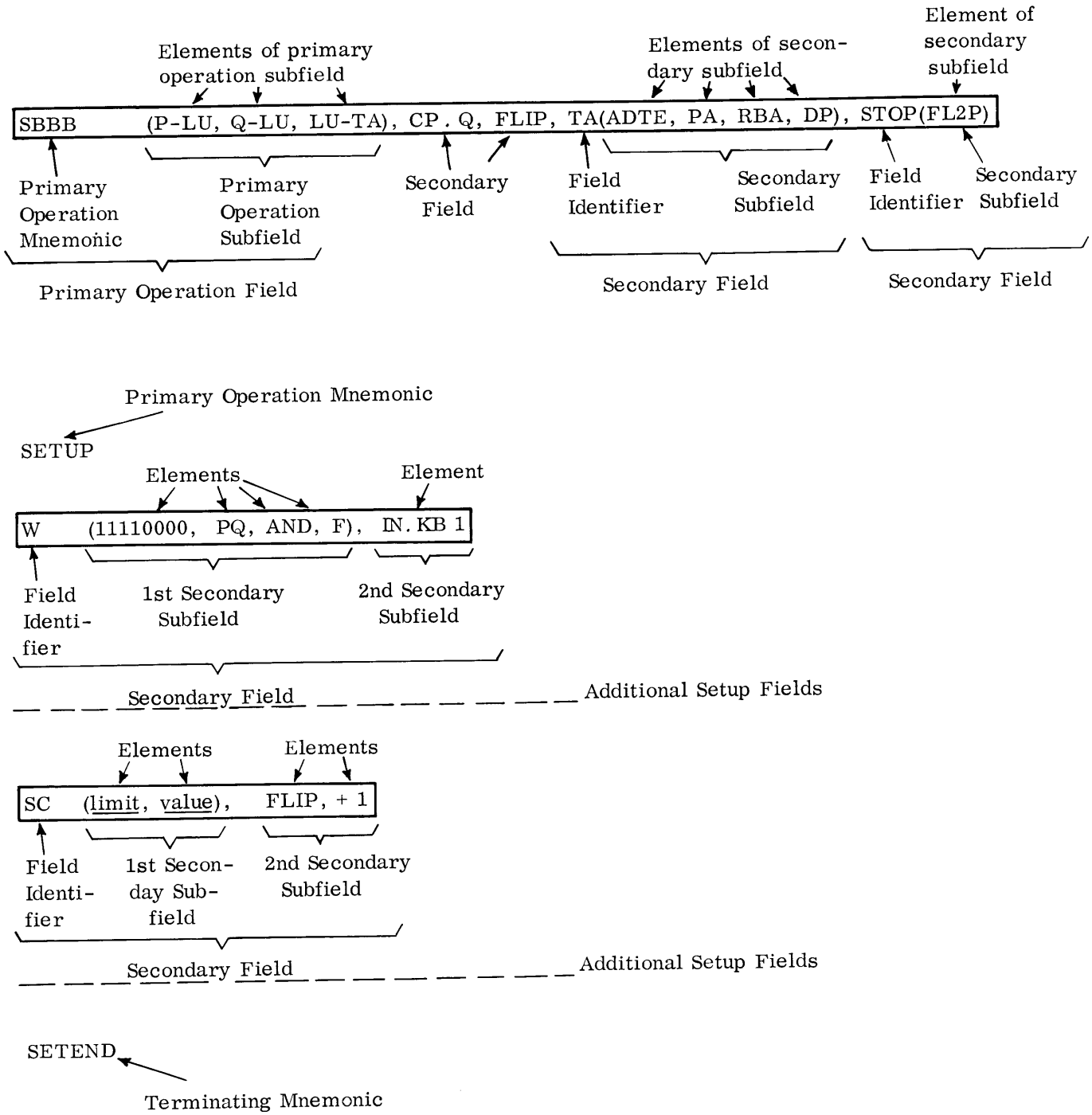


Figure 9. Statement Fields and Subfrelds

2. If all elements of a secondary subfield are null, the subfield may be dropped. If the subfield were enclosed in parentheses, the parentheses may also be dropped.

3. If all elements of all secondary subfields are null, the entire field, including field identifier, may be omitted.

4. If all elements of any subfield are nonprogrammer symbols, they can be interchanged or omitted, as desired. If all elements of any subfield are programmer symbols or mixed programmer and nonprogrammer symbols, the standard right-to-left dropout rule applies.

5. All fields in the streaming instructions and in setup may be interchanged or omitted, as desired. Fields in all other instructions follow the right-to-left dropout rule.

6. If fields or elements are omitted, the assembly program fills in zeros, with the following exceptions:

Omitted Fields	HAP Fills In
FLIM	1
TEN	3
PAD, QAD, RAD	A valid address
Search condition for SSER	PEQ (010)

7. The radix of all numeric field entries is assumed decimal unless otherwise specified. Exceptions are byte mask and match characters, whose radix is assumed to be binary; also the data field in TEY, whose radix is assumed octal.

8. The 7950 assembly program reserves a three word area for a streaming instruction and its principal adjustments, no matter how many such adjustments are used. No padding is necessary.

9. If IC + 3 is to be followed by supplementary adjustments and the right half of location IC + 3 is not used, a CNOP must be inserted at this point.

10. All symbolic names used for streaming mode operations must not exceed eight characters.

11. A period indicates a logical AND and a V denotes a logical OR. The symbol XV denotes logical exclusive OR.

12. Although logically correct, SEQ should not be used as an entry in the first level of indexing because parity errors may result.

STREAMING INSTRUCTIONS

In each of the following instruction fields a reference is made to a page number of the IBM 7950 Data Processing System Reference Manual (September 1, 1961) on which a more complete explanation of the field may be found.

Stream Byte-by-Byte

The format of this instruction is:

SBBB (data gates), luop, gs, TA (mode and cell size, parallel-serial, replace base address, demand parallel synchrony), STOP (stimulus), SETUP (name)

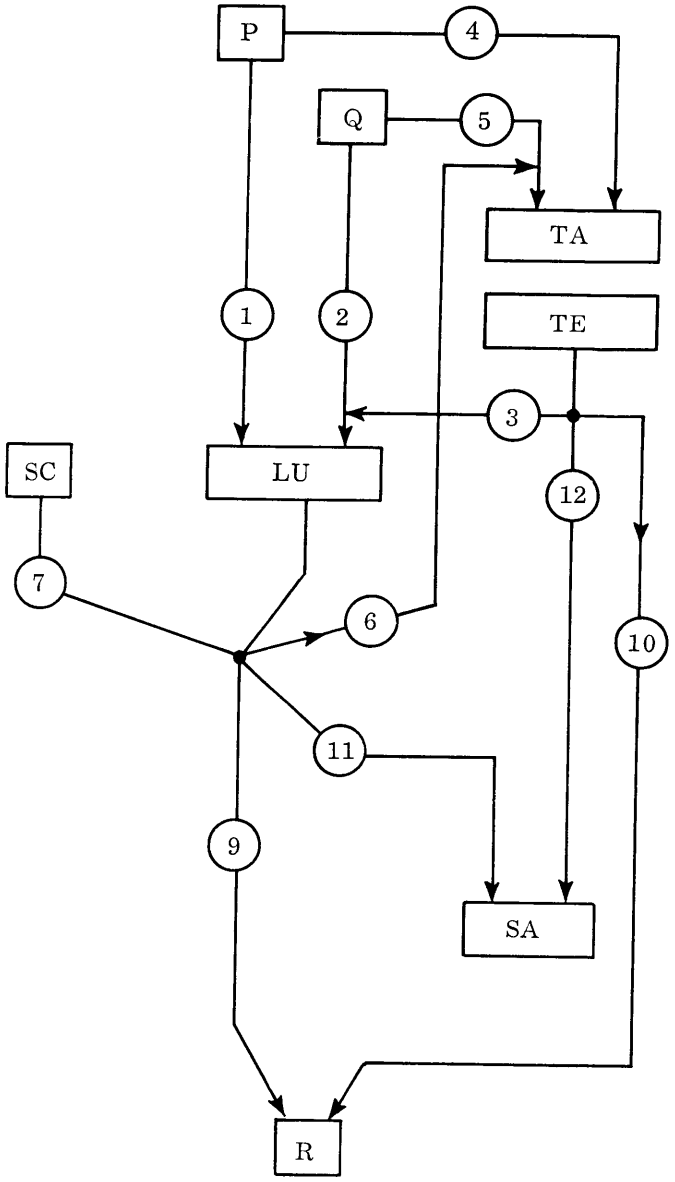
Sample Entry

SBBB (P-LU, Q-LU, LU-TA, TE-R), CP.Q, FL1P, TA (XOR, PA, RBA), STOP (FL2Q)

Coding of SBBB Fields

data gates (bits 1-12) page 8.6

This field gives the names of the data gates to be opened for the instruction. The names of the data gates are nonprogrammer symbols and must be written as shown.



Gates:

Bit Number	Name
1	P-LU
2	Q-LU
3	TE-LU
4	P-TA
5	Q-TA
6,7	SC-TA
6	LU-TA
Bit 7 = 1	SC
Bit 7 = 0	LU
7,9	SC-R
9	LU-R
10	TE-R
7,11	SC-SA
11	LU-SA
12	TE-SA

Illegal Gating Combinations:

Bit Number	Name
2,3	Q-LU & TE-LU
5,6,7	Q-TA & SC-TA
5,6	Q-TA & LU-TA
3,6,7	TE-LU & SC-TA
3,6	TE-LU & LU-TA
7,9,10	SC-R & TE-R
9,10	LU-R & TE-R
7,11,12	SC-SA & TE-SA
11,12	LU-SA & TE-SA
1,3,4	P-LU & TE-LU & P-TA

Note: If a gate is opened leading out of TE (3,10, or 12), a gate must be opened leading into TA(4,5, or 6). If SC (7) is used, there must be at least one input to LU(1,2, or 3).

This five-bit field gives the logic unit operation code. The programmer must use nonprogrammer symbols, written as shown:

Octal	Symbolic	Description	
0	CZ (assumed if no field is written)		
1	CP.Q		
2	CP.NQ		
3	C P		
4	C NP.Q		
5	C Q		
6	C PXVQ		
7	C PVQ		
10	C NP.NQ		
11	C PEQ		
12	C NQ		
13	C PVNQ		
14	C NP		
15	C NPVQ		
16	C NPVNQ		
17	C 1		
20	MAX PQ	Output larger byte	
21	MIN PQ	Output smaller byte	
22	EPZ	P if P=Q, else zero	
23	EPNB	P if P=Q, else no output	
24	EZP	P if P≠Q, else zero	
25	ENBP	P if P≠Q, else no output	
26	EZQ	Q if P≠Q, else zero	
27	ENBQ	Q if P≠Q, else no output	
30	GEP-QZ	P-Q if P≥Q, else zero	
31	GEP-QNB	P-Q if P≥Q, else no output	
32	LEQ-PZ	Q-P if P≤Q, else zero	
33	LEQ-PNB	Q-P if P≤Q, else no output	
34	MODP-Q	P-Q modulo the modulus	The modulus is specified in the LU field in setup.
35	MOD Q-P	Q-P modulo the modulus	
36	RDX P + Q	P+Q, no carry propagate	
37	MOD P + Q	P+Q, modulo the modulus	

gs (bits 19-21) page 4.56

This three-bit field is used to define the group size for the logic unit. If no code is written by the programmer, the field is made zero and the entire stream is treated as a group. The following nonprogrammer symbols are used:

Code	Binary	Definition
NOP	000	None, each byte is a group
FL1P	001	Last byte of flagged indexing level
FL2P	010	Last byte of flagged indexing level
FL1Q	011	Last byte of flagged indexing level
FL1R	100	Last byte of flagged indexing level
W	101	Signal from match unit W
Z	110	Signal from match unit Z
XVY	111	Signal from either X or Y

TA-mode and cell size (bits 24-25) (bits 26-27) page 4.31

Since the actual operation performed with the table address assembler depends on the contents of both mode and cell size, the operation codes are given in a form descriptive of the operation, not the individual fields. The code is written in nonprogrammer symbols, and, if omitted, ADTE is assumed.

Binary	Code	Description
0000	ADTE	No memory reference. Address is sent directly to TE. If bit 24 is 0, the cell-size is ignored.
0001	ADTE	
0010	ADTE	
0011	ADTE	
0100	X	The word address part of TAO fetches a word from memory to TE. The bit address is loaded into TES. (cell-size is ignored)
0101	X	
0110	X	
0111	X	
1000	OR	OR a one into the addressed bit in memory.
1001	CT8	Add a one into addressed bit, cell size 8.
1010	CT16	Add a one into addressed bit, cell size 16.
1011	CT24	Add a one into addressed bit, cell size 24.
1100	XOR	Combination of X and OR
1101	XCT8	Combination of X and count
1110	XCT16	
1111	XCT24	

The assembly program inserts 0000 and 0100 for ADTE and X, respectively.

TA - parallel-serial (bit 22) page 4.30

This bit determines whether bytes from P and Q are first OR'ed together and then added to TBA, or whether they are added serially to TBA without first OR'ing. If not coded, PA is assumed. The codes are:

Binary	Code	Description
0	PA	Parallel, OR'ed first then added
1	SE	Serial addition to TBA

TA - replace base address (bit 23) page 4.31

This bit is coded as follows:

Binary	Code	Description
0	blank	If blank, each address is formed by adding bytes to TBA.
1	RBA	In this case, the first address is formed as above, but all subsequent ones are formed relative to the one just preceding.

TA - demand parallel synchrony (bit 28) page 4.33

This bit controls stream flow to the TA and is coded as follows:

Binary	Code	Description
0	blank	Allow bytes to move synchronously to TA.
1	DP	Force synchronous movement by MU to TA.

STOP (stimulus) (bits 29-31) page 4.57

In this field the programmer writes the code for the stimulus that is to terminate the execution of the streaming instruction. If omitted, a NOP is inserted by the assembly program. Note that a streaming instruction is always terminated by a continue chain bit = 0 in an indexing level, and in addition, can be terminated by any recognized interrupt, or by the adjustment reaction "go to LC."

Either a satisfied STOP code, "go to IC," or an interrupt leaves the stream dormant; it may be resumed, provided nothing in the setup register is disturbed prior to resumption.

The STOP codes are:

Octal	Code	Stimulus
0	NOP	No STOP defined, other than continue chain = 0.
1	FL2P	End-of-level in indexing with flag 2 in P.
2	FL2Q	End-of-level in indexing with flag 2 in Q.
3	FL2R	End-of-level in indexing with flag 2 in R.
5	FL3P	End-of-level in indexing with flag 3 in P.
6	FL3Q	End-of-level in indexing with flag 3 in Q.
7	FL3R	End-of-level in indexing with flag 3 in R.

SETUP (name)

This field is given only as a convenience to the programmer, and allows either the complete TA field (with the exception of DP) or the logic unit op code or group size to be inserted in a streaming instruction from a SETUP area. If used, only those fields that are completely blank in the stream instruction will be filled in from the named SETUP area. For example, TA (mode and cell size, parallel-serial, replace base address) must be blank in the stream instruction if it is desired to make these entries from the SETUP area. Note that the presence or absence of the TA (DP) entry has no effect on the ability to make these entries. The logic unit op code or group size fields must also be completely blank if they are to be entered from SETUP.

Stream Merge

The format of this instruction is:

SMER (up-down, internal-external, simple-offset), gs, STOP(stimulus)

Sample Entry

SMER (DN, IN, SIM,), FL1R, STOP (FL2P)

Coding of SMER Fields page 8.13

up-down (bit 13)

Merge up produces a record sequence with control fields in ascending order; merge down, in descending order. If omitted, the bit is set to UP.

Bit 13	Code	Description
0	UP	Files are ordered up
1	DN	Files are ordered down

internal-external (bit 12)

If omitted, this bit is set to IN. The codes and descriptions are:

Bit 12	Code	Description
0	IN	Entire block takes less than half of internal memory available.
1	EX	Block takes more than half of internal memory.

simple-offset (bit 14)

This bit is set to SIM if omitted. Codes and descriptions are as follows:

Bit 14	Code	Description
0	SIM	The control field heads the record, and the entire record may be considered the control field.
1	OFF	For all cases where both conditions for SIM are not met.

gs (bits 19-21

Refer to the gs field description in section entitled "Coding of SBBB Fields."

STOP (stimulus) (bits 29-31)

Refer to the STOP (stimulus) description in section entitled "Coding of SBBB Fields."

Stream Search

The format of this instruction is:

SSER (store data-store address, ordered-random, up-down, simple-offset, search condition), gs, STOP (stimulus)

Sample Entry

SSER (DA, RAN, UP, SIM, PEQ), FL1R, STOP (FL2P)

store data-store address (bit 11)

If omitted, the bit is set to DA.

Bit 11	Code	Description
0	DA	Normal search operation.
1	AD	Only the address of the record satisfying the search condition is sent to R, and R indexing must be set to store a 24-bit address.

ordered-random (bit 12)

The bit is set to ORD if omitted.

Bit 12	Code	Description
0	ORD	Sets instruction for ordered records in P.
1	RAN	Assumes random ordering of P records.

up-down (bit 13)

Refer to the up-down description in section entitled "Coding of SMER Fields."

simple-offset (bit 14)

Refer to the simple-offset description in section entitled "Coding of SMER Fields."

search condition (bits 8-10)

If omitted, this entry is coded PEQ.

Binary	Code	Condition
000	- - -	Invalid
001	PLQ	$P < Q$
010	PEQ	$P = Q$
011	PLEQ	$P \leq Q$
100	PGQ	$P > Q$
101	PNEQ	$P \neq Q$
110	PGEQ	$P \geq Q$
111	- - -	Invalid

gs (bits 19-21)

Refer to the gs field description in section entitled "Coding of SBBB Fields."

STOP (stimulus) (bits 29-31)

Refer to the STOP (stimulus) description in section entitled "Coding of SBBB Fields."

Stream Select

The format of this instruction is:

SSEL (least-greatest, simple-offset), gs, STOP (stimulus)

Sample Entry

SSEL (GST, SIM,), FL1R, STOP (FL2P)

Coding of SSEL Fields page 8.36

least-greatest (bit 13)

If omitted, this bit is set to LST.

Bit 13	Code	Description
0	LST	Record with least control field is selected.
1	GST	Record with greatest control field is selected.

simple-offset (bit 14)

Refer to the simple-offset description in section entitled "Coding of SMER Fields."

gs (bits 19-21)

Refer to the gs field description in section entitled "Coding of SBBB Fields."

STOP (stimulus) (bits 29-31)

Refer to the STOP (stimulus) description in section entitled "Coding of SBBB Fields."

Stream Take - Insert - Replace

The format of this instruction is:

STIR (replace-take, instruction-data control, up-down, simple-offset), gs, STOP (stimulus)

Sample Entry

STIR (RPL, IC, UP, OFF), FL1R, STOP (FL2P)

Coding of STIR Fields page 8.42

replace-take (bit 12)

If omitted, this bit is set to RPL.

Bit 12	Code	Description
0	RPL	Q records matched in P are inserted in P file, replacing matched record.
1	TAKE	Matched records are deleted from the P file.

instruction-data control (bit 10)

If omitted, this bit is set to IC.

Bit 10	Code	Description
0	IC	Instruction control, and replace-take bit is effective.
1	DC	Data control, and the replace-take control bit is ignored.

up-down (bit 13)

Refer to the up-down description in section entitled "Coding of SMER Fields."

simple-offset (bit 14)

Refer to the simple-offset description in section entitled "Coding of SMER Fields."

gs (bits 19-21)

Refer to the gs field description in section entitled "Coding of SBBB Fields."

STOP (stimulus) (bits 29-31)

Refer to the STOP (stimulus) description in section entitled "Coding of SBBB Fields."

Stream Sequential Look-Up

The format of this instruction is:

SQNL (data gates), gs, TA (mode and cell size, parallel-serial, replace base address, demand parallel synchrony), STOP (stimulus), SETUP (name)

Sample Entry

SQNL (P-TA, Q-TA, TE-R), FL1P, TA(CT8, PA, RBA), STOP (FL2R)

Coding of SQNL Fields page 8.62

For a discussion of the coding of gs, TA, STOP, and SETUP fields, refer to section entitled "Coding of SBBB Fields."

data gates (bits 4, 5, 10, 12)

Only the following data gates are available for the SQNL instructions. For a detailed description of the data gates, refer to section entitled "Coding of SBBB Fields."

Gate No.	Name	Function
4	P-TA	Bytes go from P to TA
5	Q-TA	Bytes go from Q to TA
10	TE-R	Table extract output goes to R
12	TE-SA	Table extract output goes to SA

Either two or three gates must be specified: P-TA and/or Q-TA and TE-R and/or TE-SA.

Stream Indirect Load or Store

The format of this instruction is:

SILS (load-store), gs, TA(mode and cell size, replace base address), STOP (stimulus), SETUP (name)

Sample Entry

SILS (LD), FL1R, TA (OR, RBA), STOP (FL2P)

Coding of SILS Fields page 8.56

For a discussion of the coding of gs, TA, STOP, and SETUP fields, refer to section entitled "Coding of SBBB Fields."

load-store (bit 14)

Bit 14	Code	Description
0	LD	Address from TE goes to Q indexing.
1	ST	Address from TE goes to R indexing.

Either LD or ST must be specified.

Stream No Operation

The format of this instruction is:

SNOP

Only the primary operation mnemonic is written; no other fields are coded. For a full explanation of this instruction, refer to page 8.74 of the IBM 7950 Data Processing System Reference Manual (September 1, 1961).

ADJUSTMENTS

Each time the machine performs a streaming instruction the instruction counter is stopped at the location of the streaming instruction. The normal exit is to the instruction at the third word beyond the streaming instruction, usually called IC + 3. This leaves five half words which can be used for five adjustments, Each adjustment is written as follows:

ADJ # (stimulus, tag), reaction 1, reaction 2, { reaction 3 / or / relative address }

For a full explanation of Adjustments, refer to chapter 6, IBM 7950 Data Processing System Reference Manual.

Coding of Adjustment Fields

#

The number of the adjustment may be entered here. For programmer convenience, the adjustments may be numbered, but this will have no effect on the assembly; they will be assembled in the order written. Only numbers 1-5 may be used.

stimulus

The programmer writes here the name of the stimulus that will activate the adjustment. The stimuli and octal equivalent of the code are given in the table of Adjustment Stimuli in the following section.

tag

This field may be omitted (blank), or one of three tags may be used. Each alternative is explained below:

1. blank (00) In this case there are three reaction fields; each reaction is performed if the adjustment is activated.

2. AND (01) In this case the reactions take place only if the stimulus specified in the next adjustment is also ON. AND must not be used in ADJ 5.

3. BR (10) In this case, the first two reactions are completed, any active interrupts are taken, all quantities in setup registers are put in true form, and indexing is frozen such that the first byte of the first reading level encountered is ready to be read out. If the level being performed at adjustment time is a reading level, the indexing is frozen such that the byte that causes the adjustment is in the next time zone. The IC is then set to address the instruction whose name is given in the reaction 3 field, and the machine leaves the streaming mode.

4. CH (11) If this tag is used, reactions 1 and 2 are completed and then the adjustment addressed by the reaction 3 field (actually a relative address) is brought in. If the stimulus specified by the new adjustment is on, the reactions are performed. The tag of all adjustmdnts in a chain, except the last, must be CH. The stream stimulus mask (SSM) has no effect on the reactions of adjustments in a chain, except for the first one that starts the chain.

reaction

The programmer writes the names of the desired reactions, separated by commas. If less than three are given, a NOP will be inserted. Names, code, and a brief description are given in the table of Adjustment Reaction and Codes. In an adjustment, a NOP may be used in the first or third reaction field, but not the second, unless the tag BR or CH is used and the third field is a relative address. If this rule is not followed, a NOP in the second field causes the machine to treat the third reaction as a NOP also.

relative address

If the BR or CH tag is used, reaction 3 becomes a relative address. The reaction field is only eight bits, and the machine actually uses this as a relative address, adding it to the contents of IC in such a way that a one in the action field steps IC by a half word. Thus all extension and chained adjustments must be within the 255 half words following the streaming instruction.

The programmer must indicate the numeric value of the relative address by any permitted numeric entry mode or by giving the symbolic value of the relative address by arithmetic, such as ADJST - STREAM, where ADJST is the name of the extra adjustment or arithmetic mode instruction, and STREAM is the location of the streaming instruction.

Table of Adjustment Stimuli

Octal	Stimulus	Description	Octal	Stimulus	Description
5	FL1P	Flag 1, P Stream	43	KBZ	KB = 0
6	FL2P		45	LBZ	LB= 0
7	FL3P		46	MBZ	MB = 0
1	FL1Q	Flag 1, Q Stream	44	KB1	KB = 1
2	FL2Q		42	LB1	LB = 1
3	FL3Q		41	MB1	MB = 1
21	FL1R	Flag 1, R Stream	53	KGZ	KG = 0
22	FL2R		55	LGZ	LG = 0
23	FL3R		56	MGZ	MG = 0
24	W	W unit signal	54	KG1	KG = 1
26	X		52	LG1	LG = 1
40	Y		51	MG1	MG = 1
37	Z		67	F1	F = 1
27	XVY	X or Y unit signal	63	F1.KBZ	F = 1 and KB = 0
15	W.X	W and X unit signal	65	F1.LBZ	
16	W.Y	W and Y unit signal	66	F1.MBZ	
11	NW	Not W	64	F1.KB1	F = 1 and KB = 1
12	NX		62	F1.LB1	
13	NY		61	F1.MB1	
14	NZ		70	FZ.EG	F = 0 and end of group
17	CCZ	End of chain	73	KGZ.EG	KG=0 and end of group
25	SCLIM	SCTR = Limit	75	LGZ.EG	
30	SCNL.EG	SCTR ≠ Limit and EG	76	MGZ.EG	
10	SAGETH	SACC ≥ Threshold	74	KG1.EG	
32	SABN	SACC becomes negative	72	LG1.EG	
31	SALT.EG	SACC < Threshold and EG	71	MG1.EG	
50	ELTE	End of level in TE	77	EG (or EGLOG)	End of group
33	FZ	F = 0	47	STIM (or on)	always on
35	FZ.KB1	F = 0 and KB = 1	0	NOP	
34	FZ.LB1		60	INIT	Initial
36	FZ.MB1				

Note: Stimuli 34-36, 61-66, and 70-76 are all AND forms which can be written in reverse order. For example, FZ.MB1 may be written MB1.FZ.

Type	Group	Description	Mnemonic	Octal	IBM 7950 Section Reference
No Op			NOP	000	6.5.7
Statistical Units	RESET:	F and G	RSFG	025	4.2.1.4
		SC	RSSC	031	4.3.2
		SA	RSSA	026	6.5.4
	STEP:	SC by +1	SC + 1	023	4.3.1
		SC by -1	SC - 1	034	4.3.2
		SA by +1	SA + 1	032	6.5.4
	READ OUT SA: to R	low order 8 bits	RO8SA	052	6.5.4
		low order 16 bits	RO16SA	046	
		entire 24 bits	RO24SA	043	
	READ OUT SC:	low order 8 bits to R	RO8SC	063	6.5.4
		entire 16 bits to R	RO16SC	073	
		Add to TBA	SC + TBA	061	6.5.3
Indexing	RESET:	P This level	RSP	344	6.5.1
		Through FL1	RSFL1P	346	
		FL2	RSFL2P	345	
		FL3	RSFL3P	347	
		Q This level	RSQ	350	6.5.1
		Through FL1	RSFL1Q	352	
		FL2	RSFL2Q	351	
		FL3	RSFL3Q	353	
		R This level	RSR	324	6.5.1
		Through FL1	RSFL1R	326	
		FL2	RSFL2R	325	
		FL3	RSFL3R	327	
	ADVANCE:	P Next level	ADP	340	6.5.1
		Through FL1	ADFL1P	342	
		FL2	ADFL2P	341	
		FL3	ADFL3P	343	
		Q Next level	ADQ	354	6.5.1
		Through FL1	ADFL1Q	356	
		FL2	ADFL2Q	355	
		FL3	ADFL3Q	357	
		R Next level	ADR	320	6.5.1
		Through FL1	ADFL1R	322	
		FL2	ADFL2R	321	
		FL3	ADFL3R	323	
Table Reference Unit	Reference TBA-1		RFTBA-1	361	6.5.3
	Skip extraction		SKTA	362	
	Reset base address to TBA		RSTBA	364	
	Cancel address		OMTA	370	

Type	Group	Description	Mnemonic	Octal	IBM 7950 Section Reference
Disable	Specified trigger(s) for this byte		B12345 $010T_5T_4T_3T_2T_1$		
	for duration of group		G12345 $011T_5T_4T_3T_2T_1$		6.5.5
	Match units for runout		DISMU	315	
Pipeline	INSERT	W in LUO	INW	214	6.5.2
		X	INX	200	
		Y	INY	204	
		Z	INZ	210	
		MOD	INMOD	202	
		MOD in TE out	INMODTE	203	
	RUNOUT	P This level	ROP	220	6.5.1
		Through FL1	ROFL1P	222	
		FL2	ROFL2P	221	
		Q This level	ROQ	224	6.5.1
		Through FL1	ROFL1Q	226	
		FL2	ROFL2Q	225	
	MATCH ONLY	P This level	MOP	230	6.5.1
		Through FL1	MOFL1P	232	
		FL2	MOFL2P	231	
		Q This level	MOQ	234	6.5.1
		Through FL1	MOFL1Q	236	
		FL2	MOFL2Q	235	
	STORE ADDRESS	P	STPS	240	6.5.6
		Q	STQS	244	
	OMIT	byte after special byte from P	OMP	260	6.5.2
		Q	OMQ	264	
		TE	OMTE	266	
		Special byte output of LU	OMLU	262	
		Special byte into R	OMR	263	
	REPEAT	Special byte from P	RPP	270	6.5.2
		Special byte from Q	RPQ	274	
Misc.	Suppress LU output for duration of group		SULU	140	
	Skip space in R		SSR	304	
	Skip remaining TE extraction, this reference		SKTE	310	6.5.2

Table of Adjustment Reactions and Codes

SETUP INSTRUCTION

A setup instruction starts with the mnemonic SETUP and ends with SETEND, and causes ten full words to be reserved in storage. If the programmer writes a complete setup, he fills this storage space with a complete set of fields. For a partial setup, the programmer writes SETUP, enters his required fields, then closes with SETEND. The written setup information is inserted in the proper places and the remaining areas are either left zero or set arbitrarily to avoid errors during running of the object program, as indicated in the description of each field.

