CONTROL DATA® 6000 SERIES COMPUTER SYSTEMS

CHIPPEWA LABORATORIES FORTRAN COMPILER RUN

Preliminary Edition

CONTROL DATA CORPORATION

DEVELOPMENT DIVISION - APPLICATIONS

CHIPPEWA LABORATORIES FORTRAN COMPILER

RUN

April 15, 1966

CHIPPEWA FORTRAN COMPILER - RUN

Revision 1

Description	Add/Insert
Simplified flow charts for the CXP subroutine (pages CXP-4, CXP-5)	Added to CXP subroutine description in section 3
CHAIN subroutine description	Inserted alphabetically in section 3
Sample compilation, sheets 1 - 3	Added sheets 1 - 3 at the end of section 2

CHIPPEWA FORTRAN COMPILER - RUN

TABLE OF CONTENTS

Section 1

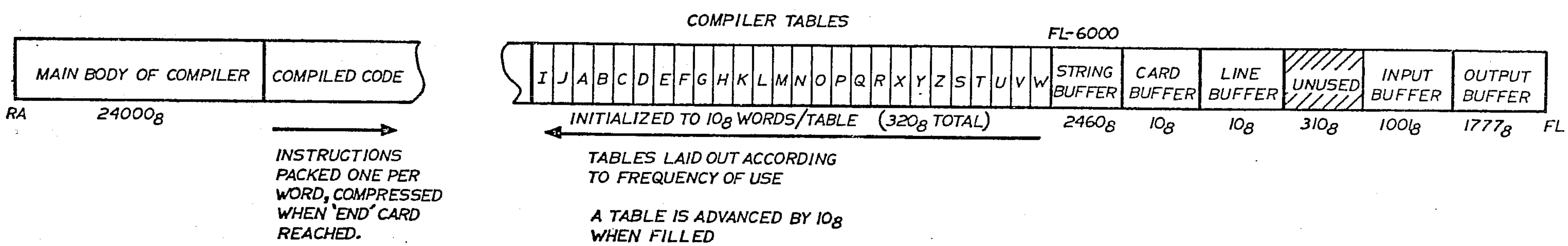
GENERAL DESCRIPTION

INTRODUCTION

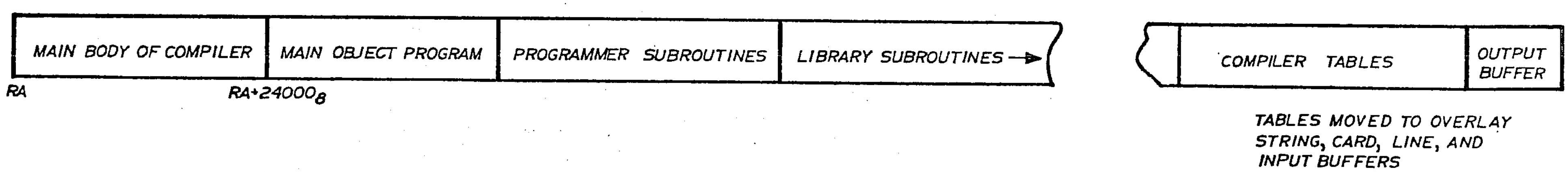
"RUN", the Chippewa Laboratory's FORTRAN compiler for the 6000 series computer systems, generates binary object code directly from FORTRAN II and FORTRAN IV source programs. The compiler also accepts programs written in a subset of the ASCENT assembly language, and in the MACHINE assembly language. The compiler also accepts certain Control Data 3000 series FORTRAN statements, such as ENCODE, DECODE, BUFFERIN, and BUFFEROUT.

The memory layout of the compiler is shown in Figure 1-1. The compiler routines are loaded into memory at location RA, and occupy $24{,}000_8$ locations. Memory space for the compiler buffers and tables, which require approximately 6300_8 locations, is allocated downward from location FL. The minimum space required by the compiler is therefore approximately $32{,}300_8$ locations. Unless adequate space has been reserved for the compiler (by specifying the proper value on the job card), the compiler will exit without attempting to compile.

The Input and Output buffers are used by the compiler in conjunction with the Chippewa Operating System's CIO peripheral package to read in source cards and to list the source and object programs. The transmittal of data between the various buffers is illustrated in Figure 1-2. Source cards from the input file on the disk are read into the Input buffer, which is 1001_8 words in length. As source cards are processed they are transferred, one at a time, into a 10_8-word card buffer. Within the card buffer, the source card is examined to determine if it is a statement card or a comments card. A statement card is transmitted to the string buffer, where it is initially packed one character per word, and another card is brought to the card buffer from the Input buffer. If the next card is a continuation card, it too is transferred to the



INITIAL COMPILER LAYOUT



COMPILER LAYOUT DURING LIBRARY SUBROUTINE LOADING

Figure 1.1

string buffer. This process is repeated until an entire statement has been loaded into the string buffer. Since the size of the string buffer, in which a statement is initially packed one character per word, is 2460_8 or 1328_{10}, the number of continuation cards is limited to 19. All source cards, including both statement cards and comments cards, are transmitted directly from the card buffer to the Output buffer for subsequent listing. Depending upon the mode of compilation, object code instructions may be converted and placed in the line buffer (10_8 words), and from there transmitted to the Output buffer for listing.

Once a statement has been transmitted to the string buffer, the first four characters of the statement are examined to determine the statement type, and the appropriate subroutine is called to process the statement. Since the string buffer is packed one character per word, in most cases the next step is to assemble the contents of the string into a sequence of variables, constants and separators. Since memory assignments cannot be made until all source statements have been processed, variables and constants are replaced in the string by various types of tags, and the variables and constants stored in the compiler tables. (The compiler tables are expanded as entries are made.) The statement is then analyzed and the generated object instructions are packed one instruction per word. In all instructions referencing memory, the K portion of the instruction will at this point contain a tag. Thus, during the compilation of a program, the generated object code expands upward from RA+24000_8, and the compiler tables expand downward from FL-6000_8. When an END card is detected, the generated instructions are packed and memory assignments made. All tags other than those defining external references are replaced with addresses, and the compiler tables are reduced accordingly. Subsequent subprograms are read from the input file and compiled, the object code for each beginning where the object code of its predecessor ended. When the end of the input file is reached, library subroutines are loaded and all subroutine references are replaced with memory addresses. The compiled program is then written on the disk, and the compilation process terminated. Since the string buffer, card buffer, and line buffer are not required for the loading of library routines, the tables are moved up to overlay these buffers before the library routines are loaded (see Figure 1-2).

In processing MACHINE or ASCENT subprograms, the compiler transfers source

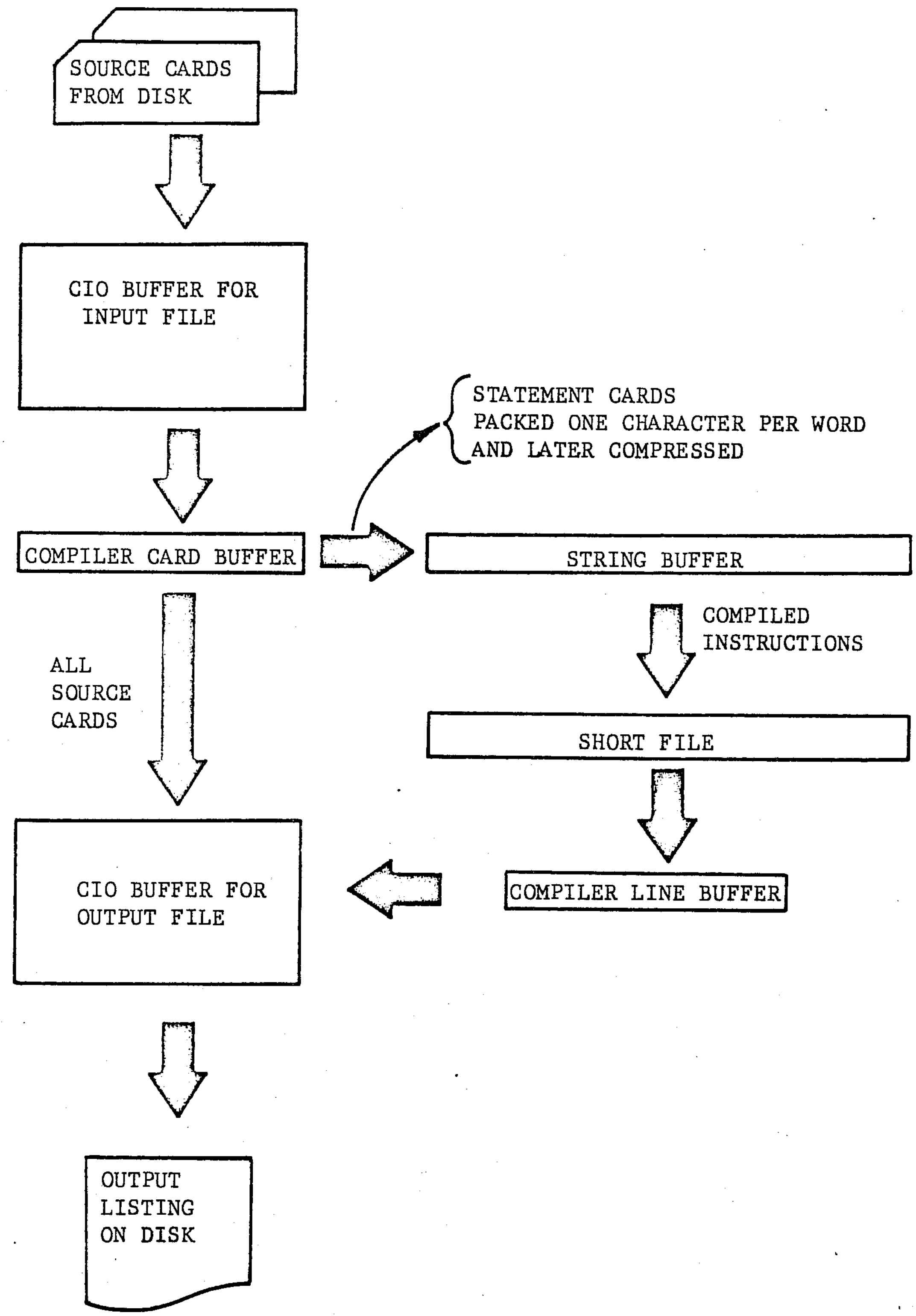


Figure 1-2

COMPILER BUFFER USAGE

cards to the string buffer in the manner described earlier. A subroutine is then called to process the assembly language record. Memory references are tagged, and the appropriate compiler tables entered. The assembled instructions are packed in the object program area. During the processing of the END card, the tags are replaced by memory addresses and constants are transferred from the appropriate table to the program area.

COMPILER TAGS AND TABLES

As constants, variables, subroutine and function names, and statement numbers are encountered in the processing of the source language statement in the string buffer, they are entered in one of the compiler tables and replaced in the string buffer by tags. The tags are entered in tables also, and their relative position within these tables corresponds to the table position of the constnat, variable, external name or statement number which the tag replaced. Many of the compiler tables are used in pairs: the address-dependent quantity is entered in one table, and the tag which replaces it in the string buffer is also entered in the corresponding location in the following table. For example, the Constant Value Table (Table A) and the Constant Tag Table (Table B) are paired tables. When a constant is encountered in the source statement, it is converted to its binary equivalent and entered in the Constant Value Table. A Constant Tag is generated and entered in the string buffer, and also in the Constant Tag Table. Thus, if the constant was entered in location 35 of the Constant Value Table, the tag which replaced it in the string will be entered in location 35 of the Constant Tag Table.

The nine different types of tags used by the compiler are uniquely identified by the value of the high-order five bits of the tag. Tags are advanced as they are assigned: the current value of each tag is maintained in a temporary (TGA, TGB, etc.). All tags, with the exception of library tags, are re-initialized prior to the compilation of each subroutine. Library tags are initialized only when the compiler is first loaded. Tags are usually advanced by one, although they may be advanced by two when a double-precision or complex value is entered in a table. Library tags are advanced by 100_8. The types of tags used by the compiler, and the numeric ranges which these tags may assume, are listed below.

Temporary Containing Current Tag Value	Tag Type	Tag Values	Listing Indicator
TGA	Program Tag	200000-217777	L
TGI	Indirect Tag	220000-237777	I
TGT	Temporary Tag	240000-257777	T
TGK	Constant Tag	260000-277777	C
TGF	Function Tag	300000-317777	F
TGW	Array Tag	320000-337777	A
TGV	Variable Tag	340000-357777	V
TGH	Statement Tag	360000-377777	N
TGL	Library Tag	400000-600000	S

The listing indicator appears on the object code listing preceding the tag number as specified in the low-order 15 bits of the tag. The tag number as given in the low-order 15 bits may range from 0 to 17777_8. Thus, the compiler permits up to 8192 tags of each type.

Constants, variables, subroutine and function references, and statement numbers are entered in the compiler tables when they are encountered in the string buffer, as are the tags which replace these quantities. There are 26 of these tables. The location of these tables is shown in the compiler layout illustration of Figure 1-1. The size of each table is initially set at 10_8 words: as tables are filled, they are expanded by 10_8 words: tables at lower memory locations are moved down to provide room for the increase. Associated with each table is a temporary which contains the parameters required to enter, search, and expand the tables. The format of these temporaries (which are labelled in the compiler as TBA, TBB, etc.) is shown below.

TBn		parameter word address	starting address	next entry address
		36	18	0

Note that this word contains its own address. This permits table scanning routines, which are entered with this word in an X register, to readily obtain the parameter word for the succeeding table in memory (i.e., parameter word address + 1 = parameter word address for the succeeding table).

TABLE TAG	TABLE NAME	TABLE ENTRY FORMATS (characters 10 … 1)
TBA	CONSTANT VALUE	CONSTANT
TBB	CONSTANT TAG	K-TAG
TBC	TEMPORARY TAG	H-TAG
TBD	PERMANENT TAG	A- OR K-TAG
TBE	FUNCTION NAME	FUNCTION NAME
TBF	FUNCTION TAG	F-TAG / AC / AT / M
TBG	DO NUMBER	STATEMENT NUMBER
TBH	DO PARAMETERS	INDEX ADDRESS / INCREMENT / LIMIT
TBK	STATEMENT NUMBER	STATEMENT NUMBER
TBL	STATEMENT TAG	A-,H-OR K-TAG
TBM	VARIABLE NAME	VARIABLE NAME
TBN	VARIABLE TAG	A-,V-,OR W-TAG / M
TBO	COMMON NAME	COMMON NAME BLOCK NAME
TBP	ARRAY TAG	A- OR W-TAG / M
TBQ	ARRAY PARAMETERS	1 / LENGTH 2 / DIMENSION 1 / LENGTH 3 / DIMENSION 1 / DIMENSION 1×2 / LENGTH 1 / A-TAG 2 / A-TAG / A-TAG 3 / A-TAG / A-TAG / A-TAG (FOR VARIABLE DIMENSIONS)
TBR	DATA STATEMENT	TRANSLATED DATA STATEMENTS
TBX	EQUIVALENCE 2ND NAME	SECONDARY NAME
TBY	EQUIVALENCE 1ST NAME	PRIMARY NAME / BASE ADDRESS
TBZ	EQUIVALENCE BIAS	BIAS
TBS	SUBROUTINE NAME	SUBROUTINE NAME / LENGTH
TBT	SUBROUTINE TAG	L-TAG / M
TBU	SUBROUTINE PARAMETER	CODE LENGTH / TOTAL LENGTH / STARTING ADDRESS / N
TBV	COMMON BLOCK	BLOCK NAME / STARTING ADDRESS LIMIT ADDRESS (2-WORD ENTRY)
TBW	PROGRAM FILE NAME	FILE NAME / ARGUMENT ADDR. BUFFER LENGTH (2-WORD ENTRY)
TBI*	ARGUMENT NAME	FUNCTION ARGUMENT NAME
TBJ*	ARGUMENT TAG	F-TAG / M

Field widths: 6 6 6 6 6 6 6 6 6 6

M=MODE
AC=ARG. COUNT
AT=ARG. TRANS. IND.

N=NO. OF ARGUMENTS

TAG	BASE VALUE	TYPE	LISTING INDICATOR
TGA (A-TAG)	200000	PROGRAM TAG	L
TGI (I-TAG)	220000	INDIRECT TAG	I
TGT (T-TAG)	240000	TEMPORARY TAG	T
TGK (K-TAG)	260000	CONSTANT TAG	C
TGF (F-TAG)	300000	FUNCTION TAG	F
TGW (W-TAG)	320000	ARRAY TAG	A
TGV (V-TAG)	340000	VARIABLE TAG	V
TGH (H-TAG)	360000	STATEMENT TAG	H
TGL (L-TAG)	400000	LIBRARY TAG	S

MODE (M)	TYPE
0	NON MODE
1	LOGICAL
2	FIXED-POINT
4	FLOATING-POINT
5	DOUBLE-PRECISION
6	COMPLEX

* THE I AND J TABLES ARE USED AS UTILITY TABLES BY A NUMBER OF ROUTINES, SUCH AS THE END STATEMENT PROCESSOR

COMPILER TABLES

Figure 1.3

The format of the table entries is shown in Figure 1-3. Note that some tag tables may hold more than one type of tag. Although there are 26 tables, several tables are used as paired tables (see table descriptions) and so, counting these tables as single multi-word entry tables, we may consider the compiler tables to be functionally fourteen in number.

The compiler tables, and the manner in which the tags and tables are used in the compilation process, are briefly described below.

Constant Value Table;
Constant Tag Table: When a constant is encountered in the translation of a source language statement, it is converted to its equivalent binary form and entered in the Constant Value Table. A Constant Tag (K-tag) is generated, and entered in the corresponding location in the Constant Tag Table. The Constant Tag replaces the original constant in the string. The instruction compiled to fetch the constant will be of the form SAi = K-tag. Constants which are integer or octal in mode, and less than 2^{16}-1 in absolute value, are not entered in the Constant Value Table (unless they are subroutine arguments). The instruction compiled for a constant of this type will be of the form SXi = K, where K is the constant value. The Constant Value Table is also used to store format statements during compilation. The format descriptors are packed in consecutive words of the Constant Value Table, 10 characters per word, and a Constant generated and entered in the Constant Tag Table for each word. Subsequent references to the format statement will be compiled with the Constant Tag associated with the first word of the format statement. Prior to entering a constant in the Constant Value Table, the table is scanned to determine if a constant of the desired value has been defined earlier and, if so, the tag associated with the earlier entry is used rather than generating a new tag and a new entry.

Temporary Tag Table;
Permanent Tag Table: These tables are most commonly used when a source program statement references another part of the program which has not yet been compiled, e.g., a GO TO n statement where statement n has not yet been processed. Reference points within the compiled program are defined by program tags (A-tags). The Temporary Tag Table provides a means of recording references to points as yet undefined in the program: The manner in which this is accomplished is illustrated in Figure 1.4.

GØ TO 10

INITIAL STATEMENT PROCESSING

- ENTER "10" IN THE STATEMENT NUMBER TABLE (TABLE K)
- GENERATE A STATEMENT TAG (H-TAG) AND ENTER IT IN THE CORRESPONDING LOCATION IN THE STATEMENT TAG TABLE (TABLE L)
- ENTER THIS STATEMENT TAG IN THE TEMPORARY TAG TABLE (TABLE C): RESERVE CORRESPONDING ENTRY IN THE PERMANENT TAG TABLE (TABLE D)
- COMPILE INSTRUCTION: 0400 H-TAG

WHEN STATEMENT 10 IS ENCOUNTERED

- GENERATE PROGRAM TAG (A-TAG) FOR FIRST INSTRUCTION COMPILED FOR THIS STATEMENT
- SEARCH STATEMENT NUMBER TABLE FOR THIS STATEMENT NUMBER: WHEN FOUND, GET CORRESPONDING ENTRY FROM STATEMENT TAG TABLE
- SEARCH THE TEMPORARY TAG TABLE FOR THE STATEMENT TAG (H-TAG) OBTAINED FROM THE STATEMENT TAG TABLE: WHEN FOUND, ENTER THE PROGRAM TAG (A-TAG) IN THE CORRESPONDING LOCATION IN THE PERMANENT TAG TABLE
- REPLACE THE STATEMENT TAG (H-TAG) IN THE STATEMENT TAG TABLE ENTRY CORRESPONDING TO THE STATEMENT NUMBER WITH THIS PROGRAM TAG (A-TAG)

USE OF TEMPORARY & PERMANENT TAG TABLES: EXAMPLE

Function Name Table;

Function Tag Table: When an arithmetic statement function is encountered, the function name is entered in the Function Name Table, and a Function Tag is generated and entered in the Function Tag Table. Subsequent references to this function name are replaced in the string by the Function Tag. When this tag is processed, instructions are compiled to pass the arguments, and then an RJ F-tag is compiled to enter the coding compiled for the arithmetic statement function.

DO Number Table;

DO Parameter Table: When a DO statement is encountered, the statement number which terminates the DO loop is entered in the DO Number Table. Instructions are then compiled to initialize the DO index. The address of the SA6 instruction compiled to store the initial value in the index (i.e., the index store), and the Variable tags (or constants) for the increment and limit values are stored in the DO Parameter Table. Each time a statement number is processed, the DO Number Table is scanned to determine if the statement number terminates a DO loop. If it does, the entry in the DO Parameter Table corresponding to the statement number is obtained, and the limit and increment tags (or constants) used to compile the index increment and test instructions. When the index store instruction was compiled, it was tagged with a Program tag (A-tag). In processing the statement number which terminates the DO, the address of the index store instruction is obtained from the DO Parameter Table entry, and a PL or NG A-tag instruction compiled to provide the loop return.

Statement Number Table;

Statement Tag Table: Whenever a statement number is encountered in a source language statement, it is entered in the Statement Number Table (unless previously entered), and a tag is entered in the corresponding entry in the Statement Tag Table. This tag may be a Statement Tag, a Program tag, or a Constant tag. If the first reference to the statement number defines it (i.e., if it is first encountered in the statement number field), a Program tag (A-tag) is entered in the Statement Tag Table. This Program tag will also be used to tag the first compiled instruction for the statement which had this number. The case where the statement number is referenced before it is defined was discussed earlier (see Figure 1.4). In this case, a Statement tag (H-tag) is generated and entered in the Statement Tag Table. This tag is later equated to a Program tag through the use of the Temporary and Permanent Tag Tables. If the statement number refers to a format statement, a Constant tag is entered in the Statement Tag Table. This Constant tag defines the starting location of the format statement in the Constant Value Table.

Variable Name Table;
Variable Tag Table: When a variable is first encountered in a source program, the variable name is entered in the Variable Name Table, and a tag is entered in the corresponding location in the Variable Tag Table. If the variable is not dimensioned, a Variable tag (V-tag) is generated and entered in the Variable Tag Table. If the variable is first encountered in a DIMENSION statement, an Array tag (W-tag) is generated and entered in the Variable Tag Table. Should the variable be an argument in a subroutine, a Program tag is entered in the Variable Tag Table. This Program tag will indicate where this value is located in the subroutine argument list which follows the subroutine entry point.

Common Name Table: When a COMMON statement is encountered, it is entered in the Common Name Table to be subsequently processed when the END statement is encountered. Common block names appear in the lower 42 bits, while common variable names appear in the upper 42 bits.

Array Tag Table;
Array Parameter Table: When a variable is encountered in a DIMENSION statement, the variable name is entered in the Variable Name Table, and an Array tag is entered in the corresponding entry in the Variable Tag Table. This Array tag is also entered in the Array Tag Table, while the dimension parameters are entered in the corresponding location in the Array Parameter Table (see Figure 1.3 for the format for 1, 2, and 3-dimensional arrays). If the dimensions are variables, the Array Parameter Table will contain a Program tag (A-tag) for each dimension parameter. This program tag will indicate where this value is in the argument list which follows the subroutine entry point. If the dimensions are constants, the Array Parameter Table will contain the values of the dimensions and (for 3-dimensional arrays) dimension product.

Data Statement Table: When a DATA statement is encountered, it is partially translated (i.e., variables are replaced by tags, constants are converted, etc.) and entered in the Data Statement Table to be subsequently processed when the END statement is encountered.

Equivalence Second Name Table;
Equivalence First Name Table;
Equivalence Bias Table: When an EQUIVALENCE statement is encountered, the variable names and bias values specified in the statement are entered in the equivalence tables. The equivalence tables are processed when the END statement is encountered.

Subroutine Name Table;
Subroutine Tag Table;
Subroutine Parameter Table: When a SUBROUTINE, FUNCTION, or CALL statement is encountered, or when a function subprogram reference is found, the subprogram name is entered in the Subroutine Name Table. A Library tag (L-tag) is generated and entered in the corresponding location in the Subroutine Tag Table. When the subprogram is compiled, the length of the compiled code, the total length (including compiled code, constants, local variables, etc.), the starting address, and the number of arguments are assembled into a single word and this word is entered in the Subroutine Parameter Table entry which corresponds to the subroutine name. The first entry made in each of these tables is for the main program.

Common Block Table: The Common Block Table is one of the tables in which entries occupy two consecutive locations. When the COMMON statement is first encountered, the common name and the block name are entered in the Common Name Table. During the processing of the END statement,' the block name, together with the starting and ending address of the block, is entered in the Common Block Table in the format shown in Figure 1.3.

Program File Name Table: When a PROGRAM card is encountered, the arguments on the card are entered in the Program File Name Table in two consecutive words. The first word contains the file name and the address of the CIO parameters for the buffer assigned (the CIO parameters occupy the first 10_8 words of the buffer area). The second word of the Program File Name Table contains the buffer length for the file. If no buffer length is specified, this length entry is set to 2010_8. (This includes the space occupied by the CIO parameters.)

Argument Name Table;
Argument Tag Table: Although the nominal purpose of these tables is to assist in the processing of arithmetic statement functions, the Argument Name and Argument Tag Tables serve as compiler utility tables, and are used for a variety of purposes. For example, these tables are used in processing the EQUIVALENCE statement, in computing array references, and in processing the END statement.

While most of the compiler tags define quantities appearing in a source statement, the program tag (A-tag) is used to define locations within the compiled object code. During the compilation process, the generated instructions are packed in the upper 30 bits of a word, one instruction per word. All address fields in the generated instructions contain tags of various types. When the END statement is encountered, these tags are replaced with addresses and the object code is then compressed. When instructions are compiled for a statement which has a statement number, a program tag is entered in the low-order 18 bits of the word containing the first instruction compiled for this statement. For example, a statement such as I = 0 might be compiled as

SX6 = 0	
SA6 = V-tag	

if it were an un-numbered statement. If, however, this statement had a statement number, it might be compiled as

SX6 = 0	A-tag
SA6 = V-tag	

This program tag serves two purposes. First, a source program transfer of control to the numbered statement will result in the compilation of a jump instruction in which the address field contains this program tag. When the END statement is processed and the object code is compressed, the program tag in instructions referencing the first instruction compiled for the numbered statement will be replaced with the address of this first instruction. Secondly, since the appearance of a program tag in the low-order bits of a word containing a compiled instruction indicates that this instruction is referenced elsewhere in the program, the instruction must be forced to the upper parcel(s) of a word when the object code is compressed.

Program tags are also used to define subroutine parameters. Since all tags (with the exception of library tags) are initialized at the beginning of subroutine compilation, there is a fixed relationship between the value of the program tag and the parameter number, as shown below.

Subroutine Word No.	Contents	Program Tag
1	Subroutine Name	L00002
2	Number of Arguments	L00003
3	Parameter 1	L00004
4	Parameter 2	L00005
⋮	⋮	
n	Parameter m	L0000(M+3)
n+1	Entry Line	L00001

Since the first six parameters (i.e., argument addresses) of a subroutine are also passed in a B register, there is a fixed relationship between the parameter number, the register in which it appears, and the program tag assigned to the parameter.

Register Associates

To assist in minimizing the number of fetches generated, the compiler utilizes 19 temporaries called Register Associates. These temporaries are associated with registers A0-A5, B1-B7, and X1-X6. As instructions are compiled for a source language statement, these Register Associates are updated to reflect the contents which the X, A, and B registers will have during the execution of the object program. For example, suppose an SA2 instruction is compiled to fetch a variable. The address field of the SA2 instruction will contain a variable tag (V-tag) which will be replaced by an address during the processing of the END statement. When this instruction is compiled, this variable tag is entered in the X2 and

A2 Register Associates, indicating that the X2 register contains the value of the variable while the A2 register contains its address. Before compiling a subsequent fetch, the Register Associates are examined to determine if the value is already available or, failing that, if the address is available and a 15-bit fetch instruction can be generated (i.e., in place of a 30-bit fetch instruction). Should subsequent instructions be compiled which use the X2 register as a result register, the X2 Register Associate will be cleared.