SETUP and SETEND Mnemonics

The mnemonics SETUP and SETEND must be used to identify the setup fields. When HAP encounters SETUP, the location counter is rounded to the next full-word address and the next ten words are cleared to zero.

SETUP normally contains a name, either absolute or symbolic, in its name field. An error message indicates the omission of a name on a setup statement. For example:

```
JOE     SETUP
        W (01000010, PVQ), IN
        PAD (400, PETE)
        QAD (500, BILL)
        RAD (600, TOM)
        SETEND
```

Setup Fields

As mentioned previously, to write a complete setup of ten words, the programmer writes SETUP (with a name) followed by the name of the setup fields and the information to be put in them, then ends with SETEND. Each field name or identifier (such as W, TA, TAQ, and so on) should appear as if it were the op-code of an instruction, and the codes in the parentheses following must appear on the same card (or a continuation card). For example:

```
        SA (U, 125, 0), KBZ   is correct, but

        X ( (8) 377, PQLU, OR, F),
        OMALL   is not correct, unless the second card is a continuation
                card.
```

W (character, connection, mode, span), action

page 4.9, Reference Manual

character: must be a numeric entry mode and is assumed to be binary unless otherwise specified.

connection: indicates which part of the stream the character is to be matched. The code must be written as shown.

Octal	Code	Description
0	NOP	No connection
1	P	P only
2	Q	Q only
3	PQ	P and/or Q

mode: may be coded for OR or AND mode. For example, PQ connection in the OR mode indicates that both the P and Q units are connected and that a stimulus signal would arise upon a successful match between the match character and either a P byte or a Q byte, or both. Similarly, PQ in the AND mode indicates that both the P and Q units are connected and that a stimulus signal would arise only upon a successful match between the match character and both the P and Q bytes simultaneously. In fact, in AND mode an adjustment stimulus is emitted only when simultaneous matches occur, or on an LUO match.

span: coded R for match on right bit only; F for match on full byte. If omitted, F is assumed.

action: Depending on the coding, one of the following actions will be taken when a successful match is made.

Octal	Code	Description
0	NOP	No-operation
1	IN	Omit both LU inputs and insert character in LU Output
2	OM	Omit byte which matched
3	OMALL	Omit matched byte and corresponding bytes from other connections
4	IN.KB1	Omit both LU inputs, insert character in LU Output, and set KB to 1
5	IN.MB1	Omit both LU inputs, insert character in LU Output, and set MB to 1

Sample Entry: W (11110000, PQ, AND, F), OM

X (character, connection, mode, span), action

page 4.51

character: See W field (preceding section)

connections:

Octal	Code	Description
0	NOP	No connection
1	P	P only
2	Q	Q only
3	PQ	P and/or Q
4	LU	LU only
5	PLU	P and/or LU
6	QLU	Q and/or LU
7	PQLU	(P and/or Q) or LU

mode: may be coded for OR or AND mode. See W field (preceding section). In the case of the PQLU connection, the programmer may use the AND mode to specify P.QVLU or use the OR mode to specify PVQVLU. With AND mode specified, a signal results from a match with the LU connection or with both the P and Q connections. If OR mode is used, a signal arises from a match with either the P or the Q or the LU connection. Only the 'OM' action has meaning if 'LU' is connected; any others specified are treated as NOPS.

span and action: See W field (preceding section). Note: A match unit action cannot be caused by a match at LU alone.

Y (character, connection, mode, span), action

page 4.53

character: See W field.

connections:

Octal	Code	Description
0	NOP	No connection
1	P	P only
2	Q	Q only
3	PQ	P and/or Q
4	TE	TE only
5	PTE	P and/or TE
6	QTE	Q and/or TE
7	PQTE	P or Q or TE (P and Q) or (P and TE)

mode: See W field and X field.

span and action: See W field.

Z (character, connection, mode, span), action

page 4.54

character: See W field.

connections:

Octal	Code	Description
0	NOP	No connection
1	TE	TE only
2	LU	LU only
3	LUTE	TE or LU

mode: the mode need not be specified, since only OR mode is used. Note: No stimulus signal will be emitted on any match.

span and action: See W field. Note: A match unit action cannot be caused by a match at LU alone. Only OM action has meaning if LU is connected.

SC (limit, value) stim, action

page 4.36

limit: bits 2-17 of setup word 35, indicating the limit to which the statistical counter (SCTR) is set. The programmer may use any symbol or a decimal number, ranging from 0 to 2^{16}-1.

value: bits 34-49 of setup word 34, contains the current value of the SCTR count. Any symbol or a decimal number may be used in the initial setup, ranging from 0 to 2^{16}-1.

stim: the programmer enters the code here for the stimulus that controls stepping of SCTR. Codes must be written as shown below.

Octal	Code	Description	Octal	Code	Description
0	NOP	No-Op	3	KG1.EG	KG = 1 AND End of Group
1	KB1	KB = 1	6	LG1.EG	LG = 1 AND EG
2	LB1	LB = 1	7	MG1.EG	MG = 1 AND EG
4	MB1	MB = 1	34	BYQ	Byte from Q
10	F1	F = 1	35	BYSA	Any byte into SACC
17	FZ	F = 0	36	TAO, TAF TAD	Address formed in TA
20	W	W Unit signal			
21	X	X Unit signal	27	W.Y	W AND Y
22	Y	Y Unit signal	16	KBZ.FZ	KB = 0 AND F = 0
23	Z	Z Unit signal	15	LBZ.FZ	LB = 0 AND F = 0
24	NX	NOT X	13	MBZ.FZ	MB = 0 AND F = 0
25	NY	NOT Y	5	LBZ	LB = 0
30	FL1P	Flag 1, P Stream	11	KB1.FZ	KB = 1 AND F = 0
31	FL1Q	Flag 1, Q Stream	12	LB1.FZ	LB = 1 AND F = 0
32	FL1R	Flag 1, R Stream	14	MB1.FZ	MB = 1 AND F = 0
26	XVY	X OR Y	37	ELTE	End of level in TE
33	BYP	Byte from P			

action: the programmer indicates whether the SCTR is to be incremented by +1 (octal code 0) or decremented by -1 (octal code 1). If omitted, a +1 is assumed.

Sample Entries:

```
SC (25), FL1P, +1
SC ( , 12), W.Y, -1
```

SA (mode, threshold, value), stim

mode: bits 56 and 57, setup word 34, used to indicate the operating mode for the statistical accumulator (SA). The entries must be coded as shown.

Octal	Code	Description
0	U	Unsigned, 1 byte
1	U16	Unsigned, 2 bytes
2	S	Signed
3	SR	Signed, negative reset

threshold: bits 32-55, setup word 33, the defining condition for the generation of the SAGETH (SA $\geq$ Threshold) stimulus. Any programmer symbol may be used.

value: bits 0-23, setup word 34, the actual SA register. Bit 23 is the sign for S and SR modes. Any numeric entry mode may be used.

stim: this field gives the stimulus that will step SA. If modes S or SR are used, the stimulus field is ignored, that is, SA cannot be stepped, except by an adjustment, if signed bytes are being accumulated. SA can only be stepped in the positive direction. The codes are:

Octal	Code	Description	Octal	Code	Description
0	NOP	No-Op	10	NZ	NOT Z
1	KBZ	KB = 0	11	EG	End of group
2	LBZ	LB = 0	12	LB1	LB = 1
3	MBZ	MB = 0	13	BYR	Byte into R
4	F1	F = 1	14	KGZ.EG	KG = 0 AND End of group
5	X	X unit signal	15	LGZ.EG	LG = 0 AND EG
6	Y	Y unit signal	16	MGZ.EG	MG = 0 AND EG
7	NW	NOT W	17	NWXY	NOT W AND NOT X AND NOT Y

Sample Entries:

SA (U, 25), X
SA (SR, 112, 0)

LU (modulus, group size), luop

page 4.11

modulus: bits 56-63, setup word 36, used in conjunction with the luop codes to specify the modulus desired (ranging from 1 to 256). Any programmer symbol may be used and it will be assembled as an eight-bit, unsigned binary number.

group size and luop: these fields are repeated here for the convenience of the programmer. If filled in, and the corresponding fields in a streaming instruction are left blank, then the programmer may, by using the coding SETUP (name) in the streaming instruction, request HAP to fill these fields from the named setup area. (See section entitled "Coding of SBBB Fields")

F (stim), limit, action

page 4.14

stim: there are nine conditions available to step the F counter. The programmer writes here the one or more he wishes to use, thus setting one or more bits from 19 to 27 in setup word 35. If any one of these conditions is met, the counter is stepped, but it is stepped only once (if at all) per byte output from the LU. The codes and conditions are:

Bit No.	Code	Description
19	KK	Prior P byte > prior Q byte AND present P byte > present Q byte
20	KL	Prior P byte > prior Q byte AND present P byte = present Q byte
21	KM	Prior P byte > Prior Q byte AND present P byte < present Q byte
22	LK	Prior P byte = prior Q byte AND present P byte > prior Q byte
23	LL	Prior P byte = prior Q byte AND present P byte = present Q byte
24	LM	Prior P byte = prior Q byte AND present P byte < present Q byte
25	MK	Prior P byte < prior Q byte AND present P byte > present Q byte
26	ML	Prior P byte < prior Q byte AND present P byte = present Q byte
27	MM	Prior P byte < prior Q byte AND present P byte < present Q byte

limit: the limit for the two-bit F counter is written as a decimal number; 1, 2, 3, or 4. If omitted, 1 is assumed. Bits 30 and 31 in setup word 35 are set by this coding. The F unit counts occurrences of a specified function of the KB, LB, and MB signals, compares the count against a specified limit, and issues a signal defined as follows (for a limit other than 1):

F = 0: Counter ≠ Limit
F = 1: Counter = Limit

action: If the F limit is one, this field may be used to modify the action of the F counter. Bits 28 and 29 of setup word 35 are used in the following coding:

Bit No.	Code	Description
- - -	NOP	No modification
29	SO	Set F = 1 on first pulse and then stay on one
28	I	Invert previous value of F for each pulse
28 and 29	ISO	Combination of 'I' and 'SO'

Sample Entries:

F (KK), 2
F (KK), 1, SO

TA (mode and cell size, parallel-serial, replace base address)

This field, like group size and luop, is repeated in the SETUP as a convenience to the programmer and is subject to the same rules. See section entitled "LU (modulus, group size), luop". Note: TAA may be used instead of TA. The demand parallel synchrony (DP) entry, usually found in the TA field, can only be entered by the SBBB or SQNL instruction. If desired, "TA (demand parallel synchrony)" can be entered by SBBB or SQNL and the remainder of the TA field can be entered from the setup area.

TBA (address, TBAHO, MDM)

page 4.26

address: bits 32-55, setup word 35. The programmer enters here any symbol he chooses for the memory address to be used as a table base address (TBA).

TBAHO: the programmer enters any additional TBA high-order bits called for by increased memory capacity. In bit positions 56 and 57 of setup word 35, the following actual binary addresses are coded: 00, 01, 10, 11.

MDM: bit 63, setup word 35. Either MDM is written or the entry is left blank. If 'MDM' is specified, each address formed references the first available memory frame. For correct lookup results, the table must be stored in duplicate. For details on how to set up these tables, see page 4.35 of the IBM 7950 Reference Manual. If 'MDM' is not specified, each address will reference a specific memory frame.

TAP (TPS, TPI, TPN, TPJ)

page 4.28

TPS: This value specifies the bit position in the assembled address to which the leftmost bit of the first stream byte from P is added. Any programmer symbol may be used.

TPI: Increment to be added to TPS before the next byte from P is positioned. Any programmer symbol may be used.

TPN: Total number of bytes from P used to form an address. Any programmer symbol may be used.

TPJ: Current position in TA register for the byte from P. Usually omitted. Any programmer symbol may be used.

Sample Entry:

TAP (0, +8, 3)

TAQ (TQS, TQI, TQN, TQJ), TAO, TAOHO, TPM

page 4.29

TQS: This value specifies the bit position in the assembled address to which the leftmost bit of the first stream byte from Q is added. Any programmer symbol may be used.

TQI: Increment to be added to TQS before the next byte from Q is positioned. Any programmer symbol may be used.

TQN: Total number of bytes from Q used to form an address. Any programmer symbol may be used.

TQJ: Current position in TA register for the byte from Q. Usually omitted. Any programmer symbol may be used.

TAO: The initial contents of the TAO (assembled address). Not usually set up. Any programmer symbol may be used.

TAOHO: Any additional TAO high-order bits called for by increased memory capacity. In bit position 56 and 57 of setup word 37, the following actual binary addresses are coded:

00, 01, 10, 11.

TPM: Current count of bytes read into TA. Not usually set up. Any programmer symbol may be used.

Sample Entries:

```
TAQ (8, , 1)
TAQ (, +4, 2)
```

TE (TEI, TEN, TEBM, TES, TEM)

page 4.46

TEI: Increment for byte read-out from the TE register. Any programmer symbol may be used.

TEN: Total number of bytes to be read out from TE. Any programmer symbol may be used.

TEBM: Byte mask to be used for bytes read from TE. A numeric entry mode must be used and it is assumed binary.

TES: Position in TE from which the first byte is to be read. Note that some TA modes cause insertion of an address in TES. Any programmer symbol may be used.

TEM: Current count of bytes read from TE. Not usually set up. Any programmer symbol may be used.

Sample Entries:

```
TE (-10, 3, 11111000, 20)
TE (4, 8, 11110000)
```

SSM (mask)

This mnemonic stands for stream stimulus mask. It is a 29-bit field (bits 0-28) in setup word 33. If a bit is set to one in SSM and the condition specified by this bit occurs while bit 42 in the interrupt mask register is set to one, then an interrupt will be taken. These SSM bits are to be coded as follows:

Bit	Code	Description
0	NW	W not match ($\overline{W}$)
1	NX	X not match ($\overline{X}$)
2	NY	Y not match ($\overline{Y}$)
3	NZ	Z not match ($\overline{Z}$)
4	W	W match
5	X	X match
6	Y	Y match
7	Z	Z match
8	FL1P	P indexing flag; FL field = 01
9	FL2P	P indexing flag; FL field = 10
10	FL3P	P indexing flag; FL field = 11
11	FL1Q	Q indexing flag; FL field = 01
12	FL2Q	Q indexing flag; FL field = 10
13	FL3Q	Q indexing flag; FL field = 11
14	FL1R	R indexing flag; FL field = 01
15	FL2R	R indexing flag; FL field = 10
16	FL3R	R indexing flag; FL field = 11
17	ELTE	End of indexing level, TE unit
18	SAGETH	SA crosses threshold in positive direction
19	F1	F = limit
20	SCLIM	SC = limit
21	KB1	K (Byte) = 1
22	LB1	L (Byte) = 1
23	MB1	M (Byte) = 1
24	KG1	K (Group) = 1
25	LG1	L (Group) = 1
26	MG1	M (Group) = 1
27	SABN	SA goes + to -
28	CCZ	CC bit = zero (P, Q, or R indexing).

Sample Entry:

SSM (FL1P, SCLIM)

DEBUG (mask)

This field, bits 60-63 of setup word 33, is used as an aid in debugging. The bits to be tested are:

ADJ	Any adjustment	Bit 61
FLAG	Any flag	Bit 62
BL	Any branch level	Bit 60
SCAN	Scan bit	Bit 63

If, for example, the 'FLAG' mask bit is activated, a 'DEBUG' interrupt (indicator bit 32) is caused each time a flag is encountered; therefore, the programmer must write a routine to handle such occurrences. Activating the 'SCAN' bit has the additional effect of freezing the status of all indicators so that they may be monitored. The indicators will be frozen for the duration of the interrupt routine provided the streaming unit is not used in this interval.

More than one mask stimulus may be activated if desired. Activation of the 'SCAN' bit alone has no meaning, since this bit cannot cause an interrupt. It should be activated in conjunction with one or more of the other three, if at all.

Sample Entry:

DEBUG (FLAG, SCAN)

PAD
QAD
RAD } (starting data address, initial index table address) page 3.2 - 3.4

starting data address: bit position 0-23 in setup word 39 (P) or 40 (Q) or 41 (R). This is the 24-bit address of the first byte to be read by P (or Q or R). Any programmer symbol may be used.

initial index table address: bit positions 32-49 in setup word 39 (P) or 40 (Q) or 41 (R). This is the address of the first word of the first level of indexing control for P (or Q or R). Any programmer symbol may be used.

Sample Entry:

PAD (JOE, PETE)

Summary of Setup Instruction

The previous sections have given all the information necessary for a full ten word setup, contained in the area reserved by the SETUP and SETEND mnemonics. These ten words must be inserted in the 7950 registers, words 32 to 41, before a streaming instruction is initiated.

However, it should be noted that setup words 42, 43, and 44 are available to the programmer. They are TEU, R1, and R2, respectively. TEU (the Table Extract register) should normally be cleared, but may be filled by the programmer if desired. R1 and R2, the left and right halves of the R register, may also be filled as desired.

INDEXING WORDS

One general indexing word format is used for all streaming instructions and for certain instructions one or two other word formats are required. To determine which requires the additional indexing formats, all streaming instructions are separated into two classes: collating and noncollating. The collating instructions include SMER, SSER, SSEL, and STIR. Noncollating instructions are SBBB, SILS, SNOP, and SQNL. Actually, the last instruction, SQNL, does not fit easily into either class, but for the purposes of this discussion it will be called a noncollating instruction.

Normal Index Format for All Stream Instructions

Each indexing level for each stream unit requires two full words of information. They must begin at a full-word boundary, and the address of the first word of the first level must be in the "initial index table address" of the PAD, QAD, or RAD setup word.

The letter P, Q, and R are used to indicate which stream unit the indexing is for, and the letter O indicates that the level is offset. Each level of indexing has the following format:

PX, QX, RX } (mode, EC/CC, FL, FF, SR, BL, Runout or R control, TRU or TRL, FS), increment, total count, byte mask/branch address, reset address, current count

or, for offset levels:

PXO, QXO, RXO } (same as above), increment offset, total count offset, byte mask/branch address, offset, RBL, reset address, current count

Coding of PX, QX, RX, and PXO, QXO, and RXO Fields

A complete explanation of each of the following fields may be found in Chapter 3 of the IBM 7950 Data Processing System Reference Manual. Where applicable, special references to the Reference Manual pages are noted.

Mode: (bit 62, higher level, first word) Refer to page 3.10.

Code	Binary
NES for nested	0 (NES if omitted)
SEQ for sequential	1

EC/CC chain: (bit 59, first word) Refer to pages 3.5 - 3.8.

Code	Binary
EC for end chain	0 (EC if omitted)
CC for continue chain	1

FL flags: (bits 57, 58, first word) Refer to page 3.34.

Code	Binary
Omit field for no flags	0
FL1	01
FL2	10
FL3	11

FF: (bit 60, first word) Refer to page 3.28.

Omit unless FF is intended. Code as FF for first to follow, so that the next indexing level acts as if it were a first level, Binary form is 1.

SR: (bit 61, first word) Refer to page 3.14, Suppress Reset.

Omit the field if normal reset is desired. (Binary 0) Code as SR if it is desired to suppress the reset at end-of-level (1).

BL: (bit 56, higher level, first word) Refer to page 3.28.

Branch level control, must be nested.
Omit if normal usage is intended. (0)
BL if this is to be a branch level (1)
(see byte mask/branch address field below)

Runout or R Control: (bits 30, 31, first word) Refer to page 3.33.

For P and Q indexing this is the runout control field, set as follows:

Code	Binary	Description
Omitted	00	No effect
R	01	Runout directly to R
M	10	Match only
RM	11	Both R and M

For R indexing, bit 30 is not used and bit 31 is the R control bit, set as follows:

Code	Binary	Description
Omitted	0	Data stored directly in the R register.
RC	1	The two words that will contain the R byte are obtained from memory to the R register before the byte enters R.

TRU or TRL: (bits 28, 29, first word) Refer to page 3.22.

Code	Binary	Description
Omitted	00	
TRU	01	Triangular indexing, with the count decremented by one at the end of the last iteration of the level.
TRL	11	Triangular indexing, with the count incremented by one at the end of the last iteration of the level.

FS: first-subsequent toggle (bit 55, first word) Refer to page 3.34.

Code	Binary	Description
F, (or omitted)	0	This bit is set to 1 by the hardware to indicate that the level has been used, and normally should not be set by the programmer.
S	1	

increment or increment offset: (bits 0-23 or 0-9 and sign, bit 24, first word).

Amount to be added to the address. Any programmer symbol may be used. The increment is 24 bits plus sign in normal indexing, and 10 bits plus sign for offset levels.

total count or total count offset: (bits 34-49 or 34-46, first word).

Total count for this level.. Any programmer symbol may be used. Count is an unsigned 16-bit field for normal levels, 13 for offset.

byte mask/branch address:

1. byte mask - (bits 24-31, second word, first level). The eight-bit byte mask appears only in a first level or a virtual first level. A numeric entry mode must be used, and binary is assumed.

2. branch address - (bits 26-31 and 50-61, second word, higher level). Any higher level may be a branch level if bit 56 of the first word is a 1. (See explanation of BL field, above) In this case no byte mask is provided, and the next pair of index words is fetched from the location given by the 18-bit branch address (formed by adjoining the two fields). The programmer may use a symbolic or absolute address, and HAP forms the address, splits it, and stores the bits in the proper field.

reset address: (bits 0-23, second word)

The location at which the initial address is stored, before the increment is added to form the current byte address. It should normally be omitted and HAP will insert zero. Any programmer symbol may be used.

current count: (bits 34-49, second word)

The machine uses this location to keep count of the number of times the increment is used. It should normally be omitted and zero is inserted by HAP. Any programmer symbol may be used.

offset: (bits 10-23 and sign, bit 26, first word) offset level only).

The amount of the offset. Any programmer symbol may be used. The amount of offset is 14 bits plus sign.

RBL: Residual byte length (bits 34-36, first word, offset level only).

If the final byte to be read by an offset level is not eight bits in length, this three-bit field gives its length. Any programmer symbol may be used.

Index Format for the Collating Instructions

In general, the collating stream instructions (SMER, SSER, SSEL, and STIR) handle two kinds of records, defined as follows:

1. Simple: Those records headed by their control field and in which the entire record may be considered as the control field.

2. Offset: Any record that does not meet both requirements for simple.

The SIM/OFF bit of the stream instruction (bit 14) specifies the kind of record to be handled and is not to be confused with the offset type of indexing (normal-offset bit, bit 25, first word of each indexing level).

Simple Records

For simple records the normal indexing parameters of the PX, QX, RX, and PXO, QXO, and RXO format are used to define the record, as follows:

1. Level 1, using PX, defines the record by putting increment = bits per byte, and total count = bytes per record. If the total number of bits per record is not a multiple of the number of bits per byte, so that the residual byte length is not zero, the indexing level itself must be offset. In this case use PXO instead of PX, and set offset = 0, and RBL to the proper value.

2. Level 2 defines an entire sequence by putting increment = bits per record, and total count = records per sequence.

Each level must be nested and have the CC bit set. Levels above the second may be used, if necessary. The IBM 7950 Reference Manual section on the collating instructions (page 8.7) should be consulted for details.

Offset Records

For offset records a special table of control field defining words is necessary. Each entry is a full word, must begin at a full-word boundary, and has the format of the first word of an offset first level of indexing.

The starting address of this special table is placed in the TBA field of the setup for the collating instruction and n_f; the total number of such words (the number of control subfields) is put in TPN. A zero in TPN is interpreted as 32 and n_f must not be greater than 32.

The first table word defines the record and has the following format:

TX (FL), byte size in increment, count of bytes in record

The second and any subsequent table words needed are used to define the control field of the record. They have the following format:

TXO (FL), byte size in increment, count of bytes in control field, offset of control field, RBL

Coding of TX and TXO Fields

FL: same as the FL field in PX or PXO.

byte size in increment:

number of bits per byte. Any programmer symbol may be used.

count of bytes in record:

number of bytes per record; applies to first index table word only. Any programmer symbol may be used.

count of bytes in control field:

number of bytes per control field; applies to all index table words except the first. Any programmer symbol may be used.

offset of control field:

number of bits from the start of the record; applies to all index table words except the first. Any programmer symbol may be used.

RBL:

residual byte length. If the final byte to be read is not eight bits in length, this three-bit field gives its length.

For one instruction, STIR, the control field definition words are decremented by one, so that word 2 can be used to define the 2-bit "action" field by putting byte size in increment = 2, count of bytes in control field = 1, RBL = 0, offset of action field = number of bits from the beginning of record. (See page 8.42 in the IBM 7950 Reference Manual.)

When the collating instructions are set for offset mode, the record defining words are fetched from this table, so that the first level indexing words for P, Q, and R are ignored by the machine. However, the address put into the "index table address" in SETUP must refer to these dummy first-level words, and not to the second level words that define a sequence.

Special TEY Format for SQNL

Sequential table lookup (SQNL) performs a series of lookups, each successive lookup after the first being performed in a table whose address depends on the result of the previous table. The SQNL table entry mode are specified by a TEY word with the following format:

TEY (TEC/TCC, TQ/TR), address, offset signed, TEN count, data

Coding of TEY Fields

TEC/TCC: (bit 33) If this is the last in the chain of looked-up words, write TEC, otherwise put TCC, which sets the bit to 1.

TQ/TR: (bit 32) If this is TQ (bit = 0), the offset is sent to the Q unit; if TR, to the R unit.

address: (bits 0-17) This address is added to the left-hand 18 bits of TE to form the address of the next word to be looked up. Any programmer symbol may be used.

offset signed: (bits 24-31, and sign, bit 23) This quantity is added to the P or Q effective address. It may be symbolic or absolute, and a one counts bits in the stream address.

TEN count: (bits 18-22) The number of bytes to be read out of TE. This corresponds to TEN in the normal TA set up. Any programmer symbol may be used.

data: (bits 34-63) The field for the looked-up data. An octal numeric entry mode must be used, and the (8) radix cannot be specified.

HMCP STATEMENTS

The programmer usually depends on the 7950 machine control program (HMCP) to control the input-output requirements of his problem program. The HMCP is an integral part of the 7950 operational system control program (HCP). It has three general functions:

1. Accessing or manipulating a logic file by READ and WRITE or SPACING commands.

2. Accessing logic records by PUT and TAKE (blocking and unblocking) commands.

3. Controlling and processing program interrupts.

For a full description of the function and operation of the HMCP, refer to the IBM 7950 HMCP Reference Manual. HMCP commands are described and itemized below.

HMCP MACRO STATEMENTS

The HMCP macro statements may be divided into the following six categories (refer to the IBM 7950 HMCP Reference Manual, chapter 6):

1. Record handling
2. Tape spacing
3. Adjust Symbol File Table
4. Control operations
5. Interrupt
6. Assignment

Record Handling Macro Statements

1. Basic read command

   MREAD, SYMBOL(I), CTLWRD(I)

   where SYMBOL indicates the file concerned, and CTLWRD indicates the location of the control word to be used.

2. Read with suppressed EOP indication

   MREAD(S), SYMBOL(I), CTLWRD(I)

3. Basic write command

   MWRITE, SYMBOL(I), CTLWRD(I)

4. Write with suppressed EOP indication

   MWRITE(S), SYMBOL(I), CTLWRD(I)

5. Basic read block command

   MTAKE, SYMBOL(I), REC AREA(I), n

   where REC AREA is the record area in core storage, and n is the number (0-15) of sequential records skipped before the first record is moved.

6. Selective read block command

   MSTAKE, SYMBOL(I), REC AREA(I), n

7. Basic write block command

   MPUT, SYMBOL(I), REC AREA(I)

8. Selective write block command

   MSPUT, SYMBOL(I), REC AREA(I), n

Tape Spacing Macro Statements

1. Space block forward

   MSPBLOK, SYMBOL(I)

2. Backspace block

   MBSBLOK, SYMBOL(I)

3. Space file forward

   MSPFILE, SYMBOL(I)

4. Backspace file

   MBSFILE, SYMBOL(I)

Adjust Symbol File Table Macro Statements

1. Set odd parity with no single error correction

   MODDNEC, SYMBOL(I)

2. Set odd parity with single error correction

   MODDECC, SYMBOL(I)

3. Set even parity with no single error correction

   MEVEN, SYMBOL(I)

4. Set high density

   MHD, SYMBOL(I)

5. Set low density

   MLD, SYMBOL(I)

6. Locate a tape unit

   MLOCATE, SYMBOL(I), UNIT#

   where UNIT# is the 729 tape unit referenced by the SYMBOL entry.

7. Turn indicator light off

   MTILF, SYMBOL(I)

Control Operations Macro Statements

1. Close the file

   MENDFILE, SYMBOL(I)
   or
   MWEF, SYMBOL(I)

2. End and release the file

   MTERMFILE, SYMBOL(I)

3. Rewind the file to the first block

   MREWIND, SYMBOL(I)

4. End the file and rewind

   MENDREW, SYMBOL(I)

5. Rewind and unload

   MREWUNL, SYMBOL(I)

6. Release the file

   MRELEASE, SYMBOL(I)

7. Copy control word

   MCOPYCW, SYMBOL(I), CW ADDR(I)

   where CW ADDR is the location where the new control word is to be entered.

8. Sound gong

   MGONG, SYMBOL(I)

Interrupt Macro Statements

1. Wait for an interrupt

   MWAIT, SYMBOL(I), BR ADDR(I)

   where BR ADDR is the address to which the program will return.

2. Pseudo branch disabled

   MPBD, BRANCH ADDRESS(I)

3. Pseudo branch enabled

   MPBE, BRANCH ADDRESS(I)

4. Interrupt table adjustment

   MINTTBL, INDICATOR + DATA ADDRESS, COUNT, CHAIN ADDRESS, MASK ON

   where INDICATOR is the mnemonic of the interrupt indicator.
   DATA ADDRESS is the symbolic or absolute address of the new interrupt table entry.
   COUNT is the number of consecutive words to be entered.
   CHAIN ADDRESS is a means for replacing non-consecutive words in the interrupt table, or words in different locations.
   MASK ON is used to set the interrupt indicator mask. A one sets them on; a zero leaves them unchanged.

5. Set mask register bit

MSETMASK, INDICATOR, COUNT, CHAIN ADDRESS, MASK ON OR OFF

where MASK ON OR OFF allows the specified mask bits to be set. A one in this field sets them on; a zero sets them off.

Assignment Macro Statement

1. Level control

MLEVEL, n

where n is the number of the level to be executed, from 1 to 15.

IOD AND IOX STATEMENTS

The programmer writes an input-output definition (IOD) statement for each logical input, output, or work file required for the execution of the program. The compiler interprets each IOD statement and forms a 15-word Symbol File Table (SFT) which is then used to control all operations relative to that symbolic file. Refer to the HMCP IOD statement chart for available IOD statements, their fields, and proper entries.

The programmer has the ability to specify his own routines for processing I-O interrupts after the standard HMCP processing routines are completed. The addresses of these special I-O interrupt routines appear as operands in an IOX statement which the programmer writes. Each file referenced by an IOD statement requires an IOX statement for special processing of the interrupts for that file. The format of the IOX statement is:

EXIT IOX, EOP, ERROR, EE, SIGNAL

Note that, unlike the IOD statement, the order of fields in the IOX statement is fixed. Commas must be used to separate the fields in both statements. (Refer to IBM 7950 HMCP Reference Manual.)

CONTROL WORD

A control word regulates the movement of data into core storage from external storage media. The programmer supplies a control word when using the basic read and write command; he need not do so when using the blocking routines. The control word has the following general form:

NAME CW(OP2), DWA, COUNT CHAIN

(Refer to the IBM 7950 HMCP Reference Manual.)

SYMBOL IOD (TYPE, DISP, USE, F / V / O, PET, CONTROL), EXIT

IQS									
READER									
PUNCH									
PRINTER									
CONSOLE									
DISK									

SYMBOL IOD (TYPE, DISP, USE, F / V / O, PET, LEVEL), EXIT

TRACTOR					1, 2, 3, etc.				
					to 15				
					(32-46)				

SYMBOL IOD (TYPE, DISP, USE, F / V / O, PET, CONTROL, UNIT, MODE, DENSITY), EXIT

TAPE									
	INPUT			D1	C1	U1	ODD	HIGH	
	WORK			D2	C2	U2	ECC	LOW	
	OUTPUT						EVEN		
					C7	U7			
				D15					
(28-31)	(60-63)	(49)	(47,48)	(51-54)	(18-20)	(21-23)	(25,26)	(24)	(14.32)

HMCP IOD Statement Chart

In a sense, certain commands or statements called 7950 control program (HCP) connect the object program to the 7950 operational system.

PARAMETER ENTRY STATEMENTS

At program pre-run time, the problem program is brought into memory and any parameters for that program are extracted from the parameter file, checked, converted, and inserted into machine memory by the problem program pre-execution supervisor. These activities are carried out according to and by means of a Symbol Translator Table (STT) which is part of the problem program and is constructed by HAP III from parameter location entry (PLE) and symbolic parameter entry (SPE) statements.

Parameter Location Entry (PLE) Statement

The format of this statement is:

```
PARAM  |                                        E
IDENT  | PLE(TYPE, MUST), LOCATION, M(#), F(FL, FFL)
```

PARAM IDENT: The symbolic name for the entry in the STT. It corresponds to the parameter identification to be specified on a parameter card. PARAM IDENT may be any combination of eight or fewer alphanumeric characters (0 to 9, A to Z) of which the first must be alphabetic.

PLE: The statement mnemonic.

TYPE: The type of parameter. Available types are:

VBU	VFL binary unsigned	INT	ALPHA integer
VBS	VFL binary signed	FLT	Floating point
VOU	VFL octal unsigned	HCS	HARVEST character
VOS	VFL octal signed	SYM	Symbolic
VDU	VFL decimal unsigned	FDD	File data description
VDS	VFL decimal signed		

MUST: This field indicates whether or not a particular parameter has to be supplied on parameter cards accompanying the request for the program in order for the program to function properly. MUST indicates that the parameter has to be supplied; a blank or null field indicates that the parameter need not be specified each time the program is run.

LOCATION: The 24-bit starting memory area into which the translated parameter value is to be inserted. Parameter values may be inserted anywhere in pre-run programmer available memory (Lower Bound - 64.0, Upper Bound 60,000.0). LOCATION may be written in absolute or symbolic.

#: Depending on TYPE, this field indicates the exact or maximum number of elements which must or may be inserted into memory. The elements are:

Numbers: VBU, VBS, VOU, VOS, VDU, VDS, INT, FLT.

Characters: HCS - No symbol separator for HCS1; comma (,) for numeric memory image representation of characters.

Symbols: SYM - Each symbol has an SPE entry; several SPE entries may point to the same PLE entry; all must have the same field length; symbols are separated by comma(,).

Fields: FDD - Fields are enclosed by parentheses; Primary and secondary fields are separated by slash (/); the PLE (#) refers to the total number of fields.

A blank or null PLE (#) implies the maximum of 2048 numbers, characters, symbols, or fields.

E
M This field indicates whether the (#) field is an EXACT or MAXIMUM number of numbers, characters, symbols, or fields. E indicates EXACT; M indicates MAXIMUM. A blank implies MAXIMUM. However, if the M field is null, the entire M (#) fields must be null.

F: This symbol serves to identify to HAP III that the following parenthetical entry contains field length information.

FL: The total field length of a parameter value element, including sign byte and fraction length, if any. It is applicable to VBU, VBS, VOU, VOS, VDU, VDS, SYM, and HCS. No blank is allowed in VBU, VBS, VOU, VOS, VDU, and VDS; a blank in SYM implies a field length of 24 bits; a blank in HCS implies a field length of eight bits. A blank entry is not applicable to INT, FLT, and FDD. The maximum VBU, VBS, VOU, VOS, VDU, VDS, and HCS field length is 64 bits and the maximum SYM field length is 24 bits.

FFL: This field indicates the fraction field length; that is, if the parameter number is fractional or mixed (integer plus fraction), it indicates the number of bits of the total element field length to be used in expressing the converted fraction. In conversion, the fraction is rounded to fit the fraction field length. For example, if the unsigned decimal number 5.5 is to be expressed in a total element field length of eight bits, with a fraction field length of four bits, the binary result is 01011000. A blank entry for fraction field length implies that parameter numbers will be integers (but not necessarily INT integers). The FFL is not applicable to INT, FLT, HCS, SYM, or FDD.

Symbolic Parameter Entry (SPE) Statement

The format of this statement is:

SYMBOLIC PARAMETER NAME	SPE, Parameter, PLE Name

Symbolic Parameter Name: Same as for PLE.

SPE: The statement mnemonic.

Parameter: The value of the symbolic parameter is expressed in absolute or symbolic. It will be assembled as a 24-bit address. The memory insertion field length is stated in the associated PLE entry; if fewer than 24 bits, the parameter field of the SPE entry will be right-truncated by the pre-execution supervisor.

PLE Name: The name of the PLE entry which contains memory insertion address, and so on. The name of the PLE entry will be assembled as an 18-bit address.

Problem Program Parameters

Expression of Parameters

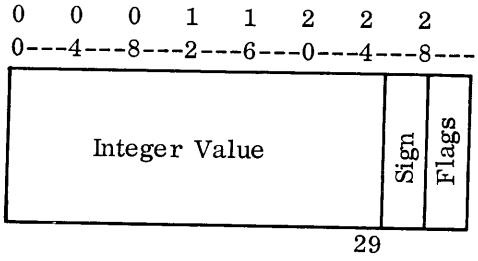
Numeric Parameters: VBU, VBS, VOU, VOS, VDU, VDS

These parameters can consist of one or more numbers, each expressed in the same form, all either signed or unsigned and with the same resulting field length and fraction length as specified by parameter location entry in the Symbol Translator Table of the problem program. If a signed number is specified by the PLE entry in the Symbol Translator Table, a 1-bit sign byte becomes part of the resulting binary number. A missing sign implies a positive value. If a fraction length is specified by the PLE entry in the Symbol Translator Table, a fraction point and a fraction value (even if the value is zero) must be given as part of the number.

All numbers are converted to binary and inserted into problem program memory according to a field length specified in the PLE entry. If any high-order significant bits are truncated by this process, the conversion is invalid and the job is rejected.

ALPHA Integers: INT

ALPHA integers are signed whole numbers with a fixed 32-bit field length. An ALPHA integer may be from 1-9 decimal digits whose absolute value is less than $2^{29} = 536,870,912$.



Machine Floating Point: FLT

A FLT floating point number is converted to a conventional 64-bit machine floating point number. It is expressed externally as:

1. Sign (may be omitted if positive)
2. Integer - with a value from 1 to 9
3. Decimal point
4. Fraction - from 1 to 15 decimal digits
5. "E" (exponent indicator)
6. Exponent sign (may be omitted if positive)
7. Exponent: from 1 to 3 decimal digits whose value is $\leq$ 308. If the exponent is zero, 5, 6, and 7 may be omitted.

Example: +2.35E - 15
2.35 (= +2.35E + Ø)
2.Ø (not 2)

Character Parameters: HCS

A character parameter must begin in the card column immediately following the equal sign which separates it from the parameter identification.

Characters may be expressed in HCS1 or in a decimal or octal memory image representation. If no indication is given in the parameter statement, HCS1 is assumed. Each HCS1 character is expressed in one card column; blank is a legitimate character. Characters in decimal form are expressed as 1 to 3 digits $\leq$ 255 which are converted to binary. Characters in octal form are expressed as 1 to 3 digits $\leq$ 377 which are converted to binary. Characters in octal or decimal are separated by commas (,); blanks are ignored.

To express a character parameter statement in a mixture of HCS1, decimal or octal characters, the following arrangement has been devised:

1. The special digraph "*2" in a character parameter statement means "decimally represented characters following." The digraph is not entered into memory.

2. The special digraph "*3" in a character parameter statement means "octally represented characters following." The digraph is not entered into memory.

3. The special digraph "*1" in a character parameter statement means "HCS1 characters following." The digraph is not entered into memory. Normally, *1 will be used to return to the HCS1 mode of expression after writing some or all of the characters in the preceding part of the statement in decimal and/or octal form.

   Example:
   ```
   EXAMPLE = NOW IS THE TIME*2,112,120,
   EXAMPLE + 1 = 6*1 FOR ALL GOOD  *
   ```

   The parameter in problem program memory appears as follows (with slashes used here for visual separation):

   ```
     Space       N         O         W       Space
   00000000/00011101/00011110/00100111/00000000/

      I          S      Space        T         H
   00011000/00100010/00000000/00100011/00010111/

      E      Space        T         I         M
   00010100/00000000/00100011/00011000/00011100/

      E        112       120         6      Space
   00010100/01110000/01111000/00000110/00000000/.

      F          O         R      Space        A
   00010101/00011110/00100001/00000000/00010000/

      L          L      Space        G         O
   00011011/00011011/00000000/00010110/00011110/

      O          D      Space
   00011110/00010011/00000000
   ```

   In expressing the value of a character parameter statement, the visual separator "**" may be used. It is not inserted into memory and, once recognized, is ignored by the system.

File Data Descriptors: FDD

The external form and restrictions on the file data descriptor parameter of a problem program is specified in Tractor File Data Descriptors, 7950 Operational System Control Program Manual. However, the same rules for parameter identification specification apply here as apply to other problem program parameters, e.g., the parameter identification may be any symbolic name.

A file data descriptor parameter is converted to the same internal form as specified in the above manual. As previously mentioned, the programmer can set a maximum or exact number of field descriptions that it will accept, but in no case will a data descriptor parameter be accepted which contains more than 49 field descriptions.

Checking Parameters by the Pre-Execution Supervisor

In general, the value statement of a JRL problem program parameter must be consistent with the TYPE specified in the Symbol Translator Table of the program. For example, if the TYPE is VDU, a plus, minus, or alphabetic (A-Z) character is invalid (excluding comment of course) and causes a job reject.

The number of elements (numbers, characters, symbols, or fields) must be equal to or less than the amount specified in the Symbol Translator Table of the program.

If a fraction length is specified in the Symbol Translator Table, a fraction point and a fraction value must be given for each VBU, VBS, VOU, VOS, VDU, VDS number.

If the programmer has specified, by means of the Symbol Translator Table, that a particular parameter must be supplied, failure to do so causes a job reject.

Parameter Checking by a Problem Program

After parameters have been translated, system checked, and inserted into machine memory at program pre-run time, control is given to the problem program to perform pre-run functions. One of these functions is the specialized checking of the parameters that have been supplied to the program.

To facilitate the problem program check of parameters, a Used bit is set in the Symbol Translator Table entry for each parameter which has been supplied and for each symbolic parameter value which has been selected. In addition, an "ERROR" bit is reserved in each Symbol Translator Table entry to enable the problem program to tag any parameters in error.

At the conclusion of a particular program pre-run, control is returned to the pre-execution supervisor with an indication of "okay" or "not okay". If not okay, the program-selected reason for reject and any Symbol Translator Table entries, which are error tagged, are listed and the job is rejected.

PROGRAM NAME AND LIMIT STATEMENTS

Every problem assembled for use within the 7950 operational system must contain, at the beginning of the program deck, a program name statement immediately followed by limit statements defining the memory limits of the required parts of such a program.

The statements, although a part of the source program, do not occupy any of problem program available memory in the resulting assembled program. The information contained in the statements is used by the HCP as an aid in handling an assembled program by (1) library maintenance, which prepares it for addition to the HCP library, (2) the program pre-execution supervisor, and (3) by the program execution supervisor.

Program Name Statement

The program name statement contains the name of the program and can be used to differentiate between various versions of a program.

Program Version Indication

A program name can consist of 1 to 15 alphanumeric characters (A-Z, 0-9), but to facilitate the naming of two or more versions of the "same" program (that is, an operational version and a concurrent improved version in the debug stage or Mod I and Mod II of a program), a 16th character may be appended to the name. In order to make this indication easier when a program name consists of fewer than 15 characters, this character is enclosed in parentheses following the program name.

Program Name Statement Format

The program name statement has the following format:

PGN, Program Name

where PGN is the statement mnemonic, and program name is the name of the program, from 1 to 15 alphanumeric characters. If there is a version indication, the format also contains one alphanumeric character between parentheses after the program name, as follows:

PGN, Program Name (a/n)

Program Limit Statements

Various parts of each program occupy different storage areas at different times. The storage areas may be either library tape space, memory space before and after prerun, or tape space on the cycle program file.

Limit Statement Formats

The formats for the limit statements are:

```
L LIB,      First Address, Last Address + 1
L RUN,      First Address, Last Address + 1, Entry Address
L SAVE,     First Address, Last Address + 1
L PRERUN,   First Address, Last Address + 1, Entry Address
L SFT,      First Address, Last Address + 1
L STT,      First Address, Last Address + 1
L IDD,      First Address, Last Address + 1
L ODD,      First Address, Last Address + 1
```

The meanings of the statement mnemonics are:

LIB	Library
RUN	Execution portion
SAVE	Uncleared memory space
PRERUN	Pre-execution housekeeping, etc.
SFT	Symbol file table
STT	Symbol translator table
IDD	Input data descriptor area
ODD	Output data descriptor area

The remaining parts of the statements are:

First Address: the starting address of the area; an 18-bit field, usually symbolic.

Last Address: the check address of the area; an 18-bit field, usually symbolic.

Entry Address: for run and pre-run only, the entry address into the program or routine. A 19-bit field, usually symbolic.

Program Name and Limit Table Coding

The memory words referred to in the following sections are the contents of the first two card images of the program binary output. (See section entitled "Binary Output File.") Refer to Figure 10 for a representation of the table in memory constructed from PGN and limit statements.

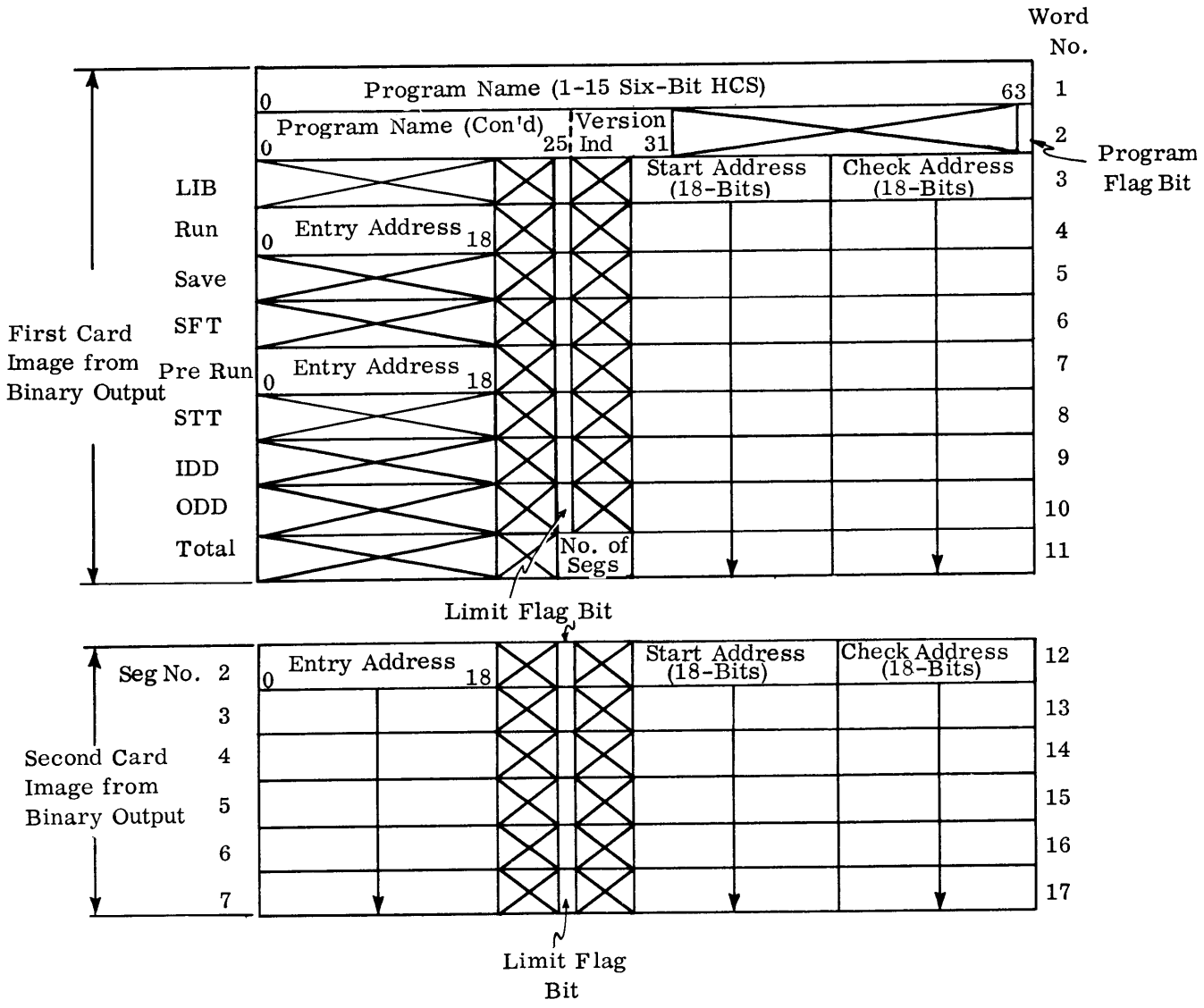


Figure 10. Program Name - Limit Table in Memory

Program Name (PGN)

The program name is converted to six-bit truncated 7950 character set (THS) coding and stored in word 1 (bits 0-63) and word 2 (bits 0-25) of the Program Name-Limit Table. If there are fewer than 15 characters in the name, the rightmost character positions are padded with THS blanks ($000\ 000_2$). If there is a version indication, the version indicator character is converted to THS and stored in word 2 (bits 26-31). If there is no version indication, a THS blank is stored in this character position. Word 2 (bits 32-63) are set to binary zeros.

Program LIMIT

The start address and check address of each area are converted to 18-bit addresses and placed in the count and refill fields, respectively, of the appropriate word. The entry addresses for RUN and PRERUN are converted to 19-bit addresses and placed in the value fields of the appropriate words.

Words 3 through 10 contain the limit information for LIB, RUN, SAVE, SFT, PRERUN, STT, IDD, and ODD (one word each) in that order.

Word 11 contains two addresses: the lowest memory location, and the word following the highest memory location of the assembled program memory area. For a program of several segments, words 12 through 17 contain the run limits and entries for the additional segments. For a single segment program, these entries are blank.

The limit statements, as a set, (but in any order) must immediately follow the PGN control statement card. Any limit statement contained in the source program which is not in the set immediately following the PGN card is treated as an invalid operation.

Error Conditions

PGN Statement Errors

If any of the following error conditions occur, the program listing is error flagged and the PGN error flag, in the heading of block one of the binary output (word 2, bit 63), is set to 1. The testing for and detection of errors occur in the order listed below; the processing of a PGN card in error is terminated after the specified action is taken, the PGN error flag is set to 1, and the listing is error flagged.

1. Any PGN statement contained in the source program not at the beginning of the program deck is ignored.

2. If the rest of the card is blank following the PGN, words 1 and 2 are set to binary zeros.

3. If the first nonblank character following PGN is not alphanumeric, words 1 and 2 are set to binary zeros.

4. If there are more than 15 alphanumeric characters following PGN, before the end of the card or before a nonalphanumeric character, word 1 (bits 0-63) and word 2 (bits 0-25) will contain the first 15 characters and word 2 (bits 25-63) will be set to binary zeros.

5. If there is a nonblank, nonalphanumeric character before the end of the card and it is neither a comma nor an open parenthesis, bits 26-63 of word 2 are set to binary zeros.

6. If the nonblank, nonalphanumeric character following the program name is an open parenthesis and

   a. if the next nonblank character is not alphanumeric, or if there are no more nonblank characters on the card, bits 26-63 of word 2 are set to binary zeros.

   b. if there is more than one character before a closed parenthesis, bits 32-63 of word 2 are set to binary zeros.

   c. if a nonblank character follows the closed parenthesis, and if it is not a comma, bits 32-63 of word 2 are set to binary zeros.

7. If the PGN control statement is omitted in the source program, words 1 and 2 are set to binary zeros.

LIMIT Statement Errors

If any of the following error conditions occur, the program listing is error flagged and the LIM error flag bit (index flag: bit 25) is set to 1 in the appropriate limit word. The testing for and detection of errors occur in the order listed below; the processing of an LIM card in error is terminated after the specified action is taken, the LIM error flag is set to one, and the listing is error flagged.

1. If any one of the complete set of eight limit statements (LIB, RUN, SAVE, PRERUN, SFT, STT, IDD, ODD) is repeated, the corresponding limit word is set to zero.

2. If any of the seven limits (LIB, RUN, PRERUN, SFT, STT, IDD, ODD) are omitted in the source program, the corresponding limit word is set to zero.

3. If any address cannot be evaluated, or if any required address field is omitted, the corresponding field in the appropriate word is set to binary zeros. Zero is not accepted as a valid address.

AVAILABLE DEBUG FACILITIES

The HAP III language contains a macro statement for the dumping of specified areas of memory in specified formats.

Also available to the HAP III programmer is a special purpose statement which enables the same program storage allocation to be retained regardless of whether the program is assembled in the debug mode or in the production mode. This statement, called a DNOP, allows debugging instructions to be performed when in the debug mode or to be "branched around" when in the production mode.

MHAPDUMP Macro Statement

The MHAPDUMP macro is the method by which a HAP program, during program execution in a Debug Cycle, can request dumps of specified areas of memory in specified formats. The form of the macro is as follows:

MHAPDUMP, Start Location, Length, Format

where "Start Location" is the 24-bit address of the memory area to be dumped, "Length" is the number of words and bits in the memory area to be dumped (from 0.01 to 4095.63 words), and "Format" is one of the options available in the Debug Dump Formatter program. Without explanation here, the available formats are as follows:

CHAR(HCS / BCD, BS) (ABS / TAG, N / C)

FLTPT(ABS / TAG, OCT / DEC, N / C)

VFL(BU, FL) (ABS / TAG, OCT / DEC / D##, N / C)

VFL(B, FL, BS) (ABS / TAG, OCT / DEC / D##, N / C)

VFL(DU / D, FL, BS) (ABS / TAG, N / C)

INSTR ABS
CTLWD (TAG, N / C)
NDXWD SYM

STR-INSTR (N)
STR-SETUP (C)
STR-INDEX(P / Q / R, N / C)

The calling sequence of this macro has the form:

```
        DNOP, tag
        CNOP
          SIC , #PGMIC
           BD, #HAPDP
           XW, Start Location, , Length [Format]
tag     SYN , #
```

DNOP Statement

The format for this statement is:

DNOP, TAG

where DNOP is the mnemonic and TAG is the symbolic tag provided by the programmer to identify another instruction in the program. When assembling in the debug mode, the DNOP is replaced by a NOP; and when assembling in the production mode, the DNOP is replaced by a B (branch) operation. In other words, the DNOP, TAG is changed to NOP, TAG in the debug mode, or to B, TAG in the production mode.

Assembling in the Debug and Production Mode

The 7950 assembly program must know whether it is assembling in the debug mode or in the production mode.

When assembling a program in the debug mode, HAP must:

1. Convert all DNOP, TAG statements to NOP, TAG instructions before assembling them.

2. Assemble all other HAP instructions conventionally.

3. Construct the HAP Debug Symbol Table and attach it to the binary output of the assembly.

When assembling a program in the production mode, HAP must:

1. Convert all DNOP, TAG statements to B, TAG instructions before assembling them.

2. Assemble all other HAP instructions conventionally.

3. Omit construction of the HAP Debug Symbol Table.

HCP MACRO STATEMENTS

The following macro statements will be discussed in this section:

MPPMSG
MENDPRUN
MCHANGE
MENDPGM

ENDPRUN and CHANGE are available for use at problem program Pre-Run time; ENDPGM and HAPDUMP are available for use at problem program exeuction time; and PPMSG is available at both Pre-Run and execution time.