Compiler Master Loop

The flow chart for the compiler master loop is shown on pages A-1 and A-2 of Appendix A. The master loop may be considered as being composed of an outer loop and an inner loop. The outer loop controls program and subprogram processing, while the inner loop controls statement processing. The main functions of the compiler master loop are described below.

COMPILER INITIALIZATION

After clearing the Chain and Error indicators, the compiler calls the peripheral processor package "CHK" to determine the status of the OUTPUT file. This status is subsequently used to determine what, if any, repositioning of this file is required. The compiler then picks up the field length from the A0 register and, unless a field length of at least 32000_8 words was specified, immediately exits. If the specified field length was adequate, the Initialize for Input/Output (IIO) routine is called to set up the compiler buffers and to process the compiler arguments from the RUN card. These arguments are passed to the compiler in locations RA+2, RA+3, etc., during the loading of the compiler. The order in which these arguments appear, and the value assigned by the compiler if an argument is omitted, are shown in Figure 1-5. IIO also enters the string buffer starting address (FL-6000) in the A0 register, where it will remain for the duration of compilation. Next, the Read Next Card (RNX) subroutine is called to bring the first card to the

COMPILER ARGUMENTS

		VALUE IF NOT SPECIFED
RA + 8	LINE LIMIT	20000_8
RA + 7	OUTPUT FILE	"OUTPUT"
RA + 6	INPUT FILE	"INPUT"
RA + 5	BUFFER LENGTH	2010_8
RA + 4	COMMON LENGTH	AS PER MAIN PROG.
RA + 3	PROGRAM LENGTH	JOB LENGTH
RA + 2	COMPILE MODE	"G"
RA + 1	0————0	
RA	0————0	

NOTE: THE FIELD LENGTH FROM THE JOB CARD IS IN A_0 ON ENTRY.

Figure 1-5

Card Buffer. This routine is used throughout the compiler to transfer a card from the Input Buffer to the Card Buffer and, if the Input Buffer is empty, to initiate a CIO call to fill the buffer. The Initialize Program Tables (IPT) subroutine is called to initialize the Library tag, set the Short File Start and Long File Start, and to set up the Subroutine Parameter, Common Block, and Program File Name Tables. These tables are initialized only at the beginning of compilation: all other tables, with the exception of the Argument Name and Argument Tag Tables, are initialized each time a program or subprogram is compiled. The Argument Name and Argument Tag Tables are initialized as required.

SUBPROGRAM INITIALIZATION

The Initialize Subroutine Tables (IST) routine is called next. This subroutine initializes the remaining tables and tags, and sets A7 to the Short File Start address. During compilation, the A7 register will always contain the address of the last instruction compiled and X7 will be used to store instructions as they are compiled. (Note: the return to compile the next subroutine is made to this point in the master loop.) RNX brought the first card into the card buffer: this card is examined to determine if it contains a + or - in column 1. If it does, then the following program or subroutine is not a source program but a binary deck, and control is transferred to the END statement processor which will load the binary object deck, extract any external references, and enter these in theSubroutine Name Table. If the first card does not contain a + or - in column 1, the Assemble FORTRAN Statement (AFS) subroutine is called. This subroutine transfers source cards from the card buffer to the string buffer, packing one character from the card into one word in the string buffer. AFS also transmits the source card to the Output Buffer (via the WNX routine) for listing. Next, AFS brings the next card to the card buffer and examines it to determine if it is a continuation card. If it is, is also is loaded in the string buffer. If the next card is a comments card, AFS transmits it to the Output Buffer. This process is repeated until AFS finds a non-comments, non-continuation card in the card buffer.

DETERMINING COMPILATION MODE

The first seven letters of the statement assembled in the string buffer by AFS are examined to determine if they are ASCENTF, MACHINE, or FORTRAN. If these letters are ASCENTF or MACHINE, the subprogram mode indocator (and subsequently the program mode indicator, if this is a main program) is set to -2 or -1, respectively. If these letters are FORTRAN, the next two letters are examined to determine if they are II, IV, or VI, and the mode indicator(s) set accordingly. If the card does not begin with ASCENTF, MACHINE, or FORTRAN, a FORTRAN IV compilation is assumed. (This mode is set by the Initialize Program Tables subroutine.) If these letters appeared on the first card, they are blanked out and the next seven letters are assembled. These letters are compared with entries in the table of Program Title Types: PROGRAM, SEGMENT, SUBROUT, FUNCTIO, END, and BLOCKDA. If not found in this table, the letters are examined to determine if they are the FUNCTION predecessors DOUBLE PRECISION, DOUBLE, READ, INTEGER, LOGICAL, or COMPLES. If they are, the function type indicator is set accordingly, these letters are blanked out, and the next seven letters assembled and checked. The header card is passed on to the inner portion of the compiler's master loop for processing. If the card was a PROGRAM card, the Program/Subprogram Indicator is set to zero: otherwise, it is set to a non-zero value. This indicator is examined by the END statement processor to determine if tags should be replaced by absolute memory addresses or by memory addresses relative to the start of the subprogram when subroutines are separately compiled.

STATEMENT PROCESSING INITIALIZATION

The statement processing portion of the compiler master loop is now entered. (The program title card, or header card, is passed on to this portion of the compiler master loop for processing.) Various statement-related flags and indicators are cleared, and a program tag is generated and saved as the Current Program Tag. The Get Statement Number (GSN) routine is called to assemble the statement number, if any, associated with this statement. Processing of this statement number will be performed after this statement has been compiled.

STATEMENT RECOGNITION

Column 1 of the source card is checked to see if it contains an F (FORTRAN II external function) and, if it does, the Process Function Name (FUN) subroutine is called to process the function. If the statement is not a FORTRAN II external function, the first two letters are examined and, if these letters are DO. If they are, the Sense DO Statement (SDO) subroutine is called to determine if the statement is a DO statement and to initiate DO statement processing. If the statement is not a DO statement, the Sense Formula (SFO) subroutine is called to determine if the statement is an arithmetic statement and to initiate arithmetic statement processing. If the statement is not a FORTRAN II external function, a DO statement, or an arithmetic statement, the first' four letters of the statement are assembled and used to scan a Statement Letter group Table. If the statement is not found in this table, and the four letters are not TYPE, a Format Error diagnostic is generated. If the statement is found in this table, the Current

Jump and Continue indicators are processed, and a jump table used to transfer control to the appropriate statement processing routine. (Note that the relative location of the statement within the Statement Letter group table indicates if the statement is executable or non-executable.

RETURN FOR NEXT STATEMENT

Statement processing routines generally enter the Process Statement Number (PSN) subroutine upon completion of statement processing, and this routine in turn returns control to the compiler master loop. If the program title card called for a MACHINE or ASCENTF assembly, the Process Machine/Ascent Records (MAA) subroutine is called.

CHIPPEWA FORTRAN COMPILER - RUN

Section 2

STATEMENT PROCESSING

INTRODUCTION

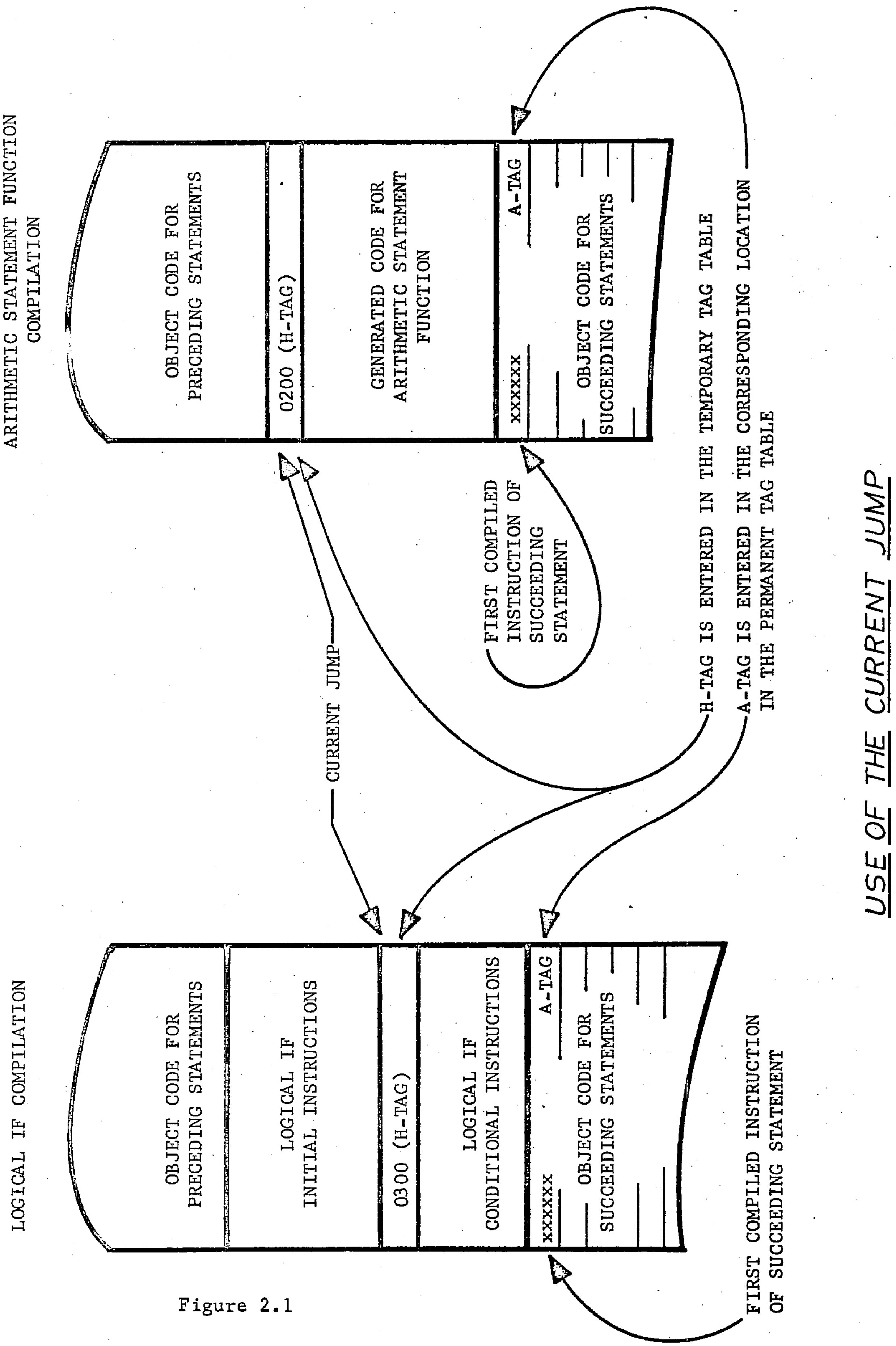
At the time the statement processing routine is entered from the compiler master loop, the source language statement in the string buffer is packed one character per word. Most statement processing routines call the Normalize Statement (TAB) subroutine. The TAB subroutine (see TAB description in Section 3) assembles the contents of the string buffer into a series of words containing variable, constants (which may occupy more than one word) and separators. The TAB subroutine also enters the format statement in the Constant Value Table. Next, the Translate Individual Quantities (TIQ) subroutine is called. TIQ translates the contents of the string buffer into a sequence of tags, separators, and constants which can easily be manipulated by the statement processing routines. The TIQ subroutine is discussed in Section 3.

There are two indicators which are tested during the processing of most executable statements as well as in the compiler master loop. One of these is the Current Jump Indicator, which is used in the processing of arithmetic statement functions and logical IF statements. Since arithmetic statement functions may occur anyplace within the source program, it is necessary to compile a jump over the code generated for a function. Therefore, before compiling the instructions for the function, a Statement Tag (H-tag) is generated and entered in the Temporary Tag Table, the corresponding entry in the Permanent Tag Table is reserved, and an 0400 H-tag instruction is compiled for the jump over the generated code for the function. The Current Jump Indicator is set to the address of this jump within the compiled code. Similarly, in the case of the Logical IF statement, a jump instruction over the coding generated for the TRUE condition must be compiled. In this case, an 0200 H-tag is compiled and the Current Jump Indicator set to the address of this jump instruction. The Current Jump Indicator is carried along during the processing of non-executable statements and other arithmetic statement functions. When an executable statement is encountered, a Program tag is used to tag the first instruction compiled for this statement. The address contained in the Current Jump Indicator is used to obtain the jump instruction, and the Statement tag (H-tag) is extracted from the jump instruction. The Temporary Tag Table is then searched for the entry containing this tag, and the Program tag (A-tag) entered in the corresponding location in the Permanent Tag Table, thus equating the two tags. The use of the Current Jump Indicator is illustrated in Figure 2.1.

A second indicator which must be examined when a statement is compiled is the Continue Indicator. If a CONTINUE statement is encountered which does not terminate a DO loop, the Continue Indicator is set. This indicates that the A-tag which otherwise would be generated for the CONTINUE statement is instead to be assigned to the first executable statement following the CONTINUE statement. This is accomplished by the compiler master loop which, during the initialization performed for each statement, generates a Program tag and enters it in the X7 register, thus tagging the first instruction compiled for the next statement. If the Continue Indicator is set, this tag is left in X7 for subsequent processing. If the Continue Indicator is not set, the X7 register is cleared before a transfer to the statement processing routine takes place.

HEADER CARDS

Five different types of header cards are acceptable to the FORTRAN compiler: PROGRAM, SEGMENT, FUNCTION, SUBROUTINE, and BLOCKDATA. All but the last may have formal parameters included on the card. BLOCKDATA is a special type of subprogram which contains only declarative statements. PROGRAM and SEGMENT are closely related because the name appearing on the card of either is the identifying name of file on the disk containing the object program as the first record. Each of these files may contain many SUBROUTINE and/or FUNCTION subprograms.



USE OF THE CURRENT JUMP

Figure 2.1

PROGRAM and SEGMENT differ in that numbered or blank common and the I/O buffers are initialized with a PROGRAM declaration but not with SEGMENT. Both are compiled to be read in from the disk beginning at RA. Any SEGMENT called will completely overlay the main program and its subroutines, but the I/O buffers and common will not be disturbed. A SEGMENT may be called repeatedly by any other segment or from the main file, and arguments may be transferred through COMMON. Any SUBROUTINE or FUNCTION referenced within a SEGMENT must be compiled with it because no portion of the main program or previous segment is available for use. The maximum number of I/O files referenced by the main program or any segments called must be declared on the PROGRAM card because only in this way is the buffer reserved for the file. All segments to be chained must be compiled with the same file names.

If no length is specified on either the RUN card or PROGRAM card for the buffers, then 2010_8 words are reserved for each file declared. An individual buffer length may be specified on the PROGRAM card which will override that specified on the RUN card, but neither length may be less than 1001_8 words. Equivalenced files will utilize the same buffer. Instructions are compiled to initialize buffer parameters, to set unused memory space to indefinites and blank or numbered common area cleared to zero upon encountering the PROGRAM card.

The mode of the FUNCTION is set from the preceding type declaration or by checking the first character of the name. This mode must agree with the type from a previous reference or the diagnostic "FUNCTION TYPE ERROR" results. An "ARGUMENT COUNT ERROR" identifies a previous call requiring more arguments than are being compiled. All arguments on a SUBROUTINE or FUNCTION card receive a location tag pointing to a reserved word beginning at the third relative word of the subprogram. Since the addresses of the first six arguments are passed to the subprogram via index registers B1-B6, instructions are compiled to pack the address, three per word, into two temporary words. B1 and B4 occupy the lower 18 bits of the two words with B2 and B3 packed in the next 18 bit portions of the first word. B5 and B6 reside in the second word in the same relation as B2 and B3.

A more detailed discussion of this initialization process is contained in the Process Name and Arguments (PPG) subroutine description in Section 3.

DO STATEMENT PROCESSING

In initialing the processing of a source language statement, the compiler master loop checks the first two letters of the FORTRAN statement to see if they are "DO". If they are, a routine called Sense DO Statement (SDO) is called to determine if the statement is actually a DO statement. The basic steps performed by SDO are tabulated in Figure 2.2. SDO scans the first part of the statement to determine if it has the sequence

$$\left[\text{DO}\right] \left[\text{number}\right] \left[\text{variable}\right] \left[=\right]$$

If this sequence is found, SDO scans the Variable Name Table for the variable and, if found, examines the corresponding tag from the Variable Tag Table to determine if the mode indicator for the variable is a 2 (integer mode). If the variable is not in the Variable Name Table, the first letter of the variable is checked to determine if it is I, J, K, L, M, or N. If the above sequence is found, and if the variable is an integer variable, SDO assumes that the statement is a DO statement, and proceeds with DO initialization processing.

The TAB subroutine is called to normalize the statement. Since executable instructions will be compiled to initialize the DO loop, the Current Jump and Continue indicators are processed. If the DO statement itself was numbered, a Program Tag (A-tag) is entered in X7 to tag the first executable instruction of the DO statement. Next, the TIQ subroutine is called to translate the statement into a sequence of separators, tags, and constants. TIQ will make any necessary entries in the Variable Name and Constant Value Tables.

The CDI (Compile DO Initial Instructions) subroutine is then called to compile the DO loop initialization instructions. The basic steps performed by this routine are illustrated in Figure 2.2. CDI enters the statement number (of the DO termination statement) in the DO Number Table (Table G), and then examines the string entry for the initial value (i.e., n_1) to determine if the initial value is a variable or a constant. If the initial value is a constant, CDI compiles an SX6 = K instruction to set the initial value. Should the initial value be a variable (i.e., as indicated by a variable tag in the string) CDI calls

DØ s $i = N_1, N_2, N_3$

SDO (SENSE DO STATEMENT)

- SENSE LETTERS "DØ"
- SENSE NUMBER IN NEXT FIELD
- SENSE VARIABLE IN NEXT FIELD
- SENSE VARIABLE MODE
- SENSE EQUAL SIGN
- NORMALIZE STATEMENT
- TRANSLATE STATEMENT
- CALL CDI TO COMPILE INITIAL INSTRUCTIONS

CDI (COMPILE DØ INITIAL INSTRUCTIONS)

- ENTER STATEMENT NUMBER IN DØ NUMBER TABLE (TABLE G)
- IF n_1 IS A CONSTANT:

  COMPILE SX6 = CON TO INITIALIZE THE INDEX

  IF n_1 IS A VARIABLE:

  COMPILE SA_i = V-TAG OR $SA_i = B_j$ TO FETCH INITIAL VALUE UNLESS ALREADY AVAILABLE IN AN X REGISTER
- COMPILE BX6 = X_j TO BRING INITIAL VALUE TO X6 (UNLESS ALREADY AVAILABLE IN X6)
- COMPILE SA6 = V-TAG TO STORE INDEX
- ASSEMBLE ADDRESS OF INDEX STORE, THE INCREMENT TAG (OR CONSTANT), AND THE LIMIT TAG (OR CONSTANT): ENTER IN THE DØ PARAMETER TABLE (TABLE H)

DO STATEMENT PROCESSING

Figure 2.2

the CIR (Compile Read Instructions) subroutine to compile a fetch instruction for the initial value. It is possible that the variable used as the initial value may (at execution time) be available in an X register. In this case, CIR will not generate a fetch instruction, but will supply CDI with the number of the X register in which the variable can be found at execution time. Whether or not a fetch instruction had to be compiled, CDI next compiles a BX6 = X_i instruction to bring the initial value to a write register.

CDI next compiles an SA6 = Variable Tag instruction to store the initial value in the index location. A program tag (A-tag) is set in the lower half of the word in which this compiled instruction is stored, since this instruction is the return point from the bottom of the DO loop. Next, CDI examines the string to determine if an increment has been specified. If an increment has not been specified, the increment value is set to 1. If there is an increment entry in the string buffer, CDI examines the entry to determine if it is a variable tag for an integer variable or a constant. If it is not, an error exit occurs. The limit field is similarly checked. An example of the initial code compiled for the DO is shown in Figure 2.3.

Finally, CDI assembles the address of the index store instruction, the increment tag or constant, and the limit tag or constant into a single word (see Figure 1-3 for the format), and enters this word in the DO Parameter Table, (Table H). When the statement number of the statement which terminates the DO loop is encountered, these parameters will be used to compile the index test instructions. Processing of the DO is now complete, and so CDI jumps to PSN (Process Statement Number) to process the statement number, if any, associated with the DO statement, and from there control is returned to the compiler master loop for processing of the next statement.

Each time the compiler master loop processes a source statement, it calls the GSN (Get Statement Number) to perform the initial statement number processing. GSN determines if the statement which is about to be processed has a statement number, and, if so, extracts this statement number from the string buffer. GSN then scans the Statement Number Table and the DO Number Table for this statement number: if the

DO LOOP INITIALIZATION CODE

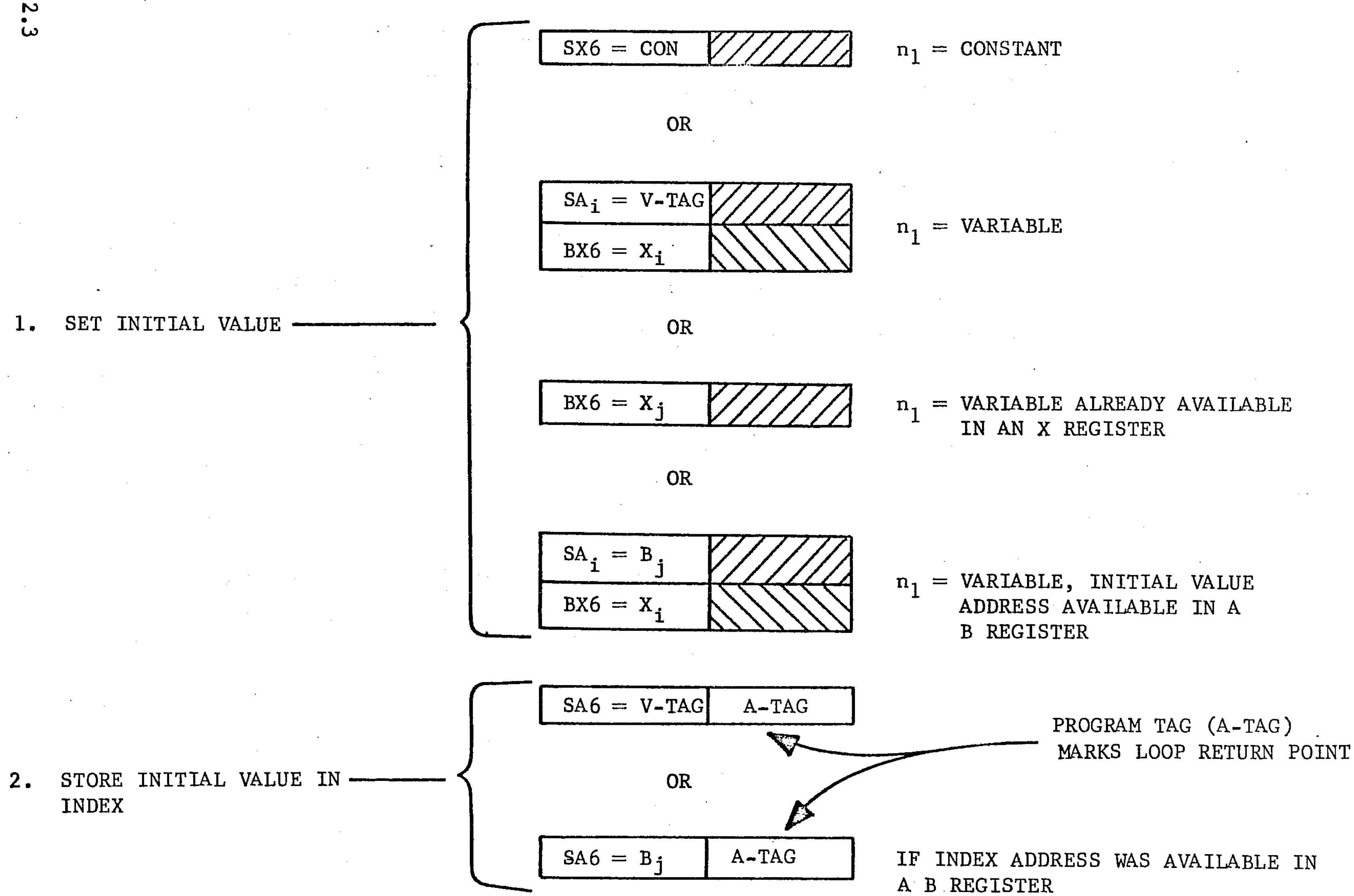
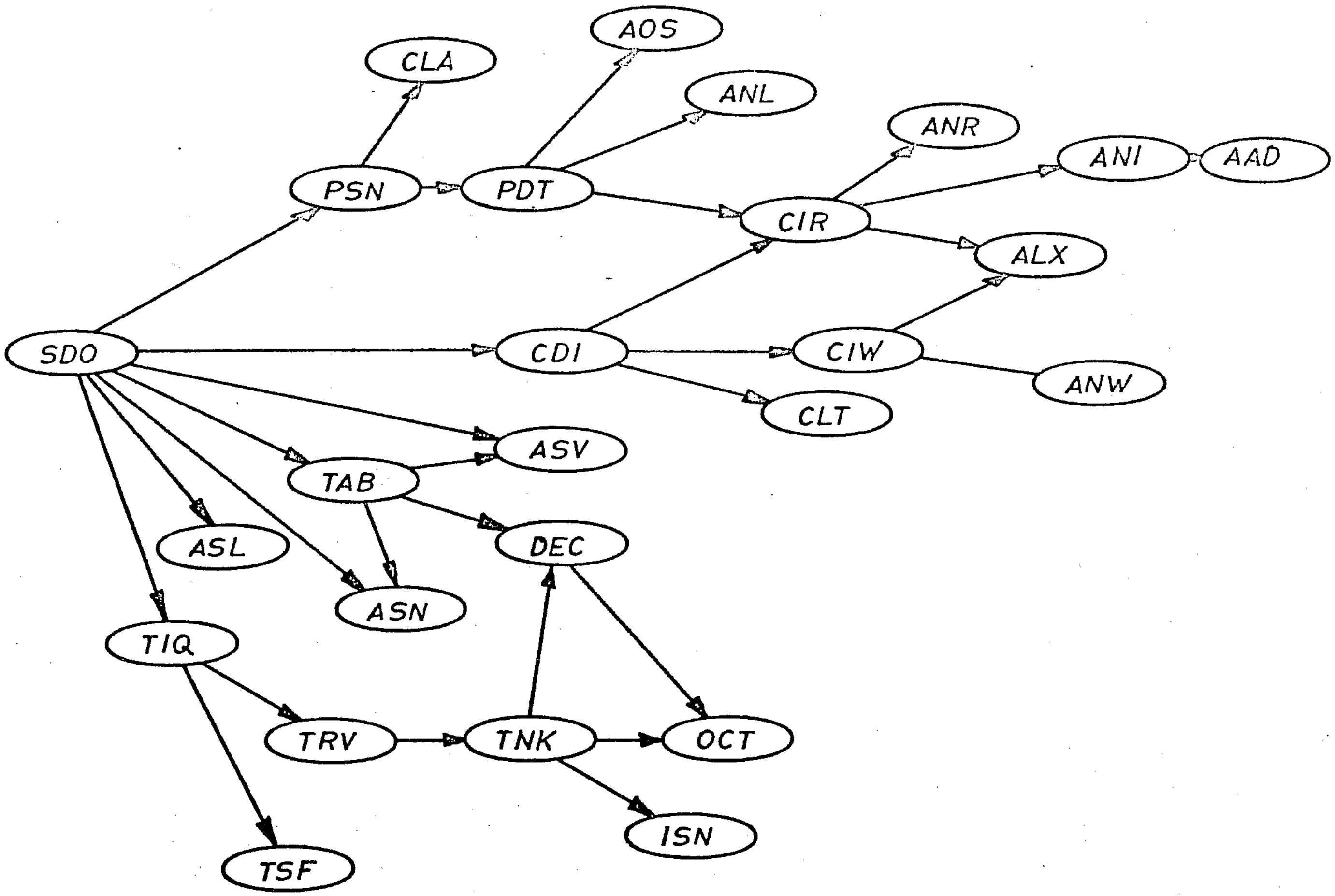


Figure 2.3

statement number is found in both tables, the DO termination indicator is set. The instructions for this statement are then compiled (unless the statement is a CONTINUE statement) and then the PSN (Process Statement Number) is called to process the statement number.