MPPMSG Macro Statement

A program, in either its Pre-Run routine or during program execution, can cause messages to be written on the System Log by means of the PPMSG macro. Each program is limited to five messages, each with a maximum of 89 character, during Pre-Run and during program execution. The form of the macro is as follows:

```
MPPMSG, Location, Length
```

where "Location" is an 18-bit address and "Length" is the number of words (4-12) which the message occupies. Three unused words (to allow space for the message header supplied by the HCP) must precede the program's message; "Location" points to the beginning of this three word area and "Length" includes the three heading words. The program's message must consist of 6-bit Type-9 BCD characters. Any unused bits in the last word of the message can contain "blanks" (20_8) or binary zeros.

The form of the calling sequence resulting from the macro is as follows:

```
SIC, #AVEIC
BD, #PPMSG
BE, Location (#Length)
```

MENDPRUN Macro Statement

The ENDPRUN macro is the method by which a program indicates to the HCP the completion or termination of its Pre-Run routine. Its form is as follows:

```
MENDPRUN, GO
MENDPRUN, STOP
```

where "GO" indicates that the program parameters, file data descriptors, etc., are compatible (to the extent that they are checked by the Pre-Run routine) with each other and with the program's requirements and thus the processing of the program and its job can continue; and where "STOP" indicates that some error or inconsistency has been detected and thus the processing of the program and its job is to be stopped.

The form of the calling sequence resulting from the macro is as follows:

```
MENDPRUN, GO        SIC, #AVEIC
                    BD, #NDPRE
                    BE, (#1)

MENDPRUN, STOP      SIC, #AVEIC
                    BD, #NDPRE
                    BE, (#2)
```

MCHANGE Macro Statement

A program, while in its Pre-Run routine, can change the RUN Limits and Run Entry Address into the program by means of the CHANGE macro. (The RUN Limits define the area of memory which is to be saved at the end of Pre-Run and brought into memory at program execution time; in the interests of minimizing System overhead, Data Reservation and other program work areas should not be included in the area defined by the program RUN Limits.) The form of the macro is as follows:

```
MCHANGE, RUNSTART, New Value
MCHANGE, RUNLENG, New Value
MCHANGE, RUNENTRY, New Value
```

The "New Value" for RUNSTART is an 18-bit address, for RUNENTRY is a 19-bit address, and for RUNLENG is the length in words of the RUN area.

The form of the calling sequence resulting from the macro is as follows:

```
MCHANGE, RUNSTART, New Value      SIC, #AVEIC
                                  BD, #CHANG
                                  BE, New Value(#1)

MCHANGE, RUNLENG, New Value       SIC, #AVEIC
                                  BD, #CHANG
                                  BE, New Value(#2)

MCHANGE, RUNENTRY, New Value      SIC, #AVEIC
                                  BD, #CHANG
                                  BE, New Value(#3)
```

MENDPGM Macro Statement

The ENDPGM macro is the method by which a program indicates to the HCP the completion, termination, or request for the restart of the program from the beginning. The form of the macro is as follows:

```
MENDPGM, GO
MENDPGM, STOP
MENDPGM, REPEAT
```

where "GO" indicates that the program has been successfully completed and thus the processing of the job can continue; "STOP" indicates that the program has detected an uncorrectable error and thus the processing of the program and job is to be stopped; and "REPEAT" indicates that the program is to be restarted from the beginning (usually because of an uncorrectable "write" error or because of an uncorrectable "read" error of a block written by the program itself). A program can be restarted only once during a cycle. The mechanism causes the program to be restarted under the same conditions as when it was first started; that is, all HCP/MCP setup is repeated, including bringing the program in from the Cycle Program File again.

The form of the calling sequence resulting from the macro is as follows:

```
MENDPGM, GO        SIC, #AVEIC
                   BD, #NDRUN
                   BE, (#1)

MENDPGM, STOP      SIC, #AVEIC
                   BD, #NDRUN
                   BE, (#2)

MENDPGM, REPEAT    SIC, #AVEIC
                   BD, #NDRUN
                   BE, (#3)
```

ENTRY MODE

The characters appearing in the statement on a symbolic card are assumed to be either alphabetic or numeric. When the characters are numeric, HAP assumes they are written in the decimal radix. It is often more convenient, however, to write a numeric entry in another radix, such as octal or binary. By means of the HAP entry mode specification, the programmer can choose other radices, or even describe other properties of the source language to HAP.

Within the data description field of an arithmetic mode instruction, the use mode, field length, and byte size describe characteristics of the data that determine the conversion of the data and its later use at execution time. These characteristics are compiled along with the data. The entry mode, however, describes the form in which the data appears on the card and; therefore, it need not be compiled. The entry mode may be employed in one of three ways:

1. As a data definition statement entry mode.
2. As a field specification entry mode.
3. As a parenthetical integer entry mode.

Data Definition Statement Entry Mode

An entry mode may be used to specify the properties of all data in a DD or DDI statement. When used thus, it is enclosed in parentheses and appears immediately before the DD or DDI mnemonic in the operation field, as follows:

```
(EM) DD(dds), D, D', D''
```

The use of the data definition statement entry mode in entering alphabetic information is explained in the discussion of "Data Definition Statements".

Field Specification Entry Modes

Some entry modes may be used to specify the properties of all the fields in a statement or to specify the properties of one specific field within the statement. One such entry mode is the F mode; another is the radix specification mode.

F Entry Mode

The F entry mode may appear only in DD or DDI statements where an unnormalized floating point or binary use mode has been specified. If the F mode is employed as a statement entry mode within such a statement, it is written enclosed in parentheses immediately before the operation code:

```
(F6) DD (BU), 12.36
```

In this case, the entry mode F implies that the data that follow are written in the decimal radix, are to be converted to binary, and may contain a decimal fraction portion. The integer following F specifies the number of fractional binary bits that are desired to the right of the binary output following conversion. In the previous example, the fractional portion to the right of the binary point was limited to six bits in length. The converted six-bit fractional portion plus the integer portion is right justified in the appropriate field (in this case a 64-bit field, so leading zeros are supplied by HAP).

Conflicts between the field length specified and the F entry mode can arise where binary use mode has been written. If the converted data entry is too large to fit in the field requested, high-order bits are discarded. Whenever the converted entry is smaller than the field size specified, the problem is less crucial. High-order zeros are supplied.

In the case of unnormalized floating point DD statements, the rules governing the interpretation of the data and its conversion are identical to the handling of binary use mode statements except that the converted data entry is always inserted right justified in the standard fractional portion of the floating point format. The correct exponent, as determined by the location of the decimal point, is supplied by HAP.

Single Field Specification: Entry mode F may also be used as a field entry mode; that is, it may be used to specify the properties of one particular field within a DD or DDI statement without influencing the treatment of any other field in the same statement. In everyday programming situations, it is common to write DD statements with several data entries in each statement. In this situation, it is often desirable to use different entry modes for each field. Thus, the programmer may write:

```
DD(BU), (F6) 12.36, (F2) 187.5, (F8) 1005.679
```

Note that when the F entry mode is used as a field entry mode it is still enclosed in parentheses and appears first in the field. The meaning is the same as when it appears as a statement entry mode; however, that meaning applies only to the data entry in the field in which it appears.

Combined Statement and Field Entry Mode: Both statement entry modes and field entry modes may appear in the same statement. When there are contradictory properties by the statement and field entry modes, the field entry mode overrules for the case of the particular field on hand. Entry modes may not appear in a manner that cause parentheses within parentheses. In the following example:

```
(F6) DD (BU), 12.36, (F3) 166.3, 1776
```

the F6 entry mode rules for data entry fields 1 and 3, while the F3 specification temporarily overrules the statement entry mode for the second data entry only.

Radix Specifier

The radix specifier is another entry mode that may be used as a statement entry mode or a field entry mode. In any programmer symbolized field except the dds field, numeric integers and bit addresses may be written in any radix from 2 through 16. The radix is specified by enclosing the appropriate decimal integer in parentheses and placing it either before the operation code if statement entry mode action is desired, or at some appropriate place in the field to which it refers when it is employed as a field entry mode. (The radix specifier usually is the first item to appear in the field.)

If used as a statement entry mode, the radix specified applies to the entire statement unless individual fields contain their own radix specifier, in which case the field entry mode overrules the statement entry mode for that field only. If used as a field entry mode, the radix applies to the entire field unless it is reset before the end of the field is reached. If no radix is specified, the base 10 is assumed.

Examples:

1. (8) 573 - 34 + 50 (all numbers are in octal)

2. (2) 11011011100011.111100 (bit address written in binary)

3. (5) SAM - 342 (the symbol SAM is not affected by the radix, having been previously converted to binary. The integer 324 is written in the number system of the base 5.)

4. (8) 7436. (10) 60 + 9 (the full word portion of this bit address is written in octal, whereas the bit address portion and the integer 9 are written in decimal.)

5. (2) DD (B, 16, 8), (10) - 972, 111011110 (the first D field is written in decimal, the second one is binary.)

The entry mode radix specifies the radix in which an integer is written on the card but says nothing about the one to which it is converted. At the completion of every HAP statement, the radix is automatically reset to 10 and remains 10 for the following statement unless it is changed therein.

Note: An address expression written in hexadecimal must begin with a numeric character.

Parenthetical Integer Entry Mode

A final HAP entry mode used as a field entry mode is the parenthetical integer entry mode. This mode permits any integer or pattern of bits to be OR'ed in any bit positions of an instruction or pseudo operation that produces binary output. The general format for this entry mode is:

$$(.n)\ A_{n+1}$$

The symbol . n represents the bit address of the rightmost bit of the field into which the integer or bit pattern is to be entered. The integer A_{n+1} is formed as an unsigned field, n+1 bits in length (because of the custom of addressing bits starting with zero), and inserted into the leftmost n+1 bits of the addressed instruction or data entry field by means of a logical OR type operation. Logical OR is used so that the parenthetical entry may be combined with the existing contents of the particular field addressed or with other parenthetical entries.

Crossing Field Lines

The field selected by the parenthetical integer entry mode may cross field lines within a statement as determined by the format of the statement. However, the parenthetical entry mode is not permitted to cross statement lines. The specification of the rightmost boundary of the addressed field via . n must therefore be less than or equal to 31 in a half-word instruction, or 63 in a full-word instruction. Nevertheless, a maximum of 24 significant bits can be converted in a parenthetical entry. If necessary, zeros are added to expand to the desired length. When the bit address is specified as . n, the parenthetical integer expression is assigned a field length of n+1 and is evaluated modulo 2^{n+1}. All parenthetical fields are regarded as unsigned by HAP, so that a negative number is compiled as the complement, re 2^{n+1} of the magnitude of the number.

In the following instruction:

```
E + I, (. 8) 41
```

the integer 41 is converted to binary and OR'ed into the first nine bits of the E + I instruction. In the case of an instruction, the position of the entry is determined by counting bits from the beginning of the instruction, starting with bit zero, no matter which subfield of the instruction the integer entry may be written. Thus, in the VFL instruction format, the parenthetical integer entry may be appended to the address field, as in this illustration:

```
+ (BU), DATA (. 23) 4, 20
```

or it may follow the offset specification as in this illustration:

```
+ (BU), DATA, 20 (. 23) 4
```

In either use, the result would be the same. The rule is that the parenthetical entry must follow all other information in the field in which it does appear.

In the Address Field of a DD Statement

When a parenthetical integer entry mode appears in the address field of a DD statement, the n specification names the rightmost bit position relative to the beginning of the field at hand, not relative to the beginning of the DD statement. In other words, the parenthetical field position is determined by counting bits from the previous comma forward. In DD statements with multiple data entries, one or many parenthetical entries may be appended to each such field. Again in the case of DD only, the .n specification is restricted to be less than or equal to the field length as given in the data description of the particular statement.

Consecutive Parenthetical Integers

There is no limit to the number of consecutive parenthetical integer entries that may be written. Although one entry can conceivably be made to serve any single instruction or data field, is it often convenient to write several different integer entry specifications when one wishes to place numbers or patterns of bits in various positions within an instruction or data field.

Restrictions on Use in Statements

The parenthetical integer entry mode must appear in a statement that compiles space in storage. Therefore, this mode cannot be used in pseudo operations that give instructions to the compiler but result in no binary output (SLC, DDI, END, and so on). The parenthetical entry mode is a modification that may be appended to a D field or to any programmer symbolized field (or in place of such a field) which is not enclosed in parentheses. Thus, an index register specification in an address field may not contain this entry mode. One exception is permitted in DD statements only. Here, a parenthetical integer entry may be written in the data description field which is enclosed in parentheses. When so written as an appendage to the field length or byte size specification (but never as a modification of the use mode), the meaning is similar to that of a statement entry mode. That is, the parenthetical integer entry acts as if it had been appended to each of the D fields that follow in the DD statement. This unusual notation permits the insertion of a pattern of bits in every data entry in a multiple D field DD statement without the necessity of repeatedly writing the parenthetical entry in every field. In all other respects the parenthetical entry mode behaves exactly as it does when used as a field entry mode.

Contents of Parenthetical Expressions

Parenthetical expressions may contain anything that goes in a normal address field (except bit address), but may not contain other information such as alphabetic messages or real numbers which are permitted in DD or DDI statements. If a programmer symbol is used as a parenthetical integer entry, any data description associated with this symbol has no effect on this particular usage of the symbol. All numbers that appear in a parenthetical field are converted to binary, never to decimal or floating point.

Radix in Parenthetical Expressions

Radix designators are permitted in parenthetical OR fields, separated by commas from the bit address designation, and the two may be in any order. Thus, (.32,8) or (8, .32) signifies a parenthetical integer entry follows that is written in the octal radix on the card and is to be inserted in the field whose rightmost boundary is bit position 32.

Examples:

1. L(BU), INFO(.50,8) 17-JOE+(10)4203(4, .22) - 33303 (.60)1030

2. L(BU), INFO(7) (.20) 1265 (.30) (10) 138-(6) 43 (.10) 553

The first example is that of a VFL instruction with three consecutive parenthetical integer entry expressions appended to the address field. Note that arithmetic between integers and programmer symbols is permitted in forming the integer entry (17_8 - JOE+4203_{10}) and that when no radix is specified with a parenthetical entry, the current operative radix is continued. No attempt is made to reset to 10. The radix is assumed to be 10 if no radix has been previously specified in the field to which the parenthetical entry is appended, and if no radix has been specified as a statement entry mode.

The second example also illustrates three separate parenthetical integer entries in the address field. Note that the radix need not be specified within the same set of parentheses as the bit address specification for the integer entry. The radices which apply in the previous examples are:

Example	Number	Radix
1	17	8
1	JOE	Does not apply
1	4203	10
1	33303	4
1	1030	4
2	1265	7
2	138	10
2	43	6
2	553	6

All numbers that appear within parentheses are interpreted by HAP as decimal numbers.

Form of Numeric Data Entries

Any number written in a DD statement for conversion by HAP must be capable of being expressed in 64 binary bits. This means that the largest fixed point quantity that can be converted by HAP is equal to 2^{64} or approximately 20 decimal digits.

The floating point data format is a special case. Here the numeric entry is always converted to a 48-bit fraction and an 11-bit exponent. Therefore, only decimal quantities that lie within the range 10^{-308} to 10^{308} can be expressed in floating point format.

Numeric entries in DD statements may be written in a variety of formats. The two basic formats are the integer format, such as

982104

and the decimal fraction format, as in

-982104.2

These illustrations are written in the decimal radix. As previously described, an entry mode in the form of a radix specifier can be employed so that the programmer may write the data entry in one of several radices. If no sign is written, the number is assumed to be positive. If a BU or DU use mode is given and a sign is written, the sign is ignored by HAP.

Some special characters may be appended to data entries to provide further flexibility in notation. Three characters used for this purpose are the E suffix, the S (sign byte), and the X (exponent) entries.

E Suffix on Data Entries

It is often convenient to express a data entry as a number raised to some power of 10. The suffix letter E is used for this purpose, as in this example:

670.7E7

The meaning of E is to multiply the number that precedes it by 10 raised to the power expressed by the number that follows it. This number is always interpreted as a decimal integer. Thus, the above example is interpreted by HAP to mean 670.7×10^7. The presence of E automatically implies that the entry is written in the decimal radix. If a floating point use mode is specified, both the E specification and the position of the decimal point affect the computation of the exponent.

Sign Byte Entry

The letter S is used to enter information into the sign byte of signed data. Any integer that follows the S is interpreted by HAP as an octal integer. It is converted to binary and inserted by means of a logical OR into any previously calculated sign byte.

The sign byte generated depends on the byte size specified in the data description; its composition is illustrated by the following table:

Byte Size	Sign Byte	
1	S	
2	ST	
3	STU	
4	STUV	Z = zone bit
5	ZSTUV	S = sign bit
6	ZZSTUV	T]
7	ZZZSTUV	U } flag bits
8	ZZZZSTUV	V]

A byte size of 1 means that the sign byte is composed only of the sign bit; hence, an octal 1 is OR'ed into the sign bit position and creates a negative sign. If the specified byte size had been 4, the suffix S10 would be required to create a negative sign. Because the logical OR is used for the insertion, the sign byte sign position can be made negative by either a negative sign written with the numeric entry or by an S-type entry.

Exponent Entry

The suffix letter x may be used if the programmer wishes to create his own exponent for a floating point data entry. The number following the x is interpreted by HAP as a decimal integer and is converted to binary and compiled as the machine exponent of the floating point number to which it is attached. It overrules and replaces the exponent computed by HAP in the conversion process, which is completely eradicated by the replacement process.

DATA DEFINITION

Data are entered and defined in the program by means of certain statements called data definition statements. In the streaming mode, the setup instruction might be considered as a data definition statement. More generally (and in the arithmetic mode particularly), data are identified by some variation of the DD statement.

Data Definition (DD) Statement

The data definition statement provides the programmer with the basic method of entering and defining data. Its format is:

DD (dds), D, D', D''

Operation Field dds

The dds in the operation field is identical in form and content to that previously described in the section entitled "Data Description (dds)" and must be written in every DD statement. Thus, a data description may be attached to a symbol at the point of definition of the symbol, or it may be written as a part of an instruction referring to the symbol.

The data description is invoked whenever the symbol appearing in the name field of the data defining statement is used in the principal address field of a 7950 instruction. Therefore, a description set down at the point of definition of the symbol is overruled by a data description appearing in the 7950 instruction that references the symbol. Whenever overruling occurs, the entire data description specifed in the data defining statement is overruled. Overruling applies only to the instruction at hand. Thus, the instruction:

```
+ (BU), SOMEMORE
```

explicitly specifies data which are binary unsigned, field length 64, and byte size 8 (field length and byte size derived from null field conventions) to be compiled with this statement, regardless of the data description written in the statement where SOMEMORE was defined.

Address Fields

The address fields D, D', and so on, shown in the format above, represent separate numeric entries which the programmer wishes defined by HAP and converted to one of several 7950 internal forms. Several numeric entries may be written in one DD statement, separated by commas. D fields are signed fields if use mode B, D, N, or U is given, (see section entitled "Data Description (dds)"), if no sign is written, the positive sign is assumed. The fields are converted and allocated locations in storage sequentially as separate pieces of data, each having the data description specified. If too many D fields are written to fit on one card, continuation cards may be used to extend the statement field of the DD statement. If a symbol appears in the name field, it is attached only to the first piece of data compiled. When one wishes to name each of the entries, each must be presented in a separate DD statement with its own name.

Programmer symbols may not appear in the address field of a DD statement (VF or EXT may be used for this purpose), because certain letters have fixed meanings not subject to programmer control when they appear in a D field. Bit addresses, similarly, are not permitted in a D field. HAP always assumes that a numeric entry is written in the decimal radix; therefore, in a DD statement, the programmer need specify only the form to which he wishes his data entry converted. This is accomplished by the use mode in the operation field dds. All seven use modes (N, U, B, BU, D, DU, and P) are acceptable in a DD statement.

DD Statement Use Modes

Use Mode N: If use mode N is specified in a DD statement, as in

```
FLOATIT   DD(N), 1000
```

the data entry 1000 is converted to its normalized floating point equivalent (in HAP format), and placed in the full-word storage location henceforth symbolically referred to as FLOATIT. Note that DD conforms to the normal HAP rounding upward conventions.

Use Mode U: If use mode U had been specified in the dds of the above example, 1000 would have been converted to floating point in the same fashion, but not normalized.

Use Mode B: Use mode B converts the numeric entry from decimal to entry. The sign byte is the low-order byte of the converted number, equal in size to the byte size specified in the dds. The converted entry is placed in a field equal in length to the number of bits specified in the field length of the dds. If the field length specifies a field that is too small to contain the converted entry, the number is inserted in the field with the unit position aligned with the rightmost bit. Any high-order bits that do not fit in the field are discarded. No rounding up of the location counter takes place. The field length specified is added to the current setting of the location counter and the numeric entry is converted and inserted in this field.

Use Mode BU: Use mode BU is essentially the same as B except that the entry is considered to be unsigned, and no sign byte is created. The entry is converted and inserted in a field of the length specified in the dds. The byte size specified has no effect on the conversion since an unsigned operation has been called for and no sign byte is compiled.

Use Mode D: When use mode D is specified, a character-by-character type of conversion is called for, wherein each decimal digit in the numeric entry is converted to the four-bit binary coded decimal form. If the byte size specified in the dds is greater than 4, high-order zeros are added. If the byte size requested is less than 4, truncation occurs.

Use Mode DU: If use mode DU is specified, the conversion process is the same as for D. However, no sign byte is compiled, as none is required for the unsigned decimal mode.

To illustrate the differences between the binary and decimal modes, consider the following HAP statements and the resulting fields compiled in storage:

HAP Statements	Field Compiled
DD(BU, 4, 1) 1	0001
DD(BU, 12, 4), 12	000000010010

Use Mode P: The P mode references the dds in another statement where the use mode must be N, U, B, BU, D, or DU. Once the reference is made, the conversion performed by HAP proceeds according to the rules already outlined.

Entering Alphabetic Information

To enter alphabetic information by means of a DD statement, a special entry mode subfield must be written, enclosed in parentheses immediately before the operation code as shown in this format:

(EM) DD (dds), D, D', D'', ...

When an entry mode is used in connection with the data of a DD or DDI statement, it may in this instance (but only in this instance) designate that alphabetic information is to be compiled.

A second character, known as the end-of-statement character, is entered within parentheses following the EM. Its presence informs HAP to perform the desired conversion until this character is reached in the message. The end-of-statement character may be any acceptable card code character except), ', ?, and blank. This character is not compiled. (See the section entitled "Sample DD Statement.")

In the data description (dds), the use mode and field length do not affect the conversion of the alphabetic characters but are important later if another 7950 instruction refers to this alphabetic data and does not overrule the implied dds. The byte size affects the conversion of the characters.

There are five entry modes available for use in entering alphabetic or alphanumeric messages. Each entry mode serves two functions: it tells HAP that a message is being entered, and it describes the character set being used and prescribes the type of conversion required. When alphabetic information is specified, only one entry per DD statement is permitted.

Entry Mode A

Entry mode A signals the appearance of a message composed only of character drawn from the standard IBM BCD character set. If byte size 6 is specified in the data description field (dds), the characters are converted to the six-bit IBM tape BCD format. If byte size 8 is given, two leading zeros are added to each six-bit byte during the conversion process.

Entry Mode IQS

IQS tells HAP that the characters in the message are drawn from the 120-character set appropriate to the console typewriter or printer, and are to be converted to their eight-bit binary equivalents.

Note: Byte size 8 is normally designated. If byte size 6 is specified, the two leading bits, although valid, will be dropped.

Entry Mode CC

CC is the mnemonic for card code, and delineates that set of characters known as IBM Hollerith characters. These characters are converted to 12-bit bytes, where each byte reflects the multiple punch actually read in the appropriate card column. Byte size 12 must be specified.

Entry Mode Type 9 (T9)

Alphabetic and numeric information are identical in either T9 or A entry mode. The only difference between the two modes is in the representation of special characters, as follows:

HAP Meaning or Use	Type 9			A		
	Sym-bol	Octal	Punch	Sym-bol	BCD	Punch
Add	+	40	11	+	60	12
Subtract	-	60	12	-	40	11
Begin subfield	(	53	11 - 3 - 8	(	34	0 - 4 - 8
End subfield	)	54	11 - 4 - 8	)	74	12 - 4 - 8
System symbol	#	13	3 - 8	$	53	11 - 3 - 8
Divide	/	21	0 - 1	/	21	0 - 1
Multiply	*	32	0 - 2 - 8	*	54	11 - 4 - 8
Bit address indicator	.	73	12 - 3 - 8	.	73	12 - 3 - 8
Instruction separator	?	74	12 - 4 - 8	;	52	11 - 0
Subfield separator	,	34	0 - 4 - 8	,	33	0 - 3 - 8
Begin comment	'	33	0 - 3 - 8	@	14	4 - 8
(unassigned)	=	14	4 - 8	=	13	3 - 8

Note the different characters used in each entry mode to indicate a system symbol, an instruction separator and a begin comment mark. A byte size of either 6 or 8 may be specified. If a byte size of 8 is called for, two leading zeros are added to each six-bit byte during the conversion process.

Entry Mode HCS

The 7950 character set (HCS) is the name given to a set of alphabetic, numeric, and special character symbols uniquely defined for the 7950 system. Each symbol may be represented by a six-bit binary code. The following table shows the relationship between HCS and type 9 entry mode, using octal notation to represent the six-bit bytes. Note that either a six-bit or an eight-bit byte may be specified for HCS. If an eight-bit byte is called for, two leading zeros are added to each six-bit byte during the conversion process.

Symbol	HCS	T9	Symbol	HCS	T9	Symbol	HCS	T9
0	60	12	A	20	61	b	0	20
1	61	1	B	21	62	.	6	73
2	62	2	C	22	63	-	7	60
3	63	3	D	23	64	(	10	53
4	64	4	E	24	65	)	11	54
5	65	5	F	25	66	+	12	40
6	66	6	G	26	67	/	13	21
7	67	7	H	27	70	'	14	33
8	70	10	I	30	71	,	15	34
9	71	11	J	31	41	#	16	13
			K	32	42	=	17	14
			L	33	43	*	76	32
			M	34	44	?	77	74
			N	35	45			
			O	36	46			
			P	37	47			
			Q	40	50			
			R	41	51			
			S	42	22			
			T	43	23			
			U	44	24			
			V	45	25			
			W	46	26			
			X	47	27			
			Y	50	30			
			Z	51	31			

Sample DD Statement

If the following statement were encountered by HAP:

(AQ) DD (BU, 60, 6), STOP HERE Q

the compiler interprets the A entry mode to mean that the alphabetic data entry on this card is composed of BCD characters which are to be converted to IBM tape BCD format. The second character in the entry mode subfield is the end-of-statement character. Blanks that appear in the message are retained, converted, and stored correctly. A blank between the comma marking the end of the operation field and the first alphabetic character is converted.

If the byte size specified is greater than 6, leading zeros are supplied by HAP. If the byte size is less than 6, leading bits are truncated.

If IQS entry mode is specified, the conversion process is similar except that the characters are converted to the eight-bit inquiry station code. When the byte size specified is greater than 8, leading zeros are inserted; when the byte size is less than 8, leading bits are truncated. Note that in a DD statement, the byte size of converted characters may range from 1 through 12, as specified in the dds. However, the byte size in a 7950 statement may range from 1 through 8 because the byte size field is restricted to three bits in length. Therefore, byte size is considered modulo 8 by HAP.

Data Definition Immediate (DDI) Statement

The DDI statement performs the same basic function as DD; that is, it provides the mechanism for entering and converting data. The data, however, if the DDI statement is used, are specifically intended to be used as an immediate operand in an immediate instruction.

More specifically, DDI is the only convenient method for compiling decimal information in the address field of an immediate instruction. Data in an immediate address are always converted to binary, never to decimal, regardless of the use mode specified in the data description.

The format of the DDI statement is:

```
ANYNAME     DDI (dds), D
```

The data entry in a DDI statement is converted according to the use mode specified in the dds. The resulting field cannot exceed 24 bits in length; if it does, high-order bits are lost. This field is inserted, right-justified, in a 24-bit field in the HAP symbol table. The field length specified in the dds is ignored at this point. When a 7950 immediate operation references this data through the symbol that appears as the name of the DDI statement, a field of the length specified in the implied dds or the overruling dds (if one is given) is extracted from the right end of the appropriate symbol table entry and is inserted left-justified in the instruction address field.

Sample DDI Statement

In the following example

```
IDATA   DDI(DU, 4, 4), 4
        LI, IDATA
```

the converted field created in the symbol table is

000000000000000000000100

while the 24-bit address field of the load immediate instruction will be compiled as:

010000000000000000000000

If the load immediate instruction had contained an overruling dds, such as:

```
LI (DU, 8, 4), IDATA
```

the address field, after compilation would contain the following:

```
000001000000000000000000
```

If a signed use mode is given, such as:

```
LI (D, 8, 4), --IDATA
```

then the symbol table entry would be

```
000000000000000000000100
```

and the instruction address field would be compiled as follows:

```
010010000000000000000000
```

Restrictions on DDI Statement

If the length of the converted data entry is greater than the field length specified, high-order bits (from the left) are truncated before insertion into the address field. Only the decimal or binary use modes, and the P mode, are allowed in a DDI statement. The floating point use modes are not appropriate in immediate addressing, and hence are not acceptable. Any entry mode that is allowed in a DD statement, including the alphabetic entry mode, is accepted in a DDI statement as well. If the field length is null in the specified dds, 24 is assumed by HAP.

Thus, DDI is purely definitive in character; it compiles no space or binary output in storage. Data are converted and entered only in the symbol table. Data so defined that are referenced symbolically by a 7950 instruction are also inserted in the address field of that instruction.

Synonym (SYN) Statement

The synonym statement (SYN) provides another mechanism for defining a symbol in terms of an integer, a bit address, or another symbol. The other symbol eventually is defined as an integer or bit address. The format of the SYN statement is:

```
ANYNAME    SYN (dds),  Y
```

When one writes

```
A    SYN, 6
```

the meaning is that whenever A is written in the program, the effect is the same as if 6 had been written. The meaning of SYN is always one of exact substitution.

SYN Address and (dds)

The entry in the address field of the SYN statement is converted to binary and inserted right-justified in a 24-bit field in the symbol table. In this process, SYN is similar to DDI, in that neither statement compiles space in storage.

SYN statements may have their own data description; any dds that appears in a SYN is attached to the symbol in the name field, but in no way affects the conversion of the entry in the address field. When a 7950 instruction references the symbolic name of a SYN statement, the dds attached to that symbol is invoked as in DD. If no dds is given in a SYN statement, none is attached to the symbolic name. Then a dds must be explicitly written in an instruction that references a symbol defined by such a SYN statement.

Index Registers with SYN

Index registers may be attached to the expression appearing in the address field of a SYN statement. Thus, in the SYN statement:

```
A    SYN, B (#3)
```

the index register specification is attached to the address expression, so that, the instruction

```
+ (N), A
```

is synonymous with

```
+ (N), B (#3)
```

If an index register is specified in the principal address field of the instruction proper, it overrules any other index register specification for that instruction only. In the above example, if the normalized floating point add instruction had been written:

```
+ (N), A (#6)
```

this would be synonymous with

```
+ (N), B (#6)
```

Circular Definition

A circular definition may arise through the use of a sequence of SYN cards, as:

```
A    SYN, B
B    SYN, C
C    SYN, A
```

All symbols in such a sequence are assigned a value of 0 by HAP.

Data Reservation (DR) Statement

This statement has the following format:

```
A     DR (dds), N
```

The DR reserves storage space for data. It causes N fields of the kind described in the data description to be reserved; that is, the location counter is skipped forward a quantity in bits equal to the product of N and the field length specified in the dds. If N is negative, no reservation is made and the location counter is not adjusted. Any symbol A appearing in the name field of a DR statement is attached to the first field reserved, as is the data description. Thereafter, whenever A appears as the principal address in an instruction, this dds is invoked in the same manner as with DD and DDI statements. Thus:

```
SAVE     DR (BU, 8, 8), 10
```

reserves ten 8-bit fields by skipping the location counter forward 80 bits. The dds (BU, 8, 8) is attached to SAVE and SAVE is attached to the first eight-bit field reserved.

When either one of the floating point use modes is given in the data description of the DR statement, the floating point data block being reserved is forced to begin at a full word address. HAP automatically rounds the location counter to the next full-word address to accomplish this; thereby insuring that each floating point data word begins at a full-word address.

If no dds is given, the symbol appearing in the name field is assigned the normalized floating point use mode by HAP.

Data Reservation and Set to Zero (DRZ) Statement

By appending a Z to the DR mnemonic, a slightly different statement, data reservation and set to zero, is formed. The format is:

```
A     DRZ (dds), N
```

This operation is identical to DR, but it performs the additional function of setting all reserved fields to zero. DR reserves fields but makes no attempt to clear them to zero.

Extract (EXT) Statement

The general form of this statement is:

```
A EXT, (I, J, COUNT)  STATEMENT
```

It differs in form from the other data definition statements but it is included in this class because it does manipulate or define data.

The extract statement as given in its form above has the following meaning:

First, compile STATEMENT as if it were any acceptable 7950 instruction or pseudo operation that produces binary output. Then extract from this statement the subfield that is equal in length to the number of bits specified by COUNT. The subfield begins at bit I of that statement and ends at bit J. The extracted subfield is then actually compiled in the position in the code where the EXT occurs.

Any symbol A appearing in the name field is assigned a data description BU, a field length equal to COUNT (or J - I + 1), and a byte size of 8, and is attached to the subfield compiled.

Any two of the three parameters, I, J, and COUNT, are sufficient to adequately describe the subfield to be extracted. All three can be written if the programmer so desires, but if fewer than three are written, the usual right to left dropout rules, as in the dds. Therefore, the permissible alternatives are:

(I, J, COUNT)
(I, J)
(I, , COUNT)
(, J, COUNT)

The terms I, J, and COUNT may contain any number of symbolic integers. A bit address is improper, however, and is treated as a 24-bit binary integer.

If EXT is used to specify the extraction of anything beyond the range of the single statement that follows it, up to 64 zeros are added.

Example:

EXT (18, 47) +(B, 18, 7), 73.16

First the full word instruction + (B, 18, 7), 73.16 is formed. Then bits 18 through 47 (the first bit in the instruction is numbered zero, according to custom) are extracted and placed in the program being compiled. The dds (BU, 30, 8) is formed. The location counter is advanced 30 bits.

RULES FOR DD STATEMENTS

The acceptable formats for entering data can be classified according to the use mode written in the data description field of the DD statement. Normally, an element listed in the general format may be omitted if it is not needed to specify the data.

The data entries in a DD statement are restricted to real numbers. Bit addresses would have no meaning here and are not allowed. In addition, programmer symbols are not permitted. As a special case, where normalized floating point has been specified in the data description, the system symbols for certain mathematical constants are accepted.

Arithmetic expressions (that is, combination of two or more numbers by means of addition, subtraction, multiplication, and division to form one data entry) are permitted in all DD statements regardless of the use mode specified. Such arithmetic is specified using standard symbols; that is, addition (+), subtraction (-), multiplication (*), and division (/). HAP performs the arithmetic and compiles a single constant. Multiplications are performed first, proceeding from left to right, and then the additions and subtractions are completed.

HAP does the necessary bookkeeping to insure that floating point data entries are always compiled at addressable full words; the location counter is rounded up to the nearest full word, if necessary, in order to accomplish this.

Rules are given below for DD statements used to enter data in floating point and VFL form.

Normalized Floating Point

Format:

Name: DD(N), ±xx...xx.x...xxE±Sn

The number is converted to a normalized floating binary number consisting of an 11-bit signed exponent, a 48-bit fraction, and a 4-bit sign byte. If no sign byte has been entered by means of an S, the sign preceding the number is used with the flag bit set to zero. If a different binary exponent is desired, it can be entered following an x, as follows:

Format:

Name: DD(N), ±xx...xx.x...xxE±yyySnXzzz

Examples:

1. DD(N), 54.73 E 4

   54.73×10^4 is converted to floating binary. The sign bit is zero (= plus), and the flag bits are zero (that is, entire sign byte is zero).

2. DD(N), -54.73 E 4

   or DD(N), 54.73 E 4 S 10

   In this case the sign bit is set to one (negative) and the flag bits are zero.

3. DD(N), -54.73 E 4 S 5

   The sign bit is one, since the number is negative, and flag bits T and V are one. U is zero.

4. DD(N), 1, 3E-5, -45.7, 12 S 17

   This example illustrates the multiple entry feature. This single DD statement compiles four 64-bit floating point words and advances the location counter accordingly.

5. DD(N), 1/3, 472*351, 4-7*5/21 S 4

   Note: Sign byte entered in last D field.

6. DD(N), 27.9/31.4/12/14 E 5, 4+3*7/5*6

   The number produced in the first case is:

$$\frac{27.9}{31.4 \times 12 \times 14 \times 10^5}$$

   in the second: $4 + \frac{3 \times 7 \times 6}{5}$

7. As an additional convenience, certain symbols are defined by which constants involving irrational numbers can be entered:

#PI	π
#E	e
#M	$\log_{10} e$
#N	$\log_e 2$
#INF	∞ (infinity)

   Thus, one can enter a number such as $4\pi \times 10^{-7}$ by writing:

   DD(N), 4 * #PI, 1E - 7.

Unnormalized Floating Point

Format:

Name (Fn)DD(U), ±xx...x.x...xE±yyySn X±n

or

DD(U), (Fn) ±xx...xx.x...xE±yyySnX±n, (Fn) ±xx...etc.

The number is converted to binary with the correct number of binary fractional places as specified by the (Fn) entry mode; and a correct exponent is computed and entered. This exponent is overruled and replaced by that following the X if X is used (necessary only if the programmer desires an incorrect exponent). The entry mode (Fn) can come before the DD, in which case it applies to all D fields of the statement, or it may form the first element of a D field, in which case it overrules the one given before the DD. Either the X or the S or both may be omitted or their order may be interchanged. Omitting S has the same effect here as in the normalized case. Omitting X allows the correct exponent to remain as computed. Leaving out the sign, decimal point, or E is permitted as in normalized numbers.

Examples:

1. DD(U), (F21) -343.7, (F10) 432

   Two numbers are compiled. In the first, 343 is converted as an integer and .7 is converted to a 21-bit fraction. They are joined and placed in the rightmost bits of the fraction portion of the floating point word, and the correct exponent (in this case 27) and sign are supplied. In the second D field, 432 is converted to a binary integer. Because ten fractional bits are specified, but no decimal fraction is written, the ten rightmost bits of the fraction field are set to zero and the number is entered with its rightmost bit in position 50.

2. (F15)DD(U), 767.52, 767.52X-12 S11

   The (F15) applies to both D fields. In the second, the computed exponent is overruled by the specified one and the number is made negative by means of the specified sign byte.

3. (F15)DD(U), 767.52, (F20) 767.52 S11 X-12, 398

   This example is identical to example 2 except that in the second field, the operation entry mode (F15) is overruled by a field entry mode (F20), and the order of S and X is interchanged, which makes no difference. (F15) still applies to 398, however.

Missing Entry Mode

If the entry mode is omitted, two cases are possible:

1. If the number entered is an integer, (F0) is understood.

2. If the number entered is a decimal fraction, it is converted to an unnormalized floating point number.

Missing Entry Mode Examples

1. DD(U), 17, 17X-35

   In the first case, 17 is converted to binary and placed in the fraction with its rightmost bit in position 60 and an exponent of 48 supplied. In the second field, the same thing is done except that the exponent is set to -35.

2. DD(U), 17.5

   In this example 17.5 is converted to normalized floating binary and stored as such. However, instructions whose normalization bits depend on the symbol in the name field of this pseudo operation have them set to unnormalized.

Note:

17 E 5	is an integer and will be recognized as such.
17 E-5	is a decimal fraction and will be normalized.
17.5 E 5	is an integer but will be treated as a fraction and normalized. Thus, a normalized integer can be assigned use mode "unnormalized."

An integer greater than 2^{48} is stored as a normalized number.

Binary Signed VFL

Formats:

(Fn)DD(B, FL, BS), ±xx...x.x...xE±yy Sn
DD(B, FL, BS), (Fn) ±xx...x.x...xE±yy Sn
(R)DD(B, FL, BS), ±xx...xx Sn
DD(B, FL, BS), (R) ±xx...xx Sn

A data definition of binary signed data may have either (Fn) or (R) entry modes, but not both at the same time. (Fn) implies that the data following it are written in a decimal radix, whereas (R) implies that the number following it is an integer. An integer subject to a radix entry mode must be written without the aid of E because E is not defined for a radix other than 10. A decimal fraction must have a controlling (Fn) entry mode. There is no easy way to convert to a fixed point number without specifying the binary scaling. In the data description either the field length or byte size or both may be omitted. The implied field length in this case is 64, the implied byte size is 1. The sign byte need not be specified unless the programmer desires to have flag or zone bits different from zero. Note that the sign bit position changes for a byte size less than 4. To make a number negative, specify the sign byte as:

BS = 1, S1
BS = 2, S2
BS = 3, S4
BS = 4, S10

If a number has no entry mode at all, it must be a decimal integer, but may in this case be written with the aid of the E notation.

Examples:

1. (F7)DD(B, , 4), .005E3S13, -17, 143.2S11, (8) 77760, 777

   Implied field length is 64. Octal specification in the fourth D field overrules (F7) written before DD, but (F7) still applies to 777.

2. (2)DD(B, 16, 8) 110101S377, (10) - 972, 11101110S201

   Binary entry, overruled in only the second D field.

3. (F12) DD(B, 24), 1.324E3, -72, 1E-4, 3.4E-4S1

   Implied byte size is 1.

4. DD(B), 1489, -1272, 1491, (F13) -972.16, 13948S1, 12E5

   Decimal integers, except where a field entry mode is written.

Binary Unsigned VFL

Formats:

(Fn)DD(BU, FL, BS), xx...x.x...xE±yy
DD(BU, FL, BS), (Fn) xx...x.x...xE±yy
(R)DD(BU, FL, BS), xx...xx
DD(BU, FL, BS), (R) xx...xx
(Az)DD(BU, FL, BS), alphabetic information to "z"
(IQSz)DD(BU, FL, BS), alphabetic information to "z"
(Pz)DD(BU, FL, BS), alphabetic information to "z"
(CCz)DD(BU, FL, BS), alphabetic information to "z"

Numeric entry is exactly the same as in binary signed data except that no sign byte is formed, and if the byte size is left out of the dds, it is set to 8. Any sign or sign byte (with S) written with mode BU is ignored. The alphabetic modes are permitted here; they are explained in the section entitled "Entry Mode." Note that the alphabetic entry mode must precede the DD, that there can be only one D field per statement, and that if the field length is omitted, it is set equal to 64. If the byte size is omitted in entry mode CC, BS = 12 is implied.

Examples:

1. (F13)DD(BU, 30), 17.2, 183, (8) 70707

2. (A*)DD(BU, 48, 6), GLORIOUS FRIDAY, THE 13TH. *

The mode and field length have no effect on the conversion and storage; they are used in compiling instructions that refer to the name of this statement. Field length 48 indicates that the programmer wants to process these characters in groups of eight.

3. (IQSS)DD(BU, 32, 8) EIGHT BIT BYTES

Decimal Signed VFL

Formats:

(R)DD(D, FL, BS), ±xx...xxx Sn
DD(D, FL, BS), ± (R) xx...xx Sn
DD(D, FL, BS), ±xx...xxEyy Sn
(Fn) has no meaning for mode = D or DU.

The two decimal modes in DD and DDI statements represent the only cases in which HAP converts numbers to an internal decimal radix. The radix entry mode indicates the radix in which the numbers are written on the card. Thus, it is possible to write an integer in binary or octal and have it converted to decimal for machine use. If no entry mode is given, decimal to decimal is implied. The E notation can be used to multiply an integer by positive powers of 10. If either the field length or byte size is omitted, the implied values are FL = 64, and BS = 4.

Examples:

1. DD(D), -9534812, + 173E5, 18E10S13

Field length = 64; byte size = 4. A 4-bit sign byte is formed. Decimal-to-decimal conversion.

2. (2)DD(D, 20), 111010001101S7

Byte size = 4. Binary-to-decimal conversion.

3. DD(D, , 8), 432E3

Field length = 64. Decimal-to-decimal conversion.

Four binary zeros are inserted in the zone positions of each byte.

Decimal Unsigned VFL

Formats:

(R)DD(DU, FL, BS), xx...xx
DD(DU, FL, BS), (R) xx...xx
DD(DU, FL, BS), xx...xxxEyyy
(Az)DD(DU, FL, BS), alphabetic information to "z"
(IQSz)DD(DU, FL, BS), alphabetic information to "z"

The numeric conversion is just as in decimal signed mode except for the omission of the sign byte. Alphabetic conversion is exactly as in the binary unsigned mode, except that instructions referring to these data are compiled as decimal operations. For alphabetic entry, implied field length is equal to byte size.

Examples:

1. DD(DU), 8430051, (8) 77241, 82E10

   Field length = 64; byte size = 4.
   An octal-to-decimal conversion is inserted between two decimal-to-decimal conversions.

2. (IQS3)DD(DU, , 8), SEEK AND SIGNAL 3

   Field length = 8.

Summary of Rules for DD Statements

Entry Mode	Appropriate Use Modes
Fn	U, B, BU
R	B, BU, D, DU, N, U
A	BU, DU, U
IQS	BU, DU, U
CC	BU, DU, U
T9	BU, DU, U
HCS	BU, DU, U

Note: Use mode N should have no entry mode.

Special Field Entry	Appropriate Use Modes
S	N, U, B, D
X	N, U

The floating decimal notation, using E to designate multiplication by powers of 10, is appropriate to all modes.

If the field length is omitted from the dds, it is assigned a value of 64. The maximum permissible field length for a DD statement is 64.

The parenthetical integer entry mode is appropriate in any DD statement, no matter what use mode has been written. The following examples illustrate the use of general parenthetical integer entry with DD:

1. DD(N), 572(.59)1, 347.89E12(.63, 2)1011

   In the second case the sign byte is specified by means of (.n) entry.

2. DD(B), (F9) -35.7(.24)SAM + 4

   The address SAM + 4 is placed in the first part of the 64-bit field, followed by the converted number -35.7.

3. (8)DD(BU), 4762(.10)707(10, .20)34

   707 is written in octal, 34 in decimal.

4. DD(BU, 12(.2)7, 8), 787, 788

   All numerals are in decimal. Binary 111 is OR'ed into the three high-order bits of each 12-bit data field created.

CONTROL STATEMENTS

Control statements are used in HAP to provide the programmer with a simple method of controlling the program. Control statements are a kind of pseudo operation, so named because they are not machine instructions; they do not exist in machine circuitry, but they resemble machine instructions in format. The general format is:

```
NAME    POP (dds),  A (I)
```

The pseudo operation or control statement code field appears first in the statement. The operation mnemonic is symbolized by POP. A dds, if appropriate, appears as a subfield of the operation field and is enclosed in parentheses.

The address field may contain a wide variety of entries that are not always addresses in the strict sense of the word. Some addresses can include index register specifications.

INPUT-OUTPUT CONTROL STATEMENTS

The programmer usually avails himself of the facilities of the 7950 machine control program (HMCP) in order to satisfy the input-output requirements of his problem program at execution time. In order to control the output of the assembly program, he uses the HAP printing and punching control statements. These statements, described previously, are listed below.

Printing Control Statements

PRNS	Print Single-Spaced
PRND	Print Double-Spaced
NOPRNT	No Printing
SPNUS	Suppress Printing of Unused Symbols
PRNID (or PRINID)	Print Identification

Punching Control Statements

PUNFUL	Punch Full Cards
PUNNOR	Punch Normally
PUNORG	Punch Origin
NOPUN	No Punch
PUNSYM	Punch Cards for Symbols
PUNALL	Punch All
PUNID	Punch Identification

DATA DEFINING STATEMENTS

Several statements define or specify data to HAP. Most of these statements are employed in the arithmetic mode (see the section entitled "Data Definition"); the major exception being the setup and TEY statements in the streaming mode of operation (see sections entitled "Setup" and "Special TEY Format for SQNL").

Arithmetic Mode Data Definition

DD	Data Definition
DDI	Data Definition Immediate
DR	Data Reservation
DRZ	Data Reservation and Set to Zero
SYN	Synonym
EXT	Extract

Streaming Mode Data Definition

SETUP	Stream Setup Statement
TEY	SQNL Data Word

MISCELLANEOUS CONTROL STATEMENTS

Miscellaneous HAP control statements may be grouped as follows:

1. Location counter control
2. Symbol change
3. Error message control
4. No operation
5. End and terminate loading

Location Counter Control

Set Location Counter (SLC)

In normal assembly operations, cards are read in sequence. The number of bits needed for each instruction or piece of data is added to a location counter maintained by HAP to aid in the assigning of addresses to instructions and data. A principle of rounding upward is followed, guaranteeing that an instruction, VF, CF, or RF, begins exactly at a half-word address, and that index words, control words, and floating point data begin only at full-word addresses.

The SLC control statement provides for setting the assembly location counter to any values at any point in the assembly process, thus giving the programmer complete control over the location of his code. The format of the statement is:

```
A    SLC, Y
```

The SLC statement resets the location counter to the value of the bit address Y. The next instruction is compiled at this address, subject only to rounding upward conventions. Following an SLC, the location counter is advanced once more in normal fashion until another SLC statement resets it.

Y must be a bit address expression, either numeric or symbolic, whose value is positive. If an integer is specified in this field, it is treated as an integer in a 24-bit address field; that is, it is interpreted as specifying a number of bits. Subject to this interpretation, it is evaluated correctly, but an error indication is given on the listing.

Any symbol in the name field is effectively ignored, but is entered in the symbol table.

If the following statement appeared in a program:

```
SLC,  100.32
```

it would cause the HAP location counter to be reset to 100.32. If the instruction following the SLC were a VFL instruction, it would be compiled at 100.32. If it were a floating point data word, it would be compiled at 101.0.

Set Location Counter Relative (SLCR)

The format of this control statement is:

```
SLCR,  Y
```

SLCR resets the HAP location counter to the address Y in much the same fashion as SLC. However, SLCR also stops binary punching, so that locations of statements following SLCR are assigned relative to the location specified in SLCR but none of the statements appears in the binary output. This is the same as if all symbols in the name field of the statements that follow the SLCR were defined by SYN statements, and is more convenient for the programmer.

In the most common usage,

```
SLCR,  0
```

resets the location counter to 0, and all symbols following are assigned locations relative to 0. One application of SLCR might occur in the definition of table formats. In the following sequence

```
          SLCR,  0
PRICE     DD (BU, 24, 8), 0
QUANTITY  DD (BU, 6, 8), 0
ONHAND    DD (BU, 10, 8), 0
```

the evaluation of the symbols are:

```
PRICE = 0.0
QUANTITY = 0.24
ONHAND = 0.30
```

If the table in question begins at location 2000.0, and this address is placed in the value field of index register 6, the relative addressing of items in the table can be accomplished as follows:

```
L, QUANTITY(#6)
*, PRICE (#6)
```

These instructions would be compiled by HAP as:

```
L, .24(#6)
*, 0.0(#6)
```

One advantage of this method is the ease with which the dds of one of the statements can be changed without requiring compensating changes in any of the others. The definitions can also be reordered with no other changes in the statements required and all address assignments are recomputed by HAP relative to the SLCR address.

SLCR is allowed to set the location counter to an address below 41_8 without causing an error message to be printed. This is not the case if SLC had been used. The locations subsequently assigned often are below 41_8 as well, but they are usually indexed to produce addresses above the first 32 storage locations. In many ways SLCR is equivalent to SLC followed by a NOPUN. An SLC must be issued to restore binary punching of the output deck.

Symbol Change

Tail and Untail

Difficulty with multiply-defined symbols can arise when two programs, written by different people at different locations, are assembled together. By appending a unique programmer symbol as a tail to every symbol in his program, a programmer can be assured that each of his symbols are uniquely defined, regardless of what other programs are assembled with his program.

The TAIL control statement has the following format:

```
TAIL,  ANYSYMBOL
```

It appends the symbol that appears in its address field as a tail to every symbol in the statements that follow the tail statement until another tail statement, or an untail statement, is given. A tail symbol can be any permitted programmer symbol; it may be composed of as many as 128 alphanumeric characters, the first of which must be alphabetic.

When tailing is used, certain restrictions apply to the basic programmer symbols to be tailed. The last two characters of the basic symbol are used for a special character that indicates tailing is being used and for a character to represent the tail symbol. Therefore, a programmer symbol of more than 126 characters cannot be tailed. As many as 256 distinct tail symbols can be used within any one program.

Tailing Levels

HAP permits up to ten levels of tailing; that is, as many as ten different tail symbols may be appended to each programmer symbol within a block of code. When only one level of tailing is used, two characters must be subtracted from the maximum size of a programmer symbol to be tailed. In multi-level tailing, an additional character must be subtracted for each additional level of tailing. If n = the number of levels of tailing, n + 1 characters must be subtracted from the maximum size programmer symbol. Thus, if six levels of tailing are to be used, the maximum size programmer symbol that may appear in that tailed block is 121 characters in length; when ten-level tailing is specified, the longest programmer symbol may be 117 characters in length.

To facilitate multi-level tailing, a subfield is added to the basic tail statement format, as follows:

```
TAIL,  (n)DOG
```

where n refers to the level of tailing to which the given tail symbol is to be assigned. If the level is not specified, the first level is assumed.

The tail continues to be added to every programmer symbol encountered at the level specified until an untail statement or a tail statement that specifies the same level is found. An untail statement untails all levels up to and including the level specified in the address field. For example, the statement

```
TAIL,  (6)DOG
```

specifies DOG as a sixth level tail, and

```
UNTAIL,  (6)
```

untails the first six levels. Note that

```
UNTAIL,  (1)
```

is equivalent to

```
TAIL,  (1)
```

but

```
UNTAIL, (2)
```

is not equivalent to

```
TAIL, (2)
```

because UNTAIL, (2) untails the first and second level, while TAIL, (2) tails the second level only with a blank, or effectively untails it. Clearly then, if it is desired to untail one level when multi-level tailing is being done, the best method is a tail statement that specifies the level but has a blank tail symbol field, as in

```
TAIL, (6)
```

The normal reference may be made from one symbol to another within the same tailed block. However, when reference is made from a block tailed by DOG, for example, to a possible multi-defined symbol BOB in another block tailed by CAT, the statement should read

```
+ (N), BOB#CAT
```

If the symbol BOB has been tailed at several levels, they must all be mentioned:

```
+ (N), BOB#CAT#TAYLE
```

If reference is made from a tailed block to a possible multi-defined symbol in an untailed block, only the # is required after the symbol, as in

```
+ (N), BOB#
```

The # alone (actually # followed by a blank) tells HAP that the reference is to the untailed symbol BOB, not the BOB defined in the tailed block.

Error Message Control

Suppress Error Messages (SEM)

This statement has the following format:

```
SEM, 1, 2, 3, ...
```

The code SEM, followed by a blank address field, causes all error messages detected in statements that follow the SEM statement to be suppressed on the output listing. Any particular message or group of messages may be suppressed by writing the numbers identifying the messages in the address field, separated by commas. Thus,

```
SEM, 8, 2
```

suppresses the printing of error message 2 and 8 only.

Resume Error Messages (REM)

The REM statement has the following format:

```
REM, 1, 2, 3, ...
```

An REM restores normal error message printing on the listing after an SEM has been used. The ability to specify individual messages or all messages at once is also available with REM. Thus, following the statement

```
SEM, 9, 16, 18
```

the control statement

```
REM, 16
```

restores normal error printing to message 16, while messages 9 and 18 remain suppressed.

No Operation

Conditional No Operation

The format of this statement is:

```
A    CNOP
```

The CNOP is used to insure that the instruction or data immediately following the CNOP is assigned a full-word address by HAP.

When a CNOP is encountered, the location counter is immediately rounded up to the nearest half-word address if it is not already at a half-word address. Then HAP examines the location counter. If it now stands at a full-word address, the CNOP is ignored. If, however, the location counter is set to a half-word address, the instruction NOP is compiled. This has the effect of advancing the location counter 32 bits or one half-word to the next full-word address.

Any symbol A appearing in the name field is assigned a full-word address when the CNOP is ignored; or a half-word address when a NOP is compiled.

In the following example:

```
                SLC, 100.32
CASE1           CNOP
                L(BU, 24, 8), ASSIST
CASE2           CNOP
                + (N), FLOATINGONE
```

the appearance of the first CNOP causes a standard NOP instruction to be compiled at location 100.32. The load instruction is compiled at 101.0. The symbol CASE1 is assigned the value 100.32. When the second CNOP is encountered, the location counter

stands at 102.0. The CNOP is then ignored, the floating point add instruction is compiled at storage location 102.0, and the programmer symbol CASE2 is assigned the value 102.0. Thus, CASE2 becomes the symbolic location of the floating point instruction.