PSN checks the DO termination indicator: if this indicator is not set, PSN scans the Statement Number Table for the statement number and, if not found, enters the statement number in the Statement Number Table. The current program tag (A-tag) is entered in the Statement Tag Table, and the PDT (Process DO Tables) subroutine is called to compile the index test instructions. PDT first checks the Continue Indicator and the Current Jump Indicator, and processes these indicators if they are set. Next, the DO Number Table is scanned and, when found, the corresponding entry in the DO Parameter Table is saved. This entry is then deleted from the DO Number Table. The address of the index store instruction is then compared with the start of the group of instructions compiled for this statement (i.e., the DO termination statement) to determine if this is a one-statement DO. If so, an indicator is set, since one-statement DO loops are later analyzed to determine if the generated object code can be improved.

Next, the index store instruction was examined to determine if it was an SA6 = Bj instruction. If it was, the index address was the parameter of the subroutine, and so a Program tag can be formed by adding the B register number to a base tag value of 200003 (see page 1-14). If the index store was an SA6 = V-tag instruction, the Variable tag (V-tag) is extracted from the instruction. The Variable tag or Program tag which defines the location of the index is passed to the Compile Read Instructions (CIR) subroutine. CIR will generate a fetch instruction (if necessary) to fetch the index value. The increment parameter is next extracted from the DO Parameter Table entry, and examined to determine if it is a Variable tag or a constant. If the increment parameter is a Variable tag, the Analyze Loop Conditions subroutine is called to determine register availability, and the CIR subroutine called to compile a fetch instruction. This process is repeated for the limit parameter.



RUN COMPILER

DO STATEMENT PROCESSING

SUBROUTINE FLOW

NOTE 1: ADF, SCT, AND SCM SUBROUTINES NOT SHOWN

NOTE 2: SUBROUTINES BEYOND THE 2nd LEVEL MAY NOT NECESSARILY BE CALLED IN NORMAL DØ PROCESSING

Figure 2.4

When the index, limit, and increment parameters have been processed and any necessary fetch instructions have been compiled, PDT compiles the index increment instruction. If the increment parameter is a constant, this will be an SX6 = Xj + K instruction, while if this parameter is a Variable tag, an SX6 = Xi + Xj instruction is compiled (since a fetch instruction was compiled to bring the increment to Xi).

Next, the limit parameter is examined to determine what type of index test instructions must be generated. If the limit parameter is a constant, an "SX6 = X6 - K, NG X7 A-tag" sequence is compiled. If the limit parameter is a Variable tag, then an instruction has been compiled to bring the limit value to an X register, and so an "IX7 = Xi - X6, PL X7 A-tag" sequence is compiled. In either case, the A-tag is that initially assigned to the index store instruction.

If this is a one-statement DO loop, the Analyze One-Statement DO subroutine is called to attempt to improve the object code generated for the statement. The DO Number Table is then scanned again to determine if this statement appears again (remember that entries in this table are cleared as they are processed). If this statement number appears again in this table, then a nested DO loop is indicated, and so PDT repeats the process described. When all entries with this statement number in the DO Number Table have been processed, PDT exits to PSN, and from there control is returned to the compiler master loop.

ARITHMETIC STATEMENT PROCESSING

Prior to searching the table of statement types, the compiler master loop calls the Sense Formula (SFO) subroutine is called to determine if the statement is an arithmetic statement and, if it is, to initiate statement processing. SFO determines if the statement is an arithmetic statement function, in which case the Compile Function Definition (CFF) subroutine is called, or a replacement type arithmetic statement, in which case control is passed to the Compile Normal Formula (CNF) subroutine.

Statement evaluation by the CFF subroutine and by the CNF subroutine is similiar in many respects. The basic steps in the evaluation process are tabulated in figures 2.5 and 2.6. Both CFF and CNF call the TAB subroutine

CNF - COMPILE NORMAL FORMULA: BASIC STEPS

- CALL "TAB" TO NORMALIZE THE STATEMENT
- CALL "TIQ" TO TRANSLATE THE STATEMENT INTO A SERIES OF SEPARATORS AND TAGS
- CALL "UNP" TO ELIMINATE PARENTHESIZED EXPRESSIONS
- CALL "CXP" TO COMPLETE EVALUATION OF THE EXPRESSION
- SET THE RESULT MODE TO AGREE WITH THE MODE OF THE TERM ON THE LEFT SIDE OF THE EXPRESSION
- COMPILE INSTRUCTIONS TO STORE THE RESULT

Figure 2.5

CFF - COMPILE FUNCTION DEFINITION: BASIC STEPS

- SET UP THE CURRENT JUMP
- ENTER EACH FUNCTION ARGUMENT IN THE ARGUMENT NAME TABLE (TABLE I)
- ENTER A TAG FOR EACH FUNCTION ARGUMENT IN THE ARGUMENT TAG TABLE (TABLE J)
- COMPILE A ZERO WORD FOR EACH ARGUMENT
- ENTER THE FUNCTION NAME IN THE FUNCTION NAME TABLE (TABLE E)
- ENTER A FUNCTION TAG IN THE CORRESPONDING ENTRY IN THE FUNCTION TAG TABLE (TABLE F)
- CALL "TAB" TO NORMALIZE THE STATEMENT
- CALL "TIQ" TO TRANSLATE THE STATEMENT INTO A SERIES OF SEPARATORS AND TAGS
- CALL "UNP" TO ELIMINATE PARENTHESIZED EXPRESSIONS
- CALL "CXP" TO COMPLETE EVALUATION OF THE EXPRESSION
- SET THE RESULT MODE TO AGREE WITH THE FUNCTION MODE
- COMPILE A JUMP INSTRUCTION TO THE FUNCTION'S ENTRY POINT

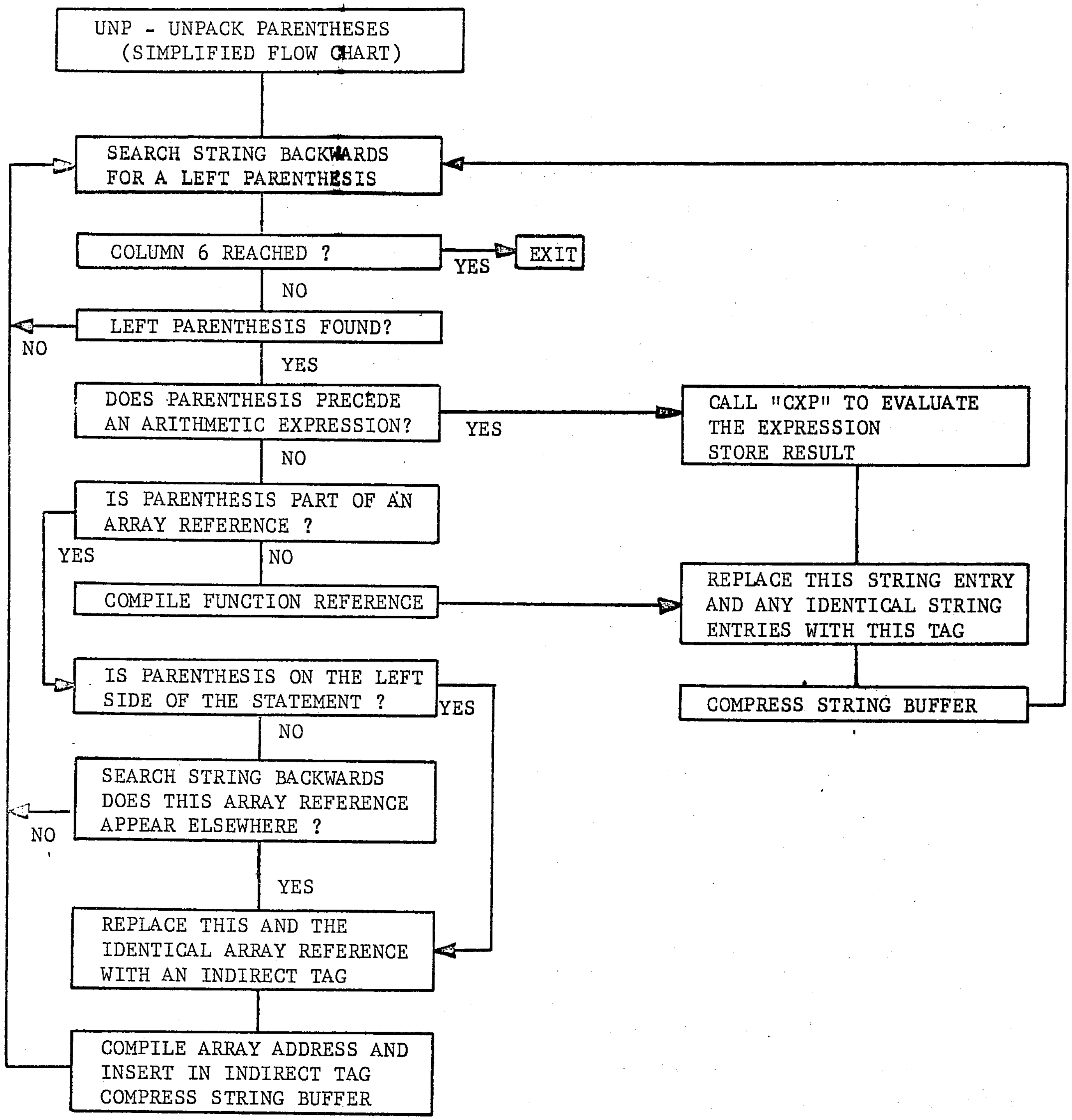
Figure 2.6

to assemble the statement in the string buffer into a series of variables, constants, and separators. CFF enters the function name in the Function Name Table and replaces it in the string with a Function tag. The function arguments are entered in the Argument Value Table and replaced in the string by function tags. The TIQ subroutine is used by both CNF and CFF to translate the constants and variables in the string into a sequence of tags, and to enter these values in the appropriate tables. In processing statements, the Function Name Table is searched for a variable before the Variable Name Table is searched. Thus, in processing an arithmetic statement of the replacement type, the Function Name Table will be empty and so the variables in the statement are determined to be active variables rather than dummy arguments. In processing an arithmetic statement function, statement variables will be found in the Function Name Table, indicating that these variables are dummy arguments.

Both CNF and CFF call the Unpack Parentheses (UNP) subroutine to eliminate array references, function references, and parenthesized expressions from the statement. A simplified flow chart of the UNP subroutine is shown in figure 2.7. When UNP finds an array reference, it compiles the instructions required to fetch the array element after scanning the statement to determine if this element has previously been referenced and is therefore available. If the parenthesized quantity is part of a function reference, UNP calls the CRF (Compile Function Reference) subroutine to construct the calling sequence for the function. If the parenthesized quantity is an expression, the Compile Expression (CXP) subroutine is called. CXP determines the dominant mode of the expression and selects the proper subroutine to compile instructions for the evaluation of the expression and the conversion of the result to the dominant mode.

When UNP has eliminated all parenthesized quantities from the statement, control is returned to CNF or CFF. These routines may call CXP directly to complete evaluation of the right-hand side of the statement. CNF and CFF then generate the instructions needed to store the result. In the case of CFF, instructions may be compiled to convert the mode of the result to the mode of the variable on the left-hand side of the statement, and a jump instruction to the function's entry/exit line is compiled.

When statement compilation is complete, control is transferred to the PSN subroutine to process the statement number, and from there control is returned to the master loop.



<u>UNP-UNPACK PARENTHESES</u>

Figure 2.7

INPUT/OUTPUT STATEMENTS

Upon encountering an input or output statement, the compiler generates a calling sequence for use by the execution time subroutines. There is no format cracking done during compilation, so all format diagnsotics are produced during execution. Each particular set of I/O statements, i.e., READ, WRITE, ENCODE, BUFFER IN, etc., use an individual execution time subroutine. These subroutines do their own processing within themselves and do not depend on a central or generalized routine for the I/O. All information necessary for the completion of the task is generated by the compiler and passed to the execution time routine with successive calls.

In order for a central memory program to communicate with an external file, all information entering or leaving the program must pass through a buffer. For every I/O file, whether it be standard input or output, scratch tape, or data tape, used by the FORTRAN program, a declaration of the file name must be made on the PROGRAM card. Each file name causes a buffer with a minimum length of 1010 words or normally 2010 words to be reserved for its use. Any file that is not assigned to a special equipment via a control card will be assigned to the disk. The execution time subroutines use the system CIO (Circular Input/Output) for the physical transfer of data.

All information written on 1" tape or binary data written on ½" tape is recorded in blocks of 1000B words (physical record). The terminating block of a transfer is called a short block whose size is between 1 and 777B words. A logical record is defined as containing any number of physical records and terminated by a short block. Coded one inch tapes use packed display code with two consecutive characters whose value is zero terminating the records. These records may not be larger than 136 characters long but are written on 1" tape in the aforementioned logical record scheme. Therefore, the system makes no distinction between coded or binary data when a one inch tape is involved. There is a difference on ½" tape. All coded information is translated to BCD and written in 136 character physical records. In this case, a logical record is the same as a physical record.

For a disk file, there is no specific record limit. The data is streamed out on the disk with a short sector (less than 100B words) being the terminating factor of a logical record. Like the one inch tape, coded and binary information appear the same to the system.

The compiler has I/O statement processors which decide from the form of statement which execution time routines are to be called. If a format statement is required, then the address of it must be available during exeuction. Since all I/O has to pass through a buffer, the address of this buffer must also be known. This information is compiled and sent to the subroutine in one entry. The I/O list is processed and one entry is made for each array or data item. It is during these entries that the format statement is cracked. A final entry is made to signal the end of the list.

The coded input statements (READ n, L; READ (i, n) L; READ INPUT TAPE i, n, L) call INPUTC. The file specified by "i" is read and the data "L" returns to the program according to the format "n". The following specifications are handled by INPUTC: E, F, D, O, A, *, I, 1, X, R, L, P. Only with "F" conversion is a scale factor allowed. The format cracking utilized in INPUTC is flow charted in Appendix C, pages 1-5. During compilation, the address of the format statement is set into B3 to be passed to the subroutine. The address of a variable format is retrieved by assigning a variable tag to the format statement; thereby fetching the proper address during execution.

Binary data may be read by READ (i) L or READ TAPE i, L. During execution, INPUTB is referenced to read file "i" and insert the data in "L". No special word count is reserved in the data itself. The number of words defined by L determines the number of physical records that are read. Binary data may be written on a file by WRITE (i) L or WRITE TAPE i, L. Either of these statements request OUTPTB to transfer the information from "L" to file "i". The number of words written by these statements must be greater or equal to the number of words read by the corresponding READ statement.

OUTPTC is the execution time subroutine called to write coded data on a file. The statements PRINT n,L; PUNCH n,L; WRITE (i, n) L; or WRITE OUTPUT TAPE i, n, L will all cause OUTPTC to be referenced. As with coded input, the format is cracked during execution. The types of format specifications allowed on output are: I, X, A, O, H, /, F, E, D, R, L, *, P. There is little difference between the procedure of format cracking used by OUTPTC and INPUTC.

ENCODE and DECODE statements are also implemented. Storage manipulation to transfer data under a specific FORMAT statement is all that is involved so no physical data file is referenced. Therefore, the list processor used by READ/WRITE compiles a calling sequence to the execution time subroutines OUTPTS and INPUTS. These subroutines work on the same format cracking scheme as OUTPTC and INPUTC.

All the aforementioned statements result in the I/O being completed by the execution time subroutines before control is returned to the central program. Therefore, the data is immediately available to the programmer after an I/O statement has been processed. However, the user may choose to buffer his own I/O in which case the BUFFER IN and BUFFER OUT statements are available. BUFFERI and BUFFEO (execution time subroutines) are called, respectively, to initiate the transfer of data via CIO. In this case, the central processor is not released by a recall (RCL) request. Instead control is returned to the central program as soon as CIO has initiated the request. Any block of data, up to normal central memory restrictions, will be handled by these statements. Before using the data it is up to the user to check the status of the buffered unit by an IF (UNIT, i). This statement compiles a calling sequence to IOCHEK which is the execution time routine used for checking the status.

The execution time subroutines receive all addresses from the program via index registers. A calling sequence is constructed by the compiler for each statement. Listed on the following page are the calling sequences compiled to be used during execution.

CALLING SEQUENCES

READ, WRITE, PRINT, PUNCH

First Entry	B1 = 0 B2 = address of buffer parameter list or complemented address of variable tape number B3 = address of format statement
Intermediate Entries	B1 = address of data item or beginning address of array B2 = array length or 0
Final Entry	B1 = -1

ENCODE, DECODE

First Entry	B1 = 0 B2 = 0 B3 = address of format statement B4 = character length
Second Entry	B1 = beginning address of packed data B2 = 0
Intermediate Entries	B1 = address of data item or beginning address of array B2 = array length or 0
Final Entry	B1 = -1

BUFFER IN, BUFFER OUT

First Entry	B1 = mode constant B2 = address of buffer parameter list or complemented address of variable tape number
Second Entry	B7 = address of first word of data block
Third Entry	B7 = address of last word of data block

END STATEMENT PROCESSING

Three conditions will cause entry to the END processing routine.

1. An END statement
2. A plus in column one of the next input card
3. An end of record on input.

In case one of the following conditons prevail:

1. The instructions for the last subprogram have been compiled one per central memory word.
2. The location tag, if any is needed, is in the lower 18 bits of the compiled instruction.
3. All information needed to make memory assignments and process generated tags is contained in the temporary tables.

At this time, the instructions are packed and the location tags along with their absolute address are saved. Various routines are then called to make temporary, common, and unique variable assignments. The tags of the variables along with their memory addresses are saved.

The routine RAD is then called to replace tags with memory addresses. It will search the complete short file, that is the last program/subprogram compiled, and replace all K portions of 30 bit instructions that have tags with memory addresses. If, for some reason, a K tag is found that has not been given a memory assignment, some type of diagnostic is given. This could be caused by a missing statement, a dimension ordering error or could be a system error.

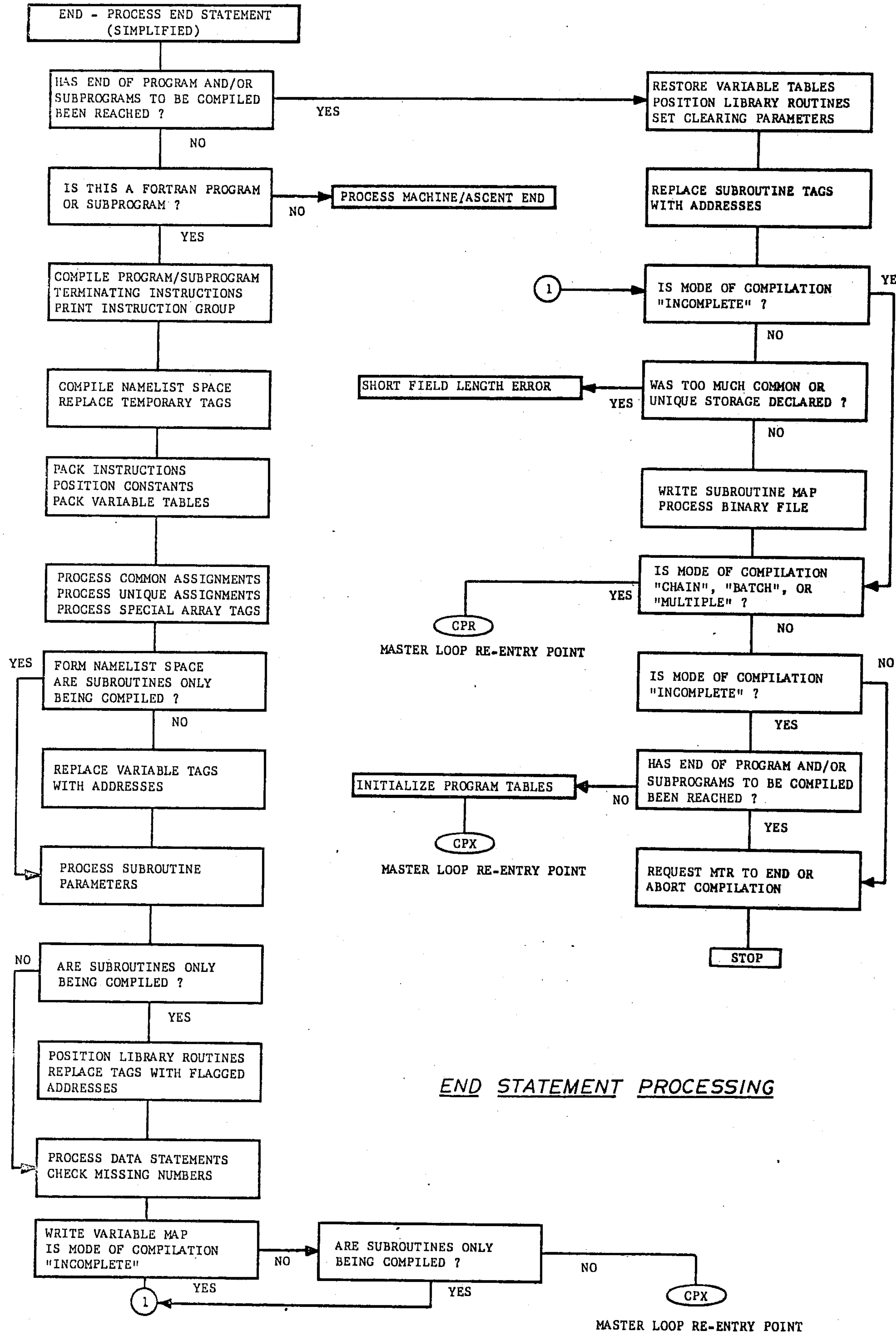
The DATA statement is then processed, the variable map written, and a return is made to the main loop for the next subprogram.

If there is a plus in column one of the next input card, a routine is entered to read in binary programs. These programs are positioned with the object deck already compiled. The information about each routine read in is extracted from the RA and RA+1 of the binary routine and entered into the subroutine tables. Processing then continues the same as if an end of record was detected originally on the input file.

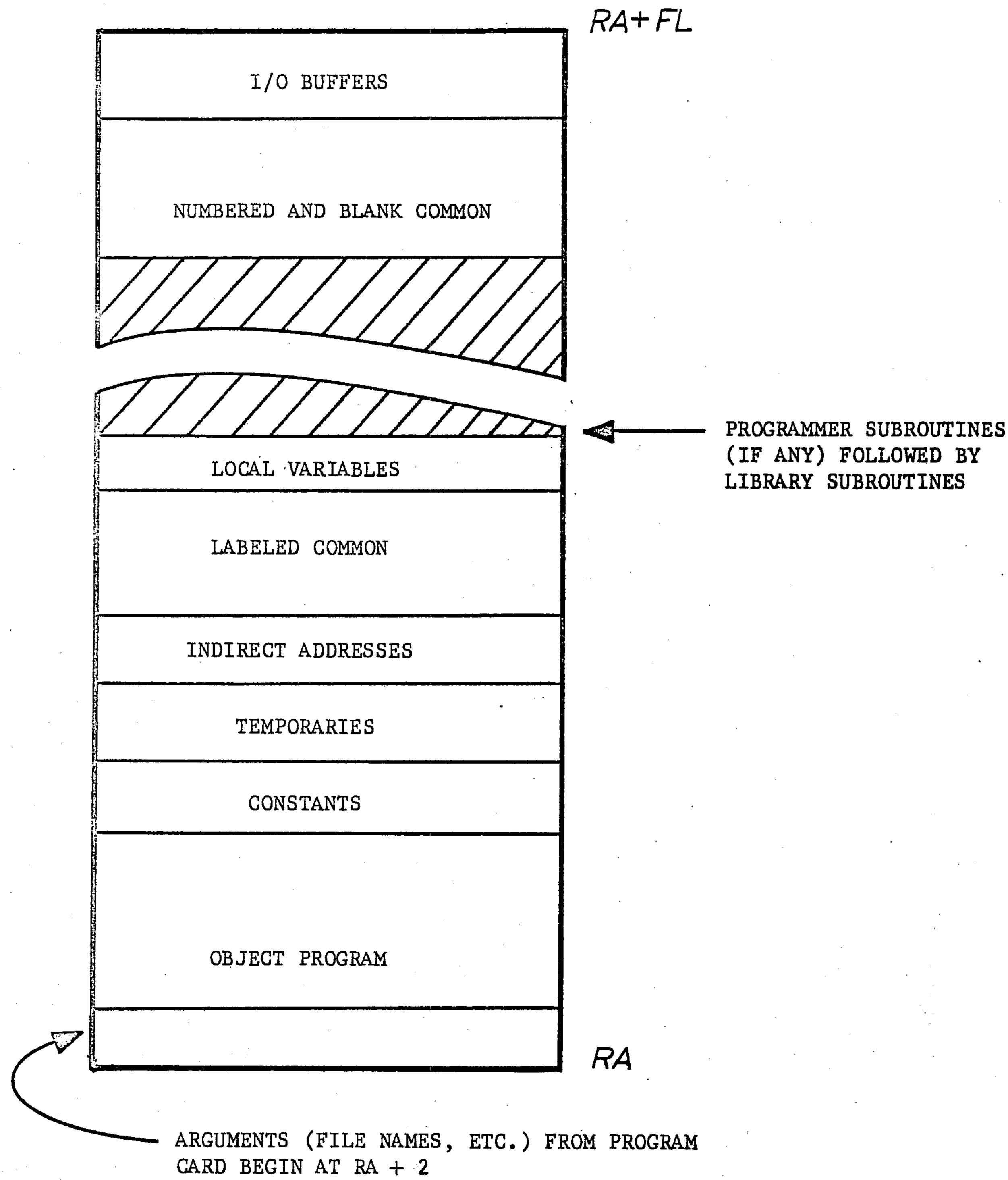
A call is made to the PP routine CLL to load all subroutines not yet defined. The starting address of subroutines is now equated to memory addresses, and RAD is called once for each program/subprogram to replace all subroutine tags with memory addresses. The complete file, if no errors have been detected, is written on the disk.

Then, depending on the mode of compilation, a return may be made for another deck, the compiler may terminate, the deck may be punched or the EXU may be called to load and execute this compiled program.

A simplified flow of the END processing is found on the next page.



END STATEMENT PROCESSING



MAIN PROGRAM ORGANIZATION

LISTING			INTERPRETATION: ASCENT EQUIVALENT			COMMENTS
	PROGRAM TEST(INPUT,TAPE1 = INPUT)				.	Header card, processed by PPG
000000	0	L00002	L00002	BSSZ	1	
000001	0	L00003	L00003	BSSZ	1	RA, RA + 1 are MTR communications area
000002	0	L00004	L00004	BSSZ	1	File Name and Buffer Address (INPUT)
000003	0	L00005	L00005	BSSZ	1	File Name and Buffer Address (TAPE1)
0000(-	0	L00001	L00001	BSSZ	1	Reserved word (subprogram entry/exit line)
000005	5110 C00001			SA1	C00001	C00001 - parameter word set up by END
	63110			SB1	X1	Extract local length
	21122			AX1	22B	
0000(	63210			SB2	X1	Extract beginning address of COMMON
	21122			AX1	22B	
	63310			SB3	X1	Extract compiled field length
	74200			SX2	A0	Pick up field length from A0
000007	37121			IX1	X2-X1	
	0331 000000			NG	X1,0	Exit if insufficient space
000010	7160 001777			SX6	001777B	
	20660			LX6	60B	Set X6 to an indefinite value
000011	56610	L00006	L00006	SA6	B1	
	6111 000001			SB1	B1 + 1	Set unused program space (i.e., from
000012	0712 L00006			LT	B1,B2,L00006	the end of local to the beginning of
	43600			MX6	0	COMMON) to indefinite: clear COMMON
000013	56610	L00007	L00007	SA6	B1	and buffer areas to zero
	6111 000001			SB1	B1 + 1	
000014	0713 L00007			LT	B1,B3,L00007	
	5110 C00002			SA1	C00002	Fetch compiled field length
000015	63110			SB1	X1	
	43052			MX0	52B	Set mask for file name
	5120 C00004			SA2	C00004	Fetch compiled buffer length
000016	63220			SB2	X2	
	67312			SB3	B1-B2	Compute starting address for 1^{st} buffer
	5130 000002			SA3	000002	Fetch first file name from RA + 2
000017	11730			BX7	X3*X0	Mask out buffer address, store file name
	56730			SA7	B3	as 1^{st} buffer parameter
	5140 C00005			SA4	C00005	Fetch compiled file name and buffer
000020	10740			BX7	X4	address, store in RA + 2 (replace
	5170 000002			SA7	000002	execution time file name)
000021	7173 000010			SX7	B3 + 10B	Initialize circular buffer pointers
	5173 000001			SA7	B3 + 1	Set FIRST
000022	5173 000002			SA7	B3 + 2	Set IN
	5173 000003			SA7	B3 + 3	Set OUT
000023	76710			SX7	B1	Pick up compiled field length
	5173 000004			SA7	B3 + 4	Set LIMIT
	43700			MX7	0	
000024	5173 000005			SA7	B3 + 5	Set sixth and seventh buffer parameters
	5173 000006			SA7	B3 + 6	to zero
000025	5150 C00003			SA5	C00003	Fetch line limit
	10750			BX7	X5	
000026	5173 000007			SA7	B3 + 7	Set line limit as eighth parameter
	67112			SB1	B1-B2	Set LIMIT for next buffer (unused in
						this example)
	NAME = 2					Source card, processed by SFO
000027	7160 000002			SX6	000002	Instructions compiled for this state-
	5160 V00001			SA6	V00001	ment by CXP

Note: words 0 - 26_8 were compiled by the PPG (Process Name and Arguments) subroutine. When the compiler master loop encounters a PROGRAM card, the PGM (Process Program Statement) subroutine is called: PGM checks to insure that a prior program or segment has not been compiled, and then calls PPG.

L0000n = PROGRAM TAG (A-TAG)
C0000n = CONSTANT TAG (K-TAG)
V0000n = VARIABLE TAG (V-TAG)

LISTING

```
          DO 3 I = 1,NAME

000030  7160 000001
000031  5160 V00002   L00013

        3     BAT = ROB

        5150 V00001
000032  5110 V00004
        10610
000033  5160 V00003
        5120 V00002
000034  7262 000001
        37756
000035  0327 L00013

          A = 16.

        5110 C00006
000036  10610
        5160 V00005

          GO TO 1

000037  0400 N00001

        1     Q = 0.

000040  43600         L00020  (N00001)
        5160 V00006

          CALL BATSUB(A,B,C,D,E,F,G,10)

000041  6110 V00005
        6120 V00007
000042  6130 V00010
        6140 V00011
000043  6150 V00012
        6160 V00013
000044  7160 V00014
        5160 S00207
000045  7170 C00007
        5170 S00210
000046  0100 S00200   L00022
        0710 L00002

          END

000047  5120 C00010
        10720
000050  0100 S00300
```

INTERPRETATION

ASCENT EQUIVALENT			COMMENTS	
			Source card, processed by SDO	
	SX6	000001	CDI(called by SDO) compiles instructions	
L00013	SA6	V00002	to initialize the DO index	
			Source card, processed by SFO. Statement	
			number processed by PSN, which calls PDT	
	SA5	V00001[1]	to compile index increment and test instrs.	
	SA1	V00004	Fetch ROB	
	BX6	X1		
	SA6	V00003	Store (ROB) in BAT	
	SA2	V00002	Fetch I	compiled
	SX6	X2 + 1	I = I + 1	by PDT
	IX7	X5-X6	NAME - I	
	PL	X7, L00013	Loop if limit not reached	
			Source card, processed by SFO	
	SA1	C00006	Fetch constant	
	BX6	X1		
	SA6	V00005	Store constant in A	
			Source card, processed by SGO	
	EQ	B0,B0,N00001	N00001 = statement tag (H-tag)	
			Source card, processed by SFO	
L00020	MX6	0	L00020 is equated to N00001 through the	
	SA6	V00006	Temporary and Permanent Tag Tables	
			Source card, processed by CLL. CLL calls	
			PRR to compile argument handling instrs.	
	SB1	V00005		
	SB2	V00007	The addresses of the first six	
	SB3	V00010	arguments are passed to the	
	SB4	V00011	subroutine in registers B1 - B6	
	SB5	V00012		
	SB6	V00013		
	SX6	V00014	Pick up address of seventh argument	
	SA6	S00207	Store 7th argument addr. in reserved word	
	SX7	C00007	Get address of 8th argument - a constant -	
	SA7	S00210	and store in reserved word	
L00022	RJ	S00200		
			Lower half of instruction word contains	
			argument count and name of caller	
			Source card, processed by END	
	SA2	C00010	Zero word passed to execution time	
	BX7	X2	subroutine END	
	RJ	S00300	Jump to execution time subroutine END	

1: the instruction in the lower half of word 31 is the limit fetch instruction for the DO index increment and test. Do index increment and test instructions are compiled by PDT (called by PSN). PDT calls the AOS subroutine (Analyze One-Statement Do) to attempt to place the increment and limit fetches (if any) at the beginning of the instructions compiled for the statement which terminated the DO loop.

SAMPLE COMPILATION: SHEET 2

LISTING — INTERPRETATION

LISTING			ASCENT EQUIVALENT			COMMENTS
	SUBROUTINE BATSUB(W,X,V,D,R,J,M,L)					Header card, processed by PPG
000077	0	L00002	L00002	BSSZ	1	Reserved for subroutine name
000100	0	L00003	L00003	BSSZ	1	Reserved for argument count
000101	0	L00004	L00004	BSSZ	1	
000102	0	L00005	L00005	BSSZ	1	
000103	0	L00006	L00006	BSSZ	1	These six words are unused, since the
000104	0	L00007	L00007	BSSZ	1	arguments to which they correspond are
000105	0	L00010	L00010	BSSZ	1	passed in registers B1 - B6
000106	0	L00011	L00011	BSSZ	1	
000107	0	L00012	L00012	BSSZ	1	Reserved for the seventh argument
000110	0	L00013	L00013	BSSZ	1	Reserved for the eighth argument
000111	0	L00001	L00001	BSSZ	1	Entry/exit line
000112	76710			SX7	B1	
	76120			SX1	B2	Save addresses of first three arguments
	20122			LX1	22B	in temporary word 1
	36717			IX7	X1 + X7	
000113	76230			SX2	B3	
	20244			LX2	44B	
	36727			IX7	X2 + X7	
000114	5170 T00001			SA7	T00001	Store first three argument addresses
	76740			SX7	B4	
	76350			SX3	B5	Save addresses of next three arguments
000115	20322			LX3	22B	in temporary word 2
	36737			IX7	X3 + X7	
	76460			SX4	B6	T0000n = Temporary Tag (T-Tag)
	20444			LX4	44B	
000116	36747			IX7	X4 + X7	
	5170 T00002			SA7	T00002	Store second three argument addresses
	RETURN					Source card, processed by RTN
000117	0400 L00001			EQ	B0,B0,L00001	Jump to entry/exit line
	END					Source card, processed by END
000120	5150 C00001			SA5	C00001	Pass a zero word to the execution time
	10750			BX7	X5	subroutine END. These instructions are
000121	0100 S00300			RJ	S00300	unused in this example: they would be
						used if the RETURN statement was not
						present.

SAMPLE COMPILATION: SHEET 3

CHIPPEWA FORTRAN COMPILER - RUN

Section 3

SUBROUTINE DESCRIPTIONS

AAR - ANALYZE ARRAY REFERENCE

This routine is called during the processing of an arithmetic expression when it has become necessary to bring the address of an array entry to B7. It is entered in an attempt to stop the storing of the array address into an indirect cell. Two conditions will make this routine flag that the store is probably necessary.

1. If there are any more array references to be processed, the store must be generated.
2. If the mode of the array is integer and if there is a division specified in the expression before this array reference is made, the address must be saved.

Otherwise, the single array reference count is increased, the next indirect tag and index 7 are packed into X6 and the routine exits.

Subroutines Called: None

Temporaries/Flags:
ARI - Array Reference Count
IGX - Current Index Assignment
SAR - Single Array Reference Count
TGI - Indirect Tag
TMF - Start of Array Reference
TMH - Start of Expression

Tables Referenced: None

Entry/Exit Register Conditions:

Entry: None

Exit: X6 - zero implies no processing done

X6 ≠ zero implies the next indirect tag and B7 have been packed in X6 and no store of the array address must be made.

ABS - PROCESS ABSOLUTE LIST

When the MAA (Process Machine or Ascent Records) subroutine encounters an ABS declarative, the ABS subroutine (Process Absolute List) is called. The ABS declarative permits the unsigned number on the right to be assembled into instructions containing the identifier in their address field. An example of the ABS declarative together with the basic steps in processing the list is shown in Figure ABS-1.

On entering the ABS subroutine, the TAB subroutine is called to normalize the list. The Variable Name Table (Table M) is scanned to determine if the variable which has been equated to the subroutine name has been entered and, if not, the variable is entered in the Variable Name Table. If the variable has previously been entered in the Variable Name Table, an error exit (Duplicate Tag Error) occurs. The variable name is also checked to insure that the first character is alphabetic and that the name is composed of two or more characters for machine programs.

Next, a check is made for the equal sign and the CVN (Convert Number) subroutine is called. If the constant is greater than -2^{16} and less than $2^{16}-1$, it is stored in the Variable Tag Table (Table N). If the constant lies between $2^{16}-1$ and $2^{17}-1$, a Statement Tag (H-tag) is generated and stored in the corresponding Variable Tag Table and the constant is stored in the Argument Tag Table (Table J).

Subroutines Called: TAB - Normalize Statement
SCT - Scan Table
CVN - Convert Octal or Decimal Number
ADF - Advance Table

Temporaries/Flags: MOD - Subprogram Mode
TGH - Statement Tag (set)
IPS - Program/Subprogram Indicator

ABS DECLARATIVE PROCESSING

EXAMPLE: ABS (JJ=100, KK=100B, LL=7777)

1. STORE THE VARIABLE NAME (e.g.,JJ) IN THE VARIABLE NAME TABLE

2. CHECK FOR AN EQUAL SIGN

3. CONVERT THE CONSTANT

4. LOOK FOR A TAG OR CONSTANT

5. STORE IN THE APPROPRIATE TABLE

6. IF THE NEXT ENTRY IS A COMMA, REPEAT 1-5

7. CHECK FOR A RIGHT PARENTHESIS

8. RETURN FOR THE NEXT RECORD

Figure ABS-1

Tables Referenced: TBM - Variable Name

TBN - Variable Tag

TBJ - Argument Tag

Entry/Exit Register Conditions: DNA

Note: Error Lists: EMC - Machine Constant

EMD - Machine Duplicate Tag Error

EMF - Machine Format Error

EMT - Machine Tag Definition Error

ACE - PROCESS ASCENT EQU

When the MAA (Process Ascent and Machine Records) subroutine encounters an EQU Ascent pseudo-operation instruction, the ACE subroutine is called.

The ACE routine writes a constant of "all fives" into the output buffer then transfers to WNX (Write Coded Record) and RNX (Read Coded Record). Next a call to TAB (Normalize Statement) reorders the string entries to one variable or separator per word. An equal sign is stored in column 9 and the location variable is stored in column 8. Further processing is handled with a jump to the ABS (Process Absolute List) subroutine.

Subroutines Called:	WNX - Write Coded Record
	RNX - Read Coded Record
	TAB - Normalize Statement
	ASV - Assemble Variable
Temporaries/Flags:	CAS - Constant of Blanks
	MHI - Machine Header Card Indicator (set)
Tables Referenced:	None

Entry/Exit Register Conditions:

A1 string address of first character beyond EQU pseudo-operation code.

ACH - PROCESS ASCENT DPC AND BCD

When the MAA (Process Ascent and Machine Records) subroutine encounters a BCD or DPC pseudo-operation, the ACH subroutine is called. The PST routine is called to process the location tag and the ARA routine is called to adjust the address and write the register. Next the first character of the address field is erased from the string, and the next ten characters are accumulated. A test is made to insure that the pseudo-op appeared in the constant section. On exit from the routine, the code is in X6.

Subroutines Called: PST - Process Statement Tag
ARA - Adjust Running Address and Write Register

Temporaries/Flags: IWC - Instruction Word Count

Tables Referenced: None

Entry/Exit Register Conditions:

A1 - Address of first non-blank following opcode.
X6 - Hollerith field

ACK - PROCESS ASCENT CON

When the MAA (Process Ascent and Machine Records) subroutine encounters a CON Ascent pseudo-operation code, the ACK subroutine is called. The constant in the address field is moved left beginning in column 7 of the string buffer. When an end of statement or a blank is encountered, a zero is written into the string and a transfer back to the main loop of MAC for further processing occurs.

Subroutines Called: None

Temporaries/Flags: None

Tables Referenced: None

Entry/Exit Register Conditions:

A1 string address of last character before the address field

ACR - PROCESS ASCENT BSS AND BSSZ

When the MAA (Process Ascent and Machine Records) subroutine encounters a BSS or a BSSZ pseudo-operation code, the ACR subroutine is called. Column 7 of the string buffer is set to a left parenthesis and the value of the address field is moved to the left beginning in column 8. Upon encountering the first blank or end of statement, a right parenthesis and a zero are stored into the next two columns of the string. Further processing is done in the Master loop of MAA.

Subroutines Called: None

Temporaries/Flags: None

Tables Referenced: None

Entry/Exit Register Conditions:

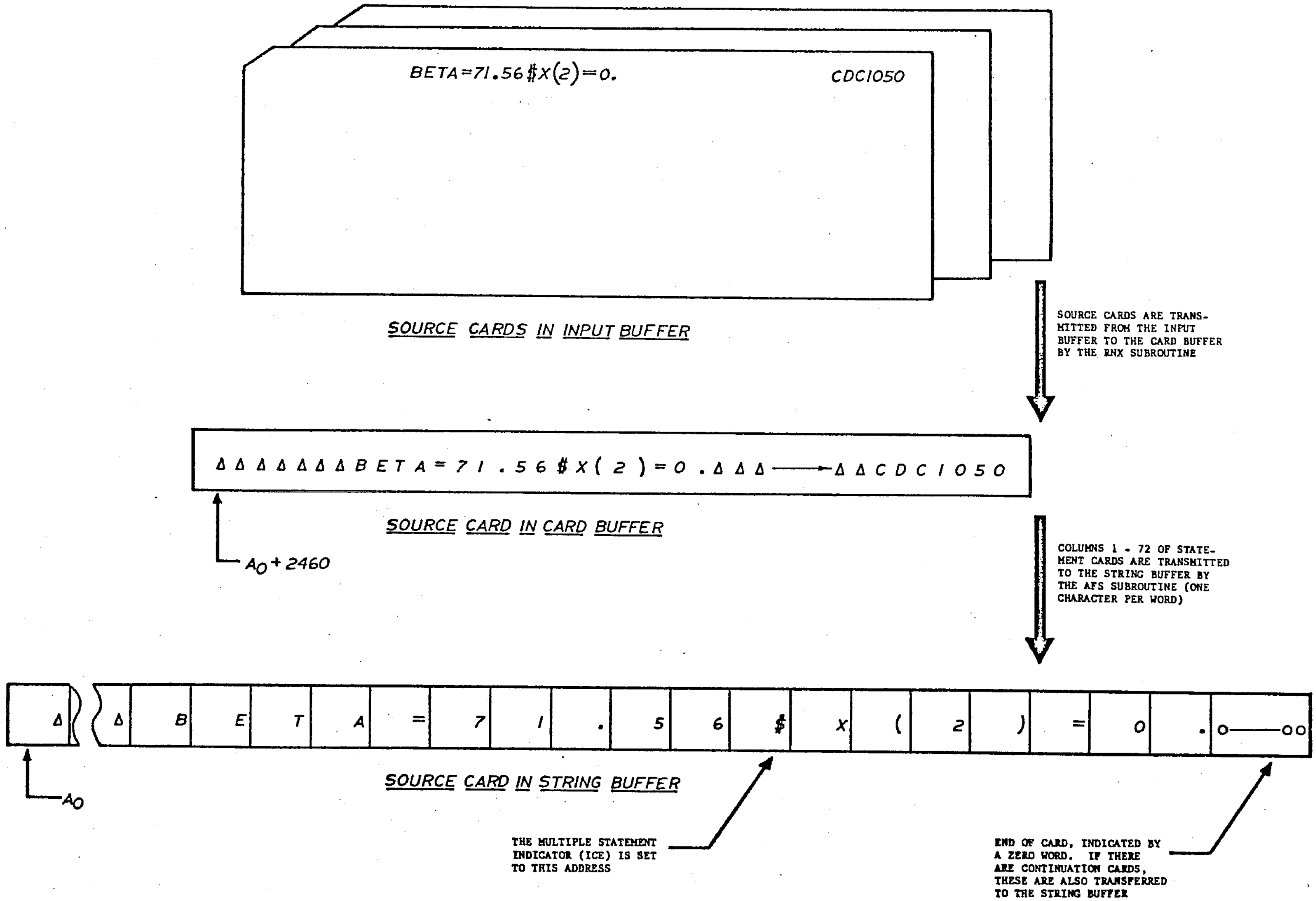
A1 - String address of the last character before the address field

AFS - ASSEMBLE FORTRAN STATEMENT

The AFS subroutine assembles a FORTRAN statement or assembly instruction from card buffer into the string buffer. If a statement is continued on one or more succeeding (continuation) cards, all such cards are also transmitted to the string buffer. Within the string buffer, information is packed one character per word, right-justified. The string buffer loading process is illustrated in figures AFS-1 and AFS-2.

On entry, the multiple statement indicator, ICE, is examined to determine if there were multiple statements on the card previously transmitted to the string buffer. If ICE is zero, then there were no multiple statements on the card previously transmitted to the string buffer: otherwise, ICE contains the address in the string buffer of the dollar sign which terminated the statement just processed. In the latter case, AFS blanks out the preceding statement and scans the remainder of the card until either a dollar sign or the end of the statement (a zero word) is encountered. If a dollar sign in encountered, the multiple statement indicator is set to the address of the dollar sign in the string buffer. Control is then returned to the calling program.

If there were no multiple statements on the previous card, AFS enters a loop which inputs, examines, and lists cards until a statement or instruction card is found. AFS calls the RNX subroutine to bring a card from the input buffer to the card buffer, and calls the WNX subroutine to transmit the card to the output buffer for listing. As each card is processed, AFS checks to see if the end of file has been reached. If the previous statement was an END statement, then PNM (program/sub-program name) will be zero: if PNM = 0 and an end of file is encountered, AFS transfers control to END (Process End Statement). In all other cases, detection of the end of file will result in an error exit. On the first entry to the loop, the card already in the card buffer is examined. If column 1 contains a period (page eject card), an asterisk or a dollar sign (remarks card), or in a FORTRAN program, the letter C (comments card), the card is listed (i.e., transmitted to the output buffer) and the next card brought to the card buffer. This process is repeated until a statement/instruction card is found: i.e., a non-



STRING BUFFER LOADING

AFS SUBROUTINE

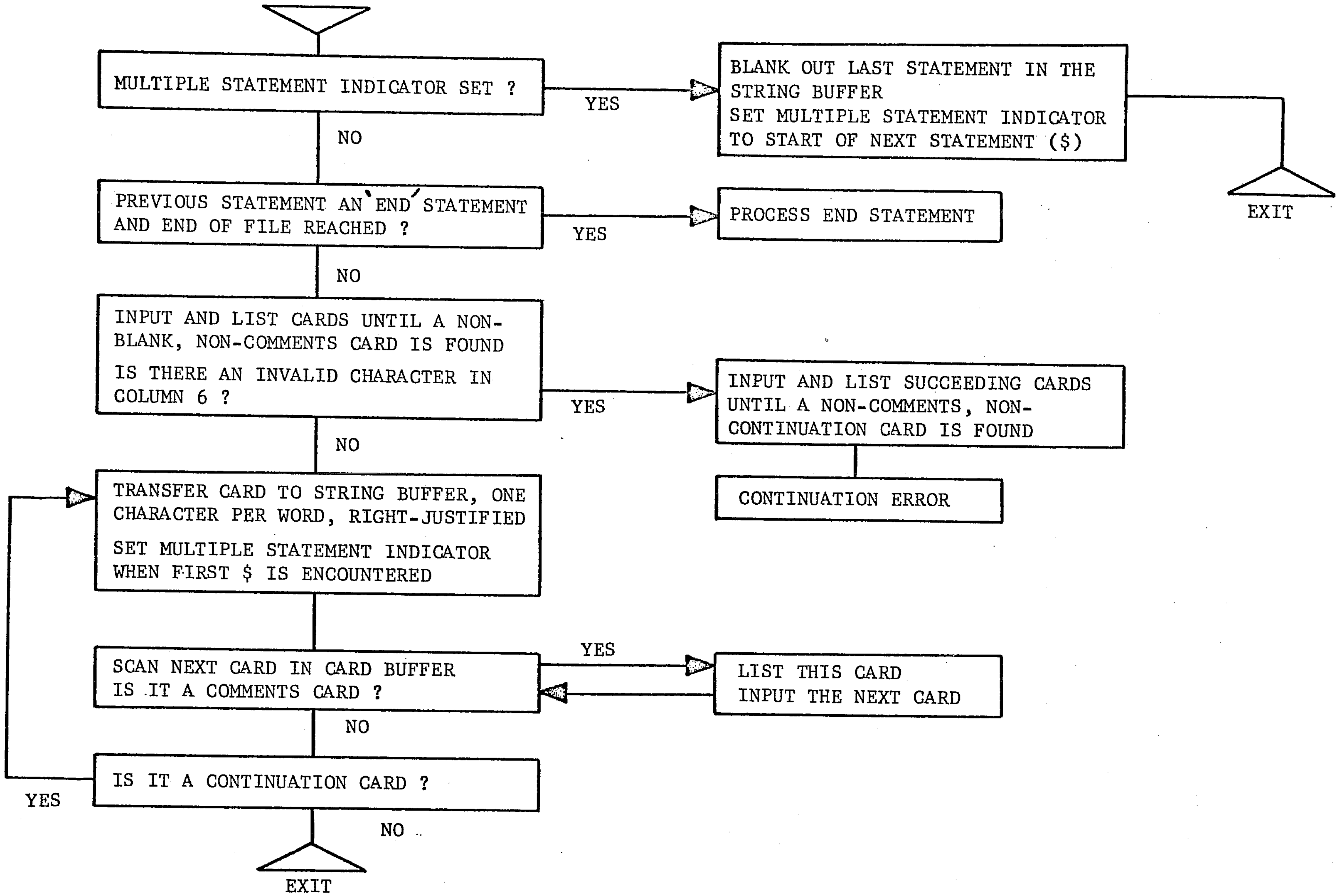
Figure AFS-1

blank card which is not one of those described above.

Next, column 6 of the card is examined. If column 6 is blank, contains a zero (FORTRAN program), or does not contain as asterisk, AFS transfers the card from the card buffer to the string buffer. All 72 characters on the card (spaces included) are transmitted to the string buffer. Characters in the string buffer are packed one per word, right-justified. If a dollar sign is encountered in the transfer process, the multiple statement indicator is set to the address of the dollar sign in the string buffer. (Note: this dollar sign is replaced with a zero word on return to the main loop of RUN.) The end of the card in the string buffer is then marked by a zero word. This zero word will be overlaid if this statement is continued on succeeding cards.

After processing a statement or instruction card, AFS inputs another card into the card buffer to determine if the statement just processed is continued: if so, the associated continueation cards must also be transferred to the string buffer. AFS examines the card to determine if it is a comments card (C in column 1). If it is, it is listed and AFS brings another card to the card buffer. If a card is found which is not a comments card, column 6 is examined to determine if it is a continuation card (non-blank and non-zero in FORTRAN, an asterisk in assembly language). If it is not a continuation card, control is returned to the calling program: the card in the card buffer will be processed on the next entry to AFS. If the card is a continuation card, it is transferred to the string buffer and the process repeated.

If the first statement/instruction card found did not contain a blank or zero in column 6 (FORTRAN program) or contained an asterisk in column 6 (assembly program), then it is assumed that an out-of-sequence continuation card has been found. AFS enters a loop in which cards are read and listed until a non-continuation, non-comments card is found, at which point an error exit takes place (continuation error).



MAJOR FUNCTIONS
AFS SUBROUTINE

Figure AFS-2

Subroutines Called: RNX - Read Coded Record
WNX - Write Coded Record
ASM - Assemble Mnemonic Code
END - Process End Statement

Temporaries/Flags: ICE - Multiple Statement Indicator (set)
MHI - Machine Heading Indicator (set)
IGS - Instruction Group Start (set)
PNM - Program/Subprogram Name
MOD - Subprogram Mode Indicator
FST - Long File Start
SIG - Compile Mode Indicator
ICA - Display Coded Running Address

Tables Referenced: none

Entry/Exit Register Conditions: n/a

ANK - ANALYZE ADDRESS GENERATING INSTRUCTIONS FOR RIGHT MEMBER

This routine is called during the processing of an expression when B7 has been used to hold the address of an array entry and there are more array references in the statement. If A0 is still available, the last compiled instruction is changed to a set A0 instruction and the A0 register associate is set to the next indirect tag which is passed back to the calling program. It also clears the instruction register X7.

Subroutines Called: None

Temporaries/Flags: TGI - Indirect Tag

VTA - A0 Register Associate

Tables Referenced: None

Entry/Exit Register Conditions:

Entry: None

Exit: X6 = zero if A0 was not available

X6 ≠ zero if A0 was used to hold the address of the array entry. Actually it would have the next indirect tag and bit 21 set to say the address was in A0.

ARA - ADJUST RUNNING ADDRESS AND WRITE REGISTER

When the MAA (Process Ascent and Machine Records) subroutine has packed as many consecutive instructions into one machine word as possible, the ARA routine is called to increment the running address by one and write the previously stored word. Should ARA be entered when the counter has been reset to zero, blanks are stored into the output buffer, then current running address is converted and stored. Should the counter be set to non-zero, the current word is written and the running address is incremented before the converting and storing of the running address.

Subroutines Called: None

Temporaries/Flags: ICT - Intraword counter
ADM - Current Running Address

Tables Referenced: None

Entry/Exit Register Conditions: N/A

ASL - ASSEMBLE LETTERS

The ASL subroutine assembles a specified number of letters from the string buffer into an assembly register. On entry to this subroutine, the B4 register contains the address of the location in the string buffer where assembly is to begin, and the B2 register contains the number of letters to be assembled. The assembled letters are returned to the calling program in the X6 register (left-justified). Spaces are ignored during assembly. If the end of the statement (indicated by a zero word in the string buffer) is encountered, or if a character is found which is not a letter, the assembly is terminated: the letters already processed, if any, are left-justified in the assembly register and control returned to the calling program.

Subroutines Called: none
Temporaries/Flags: none
Tables Referenced: none
Entry/Exit Register Conditions

Entry: B4 = address in string buffer where assembly is to begin

B2 = number of letters to be assembled

Exit: X6 = assembled letters, left-justified (zero if none assembled)

B2 = difference between number of letters requested to be assembled and number of letters actually assembled

B4 = address + 1 in the string buffer of the last letter assembled

Note: In the case where less than the requested number of letters were assembled, ASL exits with the non-alphabetic character which terminated the assembly in the X1 register.

ASM - ASSEMBLE MNEMONIC CODE

The ASM subroutine assembles the mnemonic code for an Ascent or Machine statement from the string buffer into the X6 register. First the routine scans over the leading blanks of thefield being assembled. Then up to four alphabetic characters are moved into the X6 register. The first number or separator will terminate the collecting of letters in the X6 register. This character will be left in the X1 register. The result in X6 will be left justified and the B2 register will contain a flag (see chart below) to indicate the instruction type.