End and Terminate Loading

END Statement

The format of this statement is:

```
END,  Y
```

A card containing the code END signals the end of an assembly. Therefore, an END card must appear as the last card of every symbolic program deck. When HAP recognizes an END card, it punches out a branch card with an address Y. This branch card is included as the last card of the binary output deck produced by HAP. When the binary deck is loaded, the branch card causes control to be transferred to the instruction located at Y.

Since the instruction located at Y is the first instruction in the program to be executed, Y usually specifies the location of the first instruction in a program. This use of END is illustrated in the following example:

```
        SLC, 1050.
BEGIN   L (BU, 24), DATA
        (intervening code)
        END, BEGIN
```

The END statement does not have to address the first instruction in a program. The programmer is free to select any instruction he wishes to be executed first. If the END address is a programmer symbol, HAP correctly substitutes the binary bit address equivalent. If the address is a numeric entry, it follows the rules of any 24-bit address field. An integer written in this field is interpreted as a number of bits. A bit address compiles correctly, so care must be taken to include the period unless an integer expression is specifically intended.

Any symbol appearing in the name field is effectively ignored by HAP, but the symbol is placed in the symbol table.

Terminate Loading and Branch (TLB)

This statement has the following format:

TLB, Y

The statement TLB is similar to the END statement with one major distinction: it does not stop the assembly process. Therefore, TLB may be assembled at any point in the symbolic deck where a transition card is desired. The branch card thus produced interrupts the loader when encountered in a binary deck and transfers control to the instruction at location Y. The remainder of the program must be loaded under program control.

LIBRARY SUBROUTINES

The HAP III Processor provides facilities for the inclusion of subroutines which exist in the HOPS subroutine library. Library routines are stored in symbolic form and are compiled with the problem program requesting them. Since the subroutines are in symbolic form, the problem programmer must take care to insure that the symbols in his program differ from those in the subroutines used. The programmer must also have some means for inserting library subroutines into his program and for adding additional subroutines to the library when necessary. Finally, once the subroutine is inserted in the problem program, the programmer must be able to execute it as often as required. Various macro instructions and instruction sequences are available to perform these duties.

MLIB MACRO INSTRUCTION

MLIB, Id

The MLIB macro causes the symbolic coded subroutine or subprogram (known on the library file by the identifier Id) to be inserted in the source program in place of the MLIB statement. The identifier, Id, must be a valid HAP III tag of eight alphameric characters or less. Id is the name associated with the subroutine on the library file and, in general, is the tag defining the entry point to the subroutine.

The problem programmer is responsible for tailing the symbols in the library subroutine to make certain that they are not identical to those in his own symbolic program. The macros MTAIL and MUNTAIL are employed for this purpose.

MTAIL MACRO INSTRUCTION

MTAIL, symbol

The MTAIL macro generates the HAP instruction

TAIL, (n) symbol

where the tail level "n" is determined by the MTAIL macro and is equivalent to the depth of MTAIL nesting, and where symbol is any valid HAP programmer symbol of eight or less characters. The MTAIL macro will control nested blocks of code up to a maximum depth of ten levels. Note: An MTAIL macro may not be used within the range of a HAP TAIL instruction and vice versa. The HAP TAIL instruction controls a block code completely exclusive of MTAIL control.

MUNTAIL MACRO INSTRUCTION

MUNTAIL

The MUNTAIL macro closes off an MTAIL'd block of code. It produces the HAP instruction

TAIL, (n)

where "n" is the highest tailing level currently in force. Every tailed region of a program initiated by an MTAIL must be closed by MUNTAIL.

CALLING A SUBROUTINE FROM THE LIBRARY

The problem programmer defines the relative location of the library subroutine within his program by writing the MLIB statement at the point of request. Several MLIB statements with the same identifier cause the subroutine to be provided at each point of request.

To insure that the symbols of the subroutine and those of the symbolic program are unique, the following standard sequence should be employed:

```
Id      SYN, #
      MTAIL, symbol
       MLIB, Id
    MUNTAIL,
```

where Id is the name of the subroutine, and symbol is any valid HAP programmer symbol up to eight characters in length. The above sequence holds for the requesting of lower level subroutines from within a subroutine. Nesting of subroutines is permitted up to a maximum of ten levels.

EXECUTING A SUBROUTINE

When calling for the execution of a library subroutine the following standard instruction sequence must be employed:

```
SIC, #15
  B, Id
XW, Number of output parameters, Number of input parameters, Total number of parameters
VF, Current location of the first parameter
XW, Base address of first parameter, Length of first parameter, Basic element type
VF, Current location of second parameter
XW, Base address of second parameter, Length of second parameter, Basic element type
```

where the basic element type is coded according to the following table:

Code	Type
1	F
2	INT
3	BIN1
4	BIN2
5	BASIC
6	HCS1
7	HCS2

PREPARATION OF LIBRARY SUBROUTINES

The following general rules apply when preparing library subroutines:

1. The first physical statement of the subroutine must be the first executable statement.

2. Parameters are secured from the address information provided in the calling sequence relative to #15.

3. Return from the subroutine should be made via #15 to the location following the index word of the last parameter.

HAP III MNEMONICS

Assigned HAP III mnemonics, including both operation codes and system symbols, are listed on the following pages. The numbers in the Footnote column designate notes that follow the listing. These footnotes, in general, identify a particular class of operations that may be expanded in a standard way to produce other operations. Where footnotes specify how particular modified operation mnemonics may be constructed, these mnemonics do not appear explicitly in the listings.

The following abbreviations, used in the Type column, identify the symbolic instruction type.

V	VFL
F	Floating Point
$	System Symbol (in Type 9, substitute #)
I	Index
C	Count and Branch
M	Branches and Miscellaneous
B	Branch on Bit
T	Transmits
E	I-O Select or Control Word

Type	Mnemonic	Footnote	Name	Word No.	Bit Address
$	AD	2	Address Invalid	11	16
$	AE	2	Accumulator Equal	11	61
$	AH	2	Accumulator High	11	62
$	AL	2	Accumulator Low	11	60
$	AOC	1	All Ones Count	7	44-50
$	BC	1	Boundary Control	3	57
$	BTR	2	Binary Transit	11	39
$	CA	1	Channel Address	5	12-18
$	CBJ	2	Channel Busy Reject	11	8
$	CNSL	1	Console		
$	CPUS	1	CPU Signal	11	5
$	CPU	2	Other CPU	6	0-18
$	CS	2	Channel Signal	11	13
$	DF	2	Data Fetch	11	20
$	DISK	1	Disk		
$	DS	2	Data Store	11	19
$	DTR	2	Decimal Transit	11	40
$	E	12	e		
$	EE	2	End Exception	11	11
$	EK	2	Exchange Control Check	11	3
$	EKJ	2	Exchange Check Reject	11	6
$	EOP	2	End of Operation	11	12
$	EPGK	2	Exchange Program Check	11	9
$	EXE	2	Execute Exception	11	18
$	FT	1	Factor	14	0-63
$	IA	1	Interruption Address	2	0-17
$	IF	2	Instruction Fetch	11	21
$	IK	2	Instruction Check	11	1
$	IJ	2	Instruction Reject	11	2
$	IND	1	Indicators	11	0-63
$	IQS	1	Inquiry Station		
$	IR	2	Imaginary Root	11	25
$	IT	1	Interval Timer	1	0-18
$	L	1	Left Half of Accumulator	8	0-63
$	LB	1	Lower Boundary	3	32-49
$	LC	2	Lost Carry	11	22
$	LS	2	Lost Significance	11	26
$	LZC	1	Left Zeroes Count	7	17-23
$	M	12	$\text{Log}_{10}e$		
$	MASK	1	Mask	12	21-49
$	MB	1	Maintenance Bits	4	0-63
$	MK	2	Machine Check	11	0
$	MOP	2	To-Memory Operation	11	55
$	N	12	$\text{Log}_e 2$		
$	NM	2	Noisy Mode	11	63
$	OP	2	Operation Invalid	11	15
$	PCH	1	Punch		
$	PF	2	Partial Field	11	23
$	PGO...PG 6	2	Program Indicators	11	41-47
$	PI	12	π		
$	PRT	1	Printer		
$	PSH	2	Preparatory Shift Greater Than 48	11	27
$	R	1	Right Half of Accumulator	9	0-63
$	RDR	1	Reader		
$	RGZ	2	Result Greater Than Zero	11	58
$	RLZ	2	Result Less Than Zero	11	58
$	RM	1	Remainder	13	0-63
$	RN	2	Result Negative	11	59
$	RU	2	Remainder Underflow	11	34
$	RZ	2	Result Zero	11	57
$	SB	1	Sign Byte	10	0-7
$	TC	1	Time Clock	1	28-63
$	TCI...TCK		Tape Chanels 1...K		
$	TF	2	T Flag	11	35
$	TR	1	Transit	15	0-63
$	TS	2	Time Signal	11	4
$	TX	1	Tape X (X is a numerical designation)		
$	UB	1	Upper Boundary	3	0-17
$	UF	2	U Flag	11	36
$	UK	2	Unit Check	11	10
$	UNRJ	2	Unit Not Ready Reject	11	7
$	USA	2	Unended Sequence of Addresses	11	17
$	VF	2	V Flag	11	37
$	X0	1	Index Zero	16	0-63
$	X1	1	Index One	17	0-63
$	X2	1	Index Two	18	0-63
$	X3	1	Index Three	19	0-63
$	X4	1	Index Four	20	0-63
$	X5	1	Index Five	21	0-63
$	X6	1	Index Six	22	0-63
$	X7	1	Index Seven	23	0-63
$	X8	1	Index Eight	24	0-63
$	X9	1	Index Nine	25	0-63
$	X10	1	Index Ten	26	0-63
$	X11	1	Index Eleven	27	0-63
$	X12	1	Index Twelve	28	0-63
$	X13	1	Index Thirteen	29	0-63
$	X14	1	Index Fourteen	30	0-63
$	X15	1	Index Fifteen	31	0-63
$	XCZ	2	Index Count Zero	11	48
$	XE	2	Index Equal	11	53
$	XF	2	Index Flag	11	38
$	XH	2	Index High	11	54
$	XL	2	Index Low	11	52
$	ZM	2	Zero Multiply	11	33
$	XPFP	2	Exponent Flag Positive	11	28
$	XPH	2	Exponent Range High	11	30
$	XPL	2	Exponent Range Low	11	31
$	XPO	2	Exponent Overflow	11	29
$	XPU	2	Exponent Underflow	11	32
$	XVGZ	2	Index Value Greater Than Zero	11	51
$	XVLZ	2	Index Value Less Than Zero	11	49
$	XVZ	2	Index Value Zero	11	50
$	Z	1	Word Number Zero	0	0-63
$	ZD	2	Zero Divisor	11	24

ALPHABETIC LIST OF OPERATIONS

Type	Mnemonic	Footnote	Name
V	+	3	Add
F	+	6	Add
V	+MG	3	Add to Magnitude
F	+MG	6	Add to Magnitude
V	–	3	Subtract
F	–	6	Subtract
V	–MG	3	Subtract from Magnitude
F	–MG	6	Subtract from Magnitude
V	°	4	Multiply
F	°	7	Multiply
V	°+		Multiply and Add
F	°+		Multiply and Add
F	°A +		Multiply Absolute and Add
V	°I +		Multiply Immediate and Add
V	°N +		Multiply Negative and Add
F	°N +		Multiply Negative and Add
F	°NA +		Multiply Negative Absolute and Add
V	°NI +		Multiply Negative Immediate and Add
V	/	4	Divide
F	/	7	Divide
M	B		Branch
B	BB		Branch on Bit
B	BB1		Branch on Bit and Set to One
B	BBN		Branch on Bit and Negate
B	BBZ		Branch on Bit and Zero
M	BD		Branch Disabled
M	BE		Branch Enabled
M	BEW		Branch Enabled and Wait
M	BR		Branch Relative
B	BZB		Branch on Zero Bit
B	BZB1		Branch on Zero Bit and Set to One
B	BZBN		Branch on Zero Bit and Negate
B	BZBZ		Branch on Zero Bit and Zero
V	C	10	Connect
I	C + I		Add Immediate to Count
I	C – I		Subtract Immediate from Count
C	CB	8	Count and Branch
C	CBR	8	Count, Branch, and Refill
C	CBZ	8	Count and Branch on Zero Count
C	CBZR	8	Count, Branch on Zero Count, and Refill
E	CCW		Copy Control Word
V	CM	10	Connect to Memory
V	CT	10	Connect for Test
E	CTL		Control
V	CV	5	Convert
F	D +	6	Add Double
F	D + MG	6	Add Double to Magnitude
F	D –	6	Subtract Double
F	D – MG	6	Subtract Double from Magnitude
V	DCV	5	Convert Double
F	DL	7	Load Double
F	DLWF	7	Load Double with Flag
F	D°	7	Multiply Double
F	D/	7	Divide Double
F	E +	6	Add to Exponent
F	E + AI		Add Absolute Immediate to Exponent
F	E + I		Add Immediate to Exponent
F	E –	6	Subtract from Exponent
F	E – AI		Subtract Absolute Immediate from Exponent
F	E – I		Subtract Immediate from Exponent
M	EX		Execute
M	EXIC		Execute Indirect and Count
F	F +	6	Add to Fraction
F	F –	6	Subtract from Fraction
V	K	4	Compare
F	K	7	Compare
I	KC		Compare Count
I	KCI		Compare Count Immediate
V	KE	4	Compare If Equal
V	KF	4	Compare Field
V	KFE	4	Compare Field If Equal
V	KFR	4	Compare Field for Range
E	KLN		Check Light On
F	KMG	7	Compare Magnitude
F	KMGR	7	Compare Magnitude for Range
V	KR	4	Compare for Range
F	KR	7	Compare for Range
I	KV		Compare Value
I	KVI		Compare Value Immediate
I	KVNI		Compare Value Negative Immediate
V	L	4	Load
F	L	7	Load
I	LC		Load Count
I	LCI		Load Count Immediate
V	LCV	4	Load Converted
V	LF		Load Field
V	LFT	4	Load Factor
F	LFT	7	Load Factor
E	LOC		Locate (same as Select Unit)
I	LR		Load Refill
I	LRI		Load Refill Immediate
I	LV		Load Value
I	LVE		Load Value Effective
I	LVI		Load Value Immediate
I	LVNI		Load Value Negative Immediate
I	LVS		Load Value with Sum
I	LX		Load Index
V	LTRCV	4	Load Transit Converted
V	LTRS	4	Load Transit and Set
V	LWF	4	Load with Flag
F	LWF	7	Load with Flag
V	M +		Add to Memory
F	M +	6	Add to Memory
V	M + 1		Add One to Memory
F	M + A		Add to Absolute Memory
V	M + MG	3	Add Magnitude to Memory
F	M + MG	6	Add Magnitude to Memory
V	M –		Subtract from Memory
F	M –		Subtract from Memory
V	M – 1		Subtract One from Memory
F	M – A		Subtract from Absolute Memory
V	M – MG	3	Subtract Magnitude from Memory
F	M – MG	6	Subtract Magnitude from Memory
M	NOP		No Operation
M	R		Refill
M	RCZ		Refiill on Count Zero
E	RD		Read
E	REL		Release
E	REW		Rewind
I	RNX		Rename
F	R/		Reciprocal Divide
I	SC		Store Count
E	SEOP	11	Suppress End of Operation
V	SF		Store Field
F	SHF	7	Shift Fraction
F	SHFL		Shift Fraction Left (same as SHFA)
F	SHFR		Shift Fraction Right (same as SHFNA)
M	SIC		Store Instruction Counter If
F	SLO	7	Store Low Order
F	SNRT	6	Store Negative Root
I	SR		Store Refill
V	SRD	5	Store Rounded
F	SRD	7	Store Rounded
F	SRT	6	Store Root
V	ST	5	Store
F	ST	7	Store
E	SU		Select Unit (same as Locate)
I	SV		Store Value
I	SVA		Store Value in Address
T	SWAP		Swap
T	SWAPI		Swap Immediate
T	SWAPB		Swap Backward
T	SWAPBI		Swap Backward Immediate
I	SX		Store Index
T	T		Transmit
T	TI		Transmit Immediate
T	TB		Transmit Backward
T	TBI		Transmit Backward Immediate
I	V +		Add to Value
I	V + I	9	Add Immediate to Value
I	V + C		Add to Value and Count
I	V + CR		Add to Value, Count, and Refill
I	V + IC	9	Add Immediate to Value and Count

Type	Mnemonic	Footnote	Name
I	V + ICR	9	Add Immediate to Value, Count, and Refill
I	V – I	9	Subtract Immediate from Value
I	V – IC	9	Subtract Immediate from Value and Count
I	V – ICR	9	Subtract Immediate from Value, Count, and Refill
E	W		Write
E	WEF		Write End-of-File
M	Z		Store Zero

FOOTNOTES

1. This mnemonic is a system symbol. It must be prefixed by the character "#" whenever used.

2. This mnemonic is both an indicator mnemonic and a system symbol. It must be prefixed by the "#" whenever it is used as a system symbol in a symbolic field of some instruction. This mnemonic may also be used directly to express a Branch on Indicator instruction by being substituted for the letter "I" in any of the following four formats:

BI	Branch on Indicator
BIZ	Branch on Indicator and Zero
BZI	Branch on Zero Indicator
BZIZ	Branch on Zero Indicator and Zero

The mnemonics BI, BIZ, BZI, BZIZ are not in themselves legal operation codes. Any of the integers 0 through 63 may also be substituted for I if it is desired to designate an indicator numerically.

3. This operation code may be suffixed by the letter "I" to invoke immediate addressing.

4. This VFL operation code may have the following suffixes:

I	Immediate
N	Negative
NI	Negative Immediate

5. This operation code may be suffixed by the letter "N" to invoke the negative sign modifier.

6. This floating point operation code may be suffixed by the letter "A" to invoke the absolute sign modifier.

7. This floating point operation code may have the following suffixes:

N	Negative
A	Absolute
NA	Negative Absolute

8. Count and Branch operation may have the following suffixes:

+	Add one to value
–	Subtract one from value
H	Add half to value

9. This operation code may be used to indicate either an immediate indexing operation or the secondary operation of any VFL instruction.

10. This operation mnemonic specifies, potentially, 16 connect instructions. Four binary digits are written directly after the operation code to select a particular one of the 16 instructions. This operation code is also subject to Footnote 3.

11. This code may be used as a secondary operation with I-O select orders that are subject to end-of-operation interrupts.

12. These mnemonics are mathematical constants.

APPENDIX B

Streaming Mode System Symbols

Symbol	Bit Address	Field Length	Meaning
DEBUG	33.60	4	DEBUG code
ERRIND	38.0	8	error indicator register
F	35.19	13	F unit
HR	32.0		harvest registers
MOD	36.56	8	modulus
PBS	39.55	9	bootstrap for P
PCBSL	38.8	7	bootstrap, program controlled, left part
PCBSR	38.16	16	bootstrap, program controlled, right part
PIX	39.32	18	index table, P stream
PS	39.0	24	start address, P stream
PCSCAN	38.15	1	program controlled scan bit
QBS	40.55	9	bootstrap for Q
QIX	40.32	18	index table, Q stream
QS	40.0	24	start address, Q stream
R1	43.0	64	first word of R unit
R2	44.0	64	second word of R unit
RBS	41.53	11	bootstrap for R
RIX	41.32	18	index table, R stream
RS	41.0	24	start address, R stream
SA	34.0	24	SACC register
SAMODE	34.56	2	SACC mode
SASTM	33.56	4	SACC stimulus
SATH	33.32	24	SACC threshold
SC	34.34	16	SCTR register
SCLIM	35.2	16	SCTR limit
SCMODE	34.58	1	SCTR mode
SCSTM	34.59	5	SCTR step stimulus
SSM	33.0	29	stream stimulus mask
SSS	38.36	24	stream stimulus status register
TAO	37.32	26	TA output
TBA	35.32	26	table base address
TE	42.0	64	table extract unit
TEBM	37.24	8	byte mask for TE
TEI	37.6	7	increment
TEM	37.58	6	count of bytes
TEN	37.18	6	number of bytes
TES	37.0	6	initial address, TE unit
TPI	36.6	6	increment
TPJ	36.19	5	reset address
TPM	36.27	5	count of bytes
TPN	36.13	5	number of bytes
TPS	36.1	5	initial offset, P stream
TQI	36.38	6	increment
TQJ	36.51	5	reset address
TQN	36.45	5	number of bytes
TQS	36.33	5	initial offset, Q stream
WCHAR	32.0	8	W match character
WCON	32.9	2	W connections
WM	32.14	1	W mode, OR/AND
WOP	32.11	3	W operation
WSP	32.15	1	W span bit
XCHAR	32.16	8	X match character
XCON	32.24	3	X connections
XM	32.30	1	X mode, OR/AND
XOP	32.27	3	X operation
XSP	32.31	1	X span bit
YCHAR	32.32	8	Y match character
YCON	32.40	3	Y connections
YM	32.46	1	Y mode, OR/AND
YOP	32.43	3	Y operation
YSP	32.47	1	Y span bit
ZCHAR	32.48	8	Z match character
ZCON	32.56	2	Z connections
ZOP	32.59	3	Z operation
ZSP	32.63	1	Z span bit

SYMBOLIC DESCRIPTIONS AND MNEMONICS FOR IBM 7950

The following list of mnemonics may be used with HAP II and HAP III A symbolic description of the mnemonic is given to assist the programmer. The operations symbols used are defined at the start of each section. Note that the same letter ("a" and "m" for example) has a different definition for floating point and for VFL. Carefully read the definition for each set. A more detailed description of the operation is in the IBM 7030 Reference Manual. Form A22-6530.

A specific title for each mnemonic is not given in cases where the mnemonic is derived from the basic operation by changing the sign and absolute modifiers.

In the case of VFL operations, the unsigned modifier must be implied by the data referred to or be explicitly stated in a dds.

FLOATING POINT OPERATIONS

Notation for Symbolizing the Floating Point Operations OP(dds), $A_{18}(I)$

Accumulator Operands

- a = bits (0-59) of the accumulator, and the accumulator sign, bit 4 of the sign byte register.
- b = bits (60-107) of the accumulator, and the accumulator sign.
- ab = bits (0-107) of the accumulator, and the accumulator sign.
- e(a) = bits (0-11) of a.
- f(a) = bits (12-59) of a, and s(a).
- s(a) = bit 4 of the sign byte register.
- SB(a) = bits 4-7 of the sign byte register.
- Fl (a) = bits 5-7 of the sign byte register.

Storage Operands

- m = bits (0-59) of the storage word, and its sign, bit 60.
- M = L(m) = the effective address.
- e(m) = bits (0-11) of m.
- f(m) = bits (12-59) of m, and s(m).
- s(m) = bit 60 of the storage word.
- SB(m) = bits (60-63) of the storage word.
- Fl (m) = bits (61-63) of the storage word.

$FT = Factor operand; SB($FT) = bits (60-63) of $FT.
$RM = Remainder operand.

Add

+	a+m ⟶ a	1. b is unchanged.
−	a−m ⟶ a	2. Fl (a) is unchanged.
+A	a+|m| ⟶ a	
−A	a−|m| ⟶ a	

Add to Memory

M+	m+a ⟶ m	1. Fl(m) remain unchanged.
M−	m−a ⟶ m	2. The entire accumulator and SB(a) remain unchanged.
M+A	|m|+a ⟶ m	
M−A	|m|−a ⟶ m	

Add to Fraction

F+	f(ab)+f(m) ⟶ f(ab)	1. e(m) is ignored; the add is performed with e(a) on both operands.
F−	f(ab)−f(m) ⟶ f(ab)	2. The normalized mode operates in the same way as in D+.
F+A	f(ab)+|f(m)| ⟶ f(ab)	
F−A	f(ab)−|f(m)| ⟶ f(ab)	

Add to Exponent

E+	e(ab)+e(m) ⟶ |e(ab)	1. f(m) is ignored.
E−	e(ab)−e(m) ⟶ e(ab)	2. Strap−II will assemble as unnormalized unless the normalized mode is requested by referring to normalized data or by using the dds = (N).
E+A	e(ab)+|e(m)| ⟶ e(ab)	
E−A	e(ab)−|e(m)| ⟶ e(ab)	

Add Immediate to Exponent

E+I	e(ab)+e(M) ⟶ e(ab)	1. The unnormalized mode is given unless overruled by dds = (N).
E−I	e(ab)−e(M) ⟶ e(ab)	
E+AI	e(ab)+|e(M)| ⟶ e(ab)	
E−AI	e(ab)−|e(M)| ⟶ e(ab)	

Shift Fraction

SHF	$f(ab)\cdot 2^{M} \longrightarrow f(ab)$	1. Left shift if bit 11 of M = 0.
SHFN	$f(ab)\cdot 2^{-M} \longrightarrow f(ab)$	2. Right shift if bit 11 of M = 1.
SHFA	$f(ab)\cdot 2^{\lvert M\rvert} \longrightarrow f(ab)$	3. The operation is not affected by the normalized modifier.
SHFNA	$f(ab)\cdot 2^{-\lvert M\rvert} \longrightarrow f(ab)$	4. The exponent is not adjusted for the shift. e(a) is unchanged.
SHFL	$f(ab)\cdot 2^{\lvert M\rvert} \longrightarrow f(ab)$	5. On a right shift, zeroes are introduced in bit 12.
SHFR	$f(ab)\cdot 2^{-\lvert M\rvert} \longrightarrow f(ab)$	

Double Add

D+	ab+m ⟶ ab	1. PSH indicator goes on if the exponent difference exceeds 48.
D−	ab−m ⟶ ab	
D+A	ab+|m| ⟶ ab	
D−A	ab−|m| ⟶ ab	

Add to Magnitude

+MG	R = |a|+m	1. R ⟶ a if R ≥ 0.
−MG	R = |a|−m	2. 0 ⟶ f(a) if R < 0 and e(a) is unchanged.
+MGA	R = |a|+ |m|	3. s(a) is unchanged in either case.
−MGA	R = |a|− |m|	

Double Add to Magnitude

D+MG	R = |ab|+m	1. R ⟶ ab if R ≥ 0.
D−MG	R = |ab|−m	2. 0 ⟶ f(ab) if R < 0 and e(a) is unchanged.
D+MGA	R = |ab|+ |m|	3. s(a) is unchanged in either case.
D−MGA	R = |ab|− |m|	

Add Magnitude to Memory

M+MG	R = m+|a|	1. R ⟶ m if s(R) = s(m)
M−MG	R = m−|a|	2. 0 ⟶ f(m) if s(R) ≠ s(m).
M+MGA	R = |m| + |a|	3. s(m) is unchanged in either case.
M−MGA	R = |m| − |a|	

Multiply

*	a·m ⟶ a	1. b in unchanged.
*N	a·−m ⟶ a	
*A	a·|m| ⟶ a	
*NA	a·−|m| ⟶ a	

Double Multiply

D*	a·m ⟶ ab	1. (108-127) of accumulator are unchanged.
D*N	a·−m ⟶ ab	
D*A	a·|m| ⟶ ab	
D*NA	a·−|m| ⟶ ab	

Multiply Factor and Add

°+	m·(\$FT)+ab→ab	1. The contents of \$FT remain unchanged.
°N+	−m·(\$FT)+ab→ab	
°A+	|m|·(\$FT)+ab→ab	
°NA+	−|m|·(\$FT)+ab→ab	

Divide

/	a/m ⟶ a	1. No remainder is generated.
/N	a/−m ⟶ a	2. Quotient is 48 bits.
/A	a/|m| ⟶ a	3. Pre-normalization of the operands is independent of the normalization modifier.
/NA	a/−|m| ⟶ a	4. b is unchanged.

Reciprocal Divide

R/	m/a ⟶ a	1. Performed similarly to divide.
R/N	−m/a ⟶ a	2. b is unchanged.
R/A	|m|/a ⟶ a	
R/NA	−|m|/a ⟶ a	

Double Divide

D/	ab/m ⟶ ab	1. Remainder in \$RM.
D/N	ab/−m ⟶ ab	2. 0→b except bit 60, which contains a continuation of f (a).
D/A	ab/|m| ⟶ ab	3. No rounding.
D/NA	ab/−|m| ⟶ ab	4. SB(a)→SB(\$RM).
		5. Result capable of being rounded in a subsequent instruction.