Subroutines Called: None

Temporaries/Flags: MOD - Subprogram Mode

Tables Referenced: None

Entry/Exit Register Conditions:

Entry: B4 - contains address in the string buffer where assembly is to begin

Exit: X6 - opcode left justified

X1 - next non-alpha string character

Note: The Ascent mnemonics composed of 2 letter and a number are split between X6 and X1 (i.e., SX1 X6 = SX and X1 = 34)

B2	Value	Assembled Letters	Examples
	0	4 letters except BSSZ	FORTRAN STATEMENTS
	1	3 letters BSSZ	END ASCENT PSEUDO-OPS
	2	2 letters 2 letters + Number	ASCENT MNEMONICS
	3	1 letter	Machine Mnemonic
	4	0 letters	CONSTANT

ASN - ASSEMBLE NUMBERS

The ASN subroutine assembles consecutive numbers from the string buffer into the assembly register. The routine attempts to assemble 7 numbers. On entry to this subroutine, the B4 register contains the address of the location in the string buffer where assembly is to begin. The assembled letters are returned to the calling program in the assembly register, X6 (left-justified). Spaces are ignored during assembly and, in the case of statement number assembly (i.e., assembly starting address not greater than column 5), leading zeroes are ignored. Numbers are transferred to the assembly register until either a non-space, non-numeric character is encountered or seven numbers have been assembled: The contents of the assembly register are then left-justified.

If no numbers were encountered, the assembly count (B2 register) is set to zero, and a display code zero is placed, left-justified, in the assembly register.

Subroutines Called: none

Temporaries/Flags: none

Tables Referenced: none

Entry/Exit Register Conditions

Entry: B4 = address in string buffer where assembly is to begin

Exit: X6 = assembled numbers, left-justified (display code zero if none assembled)

B2 = 7 - number of numeric characters assembled (0 if none assembled)

B4 = address + 1 in the string buffer of the last letter assembled

Note: In the case where less than the requested number of digits were assembled, ASN exits with the non-numeric character which terminated the assembly in the X1 register.

ASV - ASSEMBLE VARIABLE

The ASV subroutine assembles consecutive alphanumeric characters from the string buffer into the assembly register. The routine attempts to assemble 7 such characters. On entry to this subroutine, the B4 register contains the address of the location where assembly is to begin. The assembled characters are returned to the calling program in the assembly register, X6 (left-justified). Spaces are ignored during the assembly. Characters are transferred to the assembly register until either 7 alphanumeric characters have been assembled or a non-space, non-alphanumeric character is encountered. If 7 consecutive alphanumeric characters are found, succeeding characters in the string buffer are read and examined until a non-space character is found. The contents of the assembly register are then left-justified.

The last character scanned is then examined to determine if it is an asterisk. If it is, and if this is an assembly program, it too is packed in the assembly register. Control is then returned to the calling program.

Subroutines Called: none

Temporaries/Flags: MOD - Subprogram Mode Indicator

Tables Referenced: none

Entry/Exit Register Conditions

Entry: B4 = address in string buffer where assembly is to begin

Exit: X6 = assembled characters, left-justified (0 if none assembled)

B2 = 7 - number of alphanumeric characters assembled

B4 = address of next non-space, non-alphanumeric character in the string buffer

Note: If an asterisk was packed in the assembly register, X1 is loaded with the character immediately following the asterisk in the string buffer. B4 is not advanced but still contains the address of this character.

BNX - BINARY OUTPUT ROUTINE

This routine is called at the end of compilation when the complete program including library subroutines are all in central memory. If a binary deck was requested, either the PBS or PBC PP punch routines are called. PBS is called if the mode of compilation is incomplete as it gives a status response when the punching is complete which PBC does not do.

A request is then made to CIO to write the program as a binary file whose name is in the same as the program name and the compiler remains in recall until the output has been completed. Another CIO request is sent to rewind the file. If the mode was not compile and execute, the routine exits. Otherwise, if a program was the first routine compiled, the name of the program is written into the dayfile and the PP routine EXU is called to read the program back in. AAB is called to adjust the program field length and BNX exits.

Subroutines Called: AAB - Adjust Program Field Length

Temporaries/Flags:

BOA, BOB, BOC, BOD - Binary Buffer Parameters

FST - Long File Start

ICM - Incomplete Compile Mode Indicator

INQ - Name for Dayfile

INV - Segment Indicator

IPS - Program/Subprogram Indicator

STG - Compile Mode Indicator

ZAA - Relative Start of Current Program or Subroutine

Tables Referenced: None

Entry/Exit Register Conditions: None

BRX - READ BINARY SUBROUTINES

This routine is called to read any subroutines that have been detected by the presence of a + in column 1 of the next input card. This routine transfers the binary routines from the input buffer to the compiled program area. It does its own requests to CIO when it is necessary, and does not use the RNX routine to read cards. When all routines have been transferred or when there is no room to continue the transfer, a zero entry is made and the routine exits.

Subroutines Called: None

Temporaries/Flags: INA - CIO Input Buffer Parameters

Tables Referenced: None

Entry/Exit Register Conditions:

Entry: X7 - address start of tables
X6 - start of the region to transfer routines to

Exit: None

CDC - CONVERT INSTRUCTION OR CONSTANT TO DISPLAY CODE

The CDC subroutine converts to display code the binary instruction formed from an Ascent or Machine source card. Upon entry into this routine, B1 contains a 5, 4 or 24B indicating a short, long or full word instruction, respectively. Detecting a short instruction, CDC will write one word into the output buffer and exit. The long instruction will call KOT to convert the binary tag to a mnemonic tag and thereby cause a two word entry into the output buffer. A full word instruction appends a zero word to its entry so that the buffer has three words stored. The exact form of these entries are shown in Figure CDC-1.

Subroutines Called: KOT - Convert Binary Tag to a Mnemonic Tag

Temporaries/Flags: None

Tables Referenced: None

Entry/Exit Register Conditions:

B1 24B, 5, 4

X1 binary instruction left-justified

CDC OUTPUT BUFFER ENTRIES

SHORT INSTRUCTION	[X X I J K 0 0 0 0 0]
LONG INSTRUCTION	[X X I J A B n n n n] [n 0 0 0 0 0 0 0 0 0]
FULL WORD INSTRUCTION	[CH_1 ... CH_{10}] [CH_{11}...CH_{16} 0 0 0 0]

XX	——	OPERATION CODE
I	——	RESULTANT REGISTER
J	——	FIRST OPERAND REGISTER
K	——	SECOND OPERAND REGISTER
ABnnnn	——	MNEMONIC TAG
CH_1-CH_{16}	——	OCTAL DIGITS

Figure CDC-1

CFF - COMPILE FUNCTION DEFINITION

This routine controls the processing of all arithmetic statement functions after they are detected by the SFO routine. These functions are handled in the same manner as a function subprogram except that they may only contain one statement while a function subprogram may contain many. The function is a closed subroutine, entered via a return jump and exited from via the entry point. A zero word is compiled for each function argument plus one zero word for the entry/exit line. Since these functions are allowed to appear anyplace in the program, it is necessary to compile a jump over the closed subroutime that the arithmetic statement function generated. When such a jump is generated, the address of the jump is saved in "CJP" - current jump address.

On entry, the current jump address is examined to see if there is a current jump pending. If there is, a check is made to see that it was generated by an ASF rather than a logical if statement and a diagnostic given if not. If there is no current jump pending, a jump instruction is compiled to a statement or H tag and the address of this compiled jump is entered into CJP for later use.

The arguments to the function are then processed one at a time. The argument name is entered into the Argument Name Table and a Function Tag is generated, grouped with the mode and index assignment of the argument (if any) and entered into the Argument Tag Table. The function name is then entered into the Function Name Table and its tag and a mode indicator are entered into the Function Tag Table. This tag also replaces the name of the function in the string, and then the list of arguments are squeezed out of the string.

TAB is called to normalize the statement and TIQ is then called to translate the rest of the string entries into appropriate tags. If the dominant mode of the expression is double or complex or if the expression references other subroutines, the string is searched and for each function argument that has an index register associated with it, the argument reference count is decreased by one and the index designation for the argument in the string is deleted.

UNP is then called to generate instructions to evaluate all expressions within parenthesis and replace these expressions with tags. An attempt is made to delete an unnecessary store if the function is a very simple one. Otherwise, CXP is called to compile the final answer and bring it to X6. An attempt is then made to have the answer end up in X6 and thus eliminate any unnecessary 10 instructions. Instructions are generated to convert the expression to the mode of the function and finally a jump instruction is compiled to exit through the function's entry point.

<u>Subroutines Called</u>:	ADF - Advance Table
	CLT - Clear Temporary Tables
	CXP - Compile Expression
	SCT - Scan Table
	TAB - Normalize Statement
	TIQ - Translate Individual Quantities
	UNP - Unpack Parenthesis
<u>Temporaries/Flags</u>:	ARF - Argument Reference Count
	ARG - Argument Count
	CJP - Current Jump Address
	IGX - Current Index Assignment
	INO - Dominant Mode Indicator
	MOD - Subprogram Mode
	STN - Statement Number
	TBE - E TABLE PARAMETERS
	TBF - F TABLE PARAMETERS
	TBI - I TABLE PARAMETERS
	TBJ - J TABLE PARAMETERS
	TBM - M TABLE PARAMETERS
	TGF - Function Tag
	TGH - Temporary or Statement Tag

<u>Tables Referenced</u>:	Function Name	(E)
	Function Tag	(F)

Argument Name	(I)
Argument Tag	(J)
Variable Name	(M)

<u>Entry/Exit Register Conditions</u>: None

CHAIN

The method of chaining employed within the Fortran compiler, RUN, is a complete overlaying process. No portion of the main program is available to any segment; likewise, no portion of one segment is available to another. Only the main program or one segment resides in central memory at a time. Arguments may be passed between the main program and a segment or between segments only through blank or numbered common. A segment may be called for execution more than once but the main program should not be recalled since it clears common and the buffers. The maximum amount of numbered or blank common used by any segment must either be declared within the main program or on the RUN card. No diagnostic will result if a segment declares more common than has been previously reserved. A portion of the segment would in that case be overlayed with common.

A job that requires segmentation must have a main program. Initialization code for a program clears common and the I/O buffers. For each file designated on the PROGRAM card a buffer is allocated and the names, number, and order of these files must agree for each SEGMENT card. The segments are separated in the job deck by end-of-record (7-8-9) cards. These records are compiled and written on the disk as individual named files. The name of which is the word following segment on the SEGMENT card. Therefore, if the job deck consisted of a main program and five segments (each separated by an end-of-record card), there would be six named files on the disk for this job.

Segments are called for execution by the statement CALL CHAIN (seg), where seg is the name appearing on the SEGMENT card. During compilation a calling sequence which passes the address of the segment name in B1 to the subroutine CHAIN is generated. The subroutine fetches the segment name at execution time and sends a request to CIO (circular input/output) to rewind the file before it is loaded into central memory. Certain parameters must be initialized before calling CIO so CHAIN sets them in the first five executable words of the calling routine. That is, the CIO buffer parameters are stored in the calling routine beginning at RA+2+n where n is the number of I/O files declared.

A dayfile message informing the user of which segment is next to be executed is made by MSG (peripheral package). A limit of 100B messages is standard so if many segments are called and the MESSAGE LIMIT error is reached with of two system changes will solve the problem:

a) increase the limit in MSG.

b) remove the call to MSG in CHAIN so that no segment calls are entered into the dayfile.

Another peripheral package EXU (executed compiled program) is used to locate the file with the requested segment name on the disk and read it into central memory. The file is loaded beginning at relative zero (RA) so that there is no linkage of segments. Only one segment (that portion of the job deck between two end-of-record cards) resides in central memory at a time. EXU also requests the central processor to begin executing the new file in central memory.

CIR

CIR is called to compile read instructions. The tag of the desired read along with a mode indicator and an index assignment, if any, are specified in X6 upon entry in the following format:

M indicates the mode of the tag and can range in value from one to seven. It is examined and directs the processing in the following order and essentially controls the type of instructions compiled.

M=7 This implies that the tag portion of X6 is a value rather than a tag. If the value is zero, a MXi 0 instruction is compiled to set the value of Xi to zero, while if the value of the tag is minus zero, a MXi 60 is compiled to set the value of Xi to minus zero. CIR then exits.

M=5, 6 This indicates that the tag portion of X6 is a double or complex tag and thus requires the fetching of two central memory words. If the address of the tag is assigned to an index register (B will be the index register the address is in) a SAi Bj is compiled. If B is zero, the A register associates are searched to see if the address is associated with an address register. If so, a SAi Aj is compiled. If not, a SAi TAG is compiled. The tag itself is examined to see if it is an indirect or location tag. If so the instruction just compiled will be reading an address and a SAi Xi is compiled to bring the desired value to Xi. Then a $SA_{(i+1)}$ Ai+1 is compiled to fetch the second part of the double or complex number. The Xi and Ai register associates are set to this tag.

M=3 This indicates that the tag portion of X6 is a number whose value is less than 2^{16}. The X register associates are searched to see if this value is already in an X register. If it is, a BXi Xj is compiled and if not a SXi CONSTANT is compiled. In either case, the Xi register associate is set to this value before CIR exits.

If the mode of the tag was not one of the above, it is assumed to be a tag for a logical, integer, or real value and will require the fetching of one central memory word. The same method of determining the type of instruction to be compiled is employed as when the mode was double or complex (5,6) except that the final $SA_{(i+1)}$ Ai+1 is not compiled.

<u>Subroutines Called:</u>
ALX - Assign Long Register
ANI - Analyze Possible Index Read
ANR - Analyze Read Tag

<u>Temporaries/Flags:</u>
BIT - Bypass Interregister Transfer Indicator
INL - Logical If Indicator
INN - Mode Indicator for Read
RGX - Long Register Assignment
VTA - A Register Associates
VTY - X Register Associates

<u>Tables Referenced:</u> None

<u>Entry/Exit Register Conditions:</u>

Entry: X6 =

	42		18	
TAG		B		M

B - Index Assignment
M - Mode Indicator

Exit: X6 =

	3	
	R	

Register Assignment

CKD - CHECK MISSING DO NUMBERS

The DO Number Table is searched to see if there are any entries left that are not pseudo DO numbers. If there are, this indicates missing DO numbers and these are listed.

Subroutines Called: WST - Write Special Tag

Temporaries/Flags: None

Tables Referenced: DO Number (G)

Entry/Exit Register Conditions: None

CKL - CHECK MISSING SUBROUTINES

The list of subroutines requested via the CLL call is examined to see if there are any missing. This will be noted because of the fact that none of the missing ones will have a starting address and those missing subroutines will be listed.

Subroutines Called: WST - Write Special Tag

Temporaries/Flags: None

Tables Referenced: None

Entry/Exit Register Conditions:

Entry: B1 - start of the list of routines requested via CLL
Exit: None

CLL - PROCESS CALL STATEMENT

A CALL statement transfers control to a subroutine. Actual parameters may be exchanged between the calling program and the subroutine. No more than 60 parameter may be passed and successive calls to the same routine do not have to agree in the number of parameters used. Calling a subroutine with more actual parameters than formal parameters specified causes a diagnostic during compilation. A function call is compiled in the same manner as a subroutine call except that a value is returned in X6 from the function which must be saved upon reentry to the calling program.

A full word is reserved for each parameter on a SUBROUTINE or FUNCTION card. Each of these words receive a location tag during compilation. The first six arguments in a call are passed through index registers B1-B6 and the remaining addresses are compiled to be stored via external tags in their corresponding reserved word, which has a location tag, in the subroutine. Initial instructions in the subprogram pack the addresses, three per word, from the index registers in two temporary cells. B1 is saved in the lowest 18 bits of the first word with B2 and B3 packed in the next two 18 bit portions. The next temporary cell holds B4-B6 with the address in B4 residing in the lowest 18 bits. When one of these packed addresses is needed and is not available in the index register, then the proper temporary cell is read and the address is unpacked.

Two Fortran subroutines CHAIN and DUMP/PDUMP are specifically processed by CLL. The name of the segment called by CHAIN is replaced by a constant tag and the name is entered into the constant value table. A DUMP/PDUMP indicator is set for calls to these subroutines. After these initial checks and replacements are made, a call to CHAIN or DUMP/PDUMP is processed as any other subprogram request.

Only a routine with an external or a location tag may be called. The subroutine name is entered in the subroutine name table and may have the same name as the program. There will be no conflict because the program name is the first entry in the subroutine name table, so another entry is made for a subroutine name that is the same. When an actual parameter is the name of a function or subroutine, that name must also appear in an EXTERNAL statement in the calling program. This statement causes an external or library tag to be generated for the subprogram name. This name will have a location tag in the subprogram called which used the name as an argument. Therefore, the name of the subprogram being called may have only a library or location tag.

A call that has arguments allows an arithmetic statement function as an argument. TIQ translates the individual arguments into appropriate tags, or constants. The segment called by CHAIN has already been given a constant tag. Any arithmetic expression or subscripted variable will not be evaluated but each portion of the expression will be replaced with the tag generated by a previous definition or assigned a tag at this point.

UNP directs the processing of expressions imbedded within parenthesis and function references. The outermost set of parentheses are removed and temporary tags are generated to save the information. A call is compiled to the function referenced whether it be an arithmetic function or a function subprogram and the answer which is returned in X6 is saved. (CRF is called by UNP for this purpose).

Upon return from TIQ and UNP, all the arguments have been replaced with tags, the function references processed, and the arithmetic expressions simplified by removing the imbedded parenthesis. PRR (process function/subprogram reference) is called to pass the addresses of the arguments to the subprogram. Any subscripted variable has the variable and subscript replaced with one tag when the array address is determined in the SAD (sense and process single array address) routine. The arithmetic expression partially processed by UNP is completely evaluated by CXP (compile expression). The expression is replaced in the string buffer by a temporary tag which saves the result of the expression. When all the arguments have been evaluated, the addresses of the first six are set into index registers B1-B6, and the remaining ones are to be stored in their corresponding reserved word via an external tag. A DUMP/PDUMP call causes the number of arguments to be passed in B7 and the field length to be set in X0.

If the subprogram called was not an argument to the subroutine, then a return jump to the subprogram is compiled. The return jump instruction is forced to the upper portion of the word. The lower 30 bits contain the number of arguments in the reference to the subroutine and a tag corresponding to the location of the first word of the calling program or subroutine. The contents of the word has the subroutine's name in the left adjusted display code if the reference is from a subroutine; otherwise, the location is actually RA, which will be zero, if the call is made from a program or segment. Example -

```
             0100 S00600
             0715 L00002
    where    S00600    is location of the entry/exit word of the subroutine
                 15    is the number of arguments in the call
             L00002    is the location of the name of the subroutine.
```

A subroutine used as an argument is a special case. Instead of entering the subroutine via a return jump, instructions are compiled to insert the proper return address in the entry/exit line and generate an unconditional jump to the first executable instruction of the subroutine. In this way a subprogram may call any one of many subroutines depending upon the argument passed from the main program. Each of the subroutines used as an argument to the subprogram must have been declared external to the main program - otherwise the argument is assumed to be a simple variable.

When the call to the subroutine has been generated, then instructions are compiled to restore the argument addresses to index registers if the called subroutine was used as an argument to this subprogram or a function was used as an argument in this call. PSN is called to process the next statement when the call statement processor has completed.

SUBROUTINES CALLED:	ADF	Advance Table
	CLT	Clear Temporary Table
	CRI	Compile Restore Instruction
	PPR	Process Function/Subprogram Reference
	SCT	Scan Table
	TAB	Normalize Statement
	TIQ	Translate Individual Quantities
	UNP	Unpack Parenthesis
TEMPORARIES/FLAGS	ARF	Argument Reference Count
	FAG	Function Argument Use
	FSR	Function Statement Reference Count
	ICE	Multiple Statement Count

INF	DUMP/PDUMP Indicator
RJC	Return Jump Count
SIR	Subroutine Reference Count
TBA	A Table parameters
TBB	B Table parameters
TBM	M Table parameters
TBS	S Table parameters
TBU	U Table parameters
TGK	Constant Tag
TGL	Library Tag
TML	Argument Count
TMM	Subroutine Name
TMN	Subroutine Tag

TABLES REFERENCED:

Constant Name (A)
Constant Tag (B)
Variable Name (M)
Variable TAG (N)
Array Tag (P)
Subroutine Name (S)
Subroutine Tag (T)
Subroutine Parameter (U)

CNF - COMPILE NORMAL FORMULA

CNF is entered when SFO detects an arithmetic replacement statement. It controls the processing of the statement, the conversion of the expression evaluation to that of the answer, and the storing of the answer.

Upon entry, the last statement is checked to see if it was a conditional statement and if so, the expression is cracked immediately. If it wasn't the current jump cell (CJP) is examined. If there was a current jump, the cell is cleared and the expression is cracked. If there was no current jump, but there was a statement number, the expression is cracked. Otherwise, the continue indicator is cleared and if the last statement was not a CONTINUE, X7 is cleared to wipe out any program tag.

In order to evaluate the statement, TAB is called first to normalize the statement, TIQ is called to change the variables and constants to tags and then UNP is to control the compilation of instructions to evaluate and save all portions of the expressions that were imbedded in parentheses. Upon return from UNP, all expressions that were in parentheses are replaced with temporary tags.

A check is then made to see if the expression was a simple one. If so, and the right side is a constant, an instruction is compiled to set X6 to this constant and then go to the portion of the routine that takes care of converting and storing the answer.

If the right side was not a constant, but rather represented by a temporary tag, an attempt is made to delete a store. If the store is deleted, an attempt is made to delete a 106 instruction if there was one. Then the answer is converted and stored. If the right side was not represented by a temporary and was not a constant, or if it was a temporary but the last instruction did not store the answer into this temporary, CXP is called to compile instructions to evaluate what is left of the arithmetic expression.

Thus, CXP is the routine that finally compiles the last of the statement and brings the answer to X6 (and X7 if mode of expression is double or complex). An attempt is made to delete an extra 106 instruction by making the result of the last arithmetic operation X6. Instructions are then generated to convert the mode of the calculated answer to that of the left number of the statement if they are necessary.

The answer is now in X6 (and X7) and is ready to be stored in memory. If the address that the answer should be stored in is in an index register or if the address does not have to be calculated CIW is called to compile the proper write instruction and CNF exits to PSN (Process Statement Number). Otherwise, a search backwards of the compiled instructions is made to see if there is a register free between the present location and that of the temporary store of the address for the answer. If there is, an attempt is made to delete this address store and use the address as it is in the register. Otherwise, CIW is again called to compile the proper store instructions and then an exit is made to PSN.

Subroutines Called:

- ALX - Assign Long Register
- CIW - Compile Write Instructions
- CXP - Compile Expression
- TAB - Normalize Statement
- TIQ - Translate Individual Quantities
- UNP - Unpack Parenthesis

Temporaries/Flags:

- CJP - Current Jump Indicator
- INK - Continue Indicator
- RGX - Long Register Assignment
- STN - Statement Number

Tables Referenced: None

Entry/Exit Register Conditions

Entry: B6 $\neq$ zero if this is part of a conditional statement

Exit: None

COM - PROCESS COMMON LIST

When the MAA (Process Machine or Ascent Records) subroutine encounters a COM declarative, the COM subroutine (Process Common List) is called. The COM declarative permits the programmer to allocate blank common storage by indicating the number of words and an identifier for the first word of the array. An example of the COM declarative together with the basic steps in processing the list is shown in Figure COM-1.

On entering the COM subroutine, the TAB subroutine is called to normalize the list. If the mode indicator shows anything other than FORTRAN II, a zero block name is entered into the Common Name Table (Table O). The Variable Name Table (Table M) is scanned to determine if the identifier has been previously entered and, if not, the identifier is entered into the Variable Name Table. If the variable has previously been entered in the Variable Name Table, an error exit (Duplicate Tag Error) occurs. The variable name is also checked to insure that the first character is alphabetic and that the name is composed of two or more characters.

Next, a check is made for the equal sign, and the CVN (Convert Number) subroutine is called. A machine constant error exit is taken if the constant is negative or greater than $2^{17}-1$. If the constant is in the proper range it is stored in the Array Parameter Table (Table Q). An Array Tag (W-tag) is generated and stored in the corresponding Array Tag Table (Table P) and Variable Tag Table (Table N).

Processing of the list entries continues in the manner described above until a right parenthesis, indicating the end of the list, is encountered.

<u>Subroutines Called</u>: ADF - Advance Tables

CVN - Convert Octal and Decimal Numbers

SCT - Scan Table

TAB - Normalize Statement

CØM DECLARATIVE PROCESSING

EXAMPLE: COM(B1=1, B2=2, B3=3)

1. STORE VARIABLE NAME IN VARIABLE NAME TABLE
2. CHECK FOR EQUAL SIGN
3. CONVERT CONSTANT
4. STORE CONSTANT IN ARRAY PARAMETER TABLE
5. IF NEXT CHARACTER IS A COMMA, REPEAT 1 - 5
6. CHECK FOR A RIGHT PARENTHESIS
7. RETURN FOR THE NEXT SOURCE CARD

Figure COM-1

Temporaries/Flags: MOE - Program Mode (set)

Tables Referenced: TBM - Variable Name

TBN - Variable Tag

TBO - Common Name

TBP - Array Tag

TBQ - Array Parameter

Entry/Exit Register Conditions

CON - PROCESS CONSTANT LIST

When the MAA (Process Machine or Ascent Records) subroutine encounters a CON declarative, the CON subroutine (Process Constant List) is called. The CON declarative stores the constant in the list and tags the storage location with the identifier in the list. An example of the CON declarative together with the basic steps in processing the list is shown in Figure CON-1.

On entering the CON subroutine, the TAB subroutine is called to normalize the list. The Variable Name Table (Table M) is scanned to determine if the variable which has been equated to the constant has been entered, and, if not, the variable is entered in the Variable Name Table. If the variable has previously been entered in the Variable Name Table, an error exit (Duplicate Tag Error) occurs. The variable name is also checked to insure that the first character is alphabetic and that the name is composed of two or more characters.

Next, a check is made for the equal sign, and the CVN (Convert Number) subroutine is called. The constant is stored in the Hollerith Word Table (Table A), and a Constant Tag (K-tag) is generated and stored into the corresponding Hollerith Tag Table (Table B) and Variable Tag Table (Table N).

Processing of list entries continues in the manner described above until a right parenthesis, indicating the end of the list, is encountered.

Subroutines Called: TAB - Normalize Statement
SCT - Scan Table
ADF - Advance Table
CVN - Convert Octal and Decimal Numbers

Temporaries/Flags: TGK - Constant Tag (set)

CØN DECLARATIVE PROCESSING

EXAMPLE: CØN(C1 = 25, C2 = 777B, C3 = 6.54E-2)

1. STORE VARIABLE NAME (e.g., C1) IN VARIABLE NAME TABLE
2. CHECK FOR EQUAL SIGN
3. CONVERT CONSTANT
4. STORE CONSTANT IN CONSTANT VALUE TABLE
5. IF NEXT ENTRY IS A COMMA, REPEAT 1 - 4
6. CHECK FOR RIGHT PARENTHESIS
7. RETURN FOR NEXT SOURCE CARD

Figure Con-1

Tables Referenced: TBM - Variable Name Table

TBA - Constant Table

TBB - Constant Tag

TBN - Variable Tag

Entry/Exit Register Condition: DNA

Note: Error Exits: EMT - Machine Tag Definition Error

EMD - Machine Duplicates Tag Error

EMF - 1 Format Error

CRF - COMPILE FUNCTION REFERENCE

CRF is called to control the processing of a subroutine or function reference. Upon entry, the string entry containing the tag for the function is specified. Control will be routed to three routines depending upon the type of subroutine reference. If it is a built-in function, PBR is called to evaluate it; for an arithmetic statement function, PFR is called while PRR is called for a function/subprogram reference. Upon return from these routines, CRF will exit.

Subroutines Called: PBR - Process Built-in Function
PFR - Process Statement Function
PRR - Process Function-Subprogram

Temporaries/Flags: TML - Argument Count for Call
TMM - Name Tag for Call
TMN - Argument Tag for Call

Tables Referenced: None

Entry/Exit Register Conditions:

Entry: B5 + address of start of subroutine reference
Exit: X7 = zero

CVN - CONVERT OCTAL OR DECIMAL NUMBER

The CVN subroutine converts an octal or decimal number. It is used for constants in the address field of either a Machine or Ascent instruction. The unsigned constant is sent to DEC (Convert Decimal Number) and if the mode is Ascent, the sign is restored and a return to the calling program with the value in the X6 register occurs. However, if the mode is Machine, the B6 register is set to one and X6 contains the signed converted constant. Also leading blanks of the address field are compressed for machine instructions.

Subroutines Called: DEC - Convert Decimal Number

Temporaries/Flags: MHI - Machine Instruction

Tables Referenced: None

Entry/Exit Register Conditions:

X6 - Converted Constant
B6 - if Machine = 1

CXP - COMPILE EXPRESSION

CXP controls the evaluation of all arithmetic expressions, whether the expression is part of an arithmetic replacement statement, in an argument list, or the arithmetic expression of an IF statement. The starting address of the expression is specified upon entry and instructions will be compiled to evaluate the expression until a left parenthesis or comma is found that is not part of an array reference, or until end of statement has been reached.

Instructions are first compiled to calculate the address of all array entries within the expression. CSR (Compile Subscripted References) is called to compile these instructions and it will bring the address of the array entry to a specified index register. As each array address is calculated, the actual entry in the string is changed to an indirect tag along with an indication of which index register the address in in, and the rest of the array entry is squeezed out of the string. After index register 6 has been used, A0 is used to hold the next array address, B7 is used to hold the address of the last array entry and all addresses in between are saved in indirect cells.

It is assumed that by the time CXP is entered, all addresses for array entries that appear more than once in the statement have already been calculated and saved in an indirect cell and CXP makes no check to see if this has been done. After instructions have been compiled to determine the address of all array entries in the expression and the entries have been replaced by tags, the expression is ready to be evaluated. The HEX routine is called first to evaluate any exponentials within the statement. It will compile instructions to evaluate these exponentials, and store the answer into a temporary cell. The string entry for the exponential will be replaced by this temporary tag and the expression will be squeezed down. Since HEX has to examine every entry in the expression, it will also determine the dominant mode of the expression and, if there are any logical relations, it will set the logical relation flag.

When HEX returns to the CXP routine, the logical relation flag is checked and the HLR routine to handle logical relations is entered if any have been detected. Depending upon the dominant mode of the expression, CXP will then branch off to a routine to handle each mode. Generally, these routines are responsible for compiling instructions to evaluate the rest of the expression, converting all entries in the expression to the dominant mode if they are not in that mode already, and finally bringing the result of the expression to X6 and X7 if the dominant mode is double or complex. CXP will then change the last string entry of the expression to flag the dominant mode of the expression and exit.

Subroutines Called:

AAR - Analyze Array Reference

ANK - Analyze Address Generating Instructions for Right Number

ALX - Assign Long Register

BEX - Compile Simple Boolean Expression

CSR - Compile Subscripted Reference

FEX - Compile Simple Floating Expression

HEX - Handle Exponentials

HLR - Handle Logical Relations

JEX - Compile Simple Integer Expression

KEX - Compile Simple Complex Expression

LEX - Compile Simple Logical Relation

MEX - Compile Simple Double Expression

Temporaries/Flags:

ARI - Array Reference Count

BIT - Bypass Interregister Transfer Indicator

HIC - Highest Index Count

ICL - Simple Logical Relation Indicator

ICU - Index Tag

ICV - Upcoming Statement and Unpack Indicator

IGX - Current Index Assignment

INM - Logical Relation Indicator

INO - Dominant Mode Indicator

INX - Upcoming Statement Indicator

INY - Complete Unpack Indicator

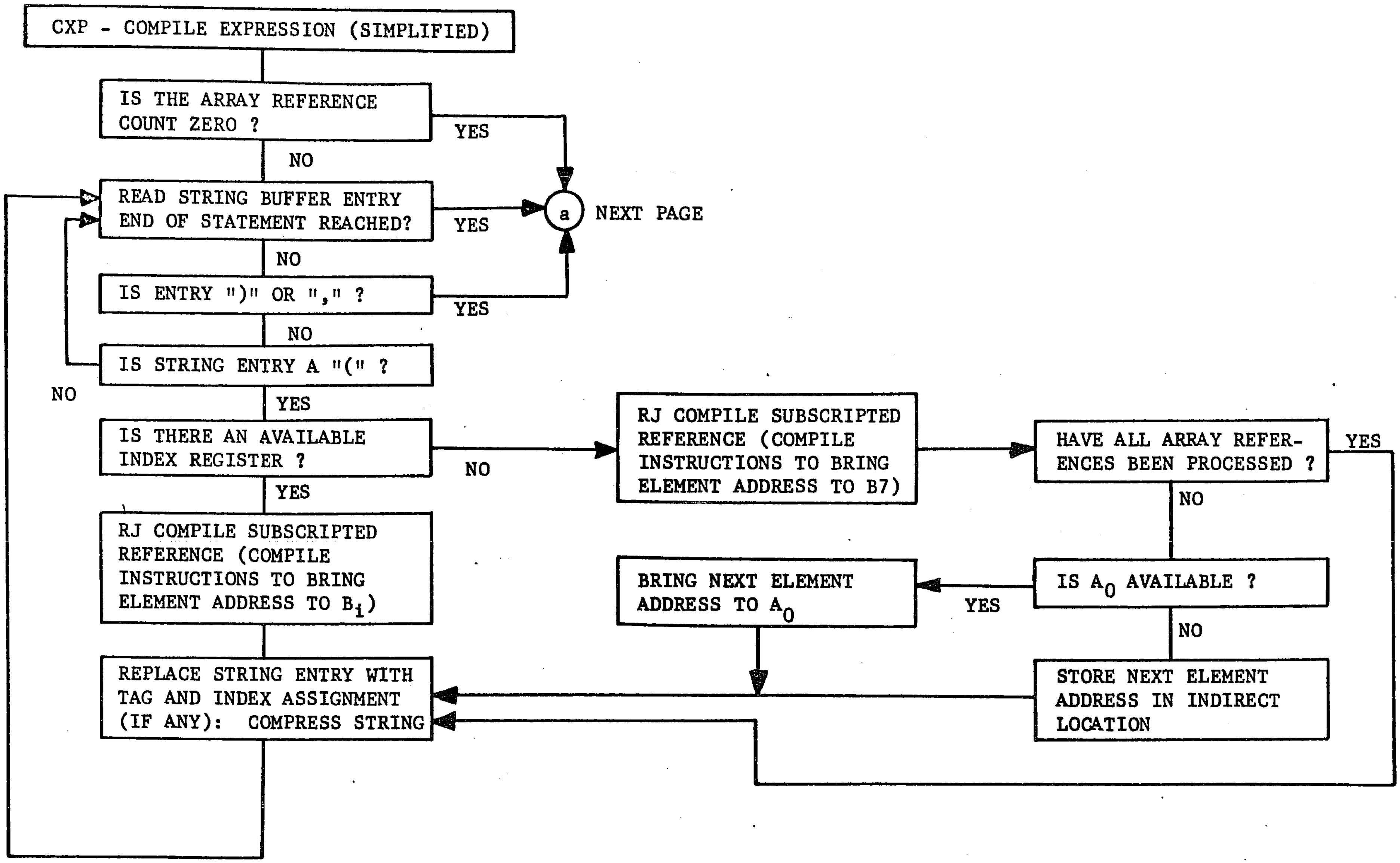
SAR - Single Array Reference Count
TGI - Indirect Tag
TGT - Temporary Tag
TMF - Start of Array Reference
TMG - Expression of Index Assignment
TMH - Start of Expression
VTA - A Register Associate
VTY - X Register Associate

Tables Referenced: None

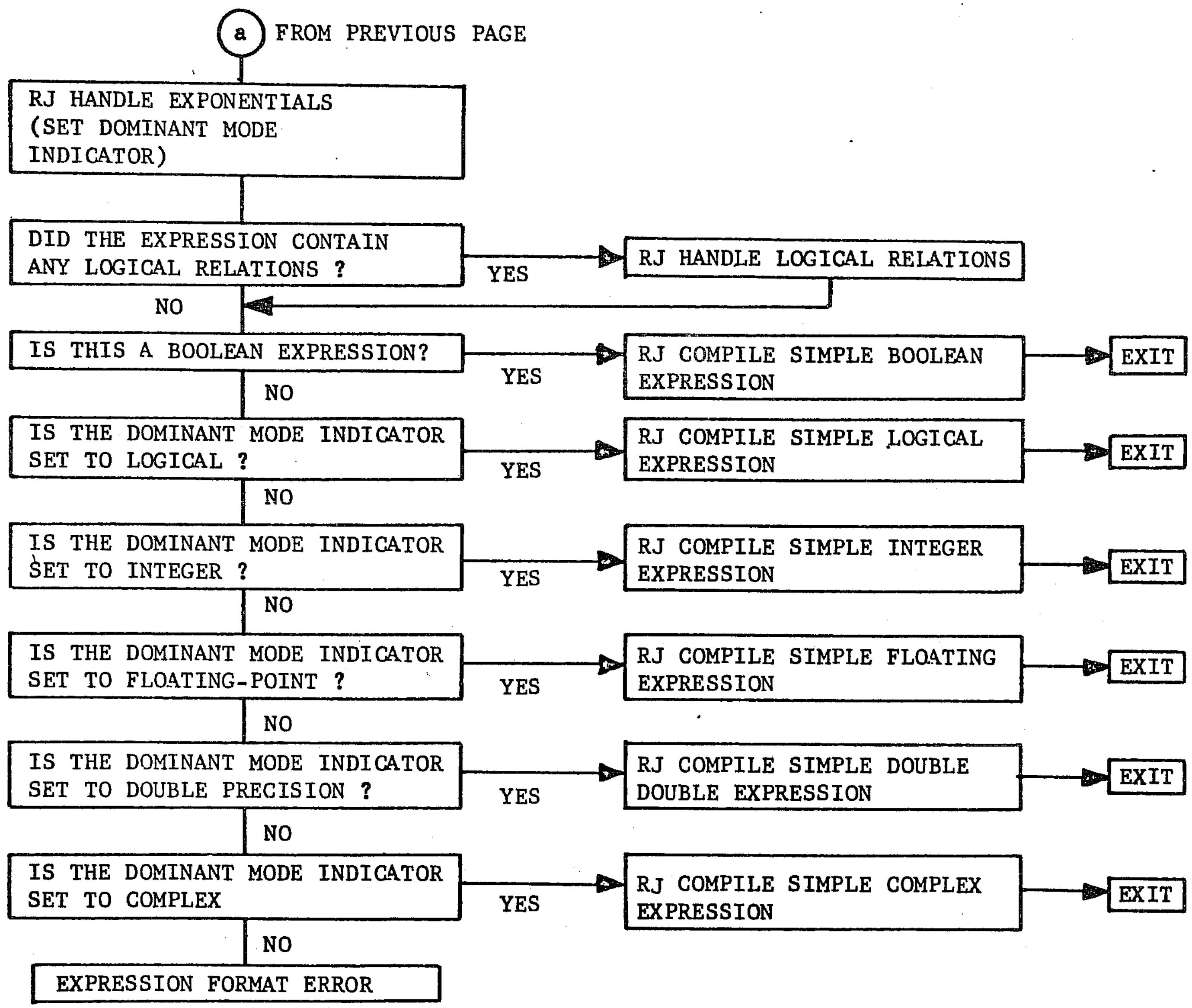
Entry/Exit Register Conditions:

Entry: B5 - Address of the start of expression

Exit: None



CXP-COMPILE EXPRESSION


a FROM PREVIOUS PAGE
RJ HANDLE EXPONENTIALS (SET DOMINANT MODE INDICATOR)
DID THE EXPRESSION CONTAIN ANY LOGICAL RELATIONS ?
YES
RJ HANDLE LOGICAL RELATIONS
NO
IS THIS A BOOLEAN EXPRESSION?
YES
RJ COMPILE SIMPLE BOOLEAN EXPRESSION
EXIT
NO
IS THE DOMINANT MODE INDICATOR SET TO LOGICAL ?
YES
RJ COMPILE SIMPLE LOGICAL EXPRESSION
EXIT
NO
IS THE DOMINANT MODE INDICATOR SET TO INTEGER ?
YES
RJ COMPILE SIMPLE INTEGER EXPRESSION
EXIT
NO
IS THE DOMINANT MODE INDICATOR SET TO FLOATING-POINT ?
YES
RJ COMPILE SIMPLE FLOATING EXPRESSION
EXIT
NO
IS THE DOMINANT MODE INDICATOR SET TO DOUBLE PRECISION ?
YES
RJ COMPILE SIMPLE DOUBLE DOUBLE EXPRESSION
EXIT
NO
IS THE DOMINANT MODE INDICATOR SET TO COMPLEX
YES
RJ COMPILE SIMPLE COMPLEX EXPRESSION
EXIT
NO
EXPRESSION FORMAT ERROR

DEC - CONVERT DECIMAL NUMBER

The DEC subroutine converts a decimal constant to its binary equivalent: it also checks for octal constants of the form nnn....nB. (Note: octal constants of the form ∅nnn....n which appear in arithmetic statements are recognized by the Translate Variable subroutine.) DEC is called when a numeric entry is recognized in a DATA statement or arithmetic statement. On entering DEC, the string buffer address of the numeric entry is contained in the B5 register. When the TAB subroutine normalized the statement, it employed the ASN subroutine to pack digits into words. The ASN subroutines assembles up to seven digits per word, so numbers in the string buffer may occupy several entries. For example, a 20 digit octal constant would appear in the string buffer as shown in figure DEC-1a. Similarly, decimal constants may occupy several words. For example. a floating-point number such as 37.84625184E-40 would appear in the string buffer as shown in figure DEC-1b.

DEC searches the next three entries following the numeric entry to determine if the number is followed by a "B". If it is, the Convert Octal Constant (OCT) routine is called to convert the number. If the numeric entry was not followed by a B, this and succeeding entries are read, converted to binary, and packed in an assembly register. Conversion continues until a non-numeric entry is encountered or until more than 18 digits have been processed. The latter condition results in an error exit. The non-numeric which terminated this part of the conversation is examined to see if it is a period (i.e., a decimal point): if it is not, then the constant is an integer constant. If the non-numeric is a decimal point, then the entries following the decimal point are read, converted to binary, and packed in the assembly register. Conversion again continues until a non-numeric entry is encountered or until more than 18 digits have been processed. The number of digits in the fractional part are saved to be used later in computing the proper exponent value, and the assembled binary number is converted to floating-point and normalized.

Word 1	Word 2	Word 3	Word 4
nnnnnnn	nnnnnnn	nnnnnn	B

Figure DEC-1a: FORMAT OF A 20-DIGIT OCTAL NUMBER IN THE STRING BUFFER

Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7
37	.	8462518	4	E	-	40

Figure DEC-1b: FORMAT OF A FLOATING-POINT CONSTANT, 37.84625184E-40, IN THE STRING BUFFER

The non-numeric entry which terminated the conversion of the fractional part of the number is then examined: if it is a D or an E, the sign of the exponent is stored and the exponent converted to binary. This exponent is then combined with the number of digits in the fractional part of the number, and converted to the appropriate powers of two.

Subroutines Called: OCT - Convert Octal Constant

Temporaries/Flags: none

Tables Referenced: Table of Powers of 10^{2^n} (REG)

Entry/Exit Register Conditions

Entry: B5 = address of numeric entry in string buffer

Exit: B5 = address + 1 of last entry processed (i.e., the string buffer address of the entry following the number)

X6 = number

X2 = 0 (second word of a double precision conversion)

B6 = Mode Indicator

Note: The mode indicator is set as follows:

Mode	Constant
1	Octal
2	Integer
4	Floating Point
5	Double Precision

FIN - FORM INSTRUCTION

In the processing of Ascent or Machine records, an intermediate language is generated and stored in the string buffer. The FIN subroutine forms the octal instruction by examining the string buffer which has been flagged by the RDA routines. FIN first determines if the instruction is a long (30-bit) instruction. An equal sign in column 8 and either a positive result in the TOS Table look-up or entries 0-3 of the TOI Table look-up (see Appendix B) are 30-bit instructions. For these long instructions, FIN can determine the instruction by examining columns 7, 12 and 13. If column 7 is a P or R and column 12 is a period, then column 13 defines which 03 instruction should be formed. The X, A and B registers are always indicated by an A, C or B respectively, and a K address is flagged with a G. If column 7 is a P or R and column 12 is not a period, then column 12 describes instructions 04-07.

If column 7 is an A, C or B, it describes the resultant or "i" register and a table look-up provides the exact instruction. The tables searched are TOI for instructions 50-77, TOP for instructions 11-13, 15-17, 30-42, 44-45 and TOS for 10, 14, 20-27, 43, 47 and certain long instructions with implied XO registers (see Figure RDA-1 for the flags corresponding to given instructions). Once the instruction is selected, a jump to FSI (Form Short Instruction) or FLI (Form Long Instruction) is taken.

Entry conditions are the flagged string, III, JJJ and KKK register constants. X5 contains the K value where applicable and X7 contains the current word address. Calls to further routines combine the opcode and registers.

<u>Subroutines Called:</u> SCS - Special Search
FSI - Form Long Instruction
FLI - From Short Instruction

Temporaries/Flags:	III - i Portion of Machine Word
	JJJ - j Portion of Machine Word
	KKK - k Portion of Machine Word
Tables Referenced:	TOI - Table of Format Checks on Instructions 50-77
	TOP - Table of Identifying Characters
	TOS - Table of Special Formats

FLI - ASSEMBLE LONG INSTRUCTION

When the FIN (Form Instruction) subroutine encounters a 30-bit Ascent or Machine instruction, the FLI subroutine is called. The FLI routine assembles these instructions as a 6-bit opcode, a 3-bit resultant register, a 3-bit operand register and a 15-bit address. Then it arranges these instructions in the parcels of a 60-bit machine word. If an RJ instruction is found, a full word is indicated. Upon entry, X0 contains the opcode right-justified, B1 and B2 contain the resultant and operand registers respectively. FLI calls the CDC (Convert Instruction or Constant Tag to Display Code) subroutine to form and insert in the output buffer the display code equivalent for this instruction. Next, the routine checks the intraword count and if the 30-bit instruction cannot be inserted, a call to the ARA (Adjust Address and Write Registers) subroutine is made and the 30-bit instruction goes to parcels 0 and 1 of the next word. If the 30-bit instruction can fit, a test is made to properly place this instruction and the intraword counter is incremented by 2. Upon exit from this routine, both the display code and binary forms have been recorded.

Subroutines Called: CDC - Convert to Display Code
ARA - Adjust Address and Write Register

Temporaries/Flags: ICT - Intraword Instruction Counter (set)

Tables Referenced: None

Entry/Exit Register Conditions:

X0 opcode (binary)
B1 - i portion of instruction else 0
B2 - j portion of instruction
X7 - current instruction word address

FSI - ASSEMBLE SHORT INSTRUCTION

When the FIN (Form Instruction) subroutine encounters a fifteen bit Ascent or Machine instruction, the FSI subroutine is called. The FSI subroutine assembles a six bit opcode, three bit resultant register (i) and two three bit operand registers (j, k), then stores the display code equivalent in the output buffer. Upon entry into this routine, XO contains a table (TOP, TOS or TOI) entry. If bit 18 of XO is set then a test is made on bits 6-11, if 6-11 are zero, B1, B2 and B3 contain the correct ijk register respectively. If bits 6-11 are equal to one, then X5 contains the corrected values for the i, j and k registers and if bits 6-11 equal two, the j and k registers must be interchanged. These bits may contain another value only if the i and j registers are equivalent. If bit 18 is not set, then B1, B2 and B3 contain the i, j and k registers and XO contains the octal op code. A call is made to CDC (Convert Instruction or Constant Tag to Display Code) then a check is made as to the correct parcel in the current word. Since only one fifteen bit instruction is assembled, the intraword counter is incremented by one. Upon exit from this routine, both the display code and binary forms of the instruction have been recorded.

Subroutines Called: CDC - Convert Instruction or Constant Tag to Display Code

Temporaries/Flags: ICT - Intraword Counter (set)

Tables Referenced: None

Entry/Exit Register Conditions:

B1	resultant register
B2	first operand register
B3	second operand register
XO	opcode
X5	register values under certain conditions
X7	current instruction word address

HOL - PROCESS HOLLERITH LIST

When the MAA (Process Machine or Ascent Records) subroutine encounters an HOL declarative, the HOL subroutine (Process Hollerith List) is called. The HOL declarative allows a ten character group to be stored in display code and tagged with an identifier. An example of the HOL declarative together with the basic steps in processing the list is shown in Figure HOL-1.

On entering the HOL subroutine a jump is taken to ASV (Assemble Variable) to isolate the identifier. If the identifier is not followed by an equal sign an error exit (Machine Format Error) is taken. The identifier is also checked to insure that the first character is alphabetic and that the identifier is composed of two or more characters. Then the Variable Name Table (Table M) is scanned to determine if the identifier had been previously entered, and, if not, the identifier is entered into the Variable Name Table. If the identifier had previously been entered in the Variable Name Table, an error exit (Duplicate Tag Error) occurs.

Next, ten characters are assembled from the string. These ten characters are entered into the Hollerith Word Table (Table A). A Constant Tag (K-tag) is generated and stored into the corresponding Hollerith Tag Table (Table B) and Variable Tag Table (Table N).

Processing of the list entries continues in the manner described above until a right parenthesis, indicating the end of the list, is encountered.

<u>Subroutines Called</u>: ASV - Assemble Variable
SCT - Scan Table
ADF - Advance Table

<u>Temporaries/Flags</u>: TGK - Constant Tag (set)

HOL DECLARATIVE PROCESSING

EXAMPLE : HOL(H1=ABCDEFGHIJ, H2=1234567890)

1. CHECK FOR AN EQUAL SIGN
2. STORE VARIABLE NAME (e.g., H1) IN THE VARIABLE NAME TABLE
3. ASSEMBLE TEN CHARACTERS
4. STORE THE HOLLERITH FIELD IN THE CONSTANT TABLE
5. GENERATE A CONSTANT TAG AND STORE IN TABLE
6. IF THE NEXT ENTRY IS A COMMA, REPEAT 1- 5
7. CHECK FOR A RIGHT PARENTHESIS
8. RETURN FOR THE NEXT RECORD

Figure Hol-1

Tables Referenced: TBM - Variable Name Table
TBA - Hollerith Word Table
TBB - Hollerith Tag
TBN - Variable Tag

Entry/Exit Register Conditions: DNA

Note: Error Exits: EMT - Machine Tag Definition Error
EMF - Machine Format Error
EMD - Machine Duplicate Tag Error

IFH - PROCESS IF SENSE STATEMENT

IFH is called from IFS when it is determined that the IF might be a SENSE SWITCH or SENSE LIGHT type IF. After some initial checking to see that the format of the statement is correct, and determining which type sense it is, the two branches of the statement are changed to tags. Instructions are then generated to read RA of the program, to mask off the declared switch or light (actually these are the same), and a zero jump is compiled to the second branch. If the statement was a SENSE LIGHT IF, instructions are compiled to turn this light on. Finally, instructions are compiled to jump to the other branch of the statement if it is not the same as the upcoming statement number, and control is transferred to PSN to process the statement number.

Subroutines Called: ASL - Assemble Letters
ASN - Assemble Numbers
ISN - Identify Statement Number
CUN - Tag Upcoming Statement Number

Temporaries/Flags: None

Tables Referenced: None

Entry/Exit Conditions:

Entry: None
Exit: None

IFL - PROCESS LOGICAL IF STATEMENT

IFL is entered once the IFS routine determines that the statement is not an I/O type IF and that the first entry past the right parenthesis is not a statement number. At this time, the last parenthesis is replaced by a zero entry to flag the end of the statement and the statement is normalized starting with the second parenthesis. The first parenthesis is replaced by an equals sign in order to simulate an arithmetic expression and the individual entries in the statement are translated to proper tags. UNP is called to control the evaluation of all portions of the expression that are imbedded in parenthesis. CXP is then called to complete the evaluation of the expression. An instruction is compiled to count the number of ones in X6 and a zero jump instruction over the coding that will be generated by the evaluation of the rest of the logical IF statement is compiled. An attempt is made here to eliminate a BX6 Xi instruction if it was the last one compiled in the evaluation of the expression. The processed portion of the IF statement is changed to blanks, the address of the zero jump instruction is saved as the current jump and a return to the main routine is made at CPQ which is the return for a conditional statement.

Subroutines Called: CXP - Compile Expression
TAB - Normalize Statement
TIQ - Translate Individual Quantities
UNP - Unpack Parenthesis

Temporaries/Flags: CJP - Current Jump
TGH - Statement Tag
TMO - Start of Conditional Statement

Tables Referenced: None

Entry/Exit Register Conditions

Entry: B3 = address of first left parenthesis in string

B4 = address plus one of last right parenthesis

Exit: B5 = 24B to flag a conditional statement next

IFS - PROCESS IF STATEMENT

The IFS routine controls the evaluation of an IF statement. The system allows four types of IF including the I/O checks, the SENSE SWITCH AND SENSE LIGHT checks, the logical type IF, and the normal arithmetic expression with two or three branches. The first seven characters past the first left parenthesis are extracted and a check is made to see if this might be a SENSE type IF. If so, control is passed to the IFH routine. If not a check is made to see if it is some type of I/O check, and if it is, it is evaluated within this routine. If it is not an I/O type IF, the first entry after the last right parenthesis is examined. If it is not a number, a logical type IF is assumed and control is transferred to the IFL routine.

I/O type IF: After extracting the tape number and determining which type of I/O check is being made, the routine to compile tape handling instructions (PMT) is called. After returning from this routine, the call to the proper routine will have been compiled and then the proper tests will have to be generated. For an "IF(UNIT,•••" I/O check, tests are generated to jump to each one of the branches. The upcoming statement number is checked for the rest of the I/O checks and one of the test jumps is eliminated if possible.

Normal IF: When it has been determined that it is a normal IF, the statement has been normalized and the first entry past the last right parenthesis is a number. In order to use the existing arithmetic statement processing, column seven is changed to an equals sign, and the routine called UNP is called to control the processing of all portions of the statement embedded in parenthesis. CXP is then called to complete the evaluation of the expression and the appropriate test jumps are then generated with checking for equal branches and branches the same as the upcoming statement number. Any unnecessary test jumps are thus eliminated. Control is then given to the PSN routine to process the statement number.

<u>Subroutines Called</u>: ASL - Assemble letters
ASN - Assemble numbers
ASV - Assemble variable
CUN - Tag upcoming statement numbers
CXP - Compile expression
ISN - Identify statement number
PMT - Compile tape handling instructions
TAB - Normalize statement
TIQ - Translate quantities
UNP - Unpack parenthesis

<u>Temporaries/Flags</u>: TMI - First statement number
TMJ - Second statement number
TMK - Third statement number

<u>Tables Referenced</u>: None

<u>Entry/Exit Register Conditions</u>

Entry: B4 = First left parenthesis of statement
Exit: None

ISL - IDENTIFY SYMBOLIC TAG

The ISL subroutine is called when a symbolic tag is being processed by PTC (Process Tag and Constant). ISL checks the tag length; it must be less than six alphanumeric characters, then the Argument Name Table (Table I) is scanned and if the variable is found to be there, a normal return is taken. If the variable does not yet appear in the table, a statement tag is generated and both the tag and name are sent to the Variable Name Table. The generated tag is returned in the X6 register.

Subroutines Called: SCM - Scan With Masking
ADF - Advance Tables

Temporaries/Flags: TGH - Statement Tag

Tables Referenced: TBI - Argument Name Table

Entry/Exit Register Conditions:

Entry: X6 - Variable Name

Exit: X6 - Generated Statement Tag

ISN - IDENTIFY STATEMENT NUMBER

The ISN subroutine searches the Statement Number Table (Table K) for a specified statement number and, if not found, enters it in the Statement Number Table. On entering the ISN subroutine, leading zeroes are deleted from the statement number, and the number checked to see if it is composed of more than five digits. If the number contains more than five digits, or contains a non-numeric character, an error exit (Statement Number Error) occurs. If the number is a valid statement number, the Statement Number Table is searched: if the number is found in this table, control is returned to the calling program. If the number is not found in the Statement Number Table, it is entered in this table, and a statement tag (H tag) is generated and stored in the corresponding entry in the Statement Tag Table (Table L). Control is then returned to the calling program.