Store Root

SRT	$\sqrt{a}$ ⟶ m	1. ab and SB(a) are unchanged.
SNRT	$-\sqrt{a}$ ⟶ m	
SRTA	$\sqrt{\|a\|}$ ⟶ m	
SNRTA	$-\sqrt{\|a\|}$ ⟶ m	

Load

L	m ⟶ a	1. 0 ⟶ Fl(a).
LN	−m ⟶ a	2. b is unchanged.
LA	|m| ⟶ a	
LNA	−|m| ⟶ a	

Double Load

DL	m ⟶ a	1. 0 ⟶ b.
DLN	−m ⟶ a	2. 0 ⟶ Fl(a)
DLA	|m| ⟶ a	
DLNA	−|m| ⟶ a	

Load with Flag Bits

LWF	m ⟶ a	1. Fl(m) ⟶ Fl(a).
LWFN	−m ⟶ a	
LWFA	|m| ⟶ a	
LWFNA	−|m| ⟶ a	

Double Load with Flag Bits

DLWF	m ⟶ a	1. 0 ⟶ b.
DLWFN	−m ⟶ a	2. Fl(m) ⟶ Fl(a).
DLWFA	|m| ⟶ a	
DLWFNA	−|m| ⟶ a	

Load Factor

LFT	m ⟶ \$FT	1. ab and SB(a) are not changed.
LFTN	−m ⟶ \$FT	2. s(m) ⟶ (60)\$FT.
LFTA	|m| ⟶ \$FT	3. 0 ⟶ (61-63)\$FT.
LFTNA	−|m| ⟶ \$FT	

Store

ST	a ⟶ m	1. Fl(a) ⟶ Fl(m).
STN	−a ⟶ m	2. a is unchanged.
STA	|a| ⟶ m	
STNA	−|a| ⟶ m	

Store Rounded

SRD	a ⟶ m	1. A one is added in bit (60)b prior to the store: a and (60)b are unchanged.
SRDN	−a ⟶ m	2. Fl(a) ⟶ Fl(m).
SRDA	|a| ⟶ m	
SRDNA	−|a| ⟶ m	

Store Low Order

SLO	b ⟶ f(m)	1. e(a) - 48 ⟶ e(m).
SLON	−b ⟶ f(m)	2. Fl(a) ⟶ Fl(m).
SLOA	|b| ⟶ f(m)	3. e(a) is unchanged.
SLONA	−|b| ⟶ f(m)	

Compare

K	a:m	1. Indicators AL, AE, and AH are set as follows:
KN	a:−m	AL is set to one if a < m
KA	a:|m|	AE is set to one if a = m
KNA	a:−|m|	AH is set to one if a > m
		2. Zero exponents of different sign are considered equal.
		3. If the exponent difference is 48 the larger of the numbers is per sign and exponents regardless of fractions.

Compare for Range

KR	a:m	1. If AH is off prior to this op, no indicators will be changed.
KRN	a:−m	2. If AH is on:
KRA	a:|m|	AL is unchanged.
KRNA	a:−|m|	AE is set to one if a < m.
		AH is set to one if a ≥ m.

Compare Magnitude

KMG	a:m	1. Same as Compare, except for accumulator comparand.
KMGN	a:−m	
KMGA	a:|m|	
KMGNA	a:−|m|	

Compare Magnitude for Range

KMGR	a:m	1. Same as Compare for Range, except for accumulator comparand.
KMGRN	a:−m	
KMGRA	a:|m|	
KMGRNA	a:−|m|	

VARIABLE FIELD LENGTH OPERATIONS

Notation for Symbolizing the Variable Field Length Operations OP(dds), $A_{24}(I)$, $OF_7(I')$

Accumulator Operands

a = the accumulator operand whose:
1. Low order bit is defined by the offset;
2. Byte size is four for decimal arithmetic, eight for binary arithmetic;
3. Length includes all bits in the accumulator to the left of the offset;
4. Sign is indicated by bit four of the sign byte register.

$\bar{a}$ = the accumulator operand, a, but without sign.

a_{20} = the accumulator operand, a, with offset = 20.

Storage Operands

m = the storage operand whose:
1. High-order bit is defined by the bit address;
2. Byte size may be any number from one to eight, but is assumed to be four in the instruction lists below;
3. Length is defined by the field length in the dds;
4. Sign is bit s in the sign byte.

$\bar{m}$ = the storage operand in which all bytes are processed as data; a positive sign is assumed.

The unsigned storage operand is designated by the dds.

Bits 7.17 and 7.18 are the leftmost two bits of $LZC.

$FT = Factor Operand; s($FT) = bit 60; FL($FT) = bits (61-63).

$TR = 64-bit Transit Register.

Integer Operations

Operations which can have an immediate operand are followed by (I), except for *+.

Add

+	a+m ⟶ a	(I) 1. If the sign changes, bits to the right of the offset are complemented.
−	a−m ⟶ a	

Add To Memory

M+	m+a ⟶ m	
M−	m−a ⟶ m	

Add to Magnitude

+MG	R=$\bar{a}$+m	(I) 1. R⟶$\bar{a}$ if R ≥ 0.
−MG	R=$\bar{a}$−m	2. 0⟶entire accumulator if R < 0.
		3. s(a) is not changed by these operations.

Add Magnitude To Memory

M+MG	R=m+$\bar{a}$	1. R⟶m if s(R) = s(m).
M−MG	R=m−$\bar{a}$	2. 0⟶m if s(R) ≠ s(m).
		3. s(m) is not changed.

Multiply

*	a·m ⟶ a_{20}	(I) 1. Multiplication takes place only if mode = B or BU.
*N	a·−m ⟶ a_{20}	2. The decimal mode gives LTRS and 00_2 to bits 7.17 and 7.18.
		3. The length of a or m must be ≤ 48 bits in binary multiply.
		4. The portion of the accummulator not containing the product is set to zero.

Multiply Factor and Add

*+	m·($FT)+a ⟶a	(I) 1. Write: *I+ and *NI+ for an immediate operand.
*N	−m·($FT)+a⟶a	2. Multiplication takes place only if mode = B or BU.
		3. Decimal mode gives LTRS and 10_2 to bits 7.17 and 7.18.

Divide

/	a/m ⟶ a	(I) 1. Divide takes place only in the binary mode.
/N	a/−m ⟶ a	2. Decimal divide gives LTRS and 01_2 in bits 7.17 and 7.18.
		3. The remainder is placed in $RM. The remainder sign, (60) $RM, is the same as the original s(a). Fl ($RM) = 0.
		4. Bits to the right of the offset are cleared.

Load

L	m ⟶ a	(I) 1. 0⟶Fl(a).
LN	−m ⟶ a	2. The entire accumulator is cleared before the load.

Load with Flag Bits

LWF	m ⟶ a	(I) 1. Fl(m)⟶Fl(a).
LWFN	−m ⟶ a	

Load Factor

LFT	m ⟶ $FT	(I) 1. 0⟶(61 − 63) $FT.
LFTN	−m ⟶ $FT	2. The offset field is ignored.

Load Transit and Set

LTRS	m ⟶ $TR	(I) 1. Offset⟶$AOC.
LTRSN	−m ⟶ $TR	2. 11_2⟶bits 7.17 and 7.18.
		3. Indicator $BTR = 1 and $DTR = 0 if mode is B or BU. Indicator $DTR = 1 and $BTR = 0 if mode is D or DU.

Store

ST	a ⟶ m	1. SB(a)⟶SB(m).
STN	−a ⟶ m	2. If the byte size is greater than four:
		Binary: zone bits of the sign byte register are stored in SB(m).
		Decimal: zone bits of the sign byte register are stored in each byte of m.

Store Rounded

SRD, SRDN — These operations are the same as the corresponding Store operations, except for:

a. Binary: a one is added one bit to the right of the offset, prior to the store.

b. Decimal: 0101 is added one byte to the right of the offset, prior to the store.

c. The accumulator is unchanged, even if rounding occurs.

Add One to Memory

M+1	m+1 ⟶ m	1. The one is added to the low order byte.
M−1	m−1 ⟶ m	2. The offset field is ignored.

Compare

K	a:m	(I) 1. The Compare operations set the AL, AE, and AH indicators.
KN	a:−m	AL is set to one if: a < m
		AE is set to one if: a = m
		AH is set to one if: a > m
		2. All bits to the left of the offset in the accumulator participate in the compare.

Compare for Range

KR	a:m	(I) 1. If the AH indicator is off prior to the operation, it is executed as a NOP.
KRN	a:−m	2. If AH is on:
		AL is unchanged.
		AL is set to one if a < m
		AH is set to one if a ≧ m

Compare If Equal

KE	a:m	(I) 1. If the AE indicator is off, no changes will occur.
KEN	a:−m	2. If the AE indicator is on, the indicators are set as in Compare, K.

Compare Field

KF	$\bar{a}$:m	(I) 1. The indicators are set as in Compare.
KFN	$\bar{a}$:−m	2. The length of the accumulator comparand is the same as the length of the storage comparand.
		3. The matching bits of both operands are compared.

Compare Field for Range

KFR	$\bar{a}$:m	(I)	1. The accumulator comparand is the same as in Compare Field, KF.
KFRN	$\bar{a}$:−m		2. The indicators are set as in Compare Range, KR.

Compare Field If Equal

KFE	$\bar{a}$:m	(I)	1. The accumulator comparand is the same as in Compare Field, KF.
KFEN	$\bar{a}$:−m		2. The indicators are set as in Compare If Equal, KE.

Logical Connectives OP(dds), A_{24} (I), OF_7 (I')

Note: If the operand from storage has a byte size (BS) less than eight, then eight minus BS (8 − BS) leading zeros are added to each byte from storage before the connect takes place. However, the storage operand is not changed in Cxxxx or CTxxxx.

Connect to Accumulator

$Cx_1x_2x_3x_4$ Result⟶a

Connect to Memory

$CMx_1x_2x_3x_4$ Result⟶m

Connect for Test

$CTx_1x_2x_3x_4$ Result is not stored.

$x_1x_2x_3x_4$ is a four-bit binary configuration to describe the type of connective; it is summarized:

Let: m = a bit from storage (may be an inserted leading zero if the byte size is less than 8.)
a = a bit from the accumulator corresponding to m. The accumulator byte size always = 8.
x_1 = desired result if m = 0 and a = 0
x_2 = desired result if m = 0 and a = 1
x_3 = desired result if m = 1 and a = 0
x_4 = desired result if m = 1 and a = 1

Example: C1010 (BU, 64, 4), 0 will complement the entire 128-bit accumulator.

Pseudo-Connectives

LF	(Load Field)	LF = C0011
SF	(Store Field)	SF = CM0101

Immediate Connects

To indicate immediate addressing, write: $CIx_1x_2x_3x_4$, $CTIx_1x_2x_3x_4$, and LFI.

$AOC = All ones count register.
$LZC = Left zeros count register.

After a connective operation the two registers, $AOC and $LZC contain the indicated counts of the result. Because the result may not occupy the entire accumulator, $AOC and $LZC may not give the total count of ones and left zeros of the accumulator. However, these counts always give the correct count in CM or SF.

Convert Instructions

Definitions:

a_D = accumulator in decimal, four-bit bytes with specified offset.
a_B = accumulator in binary with specified offset.
a_{B20} = accumulator in binary with offset = 20.
a_{B68} = accumulator in binary with offset = 68.
m_B = storage operand in binary with specified byte size and field length.
m_D = storage operand in decimal with specified byte size and field length.
$TR = 64-bit transit register with a sign byte in the rightmost four bits.

Note: The conversion goes: from decimal to binary if the mode given is decimal; from binary to decimal if the given mode is binary.

Convert

CV	$a_D \longrightarrow a_{B68}$	if mode = D or DU	1. In binary a field of 48 bits is used.
or	$a_{B68} \longrightarrow a_D$	if mode = B or BU	2. The entire accumulator to the left of the offset is used.
CVN	$-a_D \longrightarrow a_{B68}$		
or	$-a_{B68} \longrightarrow a_D$		

Double Convert

DCV	$a_D \longrightarrow a_{B20}$	1. In binary, a field of 96 bits is used.
or	$a_{B20} \longrightarrow a_D$	2. The entire accumulator to the left of the offset is used.
DCVN	$-a_D \longrightarrow a_{B20}$	
or	$-a_{B20} \longrightarrow a_D$	

Load Converted

LCV	$m_D \longrightarrow a_B$	(I)	1. s(m)⟶s(a)
or	$m_B \longrightarrow a_D$		2. 0⟶Fl(a)
LCVN	$-m_D \longrightarrow a_B$	(I)	3. The entire accumulator is cleared before the load.
or	$-m_B \longrightarrow a_D$		

Load Transit Converted

LTRCV	$m_D \longrightarrow \$TR_B$	(I)	1. The accumulator and offset are ignored.
or	$m_B \longrightarrow \$TR_D$		2. 0⟶Fl($TR)
LTRCVN	$-m_D \longrightarrow \$TR_B$	(I)	3. s(m)⟶s($TR)
or	$-m_B \longrightarrow \$TR_D$		4. The entire $TR is cleared before the load.

Progressive Indexing

Any VFL or Connective operation (when not immediate) may have a second operation enclosed in parentheses. The second operation may be V ± I, V ± IC or V ± ICR.

Format: $OP(OP_2)(dds), A_{24} (J), OF_7 (I')$

Notes: 1. The original value field J is the effective address of operation.
2. A_{24} is the immediate operand specified by J in V ± I, and so on, and the value field of J is incremented by ± A_{24} according to ± I. The incrementing takes place subsequent to note 1.
3. J may be $XO.

INDEXING OPERATIONS

Notation for symbolizing the Indexing Operations

Index Word Operands

J = bits (0 − 63) of the index word
V = bits (0 − 24) of J.
C = bits (28 − 45) of J.
R = bits (46 − 63) of J.

Storage Word Operands

m = bits (0 − 63) of a storage word.
V(m) = bits (0 − 24) of m if the second operand is V. (sign of V is in bit 24)
V(m) = bits (0 − 17) of m if the second operand is C or R.

Immediate Operands

m = bits (0 − 18) of the effective address if the second operand is V.
m = bits (0 − 17) of the effective address if the second operand is C or R.

Notes: 1. For clarity, the titles to the indexing and the branch operations have been omitted.

2. The indicators XF, XCZ, XVLZ, XVZ, and XVGZ are set by all of the direct and immediate index operations except KV, KC, KVI, KVNI, and KCI. These indicators are set before the refill (if any) takes place.

KV, KC,. . . .,KCI set the index compare indicators XL, XE, and XH.

Direct Index Arithmetic OP, J, A_{19} (I)

LX	m ⟶ J	A_{18}
LV	V(m) ⟶ V	1. M = A_{19} (I)
LC	V(m) ⟶ C	2. m = (M)
LR	V(m) ⟶ R	3. C_2 = The count field of J after modification
SX	J ⟶ m	1. A_{18}
SV	V ⟶ V(m)	
SC	C ⟶ V(m)	1. 0 ⟶ (18 - 24) of m.
SR	R ⟶ V(m)	1. 0 ⟶ (18 - 24) of m.
V+	V+V(m) ⟶ V	1. There is no V – etc.
V+C	V+V(m) ⟶ V C–1 ⟶ C_2	
V+CR	V+V(m) ⟶ V C–1 ⟶ C_2 (R) ⟶ (J) if $C_2 = 0$	
SVA	V ⟶ V(m)	1. V is truncated to 18, 19, or 24 bits, as is appropriate for the instruction containing V(m).
LVE	$(M)^n$ ⟶ V	1. (M) means contents of M $(M)^1$ " " " (M) . . . $(M)^n$ " " " $(M)^{n-1}$
KV ⟶ V:V(m) KC ⟶ C:V(m)		1. Indicators: XL, XE, XH are set by KV and KC. This setting is the only output of KV and KC.
RNX	J⟶(R($XO)) M ⟶ R($XO) m ⟶ J	1. Used for saving and restoring index registers.
LVS	(special format):	LVS, J, A^1, A^2, . . . , A^n
$\sum_{i=1}^{n}$	$V(A^i)$ ⟶ V(J)	1. The sum may include any subset of the index words, each one appearing no more than once. 2. No indexing of the address field is allowed.

Immediate Index Arithmetic OP, J, A_{19}

Notes:
1. None of the immediate index instructions allow for indexing of the address. A_{19} is the effective address and is represented by A below.
2. The output of KVI, KVNI, and KCI is the setting of indicators XL, XE, and XH.

LVNI	–A ⟶ V	1. (19 - 23) of V are set to 0.
LVI	A ⟶ V	1. (19 - 24) of V are set to 0.
LCI	A ⟶ C	
LRI	A ⟶ R	
V+I	V+A ⟶ V	1.
V–I	V–A ⟶ V	1.
V+IC	V+A ⟶ V C–1 ⟶ C	1.
V–IC	V–A ⟶ V C–1 ⟶ C	1. A is appended by 5 zero bits for the operation.
V+ICR	V+A ⟶ V C–1 ⟶ C_2 (R) ⟶ (J) if $C_2 = 0$	1.
V–ICR	V–A ⟶ V C–1 ⟶ C_2 (R) ⟶ (J) if $C_2 = 0$	1.
C+I	C+A ⟶ C_2	
C–I	C–A ⟶ C_2	
KVI	(0 - 18) of V:A	1. (19 - 24) of V are compared with zeros.
KVNI	(0 - 18) of V:A	1. (19 - 23) of V are compared with zeros and (24) of V is compared with 1 (minus).
KCI	C:A	

Count and Branch Operations OP, J, B_{19} (K)

CB	C_1 - 1 ⟶ C_2 IC_1 + 0.32 ⟶ IC if $C_2 = 0$ M ⟶ IC if $C_2 \neq 0$	1. K may be only 0 or 1. 2. M = the effective address of B_{19} (K). 3. IC_1 is the value of the instruction counter where the CB instruction is located. 4. C_1 and C_2 are the count field of J before and after the count portion of the instruction, respectively.
CBR	C_1 – 1 ⟶ C_2 IC_1 + 0.32 ⟶ IC and (R)⟶(J) if $C_2 = 0$ M ⟶ IC if $C_2 \neq 0$	
CBZ	C_1 – 1 ⟶ C_2 IC_1 + 0.32 ⟶ IC if $C_2 \neq 0$ M ⟶ IC if $C_2 = 0$	
CBRZ or CBZR	C_1 - 1 ⟶ C_2 IC_1 + 0.32 ⟶ IC if $C_2 \neq 0$ M ⟶ IC and (R)⟶(J) if $C_2 = 0$	

Note: In addition to the stated functions, the value field of J may be modified by placing + , – , or H after the above mnemonics. The modification of V takes place regardless of C_2 and before the refill (if any).

Example: In addition to the given functions of CB, we have:

CB	leave V alone
CB+	V + 1.0 ⟶ V
CB–	V – 1.0 ⟶ V
CBH	V + 0.32 ⟶ V

Unconditional Branch Operations: OP, A_{19} (I)

B BR	M ⟶ IC $M+IC_1$ + 0.32 ⟶ IC	1. The unconditional branch instructions are the only branch instructions which allow a 4 bit index field, I. The conditional branch instructions may have only a 1-bit index field, K. 2. IC_1 is the value of the instruction is located (i.e., the leftmost bit of the instruction).
BE	Enable M ⟶ IC	
BD	Disable M ⟶ IC	
BEW	Enable M Wait ⟶ IC	
NOP	IC_1 + 0.32 ⟶ IC	

Branch on Bit Operations: OP, A_{24} (I), B_{19} (K)

BB	IC_1 + 0.32 ⟶ IC if $m_1 = 0$ M_2 ⟶ IC if $m_1 = 1$	1. m_1 = (A_{24}(I)), the bit being tested. 2. M_2 = B_{19}(K), the branch address. 3. K = 0 or 1; I = 1 –15.
BZB	IC_1 + 1.0 ⟶ IC if $m_1 = 1$ M_2 ⟶ IC if $m_1 = 0$	

Note: The BB and BZB may have a suffix, Z, 1, or N, which, respectively, will set m_1 to zero or to one, or negate it. This function is independent of the success of the branch. For example, the following branch on bit instructions are permissible and perform the stated functions as well as:

BB	BZB	leave m_1 alone
BBZ	BZBZ	0 ⟶ m_1
BB1	BZB1	1 ⟶ m_1
BBN	BZBN	$-m_1$ ⟶ m_1

Branch on Indicator Operations BIND, B_{19} (K)

BIND	IC_1 + 0.32 ⟶ IC if ind. = 0 M ⟶ IC if ind. = 1	1. The indicators may not be set to 1 or negated with a BIND operation.
BZIND	IC_1 + 0.32 ⟶ IC if ind. = 1 M ⟶ IC if ind. = 0	

Notes: 1. The letters "IND" in BIND are replaced by the appropriate indicator mnemonics as shown in note 2 below.

2. The above operations can have a suffix, Z, which will cause the indicator being tested to be set to zero independently of the success of the branch. For example, BZXPOZ will set indicator XPO to zero arbitrarily. We may have: BXPO; BZXPO; BXPOZ; and BZXPOZ. The following list indicates all of the indicator mnemonics which may be used in BIND, $B_{19}(K)$, and their bit addresses.

Mnemonic	Name	Bit Address
	EQUIPMENT CHECK	
MK	Machine Check	11.0
IK	Instruction Check	11.1
IJ	Instruction Reject	11.2
EK	Exchange Control Check	11.3
	ATTENTION REQUEST	
TS	Time Signal	11.4
CPUS	CPU Signal	11.5
	INPUT-OUTPUT REJECTS	
EKJ	Exchange Check Reject	11.6
UNRJ	Unit Not Ready Reject	11.7
CBJ	Channel Busy Reject	11.8
	INPUT-OUTPUT STATUS	
EPGK	Exchange Program Check	11.9
UK	Unit Check	11.10
EE	End Exception	11.11
EOP	End of Operation	11.12
CS	Channel Signal	11.13
	(not available)	11.14
	INSTRUCTION EXCEPTION	
OP	Operation Invalid	11.15
AD	Address Invalid	11.16
USA	Unended Sequence of Addresses	11.17
EXE	Execute Exception	11.18
DS	Data Store	11.19
DF	Data Fetch	11.20
IF	Instruction Fetch	11.21
	RESULT EXCEPTION	
LC	Lost Carry	11.22
PF	Partial Field	11.23
ZD	Zero Divisor	11.24
	RESULT EXCEPTION-FLOATING POINT	
IR	Imaginary Root	11.25
LS	Lost Significance	11.26
PSH	Preparatory Shift Greater than 48	11.27
XPFP	Exponent Flag Positive	11.28
XPO	Exponent Overflow	11.29
XPH	Exponent High	11.30
XPL	Exponent Range Low	11.31
XPU	Exponent Underflow	11.32
ZM	Zero Multiply	11.33
RU	Remainder Underflow	11.34
	FLAGGING	
TF	T Flag	11.35
UF	U Flag	11.36
VF	V Flag	11.37
XF	Index Flag	11.38
	TRANSIT OPERATIONS	
BTR	Binary Transit	11.39
DTR	Decimal Transit	11.40
	PROGRAMMER INDICATORS	
PGO or PG		11.41
PG1		11.42
PG2		11.43
PG3		11.44
PG4		11.45
PG5		11.46
PG6		11.47
	INDEX RESULT	
XCZ	Index Count Zero	11.48
XVLZ	Index Value Less than Zero	11.49
XVZ	Index Value Zero	11.50
XVGZ	Index Value Greater Than Zero	11.51
XL	Index Low	11.52
XE	Index Equal	11.53
XH	Index High	11.54
	ARITHMETIC RESULT	
MOP	To-Memory Operation	11.55
RLZ	Result Less than Zero	11.56
RZ	Result Zero	11.57
RGZ	Result Greater than Zero	11.58
RN	Result Negative	11.59
AL	Accumulator Low	11.60
AE	Accumulator Equal	11.61
AH	Accumulator High	11.62
	MODE	
NM	Noisy Mode	11.63

TRANSMIT OPERATIONS: OP, J, $A_{18}(I)$, $A'_{18}(I')$

Notes:
1. Full words are transmitted in all Transmit and Swap instructions.
2. In the immediate operations, J is the count of the number of full words transmitted. J must be ≤ 16. If $J = 0$, 16 words are transmitted.
3. In the others (the direct transmission) the count field of J has the number of full words to be transmitted.

Transmit Forward

T $(M_1) \longrightarrow (M_2)$
$(M_1+1) \longrightarrow (M_2+1)$
etc.

1. M_1 is the effective address of $A_{18}(I)$
2. M_2 is the effective address of $A'_{18}(I')$

Transmit Forward Immediate

TI $(M_1) \longrightarrow (M_2)$
$(M_1+1) \longrightarrow (M_2+1)$
etc.

Transmit Backward

TB $(M_1) \longrightarrow (M_2)$
$(M_1-1) \longrightarrow (M_2-1)$
etc.

1. Both blocks are referred to in a backward direction.

Transmit Backward Immediate

TBI $(M_1) \longrightarrow (M_2)$
$(M_1-1) \longrightarrow (M_2-1)$
etc.

Swap Forward

SWAP $(M_1) \longleftrightarrow (M_2)$
$(M_1+1) \longleftrightarrow (M_2+1)$
etc.

Swap Forward Immediate

SWAPI $(M_1) \longleftrightarrow (M_2)$
$(M_1+1) \longleftrightarrow (M_2+1)$
etc.

Swap Backward

SWAPB $(M_1) \longleftrightarrow (M_2)$
$(M_1-1) \longleftrightarrow (M_2-1)$
etc.

Swap Backward Immediate

SWAPBI $(M_1) \longleftrightarrow (M_2)$
$(M_1-1) \longleftrightarrow (M_2-1)$
etc.

MISCELLANEOUS OPERATIONS: OP, $A_{19}(I)$

Store Instruction Counter If

SIC — $IC_1+1.0 \longrightarrow (0-18)$ of $A_{19}(I)$ if the following half word branch instruction is executed.

1. SIC; NOP will not store the IC.

Refill

R — $(R_M) \longrightarrow (M)$

1. R_M = refill field of word M.

Refill If Count Is Zero

RCZ — $(R_M) \longrightarrow (M)$ if C field of M = 0

Execute

EX — Execute $\longrightarrow (M)$

1. The instruction located at M is executed.
2. Control then goes to the instruction following EX.

Execute Indirect and Count

EXIC — Execute $\longrightarrow (M)^1$
$(M) + 1 \longrightarrow (M)$

1. The instruction whose address is located in M is executed.

Store Zero

Z — $0 \longrightarrow (M)$

1. Full word of zeros.

INPUT-OUTPUT INSTRUCTIONS: OP, $A_7(I)$, $A_{18}(I')$

Locate LOC
Select Unit SU

$A_7(I)$ represents a channel address; $A_{18}(I')$ represents:
1. The address of one of several units attached to channel $A_7(I)$; in this case LOC or SU must be given before a RD or W addressing this channel;
2. An address on the disk specified by $A_7(I)$. LOC = SU.

Read

RD — $A_7(I)$ represents a channel address; a reading operation is initiated for this channel (or for a unit attached to this channel if more than one unit is available and has been readied by a LOC instruction). $A_{18}(I')$ is the address of a control word.

Write

W — Initiates a writing operation. Analogous to RD except that the skip flag of the control word is ignored.

Release

REL — Immediately terminates any operation in progress at the unit specified in $A_7(I)$, the channel address, or in the last unit at $A_7(I)$ selected by a LOC instruction, if $A_7(I)$ consists of more than one unit.

Copy Control Word

CCW — The current control word corresponding to the addressed channel $A_7(I)$ is sent to $A_{18}(I')$.

LOC(SEOP)
RD(SEOP)
W(SEOP)
REL(SEOP)
CTL(SEOP)
SU(SEOP) — Same as LOC, SU, RD, W, REL, CTL except the SEOP bit in control word is set to 1; thus, program interruption on completion of an operation is suppressed, provided no exception conditions, such as unit check and end exception, are encountered.

Control

CTL — Initiates performance of certain functions at the channel indicated by $A_7(I)$, or at the last unit selected by an LOC instruction. The functions are indicated:

General I/O Unit (Standard for $A_{18}(I)$)
$A_{18}(I') = 016_8$ Reserved Light Off
017_8 Reserved Light On
116_8 Read-Write Check Light On
057_8 ECC Mode
157_8 No ECC Mode

Card Reader and Card Punch
Standard, except $A_{18}(I') = 2$ also causes a card to be offset in the stacker.

Tape Units
Standard, but in addition:
$A_{18}(I') = 057_8$ ECC Mode, Odd Parity
157_8 No ECC Mode, Odd Parity
156_8 No ECC Mode, Even Parity
136_8 Rewind Tape
076_8 Space Block (record)
176_8 Backspace Block (record)
077_8 Space File
177_8 Backspace File
117_8 Write Tape Mark (EOF mark)
056_8 Erase Long Gap
036_8 High-Density Mode (556 bits/inch)
037_8 Low-Density Mode (200 bits/inch)
016_8 Remove End of Tape Condition
137_8 Rewind and Unload

Inquiry Station, Printer, Console
Standard, except codes 057_8 and 157_8 are missing. On Console, $A_{18}(I') = 177_8$ causes the gong to sound.