Subroutines Called: SCT - Scan Tables
ADF - Advance Tables

Temporaries/Flags: TGH - Statement Tag (set)

Tables Referenced: Statement Number Table (Table K)
Statement Tag Table (Table L)

Entry/Exit Register Conditions

Entry: X6 = Statement number

Exit: X6 = corresponding tag from Statement Tag Table

KOT - CONVERT BINARY TAG TO MNEMONIC TAG

The KOT subroutine converts the binary tag associated with the instruction to a mnemonic tag which is printed with a compiled listing. Input to the routine is from CDC (Convert Instruction or Constant Tag to Display Code) which places the binary tag left-justified in the X1 register. Output is the alphanumeric tag in display code. Tags in the range of 200000 to 600000 are converted by examining the upper 5 bits and selecting a letter (L, I, T, C, F, A, V, N or S) for this configuration. The sixth bit, if set, generates a 1 as the second character and if unset, generates a zero. The remaining four digits are merely converted to their display code equivalent. The program tag description given in the appendix shows the exact letter given above for any given numeric tag.

Subroutines Called: None

Temporaries/Flags: MOD - Machine/Ascent Indicator

Tables Referenced: None

Entry/Exit Register Conditions:

X1	binary tag left-justified
X6	mnemonic tag left-justified

LST - PROCESS INPUT/OUTPUT LIST

The calling sequence to the execution time I/O routines is constructed by LST. The statement processor, RIT (READ), WOT (WRITE), PNC (PUNCH), etc., decides from the form of the statement which execution time routine is to be referenced. Also, the file name from the logical unit parameter has been constructed and B3 contains the address of the format statement.

At least three calls are made to the I/O subroutines - 1) initialization 2) intermediate 3) termination. There will be an intermediate entry for each array or data item to be transferred. Naturally, an I/O statement without a list would have only two entries made to the subroutine. The ENCODE/DECODE statements have a different initialization entry for their subroutines but the intermediate and termination entries are the same. The subroutines that will be referenced are:

INPUTC	for coded reading
INPUTB	for binary reading
INPUTS	for DECODE
OUTPTC	for coded writing
OUTPTB	for binary writing
OUTPTS	for ENCODE

The calling sequence for these subroutines, except INPUTS and OUTPTS is:

Initialization:	B1 = 0
	B2 = address of parameter or complemented address of variable tape number
	B3 = address of format statement
Intermediate:	B1 = address of data item or beginning address of array
	B2 = array length or 0
Termination:	B1 = -1

For INPUTS and OUTPTS:

Initialization I	B1 = 0	
	B2 = 0	
	B3 = address of format statement	
	B4 = number of coded characters	
Initialization II	B1 = address of packed data	
	B2 = 0	

All files to be referenced within a program must be declared on the PROGRAM card. Each name is entered into the File Name Table. Wheneven a reference is made to a file, the location of the buffer parameter list may be retrieved from this table if the logical unit number is not a variable. The address is then set in B2 to be sent to the subroutine. In the case that the logical unit number may vary, PMD (Process Tape Medium for Input/Output) will pass the address of the variable in complemented form to the subroutine. The initialization entry to the subroutine is then made with B1, B2, and B3 set accordingly. TSF (Tag Special Function) finds the tag associated with the subroutine to be called and this tag is used in the return jump instruction.

TAB (Normalize Statement) removes unnecessary blanks, and packs the variables and constants into one-word string buffer entries. Replacing the variables with tags is done by TIQ (Translate Individual Quantities). A simple variable address is set in B1 and B2 is zero unless the variable is double or complex in which case B2 = 2.

An implied DO-loop transfer makes an entry into the subroutine for each item in the array. Example: READ (25, 10) (T(I), I=1, 8) causes the subroutine to be referenced eight times. However, if T had been dimensioned then READ (25, 10) T would transfer the whole array with only one entry becuase the array length is retrieved from the Array Parameter Table and set in B2. The implied DO-loop code is generated by HBL (Process Left Parenthesis) and HCL (Process Equal Sign).

An array variable which is subscripted in the list requires CSR (Compile Subscripted Reference) to fetch the address of the word within the array. This address is sent in B1 and B2 will be zero since only one word of the array is being transferred.

If the variable was an argument to the routine, then it would have a location tag. If this tag is not in the Array Tag Table, then it must not have been dimensioned. GAT (Compile Argument Address Pick) gets the address that was passed to the routine and sets it into B1.

A variable with a location tag that was found in the Array Tag Table may

1) be followed by a subscript
2) have had fixed dimensions and the entire array is to be transferred
3) have variable dimensions and the entire array is to be transferred

In the first case, the subscript is handled in the same way as an array that was not passed to the routine as an argument. CSR (Compile Subscripted Reference) compiled instructions to fetch the proper word within the array and this address is set in B1.

An array variable used as an argument which appears in a DIMENSION statement may have the dimensions as constants or variables. A variable dimension also enters the routine as an argument. If the dimensions are constants, then the array length of the variable is read from the Array Tag Table and saved in the Constant Value Table. CIR (Compile Read Instructions) is called to fetch this value as a different argument is passed to the routine. B2 is then set to this array length and the beginning address of the array is found by GAT (Compile Argument Address Pick).

Variable dimensioned arrays have an entry in the Array Tag Table but the array parameters are given location tags instead of constant values. By scanning the Array Tag Table and checking the 2^{16} bit of the corres-poinding entry, it can be determined whether or not the dimensions have location tags. Instructions are compiled to construct the length of the array. If it is a single dimensioned array, then just the address of the one variable is sent in B2. A two dimensioned array must use the product of these two variables as a length. So with three dimensions, another product of the third dimension and the first two necessary for the length. In all cases the beginning address of the array is sent to the subroutine in B1 and the length is set in B2. A double or complex variable always has an array length twice the size available from the Array Parameter Table set in B2.

When the last data item has been processed, then a final entry with B1 = 1 is compiled to the subroutine. CRI (Compile Restore Instruction) is called to restore the addresses of the arguments to index registers if there were any arguments passed to the routine.

<u>Subroutines Called</u>:	ADF - Advance Table
	ASV - Assemble Variable
	CLA - Clear All Registers
	CIR - Compile Read Instructions
	CRI - Compile Restore Instructions
	CSR - Compile Subscripted Reference
	GAT - Compile Argument Address Pick
	HCL - Process Equal Sign
	HBL - Process Left Parenthesis
	PMD - Process Tape Medium
	SCM - Scan Table With Mask
	SCT - Scan Table
	TAB - Normalize Statement
	TIQ - Translate Individual
	TSF - Tag Special Function
<u>Temporaries/Flags</u>:	TMA - Pseudo Statement Number
	TMD - Subroutine Tag
<u>Tables Referenced</u>:	Constant Value (A)
	Constant Tag (B)
	Array Tag (P)
	Array Parameter (Q)

<u>Entry/Exit Register Conditions</u>:

Entry: X4 - Subroutine Name

X5 - Filename

B6 - NZ if ENCODE/DECODE

Exit: None

MAA - PROCESSING MACHINE OR ASCENT RECORDS

When the Run compiler interprets an Ascent or Machine header card, all subsequent records, without an * in column 1 until the next END card, are processed by the MAA subroutine. MAA first concerns itself with the operation field of the current record. Examining this field determines one of four types for this record (reference Figure MAA-1): 1) a constant or Machine register notation, 2) an Ascent mnemonic, 3) an Ascent pseudo-op or 4) a Machine declarative or FORTRAN non-executable.

The first group is processed in the MAA routine while the other three are linked with a series of subroutines. Upon determining a Machine operation or constant, MAA calls the PST (Process Location Tag) subroutine to store and tag the statement label or suppress any leading blanks. Next, a check is made for a left parenthesis indicating a block reservation request. Several checks are made for the correct form for this instruction, if there is a positive decimal or octal number enclosed within parenthesis, followed by an end of statement, and a core overflow will not occur, then MAA allocates and initializes to zero the given number of cells, and a jump to a common return area occurs.(reference Figure MAA-2). Should the type one processing encounter a constant, a jump is taken to ARA (Adjust Address and Write Register) to write the previous word. Then a jump to TAB (Normalize Statement) reorders the string buffer and a jump to CVN (Convert Constant) stores the octal equivalent to the FORTRAN acceptable constant. If this section has been entered from another type processing a check is made on the constants range. In any event, the common return area is entered. (reference Figure MAA-3). Sensing a dollar sign in the instruction field transfers control to the type 2 processing of a NO instruction. Another acceptable form for type 1 is the constant section header card. This card will cause a call to ARA (Adjust Address and Write Register) subroutine, set the instruction word counter, write blanks into the output buffer, write the record, read the next record, check for an end of file which is illegal at this point, then return to the main Run loop.(reference Figure MAA-4).

Type 1 processing of a Machine register notation calls the PAF (Process

TYPE I

NO LETTERS

- HEADER CARD
- CONSTANTS

MACHINE OPCODES

TYPE II

TWO LETTER
ASCENT MNENOMICS

- 2 LETTER OPCODES
- 2 LETTER + DIGIT OPCODES

TYPE III

ASCENTF PSUEDO-OPERATIONS

- BSS
- EQU
- DPC
- BCD
- CON
- BSSZ

TYPE IV

MACHINE
DECLARATIVES

- CON
- ABS
- SUB
- RES
- COM
- HOL
- FORTRAN STATEMENTS

OPERATION CODE FIELD TYPES

Figure MAA-1

Additive Field) subroutine. This routine flags the string buffer with an intermediate language identical to RDA's processing of Ascent instructions (reference Figure RDA-1). If a tag or constant has not been processed, the PTC (Process Tag and Constant) subroutine is called, then FIN (Form Instruction) is called and the processing is continued as in type 2.

The common return section indicates a full word has been processed, calls CDC (Convert to Display Code) for the output listing, writes the record into the buffer area, reads the next record, checks for an illegal end of file, then returns for processing this record. The return calls AFS (Assemble FORTRAN Statement) and if an * is found, the next record is read and checked; if not, MAA is re-entered from the Run main loop. The processing of type 2 Ascent mnemonics also calls the PST (Process Location Tag) subroutine then calls the RDA (Reduce Ascent Format) subroutine to flag the string buffer with an intermediate language (reference Figure RDA-1). A test is made to determine if the Ascent instruction had a constant or tag in the address field, or a literal in the instruction field. If the latter condition exists, control is transferred to type 1 processing described above. If the former condition exists, a call to PTC (Process Tag and Constant) will convert these fields before calling the FIN routine. If neither of the above conditions exist, then a direct transfer to FIN (Form Instruction) routine which interprets the string buffer and generates the 15 or 30 bit instruction. These routines also store the binary word. Then the buffer is written and the next record read, checked for an end of file, and control returns to the Run main loop.

Type 3 processing is merely a table look-up resulting in an unconditional transfer to an open routine which generally returns to the common return area of the type 1 processing (reference MAA-5). These open routines are ACE (Process Ascent EQU), ACH (Process Ascent BCD and DPC), and ACR, ACK (Process Ascent CON). A transfer to this section without a find results in an error exit.

The processing of type 4 instructions checks the relative position of the declarative or FORTRAN statement. These instructions must appear at the beginning of the program. If a FORTRAN statement is detected,

BLOCK RESERVATION PROCESSING

EXAMPLE : BLKI (100)

- CHECK THE FORMAT
- CONVERT THE CONSTANT
- INCREMENT THE RUNNING RELATIVE ADDRESS
- CHECK THE TOTAL FIELD LENGTH
- INITIALIZE THE STORAGE AREA
- CONVERT TO DISPLAY CODE
- OUTPUT TO BUFFER
- INPUT THE NEXT RECORD AND RETURN FOR PROCESSING

Figure MAA-2

CONSTANT SECTION PROCESSING

EXAMPLE : CONI 15.64E03

- NORMALIZE STATEMENT
- CONVERT THE CONSTANT
- STORE CONSTANT INTO OUTPUT BUFFER
- INDICATE FULL WORD INSTRUCTION
- CONVERT TO DISPLAY CODE
- TRANSFER TO OUTPUT BUFFER
- READ THE NEXT RECORD
- IF NOT AN END OF FILE, RETURN FOR PROCESSING

Figure MAA-3

CONSTANT HEADER CARD PROCESSING

FORMAT - COLUMN 7-8

- CHECK FOR THE FIRST HEADER CARD
- SET CONSTANT SECTION FLAG
- WRITE BLANKS INTO THE OUTPUT BUFFER
- READ THE NEXT RECORD
- IF NOT AN END OF FILE, RETURN FOR PROCESSING

Figure MAA-4

Additive Field) subroutine. This routine flags the string buffer with an intermediate language identical to RDA's processing of Ascent instructions (reference Figure RDA-1). If a tag or constant has not been processed, the PTC (Process Tag and Constant) subroutine is called, then FIN (Form Instruction) is called and the processing is continued as in type 2 (reference MAA-6). The common return section indicates a full word has been processed, calls CDC (Convert to Display Code) for the output listing, writes the record into the buffer area, reads the next record, checks for an illegal end of file, then returns for processing this record. The return calls AFS (Assemble FORTRAN Statement) and if an * is found, the next record is read and checked; if not, MAA is re-entered from the Run main loop.

The processing of type 2 Ascent mnemonics also calls the PST (Process Location Tag) subroutine then calls the RDA (Reduce Ascent Format) subroutine to flag the string buffer with an intermediate language (reference Figure RDA-1). A test is made to determine if the Ascent instruction has a constant or tag in the address field, or a literal in the instruction field. If the latter condition exists, control is transferred to type 1 constant processing described above. If the former condition exists, a call to PTC (Process Tag and Constant) will convert these fields before calling the FIN routine. If neither of the above conditions exist, then a direct transfer to FIN (Form Instruction) routine which interprets the string buffer and generates either a 15-bit or 30-bit instruction. The routines called by FIN store the binary word. Then the buffer is written and the next record is read, checked for an end of file which is illegal at this point and then transferred to the main loop of Run.

Type 3 processing is merely a table look-up resulting in an unconditional transfer to an open routine which generally returns to the common return area of the type 1 processing (reference MAA-5). These open routines are ACE (Process Ascent EQU), ACH (Process Ascent BCD and DPC), ACR (Process BSS and BSSZ, ACK (Process Ascent CON). A transfer to this section without a find results in an error exit.

The processing of type 4 instructions checks the relative position within the program of the declarative or FORTRAN statement. These instructions must appear at the beginning of the program. If a FORTRAN statement is detected,

PSUEDO OPERATION PROCESSING

EXAMPLE: TAGA BSS 5
BSSZ 7
CON 6.3

1. MOVE OPCODE TO COLUMN 7 OF THE STRING
2. END STRING AT FIRST BLANK
3. IF BSS/BSSZ: STORE A RIGHT PARENTHESIS IN THE STRING
4. GO TO TYPE I PROCESSING

EXAMPLE: TAGE EQU 777

1. PROCESS LOCATION TAG
2. INSERT EQUAL SIGN
3. GO TO TYPE IV (ABS) PROCESSING

EXAMPLE: TAGD BCD $n A_0 A_1 \ldots A_n$ $n \leq 10$
TAGF DPC $*A_0 \ldots A_9*$

1. PROCESS LOCATION TAG
2. ACCUMULATE TEN CHARACTERS
3. NORMAL EXIT IF IN CONSTANT SECTION
4. ERROR EXIT IF NOT IN CONSTANT SECTION
5. GO TO TYPE I PROCESSING

Figure MAA-5

MACHINE INSTRUCTION PROCESSING

EXAMPLE: $T = (C + TAG)$

- PROCESS ADDITIVE FIELD
- PROCESS TAG OR CONSTANT, IF ANY
- FORM INSTRUCTION
- WRITE CODED RECORD
- READ NEXT RECORD
- IF NOT AN END OF FILE, RETURN FOR PROCESSING

Figure MAA-6

PROGRAM ORDER

- HEADER CARD
- FORTRAN CARDS, IF ANY
- DECLARATION CARDS, IF MACHINE HEADER CARD
- INSTRUCTION CARDS
- CONSTANT HEADER CARD
- CONSTANT CARDS
- END CARD

ASCENT MNEMONIC PROCESSING

EXAMPLE: SX1 = X2+ TAG

1. PROCESS TAG IN THE LOCATION FIELD
2. REDUCE THE ASCENT CODE
3. FORM THE 30-BIT or 15-BIT INSTRUCTION
4. OUTPUT DISPLAY CODE TO BUFFER
5. RETURN FOR THE NEXT RECORD

then a transfer back to the RUN compiler is initiated to complete the processing. If a declarative is encountered, then a table look-up and jumps similar to type 3 processing occurs. These routines return to an unconditional transfer to the Run compiler. The routines involved in the six declarative processing are CON, COM, ABS, HOL, RES, and SUB.

The common error exit for the MAA subroutine sets the buffer to a string of * and sets several flags. The next record is read and checked for an END card or a second END card which is processed by ENO; a single END card is processed by MND, and all other cards return to the Run compiler and AFS then returns to MAA.

Subroutines Called: Reference Figure MAA-7

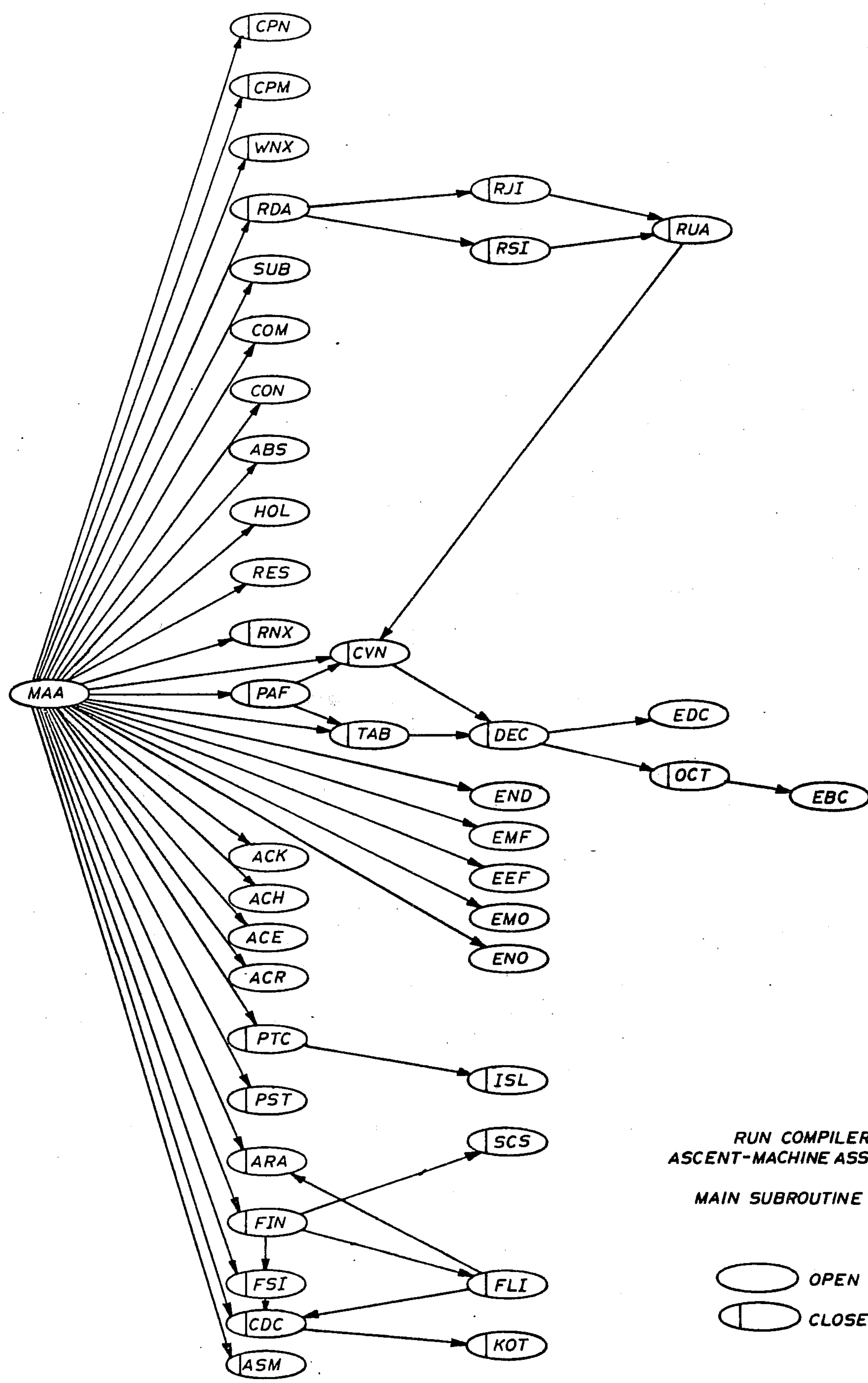
Temporaries/Flags:

- ADM - Running Relative Address
- ICE - Dollar Sign Pointer (set)
- INJ - Continue Indicator (set)
- ICT - Intraword Instruction Counter (set)
- FLH - Subprogram Error Flag (set)
- FLF - Job Error Flag (set)
- PNM - Program/Subprogram Name
- IWC - Number of Instruction Words (set)
- ZAA - Relative Start of Current Program or Subroutine

Tables Referenced:

- MTB - Table of Tag-defining Operation Codes
- MTA - Table of Ascent Pseudo Operations
- TBJ - Argument Tag Table

Entry/Exit Register Conditions: N/A



RUN COMPILER
ASCENT-MACHINE ASSEMBLY

MAIN SUBROUTINE CALLS

MCA - MAKE RELATIVE COMMON ASSIGNMENTS FOR FORTRAN IV

This routine is called during the processing of the END card to make the common assignments. Upon entry, CTY (Common Block Type Indicator) is set to zero if blank common is to be processed, and set to one if labeled common is to be processed. After blank common is processed, the routine processes numbered common and then exits. The PCA routine (Process Common Assignments) calls this routine to make numbered and blank common assignments while the PUA (Process Unique Assignments) routine calls this routine to process labeled common. In the processing of common, the Common Name Table is searched until a non-eliminated block name is found. If the block has not yet been declared in a previous program or subroutine, the current common block relative address (in the case of blank common) or the base address of the variables (in the case of labeled common) is entered, along with the block name or number, into the Common Block Name Table. If the block was declared, its starting address is extracted from the Common Block Name Table. Each succeeding variable tag is examined and assigned memory location(s) until another block name is found or until the end of the table is detected. Each variable whose tag is not an address tag is entered into the J Table along with its relative address. If the variable has an address tag, it is not entered into Table J. The Primary Name Table is then searched to see if any variables were declared equivalent to this one. If there are some, these entries in the Primary Name Table and Secondary Name Table are cleared. If the secondary tag is in the Variable Tag Table, the tag and address are entered into the J Table. Whether or not the tag is in the Variable Name Table, the block length is extended if the equivalences make it necessary.

When all variables belonging to the block are processed, the limit address of the block is entered into the Common Block Name Table unless the block was previously defined. If it was previously defined, the limit address is compared to the previous limit address to make sure this address is not greater. If it is a diagnostic is given. The routine then exits.

<u>Subroutines Called</u>: ADF - Advance Table
SCT - Scan Table

<u>Temporaries/Flags</u>:
BAV - Base Address of Variables
CBA - Current Common Block Relative Address
CTY - Common Block Type Indicator
TBJ -
TBO - Table Parameters
TBP -
TBV -
TMC - Free Temporary
TMP - Current Extended Common Block Length
TMQ - Current Common Block Name
TMR - Base Address for Equivalence Group

<u>Tables Referenced</u>:

Argument Name	(J)
Common Name	(O)
Array Tag	(P)
Common Block Name and Address	(V)

<u>Entry/Exit Register Conditions:</u> None

MTU - MOVE TABLES UP

This routine is responsible for relocating the 26 temporary tables and adjusting the TBn parameters to reflect this change. It is called prior to processing of library subroutines and usually overlays the input buffer to make more room available for the loading of library subroutines.

Subroutines Called: None

Temporaries/Flags: None

Tables Referenced: Argument Name (I)
(Referenced because it is the first table)
Program File Name (W)
(Referenced because it is the last table)

Entry/Exit Register Conditions:

Entry: B1 - how much the tables should be moved
Exit: None

PAF - PROCESS ADDITIVE FIELD

When the MAA (Process Ascent and Machine Records) subroutine encounters a Machine instruction, PAF is called to examine this instruction. This routine collects the terms, stores the register values in constants III, JJJ, and KKK, then converts the register notation to the notation used by RDA for the Ascent instructions (reference Figure RDA-1). The string buffer is flagged identically to the way RDA flags it so that FIN (Form Instruction) can be called to further process these Machine instructions. PAF uses the RCD Table look-up to convert the letters to the conventional A, B, and X registers. Error exits are taken if too many terms or a blank within the address field occurred.

Subroutines Called: TAB - Normalize Statements
CVN - Convert Number
ADF - Advance Tables
ASN - Assemble Number
ASV - Assemble Variable

Temporaries/Flags: III - i Portion of Machine Word
TGK - Constant Tag
JJJ - j Portion of Machine Word
KKK - k Portion of Machine Word

Tables Referenced: RCD - Table of Operational-Register Codes
TBA - Constant Value Table
TBB - Constant Tag

Entry/Exit Register Conditions:

B6 - non-zero if a tag or constant has not been processed

PAT - PROCESS ADDITIVE ADDRESS

When the MND (Process Machine or Ascent End) subroutine encounters a tag plus a constant in the address field, the PAT Subroutine is called. The Variable Tag (V-tag) is in the upper 18 bits of the Argument Tag Table (Table J) and the Constant tag is in the lower 18 bits. These tags are merged together so that upon exit the X6 register contains the address of the location of the V-tag plus the constant.

Subroutines Called: SCM - Scan With Masking

Temporaries/Flags: BAK - Base Address for Constants

Tables Referenced: TBJ - Argument Tag

Entry/Exit Register Conditions:

X6 - Combined tag

PBR - PROCESS BUILT-IN-FUNCTION REFERENCE

PBR is entered when a built-in function has been detected. It merely routes the processing to one of the other routines depending upon the number of arguments the built-in function has and whether or not it is a logical function.

Subroutines Called: CMA - Compile Multiple Argument Function
COA - Compile One Argument Function
CTA - Compile Two Argument Functions
KMA - Compile Multiple Double Argument Functions
KSF - Compile Special Logical Function

Temporaries/Flags: FIV - Start of the last of library functions

Tables Referenced: None

Entry/Exit Register Conditions:

Entry:. B4 - address of string entry of function

Exit: None

PCA - PROCESS COMMON VARIABLE ASSIGNMENTS

This routine is called during the processing of an END card after the instructions have been packed and the constants moved into the program area. If any variables were declared common, the equivalence tables are searched to process variables which were declared equivalent to a variable in common. If the mode is FORTRAN IV, the secondary name is removed from the common table and replaced by the primary name. If the mode is FORTRAN II the secondary name is replaced by the primary name and all equal primary names in the Primary Name Table are changed to the secondary name.

After the equivalence tables have been processed, the relative common assignments are made by calling either the FORTRAN II routine (MKA) or the FORTRAN IV routine (MCA). The FORTRAN IV routine will first make blank common and then numbered common block assignments. When the memory assignment has been made the variable tags and starting address will be entered into the J Table by MKA or MCA. This routine will then set bit 16 of the address to flag them relocatable to the start of common.

<u>Subroutines Called</u>:
- ADF - Advance Tables
- MCA - Make FORTRAN IV Relative Assignments
- CKA - Make FORTRAN II Relative Assignments
- SCT - Scan Table

<u>Temporaries/Flags</u>:
- CBA - Current Common Block Relative Address
- CSA - Common Starting Address
- FLC - Program Common Field Length
- IPS - Program/Subprogram Indicator
- LBA - Latest Buffer Address
- MOD - Subprogram Mode
- MOE - Program Mode
- TBJ - } Table Parameters
- TBO -
- TBV -

TBX -
TBY - Table Parameters
TBZ -

<u>Tables Referenced</u>:

Argument Name Table	(J)
Common Name Table	(O)
Common Block Name and Address	(V)
Equivalence Secondary Name	(X)
Equivalence Primary Name	(Y)
Equivalence Bias	(Z)

<u>Entry/Exit Register Conditions</u>: None

PCT - PROCESS SPECIAL ARRAY TAGS

PCT is entered during the processing of the END statement to process any temporary array tags in the Temporary Tag Table. For each array tag in this table, Table J is searched to find the starting address of the array. The array address increment is added to the starting address of the array, incorporated with the corresponding permanent tag in Table C and then entered into the J Table.

This type of entry is made when calculating array addresses such as A(I+10)=.

Subroutines Called: ADF - Advance Tables
SCM - Scan With Masking

Temporaries/Flags: TBC - Table Parameters
TBD -

Tables Referenced: Temporary Tag (C)
Permanent Tag (D)

Entry/Exit Register Conditions: None

PFR - PROCESS STATEMENT FUNCTION REFERENCE

PFR compiles instructions to transfer arguments to an arithmetic statement function and a return jump to the function. Two methods are employed in the passing of arguments. If the function does not reference any subroutines, the actual argument itself rather than the address is transferred to the space reserved for it at the start of the function. If the function does reference subroutines, the number of arguments is saved, the addresses of the first N arguments are transferred via index registers and the value of the remaining arguments are transferred to the reserved space. N will equal five minus the number of arguments and the first index used will be the number of arguments plus one. For example, if the function had four arguments, the address of the first two would be passed in B5 and B6 while the values of the last two would be transferred to the corresponding cells at the beginning of the function.

Subroutines Called:

- ADF - Advance Tables
- CIR - Compile Re
- CLA - Clear Index and Address and Input Tags
- CXP - Compile Expression
- GAT - Compile Argument Address Pick
- SAD - Sense and Process Single Array Address
- SCT - Scan Table

Temporaries/Flags:

- ARG - Argument Count
- IGX - Index Assignment
- TBA - Constant Name Table Parameters
- TBB - Constant Tag Table Parameters
- TGK - Constant Tag Table Parameters
- TGL - Argument Count for Call
- TMM - Name Tag For Call
- TMN - Argument Tag for Call
- VTY - X6 Register Associate

Tables Referenced:

- Constant Name (A)
- Constant Tag (B)

PGP - PROCESS SUBPROGRAM PARAMETERS

If the compilation mode is incomplete, the first part of the program is changed to the following format, with the needed information extracted from the table entries for each subroutine. The format of the first N+2 words where N is the number of parameters is as follows:

Name 42		total length 18	
buffer start 18	addr. of first inst. 18	base addr. of temps. 18	no. of arguments 6

base addr. of constant 18	base addr. of indirect 18	addr. of local var. 18	mode 6

If the compilation mode is not incomplete, the name of the routine and the total length are entered into the first program location, the instruction word count entered into the Subroutine Parameters Table and also the total length. The start of the new short file is set and the unused space indicator is updated if need be.

Subroutines Called: SCT - Scan Table

Temporaries/Flags:

BAI - Base Address for Indirects
BAK - Base Address for Constants
BAT - Base Address for Temporaries
FST - Long File Start
ICB - Argument Count
ICM - Incomplete Compile Mode Indicator
ICO - Base Address for Variables
INT - First Instruction Address
INV - Unused Compiler Space
IWC - Number of Instruction Words
MOD - Subprogram Mode
PNM - Program/Subprogram Name
TBI -

TBI -

TBJ - Table Parameters

TBS -

ZAA - Relative Start of Current Program or Subroutine

ZAB - Short File Start

Tables Referenced:

Argument Name	(I)
Argument Tag	(J)
Subroutine Name	(S)
Subroutine Parameters	(U)

PIG - PRINT INSTRUCTION GROUP

This routine is called during the processing of the statement number and during the processing of the END card. It is responsible for listing all object code compiled for the last statement along with the current running address. The listing will correspond exactly to the actual program that is loaded into core for execution except that the K portions of the 30 bit instructions will necessarily contain tags rather than absolute addresses as the assignment of variables and temporary cells is not made until the subprogram is completely compiled.

If no object listing is required, this routine merely calls PJG to process the group ending address and then exits.

Otherwise, the instructions are examined one at a time. The current running address is updated and listed when it is detected that the examined instruction will not fit into the word that is presently being processed. The method of forcing instructions to start a new word is exactly the same that is used when the instructions are packed during the processing of the END statement. The 15 bit instructions are converted to display code and listed. If the K portion of the 30 bit instruction is a tag, the first two numbers of the tag, which describe which type of tag it is, are replaced by a letter and then the instruction is converted to display code and listed. If there is an address tag associated with the instruction, it is also listed. After all instructions have been examined, the execution address of the next instruction group is saved, a line of blanks is written and the routine exits.

<u>Subroutines Called</u>:

- KOT - Convert Binary Tag to Mnemonic Tag
- PJG - Process Group Ending Address
- WNX - Write Coded Record

<u>Temporaries/Flags</u>:

- ADM - Running Relative Address
- ICT - Intraword Instruction Counter
- IGE - Instruction Group End
- IGS - Instruction Group End
- INU - Unused Compiler Space

<u>Tables Referenced</u>: TBC - Temporary Tag Table
TBD - Permanent Tag Table
TBI - Argument Name Table

<u>Entry/Exit Register Conditions</u>: None

PKI - PACK INSTRUCTIONS

In processing the END statement, the instructions are packed. PKI is called after the temporary tags have been replaced with permanent tags, but before the variable tags are replaced by addresses. The 15 and 30 bit instructions are shifted into consecutive words with the unused portions filled with pass (46000) instructions. An instruction with an operation code of 01, 02, or 04 does not allow any more instructions packed into the word with the exception that an 07 instruction following an 01 may occupy the lower bits of the word. All tagged instructions (location tags) are saved along with their corresponding addresses in the J Table. These addresses are relocatable to the beginning of the program or subprogram if, in this case, the subroutines are being compiled separately. Bit 17 is set to indicate that the address is relocatable from the beginning of the subroutine.

When the end of the instructions is found, the number of words of instructions is saved.

Subroutines Called: ADF - Advance Tables

Temporaries/Flags:
- ICM - Incomplete Compile Mode Indicator
- IGE - Instruction Group End
- IPS - Program-Subprogram Indicator
- INC - Number of Instruction Words
- ZAA - Relative Start of Current Program or Subprogram
- ZAB - Short File Start

Tables Referenced: Argument Tag Table (Table J)
(not used as such at this time)

Entry/Exit Register Conditions: None Used

PLR - POSITION LIBRARY SUBROUTINES AND EQUATE TAGS WITH ADDRESSES

This routine is responsible for reading in binary decks that appear in the input file, making a call to CLL (PP routine) to bring in any subroutines that have not yet been defined, and then to relocate the flagged addresses within these routines to absolute memory addresses.

The routine is called after it has been detected that all source input has been compiled. It first calls MTU to move the 26 temporary tables up over the string and over the input file buffer if no binary decks appear in the file in order to make more room to load in library subroutines. BRX is called to read in the binary routines appearing in the input file if any. CLL is called to read in the routines that have not yet been defined, and the compiler enters RECALL until CLL has terminated. Each routine is then checked to make sure that it was not called with more parameters than the routine just read in was assembled to handle. Each subroutine is examined one word at a time and all addresses that were flagged as program relocatable are relocated as are all locations that were flagged common relocatable. The Subroutine Tag Table is then searched and all tags along with the address of their entry/exit line are entered into the J Table. Each argument is given a tag and address in the J Table also.

Subroutines Called:	ADF	- Advance Table
	BRX	- Read Binary Subroutines
	CKL	- Check Missing Subroutines
	MTU	- Name Tables Up
	SCT	- Scan Table
	WNX	- Write Coded Record
Temporaries/Flags:	CAS	- Word of Blank Display Codes
	CSA	- Common Starting Address
	ICM	- Incomplete Compile Mode Indicator
	TBI	-
	TBS	- Table Parameters
	TBT	-
	TBU	-

ZAA - Relative Start of Current Program or Subprogram

ZAB - Short File Start

Tables Referenced:

Argument Name	(I)
Subroutine Name	(S)
Subroutine Tag	(T)
Subroutine Parameters	(U)

Entry/Exit Register Conditions: None

PPG - PROCESS NAME AND ARGUMENTS

Upon encountering a header card, PROGRAM, SEGMENT, SUBROUTINE, FUNCTION, or BLOCKDATA, PPG is called to compile initialization instructions. Whether or not the routine being compiled is Fortran, the arguments are counted, the name is saved, and the relative start of the routine is saved. Inconsistancies in the calling of a subprogram with more arguments than specified or declaring a function a different type than a previous call are checked in this routine.

In the case of a Fortran program card entry, the name of the program is saved. Each I/O file declared has a word reserved for it beginning at RA+2. The first two words, RA and RA+1, are system communication words and are given location tags. For every file designated an I/O buffer is reserved except for equivalenced files. The file appearing on right side of the equals for equivalencing must have already been defined because the file being equated to it must share the buffer. Also a buffer may be given an individual length which would appear on the right side of an equals sign. Each file is given a beginning address which will point to the parameter list of its own buffer or its equivalenced file buffer. This beginning address along with the file name is entered into the file name table (W). A buffer length of 2010_8 is assigned to each file if no length was specified on the "RUN" card or the file was not equated to a number. Either of these specified lengths must be greater than 1001_8 or that amount of space is saved anyway. Eight buffer parameters are saved for each buffer. These are used by CIO[1] (circular input/output). The first of these parameters contains the name of the file (in left adjusted display code) onto which the transfer of data is to be made. This name will always correspond to the file on which reading or writing is to be done for this execution. The names may be changed on the program call card, used to call a compiled program for another execution, to transfer the data to a file different than the one named in the compilation.

A word is saved in the constant value table for use during initialization. When a record has been compiled, the END processor fills this word with the field length, the beginning address of blank or numbered common (or the beginning address of the buffers is no common has been defined) and the

1. Chippewa Operating System, Internal Reference, E012, November 1965.

local length of the program. Instructions are generated to initialize,

B1 - local length

B2 - beginning address of common or buffers

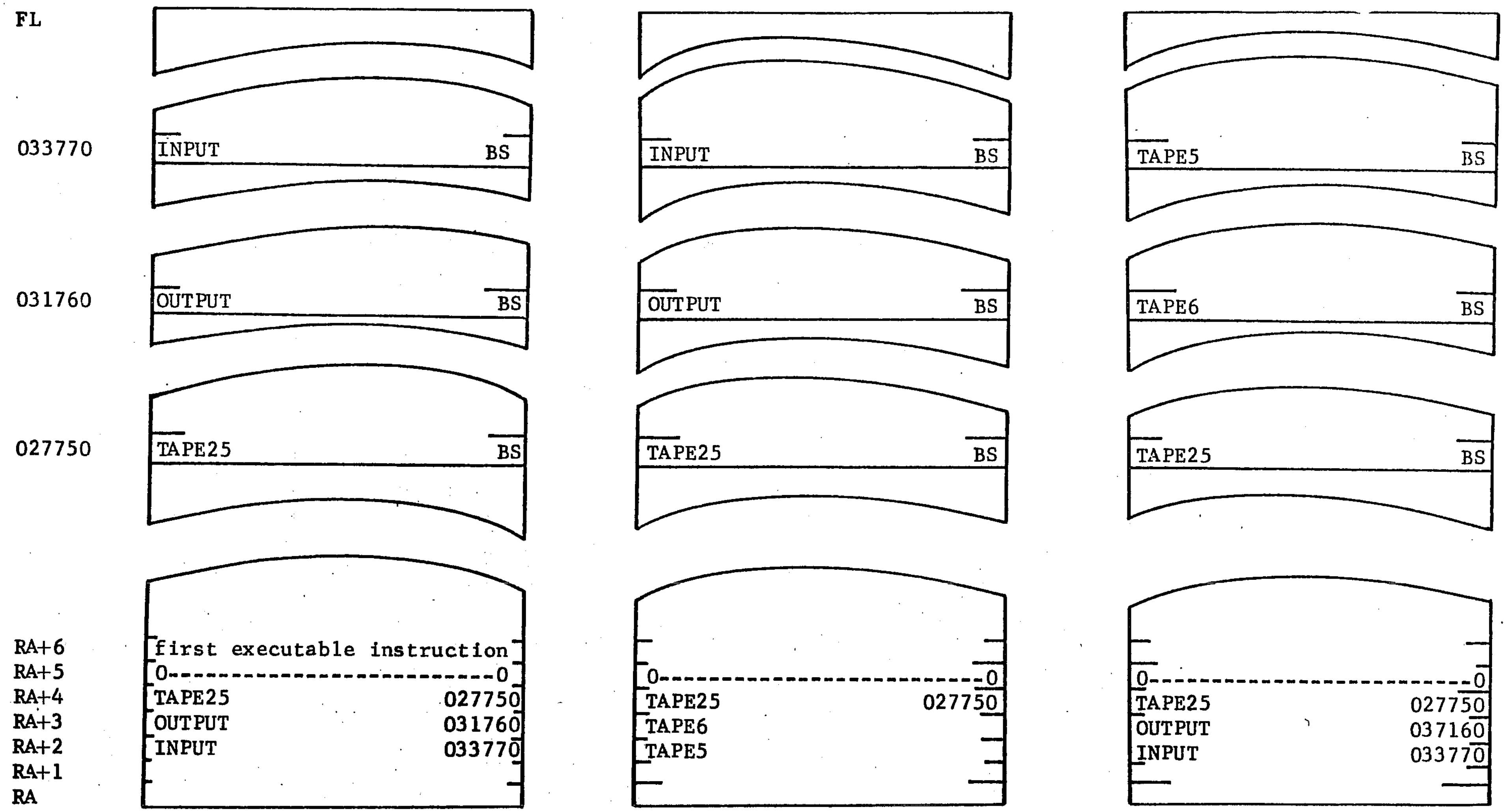
B3 = compiled field length

X2 - requested field length

The field length requested for this execution of the program may not be less than the compiled field length or an error exit is taken. The area reserved for common and the buffers is cleared to zero and the unused program space (area between the local length and common) is set to indefinites.

When a program is ready for execution, the names of the files requested appear in RA+2 through RA+n+2. The names may change from one execution to another so the originally compiled file name along with the beginning address of its buffer is saved in a tagged location. The name from RA+2 is transferred to the first word of the buffer parameters and the compiled file name replaces it at RA+2. Whenever an I/O request is made on an original file, the information will be transferred to the corresponding file named on the program call card. For example: If a program was compiled with the data entering it via INPUT, the program card would look like PROGRAM BIG (INPUT). All read requests would be compiled to take the data from the input file. If the compiled program is called for execution again and the data is to be read from an input tape, the call could be BIG(TAPE5). The name, TAPE5, would be set in RA+2, but the program initialization instructions would transfer the name to the buffer parameter list and set the name INPUT and the beginning address pointing to TAPE5 in RA+2. Whenever CIO is called to make a transfer, the file name in the parameter list identifies the file. The remaining I/O parameters are initialized. The line limit which is the seventh argument on the RUN card is transferred to the eighth word of the parameter list. This limit applies to the number of lines of listable output. and is set to 20000_8 if no special allotment is made. Instructions to set up the buffer parameters in this way are repeated for each file designated.

The SEGMENT card processing saves two words for system communication before the reserved words for the arguments. Each file name along with a beginning address for the buffer is entered into the file name table (W). Changing the file names on the SEGMENT card will have the same effect as calling a



Program - compiled and executed with header card:
PROGRAM A (INPUT,OUTPUT,TAPE25)

Program loaded with program call card:
A (TAPE5,TAPE6)

Program executed after being loaded with program call card:
A (TAPE5,TAPE6)

FILE BUFFER ASSIGNMENTS

program with different file names as was previously described. No buffer space is relinquished from segment to segment. All of the files used by the program and the segments must be declared on the PROGRAM card. An entry/exit word is reserved with a 200001 tag after all the arguments have been processed. Instructions are generated to set the index registers B1-B3 to the same values as the PROGRAM initialization. No memory is cleared or set to indefinites and the I/O buffers are not initialized.

Two system communication words are reserved when a Fortran SUBROUTINE card is encountered. One word for each argument is also saved and each argument is given a location tag (A). An argument list error is generated whenever a variable is used more than once for an argument. The entry/exit word is given a 200001 tag (first location tag). An entry is made into the subroutine name table (S), subroutine tag table (T) and subroutine parameter table (U) if the name has not already been entered. The relative start of the subroutine along with the number of arguments are set in the subroutine parameter entry. If the name appears in the subroutine name table, that is the subroutine that has previously been called, then the number of arguments used by the call must be equal to or less than the number of arguments being compiled or an argument count diagnostic results. Instructions are generated to pack the addresses of the arguments passed to the subroutine in index registers into ten temporary tagged words. B1-B3 are set into the first word and B4-B6 into the second.

The only difference between processing a FUNCTION card and a SUBROUTINE card is that the mode of the function must be checked. If the compiled type is different from the called type, a function type error results. The function name is entered into the variable name table and it is given a V-type tag. This tag along with the mode is entered into the variable tag table. An entry of the same type is made into the subroutine name table except a L-tag is inserted in the subroutine tag table. Since there is no difference in compiling a subroutine or a function, the RETURN statement processor checks the name of the routine for an entry in the variable name table. If this entry is formed, then the subprogram must be a function and the answer will be set into X6.

A BLOCKDATA statement causes three system communication words to be reserved and tagged. The name BLKDAT is entered into the subroutine name table and given an L-type tag. No other processing of this statement is done.

The arguments defined on an ascent or machine subprogram header card are entered into the variable name table but no word is reserved for it. The name is entered into the subroutine name table and the number of arguments are checked. No special initialization instructions are generated. Table entries are made so that subroutine linkage between the Fortran program and the coded routine can be made.

SUBROUTINES CALLED:	ADF	Advance Tables
	ALX	Get Register Assignment
	KON	Convert Octal Argument
	SCM	Scan Tables with Mask
	SCT	Scan Table
	TAB	Normalize Statement
	TRV	Translate Variable
TEMPORARIES/FLAGS	ARG	Argument Count
	CAS	Space Codes
	FTY	Function Type
	ICB	Argument Count
	ICK	Block-Data indicator
	ICY	Line Limit
	INQ	Name for Dayfile
	INT	First Instruction Address
	INV	Segment Indicator
	INW	Chain-mode Indicator
	IPS	Program Indicator
	JPS	Current Subprogram
	LBA	Latest Indicator Buffer Address
	PNM	Program/Subprogram name
	STG	Compile mode Indicator
	TJP	Subprogram Type
TABLES REFERENCES:	Constant Name (A)	
	Constant Tag (B)	
	Variable Name (M)	
	Variable Tag (N)	
	Subroutine Name (S)	
	Subroutine Tag (T)	

Subroutine Parameter (U)

File name (W)

PRR - PROCESS FUNCTION/SUBPROGRAM REFERENCE

When a FUNCTION or SUBROUTINE is called, PRR handles the passing of the arguments between the calling program and the subprogram. All arithmetic expressions have been stripped of their outermost parenthesis so that a somewhat simplified expression is evaluated in PRR. Each argument except constants have been replaced with appropriate tags. Further processing is required for a subscripted variable or an arithmetic expression.

Any argument that is not followed by either a comma or a right parenthesis must be a subscripted variable or an arithmetic expression. SAD (Sense and Process Single Array Address) makes the decision as to which it is. An array will have as its second character a left parenthesis and also a comma-right parenthesis or two right parentheses sequence following it. CSR (Compile Array Address) gets the address of the word within the array and returns it to SAD. The variable and its subscript are replaced with a new tag in the string buffer. If SAD did not locate a subscripted variable, then a zero in B6 is returned to PRR. CXP (Compile Expression) will evaluate the expression with the result in X6. Upon return to PRR, the result is stored in a temporary tagged location.

If the argument in the string has a mode indicator of 3 in the lower six bits, then the argument is a constant whose value is less than $2^{16}-1$. This value is entered into the Constant Value Table (A) and is given a constant tag. A simple variable argument remains in the string buffer with no operations being performed on it.

All of the arguments in the call have now been processed, so the next step is to generate instructions to pass them to the subprogram. Any argument in the call that was passed to the routine as an argument has a location tag. Therefore, the address in the location tag must be used as the address of the argument. In this case, GAT (Compile Argument Address Pick) retrieves the address from the location tag and returns it to PRR. The addresses of the first six arguments are set into B1-B6. When the index registers are exhausted, the addresses of the remaining arguments are stored in the reserved word of the subprogram via external tags. Instructions are generated to set the tag in X6 or X7 and then it is stored by an external tag (400000). This tag will be linked with the word reserved in the subprogram for the argument.

When the arguments have been set in either index registers or external tags, then a return jump will pass control to the subprogram. A call to DUMP/PDUMP also causes the number of arguments to be set in B7 and the program total field length to be sent in X0. If the subroutine being called was used as an argument to this routine, then it will not be entered by a return jump. Into the entry/exit line of the subprogram is stored a jump back to the calling routine and the subprogram is entered by a jump to the word after the entry/exit line.

A subprogram which is being called but was not used as an argument is entered with a return jump. The return jump instruction will be forced upper and the lower 18 bits will contain the number of arguments in the call and a location tag pointing to the name of the calling routine.

Example: 0100 S00600
0715 L00002

where	S00600	is the location of the entry/exit word of the subroutine
	15	is the number of arguments
	L00002	is the location of the name of the subroutine

No more than 60 parameters may be passed to a subroutine. If the number of arguments in this call to the subprogram is greater than the number of any previous call, then the new number is saved in the argument count byte of the Subroutine Parameter Table.

CRI (Compile Restore Instructions) is called if the subprogram being referenced had been previously used as an argument. The location to which control is returned by this called subprogram is given a location tag. This tag will be the same one that was stored in the entry/exit line of the called subprogram.

PRR is called by CLL (Process Call Statement) and CRF (Compile Function

Reference).

<u>Subroutines Called</u>: ADF - Advance Table
ALX - Get Long Register Assignment
CLA - Clear Tables I and J
CRI - Compile Restore Instruction
CXP - Compile Expression
GAT - Compile Argument Address Pick
SAD - Process Single Array Address
SCT - Scan Table

<u>Temporaries/Flags</u>: ARF - Argument Reference Count
FLT - Program Total Field Length
FSR - Function Statement Reference Count
IGX - Current Index Register
INF - DUMP/PDUMP Indicator
SRI - Subroutine Reference Count
TML - Argument Count
TMM - Subprogram Name
TMN - Subprogram Tag

<u>Tables Referenced</u>: Constant Value (A)
Constant Tag (B)
Subroutine Name (S)
Subroutine Parameter (U)

PSC - POSITION CONSTANTS

PSC is called to position the constants into the program after the instructions have been packed. It first sets the base address for the constants (BAK) to the short file start (ZAA) + the number of instruction words (IWC). It then sets the base address for the temporaries (BAT) to the constants base address plus the number of constants. If there is room for the constants, the base address for indirects is set to the base address of temporaries plus the number of temporaries and the base address for variables (BAV, ICO) is set to t he base address for indirects plus the number of indirects.

The constants are then transferred to the program area and the program is set up as follows:

PROGRAM (PACKED INSTRUCTIONS)	CONSTANT SECTION	INDIRECT SECTION	TEMPORARY SECTION	VARIABLE SECTION . . .
	BAK	BAI	BAT	BAV

Subroutines Called: None

Temporaries/Flags:

BAK - Base Address for Constants
BAI - Base Address for Indirects
BAT - Base Address for Temporaries
BAV - Base Address for Variables
FST - File Start
ICO - Baee Address for Variables
IWC - Instruction Word Count
TGI - Indirect Tag
TGK - Constant Tag
TGT - Temporary Tag
ZAA - Short File Start

PST - PROCESS LOCATION TAG

The PST subroutine processes the location tag for all Ascent or Machine records. The Location Tag is assembled by the ASV (Assemble Variable) subroutine and if the tag is non-alphabetic, the only acceptable value is a plus sign. The PST Subroutine will also process the blank location field if parcel 3 of the current word is not full. The running address is set to blanks. When a plus sign has been detected, the running address is stored and incremented by one. The check for an alphabetic tag involves a search in the Argument Name Table (Table I) and a find in this table causes a search of the Argument Tag Table (Table J) to see if the variable has been doubly defined. If the variable was not in the Argument Name Table, it is stored there and a Statement Tag (H-tag) is generated and stored in the corresponding tag table. In generating the entry for the tag table, a check is made for common relocation and a bit is set if necessary. Once the location tag has been processed, the PST subroutine moves the machine operation field if the string buffer had blanks beginning in column 7 and tags as the end of the string at the first blank, thus the machine instructions must not have blanks in the address field.

<u>Subroutines Called</u>: ASV - Assemble Variable
ARA - Adjust Running Address and Write Register
SCM - Scan Tables With Masking
ADF - Advance Tables

<u>Temporaries/Flags</u>: ICT - Intraword Counter
ADM - Current Running Address
IPS - Program Mode
STG - Compile Mode
TGH - Statement Tag (set)

<u>Tables Referenced</u>: TBI - Argument Name
TBJ - Argument Tag

<u>Entry/Exit Register Conditions</u>: n/a

PTC - PROCESS TAG AND CONSTANT

The PTC subroutine processes tags and constants in the Ascent or Machine coded routines. Upon entry into the routine from MAA, X5 contains the tag and/or X4 contains the constant. If a tag does not exists and the constant is in the range of $-2^{16}-1$ to $2^{16}-1$, the number is left-justified and placed in the X5 register. If the constant is greater than $2^{16}-1$ and less than $2^{17}-1$, a Statement Tag (H-tag) is generated and the constant and tag are stored in the Argument Tag Table (Table J). Then the tag is returned in the X5 register.

If there is a tag in the X5 register upon entry, the Variable Name Table (Table M), Common Name Table (Table O), and the Equivalence Name Tables (Table X and Y) are scanned. Should the name not appear in any of the tables, a jump to ISL (Identify Symbolic Tag) for a tag is executed, and the tag is returned in the X5 register and the constant, if itexists, is processed as above, but is returned in the X4 register. If the variable is in the common or equivalence tables, the variable and a Variable Tag (V-tag) are stored in the Variable Name Table and the V-tag is returned in the X5 register. If the variable is already in the Variable Name Table (Table M) the previously stored tag is returned in the X5 register.

Subroutines Called:	ADF - Advance Tables
	ISL - Identify Symbolic Tag
	SCT - Scan Table
Temporaries/Flags:	IPS - Program Type
	TGH - Statement Tag (set)
	TGV - Variable Tag (set)

<u>Tables Referenced</u>: TBJ - Argument Tag
TBN - Variable Tag
TBM - Variable Name
TBY - Equivalence Primary Name
TBX - Equivalence Secondary Name
TBO - Common Name

<u>Entry/Exit Register Conditions</u>:

Entry: X4 - constant or zero
X5 - tag or zero

Exit: if X5 = 0 on entry
X5 = constant $-2^{16-1} - 2^{16-1}, -0$
= H-tag $2^{16-1} - 2^{17-1}$

if X5 $\neq$ 0, X4 = 0
X5 = Variable Tag

if X5 $\neq$ 0, X4 = 0
X5 = Variable Tag
X4 = constant
Statement Tag (H-tag)

PUA - PROCESS UNIQUE VARIABLE ASSIGNMENTS

After all blank and numbered common locations have been assigned, PUA is entered to make unique assignments. If the mode of compilation is FORTRAN IV, the MCA routine is called to make labeled common assignments. The equivalence tables are then examined and the PXG routine which processes equivalence groups is called for each primary name that has not yet been processed. After all equivalences have been handled, the Variable Name Table is searched to assign core locations for all variables that have not yet been assigned. The tag for the variable along with the starting address is entered into Table J. The length of the array or variable is added to the starting address in order to determine the starting address of the next array or variable. When all variables have been assigned, the starting address for the next one is set as the relative start of the next program and the routine exits.

Subroutines Called:

ADF - Advance Tables
MCA - Make FORTRAN IV Relative Assignments
SCT - Scan Table

Temporaries/Flags:

BAV - Base Address for Variables
CTY - Common Block Type Indicator
IPS - Program/Subprogram Indicator
MOD - Subprogram Mode
MOE - Program Mode
TBN -
TBP -
TBX - Table Parameters
TBY -
TBZ -
TMC - Free Temporary
ZAA - Relative Start of Current Program or Subroutine

Tables Referenced:	Variable Tag	(N)
	Array Tag	(P)
	Equivalence Secondary Name	(X)
	Equivalence Primary Name	(Y)
	Equivalence Bias	(Z)

Entry/Exit Register Conditions: None