IBM 7950 Reference Card

Form R22-976:
Printed in U.S.A

ADJUSTMENT AND SETUP STIMULI

	STIMULUS MNEMONIC	ADJSTM (Octal)	SASTM (Octal)	SCSTM (Octal)	SSS 38.	SSM 33.
Indexing	**FL1P**	5		30	40	8
	FL2P	6		—	41	9
	FL3P	7		—	42	10
	FL1Q	1		31	43	11
	FL2Q	2		—	44	12
	FL3Q	3		—	45	13
	FL1R	21		32	46	14
	FL2R	22		—	47	15
	FL3R	23		—	48	16
Match Stim	**W**	24	—	20	36	4
	X	26	5	21	37	5
	Y	40	6	22	38	6
	Z	37	—	23	39	7
	NW	11	7	—	—	0
	NX	12	—	24	—	1
	NY	13	—	25	—	2
	NZ	14	10	—	—	3
	W.X	15	—	—		
	W.Y	16	—	27		
	XVY	27	—	26		
	NWXY	—	17	—		
Other Data	**BYP**	—	—	33		
	BYQ	—	—	34		
	BYR	—	13	—		
	BYSA	—	—	35		
	ELTE	50	—	37	49	17
	EG	77	11	—		
	TAF	—	—	36		
Misc	**INIT**	60				
	NOP	0	0	0		
	CCZ	17	—	—	—	28
	STIM	47				

LU Stim		ADJSTM (Octal)			SASTM (Octal)		SCSTM (Octal)			SSS 38.	SSM 33.
	AND'ing*	no	**EG**	**FZ**	**F1**	no **EG**	no	**EG**	**FZ**		
	FZ.	33	70				17				
	F1.	67				4	10			51	19
	KBZ.	43			63	1			16		
	KB1.	44		35	64		1		11	53	21
	LBZ.	45			65	2	5		15		
	LB1.	42		34	62	12	2		12	54	22
	MBZ.	46			66	3			13		
	MB1.	41		36	61		4		14	55	23
	KGZ.	53	73			14					
	KG1.	54	74					3		56	24
	LGZ.	55	75			15					
	LG1.	52	72					6		57	25
	MGZ.	56	76			16					
	MG1.	51	71					7		58	26

SA & SC Stim	AND'ing	no	**EG**								
	SAGETH	10								50	18
	SALT.		31								
	SABN	32									27
	SCLIM	25								52	20
	SCNL.		30								

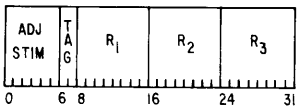
Other SSS Bits			
	ACR6	59	
	FRCT**		
	F reset	61	
	F counter	62, 63	

Mnemonics in the left column of the above table must be used in ADJ, SA, SC, and SSM statements. No mnemonic may be used in a statement for which it does not exist; for example, W as an SA stimulus

*When no AND'ing is desired for the AND'ing stimuli, delete the period. When AND'ing is desired, add the mnemonic at the top; i.e., MB1.FZ

**The system symbol FRCT includes the F reset and the F counter

ADJUSTMENT REACTIONS



ADJ (stimulus, tag), reaction 1, reaction 2, reaction 3
Tag Field: 00 blank 10 **BR**
01 **AND** 11 **CH**

For **BR** and **CH**, reaction 3 *relative address*

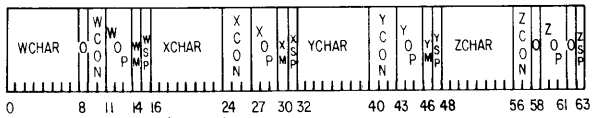
Type	Group	Description	Mnemonic	Octal	Hex
NO OP			**NOP**	000	0
STATISTICAL UNITS	RESET:	F and G	**RSFG**	025	1
		SC	**RSSC**	031	1
		SA	**RSSA**	026	1
	STEP:	SC by +1	**SC+1**	023	1
		SC by −1	**SC−1**	034	1
		SA by +1	**SA+1**	032	1
	READ OUT SA: to R	low order 8 bits	**RO8SA**	052	2
		low order 16 bits	**RO16SA**	046	2
		entire 24 bits	**RO24SA**	043	2
	READ OUT SC:	low order 8 bits to R	**RO8SC**	063	3
		entire 16 bits to R	**RO16SC**	073	3
		Add to TBA	**SC+TBA**	061	3
INDEXING	RESET:	P — This Level	**RSP**	344	E
		thru FL1	**RSFL1P**	346	E
		FL2	**RSFL2P**	345	E
		FL3	**RSFL3P**	347	E
		Q — This Level	**RSQ**	350	E
		thru FL1	**RSFL1Q**	352	E
		FL2	**RSFL2Q**	351	E
		FL3	**RSFL3Q**	353	E
		R This Level	**RSR**	324	D
		thru FL1	**RSFL1R**	326	D
		FL2	**RSFL2R**	325	D
		FL3	**RSFL3R**	327	D
	ADVANCE:	P — Next Level	**ADP**	340	E
		thru FL1	**ADFL1P**	342	E
		FL2	**ADFL2P**	341	E
		FL3	**ADFL3P**	343	E
		Q — Next Level	**ADQ**	354	E
		thru FL1	**ADFL1Q**	356	E
		FL2	**ADFL2Q**	355	E
		FL3	**ADFL3Q**	357	E
		R — Next Level	**ADR**	320	D
		thru FL1	**ADFL1R**	322	D
		FL2	**ADFL2R**	321	D
		FL3	**ADFL3R**	323	D
TABLE REFERENCE UNIT	Reference TBA-1		**RFTBA-1**	361	F1
	Skip Extraction		**SKTA**	362	F2
	Reset Base Address to TBA		**RSTBA**	364	F4
	Cancel Address		**OMTA**	370	F8
DISABLE	Specified Trigger(s) for this byte		**B12345**	010 $T_5T_4T_3T_2$	
	Specified Trigger(s) for duration of group		**G12345**	011 $T_5T_4T_3T_2$	
	Match Units for Runout		**DISMU**	315	CD
PIPELINE	INSERT	W in LUO	**INW**	214	8C
		X	**INX**	200	80
		Y	**INY**	204	84
		Z	**INZ**	210	88
		MOD	**INMOD**	202	82
		MOD in TE out	**INMODTE**	203	83
	RUNOUT	P — This Level	**ROP**	220	90
		thru FL1	**ROFL1P**	222	92
		FL2	**ROFL2P**	221	91
		Q — This Level	**ROQ**	224	94
		thru FL1	**ROFL1Q**	226	96
		FL2	**ROFL2Q**	225	95
	MATCH ONLY	P — This Level	**MOP**	230	98
		thru FL1	**MOFL1P**	232	9A
		FL2	**MOFL2P**	231	99
		Q — This Level	**MOQ**	234	9C
		thru FL1	**MOFL1Q**	236	9E
		FL2	**MOFL2Q**	235	9D
	STORE ADDRESS	P	**STPS**	240	A0
		Q	**STQS**	244	A4
	OMIT byte after special byte from	P	**OMP**	260	B0
		Q	**OMQ**	264	B4
		TE	**OMTE**	266	B6
	Special byte output of LU		**OMLU**	262	B2
	Special byte into R		**OMR**	263	B3
	REPEAT special byte from	P	**RPP**	270	B8
		Q	**RPQ**	274	BC
MISC	Suppress LU output for duration of group		**SULU**	140	60
	Skip Space in R		**SKR**	304	C4
	Skip remaining TE extractions, this ref.		**SKTE**	310	C8

Note: Hexadecimal (base 16) A=10, B=11, C=12, D=13, E=14, F=15

SETUP

All mnemonics used in the setup word diagrams are system symbols. SA and SC stimuli are in the "Adjustment and Setup Stimuli" Table. Boldface type indicates nonprogrammer symbols and system symbols. Italicized items may be any programmer symbol

32



W (CHARACTER, connection, mode, span), action

Connection Codes

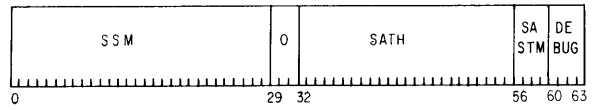
Octal	W	X	Y	Z
0	**NOP**	**NOP**	**NOP**	**NOP**
1	**P**	**P**	**P**	**TE**
2	**Q**	**Q**	**Q**	**LU**
3	**PQ**	**PQ**	**PQ**	**LUTE**
4	—	**LU**	**TE**	—
5	—	**PLU**	**PTE**	—
6	—	**QLU**	**QTE**	—
7	—	**PQLU**	**PQTE**	—

Action Codes

Octal	Code	Action
0	**NOP**	No operation
1	**IN**	Insert char in LUO
2	**OM**	Omit matching byte
3	**OMALL**	Omit all on match
4	**IN. KB1**	Insert and force KB1
5	**IN. MB1**	Insert and force MB1
6	not used	No operation
7	not used	No operation

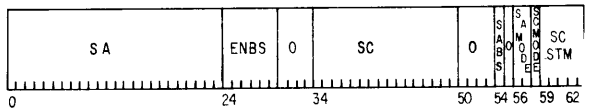
If mode is coded **AND**, a 1 enters WM; if **OR** or blank, a 0 enters WM.
In **AND** mode an adjustment stimulus is emitted only when synchronous simultaneous matches occur or on an LUO match.
Span bit 1 is coded **R**, match on right bit only; **F**, match full byte.
This format is also used for X, Y, and Z match units. All MATCH CHARACTERS must be numeric and are assumed to be binary

33



SSM(mask) mask code mnemonics are in the "Adjustment and Setup Stimuli" Table
DEBUG(mask) code: **BL** — Bit 60 — Any branch level
ADJ — Bit 61 — Any adjustment
FLAG — Bit 62 — Any flag
SCAN — Bit 63 — Must be on to enable a debug scan

34



SA (mode, *threshold*, *value*), stim
SC (*limit*, *value*), stim, action

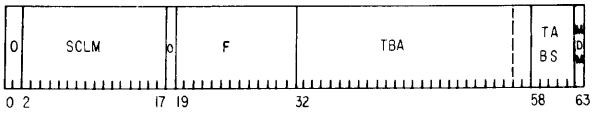
Samode Codes

Octal	Code	Action
0	**U**	Unsigned, 1 byte
1	**U16**	Unsigned, 2 bytes
2	**S**	Signed
3	**SR**	Signed, negative reset

Scmode Codes

Octal	Code	Action
0	**+1**	Step plus one
1	**−1**	Step minus one

35



F (stim), limit, action

F Stim, Limit, and Action Mnemonic Codes

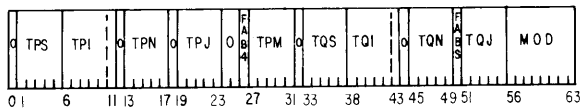
Bit	Code	Meaning	Bit	Code	Meaning
19	**KK**	OKB1.KB1	26	**ML**	OMB1.LB1
20	**KL**	OKB1.LB1	27	**MM**	OMB1.MB1
21	**KM**	OKB1.MB1	28	**I**	Invert with F limit 1
22	**LK**	OLB1.KB1	29	**SO**	Stay On with F limit 1
23	**LL**	OLB1.LB1	30	**1, 2,**	Limit 1 to 4, blank sets
24	**LM**	OLB1.MB1	31	**3, 4**	F limit 1
25	**MK**	OMB1.KB1	28 & 29	**ISO**	Stay On with F limit 1

TBA(*address*, TBAHO, **MDM**)
TBA high order bits — code this in actual: **00, 01, 10, 11.**

Logic Unit Signal Generation Table

	LUOP	KB1 if	LB1 if	MB1 if
0-17	Connective	Result has odd parity	Result zero	Result has even parity and not 0
20-35	Compare or subtract	P > Q	P = Q	P < Q
36	RDXP+Q	P + Q ≥ 256	—	P + Q < 256
37	MODP+Q	suppressed	suppressed	suppressed

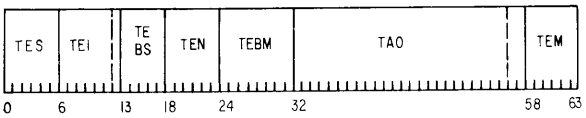
36



TAP (*TPS, TPI, TPN, TPJ*)
TAQ (*TQS, TQI, TQN, TQJ*), TAO, TAOHO, *TPM*
LU (*modulus*, group size), luop

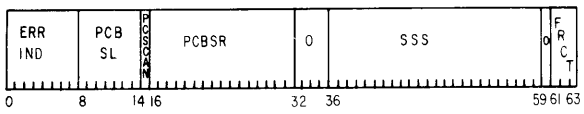
TAO high order bits — code this in actual: **00, 01, 10, 11.**

37



TE (*TEI, TEN,* TEBM, *TES, TEM*)
Note: TEBM must be numeric and is assumed to be binary.

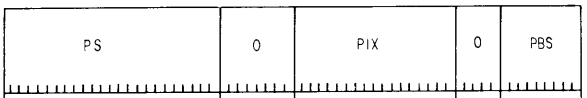
38



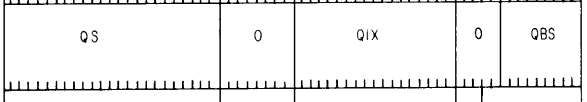
The error indicators are: 0 = P 1 = Q 2 = R 3 = SACC 4 = TA 5 = TE 6 = SLAM 7 = SIGMA

INDEX SETUP WORDS

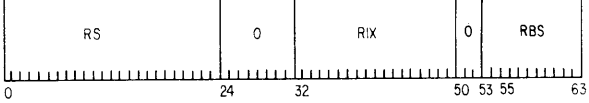
39



40



41



PAD
QAD (*starting data address, initial index table address*)
RAD

OTHER SYSTEM SYMBOLS

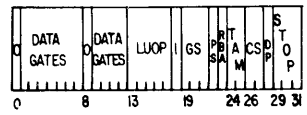
Address	Length	Code	Meaning
3.57	1	**BC**	Boundary control
3.58	1	—	Error inject CPU
3.59	1	—	Error inject HPU
3.62	1	**SI**	Setup interrupt
3.63	1	**FMP**	High performance storage protect
32.	—	**HR**	Setup registers
42.	64	**TE**	Table extract
43.	64	**R1**	R data register 1
44.	64	**R2**	R data register 2

INDICATORS

Address	CPU Code	HPU Code	HPU Meaning
11.25	**IR**	**INC**	Incomplete instruction
11.26	**LS**	**LST**	Lost stimulus
11.27	**PSH**	**EW**	Extract wraparound
11.28	**XPFP**	**MCO**	Memory count overflow
11.29	**XPO**	**SAO**	SA overflow
11.30	**XPH**	**SCO**	SC overflow
11.31	**XPL**	**LWM**	Look up in wrong memory
11.32	**XPU**	**DSCT**	Debug scan taken
11.33	**ZM**	**STK**	STIR check
11.34	**RU**	**RNIF**	Record not in file
11.35	**TF**	**BPR**	Break point reached
11.41	**PG0**	**HMK**	Setup arithmetic error
11.42	**PG1**	**ISSM**	Interrupt on SSM
11.43	**PG2**	**F3P**	EOL-P with flag 3
11.44	**PG3**	**F3Q**	EOL-Q with flag 3
11.45	**PG4**	**F3R**	EOL-R with flag 3
11.46	**PG5**	**DTS**	Delayed time signal

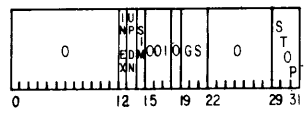
HAP OPERATION CODES

SBBB



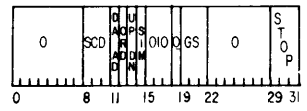
SBBB (data gates), luop, gs, **TA** (mode and cell size, parallel-serial, replace base address, demand parallel synchrony), **STOP** (stimulus), **SETUP** (*name*)

SMER



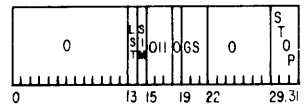
SMER (up-down, internal-external, simple-offset), gs, **STOP** (stimulus)

SSER



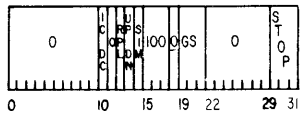
SSER (store data-store address, ordered-random, up-down, simple-offset, search condition), gs, **STOP** (stimulus)

SSEL



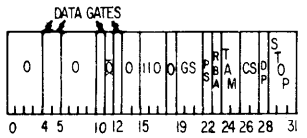
SSEL (least-greatest, simple-offset), gs, **STOP** (stimulus)

STIR



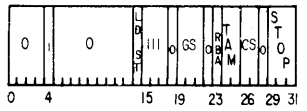
STIR (replace-take, instruction-data control, up-down, simple-offset), gs, **STOP** (stimulus)

SQNL



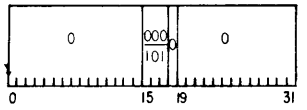
SQNL (data gates), gs, **TA** (mode and cell size, parallel-serial, replace base address, demand parallel synchrony), **STOP** (stimulus), **SETUP** (*name*)

SILS



SILS (load-store), gs, **TA** (mode and cell size, replace base address), **STOP** (stimulus), **SETUP** (*name*)

SNOP



Note: Boldface type is used for nonprogrammer symbols and system symbols. Italicized items may be any programmer symbol

SBBB MNEMONIC CODES

DATA GATES — 12 Bit Field; Bit 8 Always 0

Bit Number	Mnemonic	Bit Number	Mnemonic
1	**P-LU**	9	**LU-R**
2	**Q-LU**	10*	**TE-R**
3	**TE-LU**	11	**LU-SA**
4*	**P-TA**	12*	**TE-SA**
5*	**Q-TA**	6, 7	**SC-TA**
6	**LU-TA**	7, 9	**SC-R**
7	**SC** (bit 7 = 1)	7, 11	**SC-SA**
7	**LU** (bit 7 = 0)		

*Gates which may be used with SQNL

LUOP — Logic Unit Operation Code (5 Bits)

Octal	Mnemonic	Function	Octal	Mnemonic	Function
0	**CZ**	zero	20	**MAXPQ**	Maximum of P and Q
1	**CP.Q**	PQ	21	**MINPQ**	Minimum of P and Q
2	**CP.NQ**	$P\bar{Q}$	22	**EPZ**	P if P=Q or zero
3	**CP**	$P\bar{Q} \vee PQ$	23	**EPNB**	P if P=Q or no byte
4	**CNP.Q**	$\bar{P}Q$	24	**EZP**	P if P≠Q or zero
5	**CQ**	$\bar{P}Q \vee PQ$	25	**ENBP**	P if P≠Q or no byte
6	**CPXVQ**	$\bar{P}Q \vee P\bar{Q}$	26	**EZQ**	Q if P≠Q or zero
7	**CPVQ**	$\bar{P}Q \vee P\bar{Q} \vee PQ$	27	**ENBQ**	Q if P≠Q or no byte
10	**CNP.NQ**	$\bar{P}\bar{Q}$	30	**GEP−QZ**	P−Q if P≥Q or zero
11	**CPEQ**	$\bar{P}\bar{Q} \vee PQ$	31	**GEP−QNB**	P−Q if P≥Q or no byte
12	**CNQ**	$\bar{P}\bar{Q} \vee P\bar{Q}$	32	**LEQ−PZ**	Q−P if Q≥P or zero
13	**CPVNQ**	$\bar{P}\bar{Q} \vee P\bar{Q} \vee PQ$	33	**LEQ−PNB**	Q−P if Q≥P or no byte
14	**CNP**	$\bar{P}\bar{Q} \vee \bar{P}Q$	34	**MODP−Q**	Modular P−Q
15	**CNPVQ**	$\bar{P}\bar{Q} \vee \bar{P}Q \vee PQ$	35	**MODQ−P**	Modular Q−P
16	**CNPVNQ**	$\bar{P}\bar{Q} \vee \bar{P}Q \vee P\bar{Q}$	36	**RDXP+Q**	P+Q no carry out
17	**C1**	$\bar{P}\bar{Q} \vee \bar{P}Q \vee P\bar{Q} \vee PQ$	37	**MODP+Q**	Modular P+Q

GS — Group Size (3 Bits)

Octal	Mnemonic		Octal	Mnemonic	
0	**NOP**	None	4	**FL1R**	Flag 1 in R
1	**FL1P**	Flag 1 in P	5	**W**	Match signal from W
2	**FL2P**	Flag 2 in P	6	**Z**	Match signal from Z
3	**FL1Q**	Flag 1 in Q	7	**XVY**	Match signal from X or Y

TA — Mode and Cell Size (4 Bits)

Binary*	Mnemonic	Function
00xx	**ADTE**	Address is sent directly to TE
01xx	**X**	Extract word from memory, send to TE
1000	**OR**	OR a one into addressed bit in memory
1001	**CT8**	Add a one into addressed bit, cell size 8
1010	**CT16**	Add a one into addressed bit, cell size 16
1011	**CT24**	Add a one into addressed bit, cell size 24
1100	**XOR**	Combination of X and OR
1101	**XCT8**	Combination of X and count
1110	**XCT16**	Combination of X and count
1111	**XCT24**	Combination of X and count

*x may be either 1 or 0

TA — One-Bit Fields

Field	Binary	Mnemonic	Function
PS	0	**PA**	Bytes OR'ed before adding
	1	**SE**	Bytes added sequentially
RBA	0	blank	Form each address on TBA
	1	**RBA**	Form address on preceding address
DP	0	blank	Allow bytes to move asynchronously to TA
	1	**DP**	Force synchronous movement by MU to TA

STOP — Stop Stimulus (3 Bits)

Octal	Mnemonic	Stimulus	Octal	Mnemonic	Stimulus
0	**NOP**	only CC = 0	4	NOP	only CC = 0
1	**FL2P**	flag 2 in P	5	**FL3P**	flag 3 in P
2	**FL2Q**	flag 2 in Q	6	**FL3Q**	flag 3 in Q
3	**FL2R**	flag 2 in R	7	**FL3R**	flag 3 in R

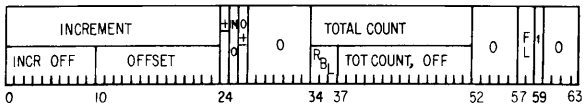
OTHER INSTRUCTION MNEMONIC CODES

Use	Field Name	Binary	Mnemonic	Function
SMER, SSER, STIR	UP	0	**UP**	Files are ordered up
		1	**DN**	Files are ordered down
SMER	IN	0	**IN**	Entire sort is done internally
		1	**EX**	The sort requires external I-O
SMER, SSER, STIR, SSEL	SIM	0	**SIM**	Control field is entire record
		1	**OFF**	Offset control field
SSER	DA	0	**DA**	Entire records are stored
		1	**AD**	Record address is stored
SSER	ORD	0	**ORD**	Records are ordered
		1	**RAN**	Records are not ordered
SSER	SCD	000	—	Invalid — do not use.
		001	**PLQ**	P < Q
		010	**PEQ**	P = Q
		011	**PLEQ**	P ≤ Q
		100	**PGQ**	P > Q
		101	**PNEQ**	P ≠ Q
		110	**PGEQ**	P ≥ Q
		111	—	Invalid — do not use.
SSEL	LST	0	**LST**	Select the least
		1	**GST**	Select the greatest
STIR	RPL	0	**RPL**	Replace matched records
		1	**TAKE**	Delete matched records
STIR	IC	0	**IC**	Instruction control, RPL bit used
		1	**DC**	Data control, ignore RPL bit
SILS	LD	0	**LD**	Address from TE → Q indexing
		1	**ST**	Address from TE → R indexing Either LD or ST must be specified

ARITHMETIC MODE INSTRUCTIONS

BES, *address* (I)	xxxxxx. 24_8 $0x_{16}$	Branch enable to stream initiate
BES(R), *address* (I)	xxxxxx. 64_8 $0x_{16}$	Branch enable to stream resume
CLM(S), *address* (I)	xxxxxx. 32_8 $0x_{16}$	Clear memory small block
CLM(L), *address* (I)	xxxxxx. 72_8 $0x_{16}$	Clear memory large block

COLLATING INSTRUCTION TX TABLE WORDS



First word not offset: **TX**(FL), *byte size in increment, count of bytes in record*

First word offset: **TXO**(FL), *byte size in increment, count of bytes in record, , RBL*

Second word STIR only: **TXO**(FL), 2, 1, *offset of action field*

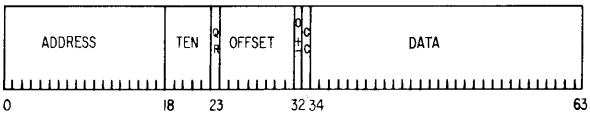
All following words: **TXO**(FL), *byte size in increment, count of bytes in control field, offset of control field, RBL*

Note: TXO sets NO, bit 25, to a one

STIR ACTION FIELD

The two-bit data action field is coded: 0 = insert, 1 = take, 2 = replace, 3 = check

SQNL DATA WORD FORMAT

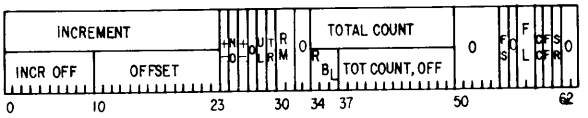


TEY(TEC/TCC, TQ/TR), *address, offset signed, TEN count,* DATA

Note: The initial SQNL data word must be stored in word 42 of setup in the following format: TEY(TEC), *address*

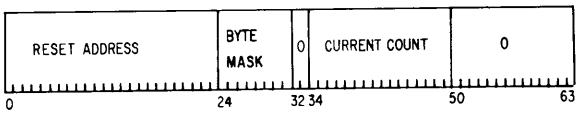
INDEXING WORDS

45 49 53



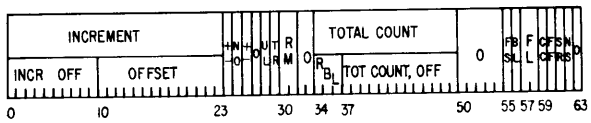
First level, First word for P and Q. R is identical except bit 30 unused.

46 50 54



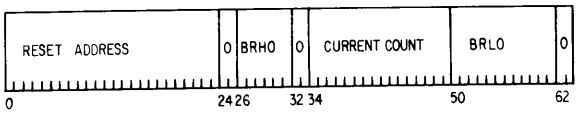
First level, Second word for P, Q, and R.

47 51 55



Higher level, First word.
Note: Identical to First level, First word except for bits 56 and 62.

48 52 56



Higher level, Second word for P, Q, and R.

Non-offset levels:
PX, **QX**, **RX** (mode, EC/CC, FL, FF, SR, BL, Runout or R control, TRU or TRL, FS), *Increment, Total count,* BYTE MASK*/Branch address, Reset address, Current count*

Offset levels:
PXO, **QXO**, **RXO** (same as above), *Increment offset, Total count offset,* BYTE MASK*/ Branch address, Offset, RBL, Reset address, Current count*

Note: BYTE MASK must be numeric and is assumed to be binary

MNEMONIC CODED FIELDS

Field	Binary	Mnemonic	Function
TR	0	—	If zero, ignore UL field. If one, consult UL for triangular indexing.
	1	**TRU** or **TRL**	
UL	0	**TRU**	UL = 0, decrement N
	1	**TRL**	UL = 1, increment N
Runout	00	—	No effect
RM	01	**R**	Run-out directly to R
(for P	10	**M**	Match only
and Q)	11	**RM**	Both R and M
R control	0	—	Data stored directly in R
(bit 31 in R)	1	**RC**	Fetch before storing in R
FS	0	—	First-subsequent toggle. First = 0,
	1	**S**	subsequent = 1. Not set by programmer
BL	1	**BL**	Branch level, must be nested
FL	00	—	Flag control field. 00 = no flag
	01	**FL1**	Flag 1
	10	**FL2**	Flag 2
	11	**FL3**	Flag 3
CC	0	**EC**	End chain
	1	**CC**	Continue chain
FF	1	**FF**	A First level follows this level
SR	1	**SR**	Suppress Reset at end of level
NS(mode)	0	**NES**	Nested mode
	1	**SEQ**	Sequential mode

PROGRAMMER SYMBOLIZED FIELDS

Field	Binary	Function
Increment	24 bits, sign	Amount added to address
Incr Offset	10 bits, sign	Amount added to address (in offset)
Offset	14 bits	Amount of offset
Bit 24	0	Increment sign (+)
	1	Increment sign (−)
Bit 25	0	normal } This bit, NO, is set by either
	1	offset } PXO, QXO, or RXO
Bit 26	0	Offset sign (+)
	1	Offset sign (−)
Total count	16 bits	Total count to be reached
Tot count, off.	13 bits	Total count to be reached (in offset)
RBL	3 bits	Residual Byte Length, for last offset byte
Reset address	24 bits	Location in which initial address is stored
Current count	16 bits	Number of times increment is used
BYTE MASK	8 bits	Byte mask in first levels
Branch	6 bits	Branch address, high order } in higher levels
Address	12 bits	Branch address, low order }

IBM 7950 HAP III Printed in U.S.A. R22-9789

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York