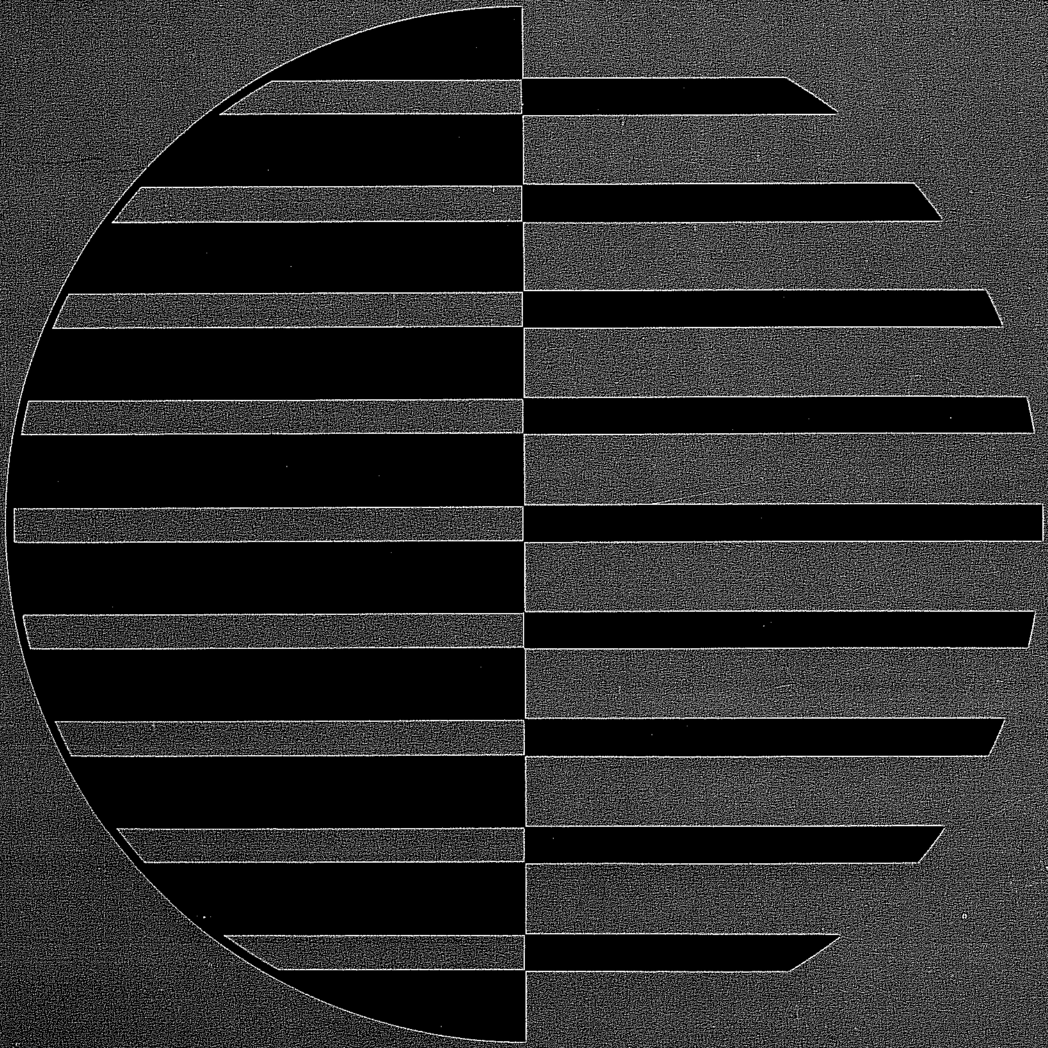
CONTROL DATA® 6600 Computer System Programming System/ Reference Manual

Volume 1 ASCENT

Assembly System CENTral Processor



SECOND EDITION

Peripheral and Control Processor Instruction Execution Times

Mnemonic & Octal Code		Name	Time (Major Cycles)
PSN	00	Pass	1
LJM	01	Long jump to m + (d)	2-3
RJM	02	Return jump to m + (d)	3-4
UJN	03	Unconditional jump d	1
ZJN	04	Zero jump d	1
NJN	05	Nonzero jump d	1
PJN	06	Plus jump d	1
MJN	07	Minus jump d	1
SHN	10	Shift d	1
LMN	11	Logical difference d	1
LPN	12	Logical product d	1
SCN	13	Selective clear d	1
LDN	14	Load d	1
LCN	15	Load complement d	1
ADN	16	Add d	1
SBN	17	Subtract d	1
LDC	20	Load dm	2
ADC	21	Add dm	2
LPC	22	Logical product dm	2
LMC	23	Logical difference dm	2
PSN	24	Pass	1
PSN	25	Pass	1
EXN	26	Exchange jump	min. 20
RPN	27	Read program address	1
LDD	30	Load (d)	2
ADD	31	Add (d)	2
SBD	32	Subtract (d)	2
LMD	33	Logical difference (d)	2
STD	34	Store (d)	2
RAD	35	Replace add (d)	3
AOD	36	Replace add one (d)	3
SOD	37	Replace subtract one (d)	3
LDI	40	Load ((d))	3
ADI	41	Add ((d))	3
SBI	42	Subtract ((d))	3
LMI	43	Logical difference ((d))	3
STI	44	Store ((d))	3
RAI	45	Replace add ((d))	4
AOI	46	Replace add one ((d))	4
SOI	47	Replace subtract one ((d))	4
LDM	50	Load (m + (d))	3-4
ADM	51	Add (m + (d))	3-4
SBM	52	Subtract (m + (d))	3-4
LMM	53	Logical difference (m + (d))	3-4
STM	54	Store (m + (d))	3-4
RAM	55	Replace add (m + (d))	4-5
AOM	56	Replace add one (m + (d))	4-5
SOM	57	Replace subtract one (m + (d))	4-5
CRD	60	Central read from (A) to d	min. 6
CRM	61	Central read (d) words from (A) to m	5 plus 5/word
CWD	62	Central write to (A) from d	min. 6
CWM	63	Central write (d) words to (A) from m	5 plus 5/word
AJM	64	Jump to m if channel d active	2
IJM	65	Jump to m if channel d inactive	2
FJM	66	Jump to m if channel d full	2
EJM	67	Jump to m if channel d empty	2
IAN	70	Input to A from channel d	2
IAM	71	Input (A) words to m from channel d	4 plus 1/word
OAN	72	Output from A on channel d	2
OAM	73	Output (A) words from m on channel d	4 plus 1/word
ACN	74	Activate channel d	2
DCN	75	Disconnect channel d	2
FAN	76	Function (A) on channel d	2
FNC	77	Function m on channel d	2

PREFACE

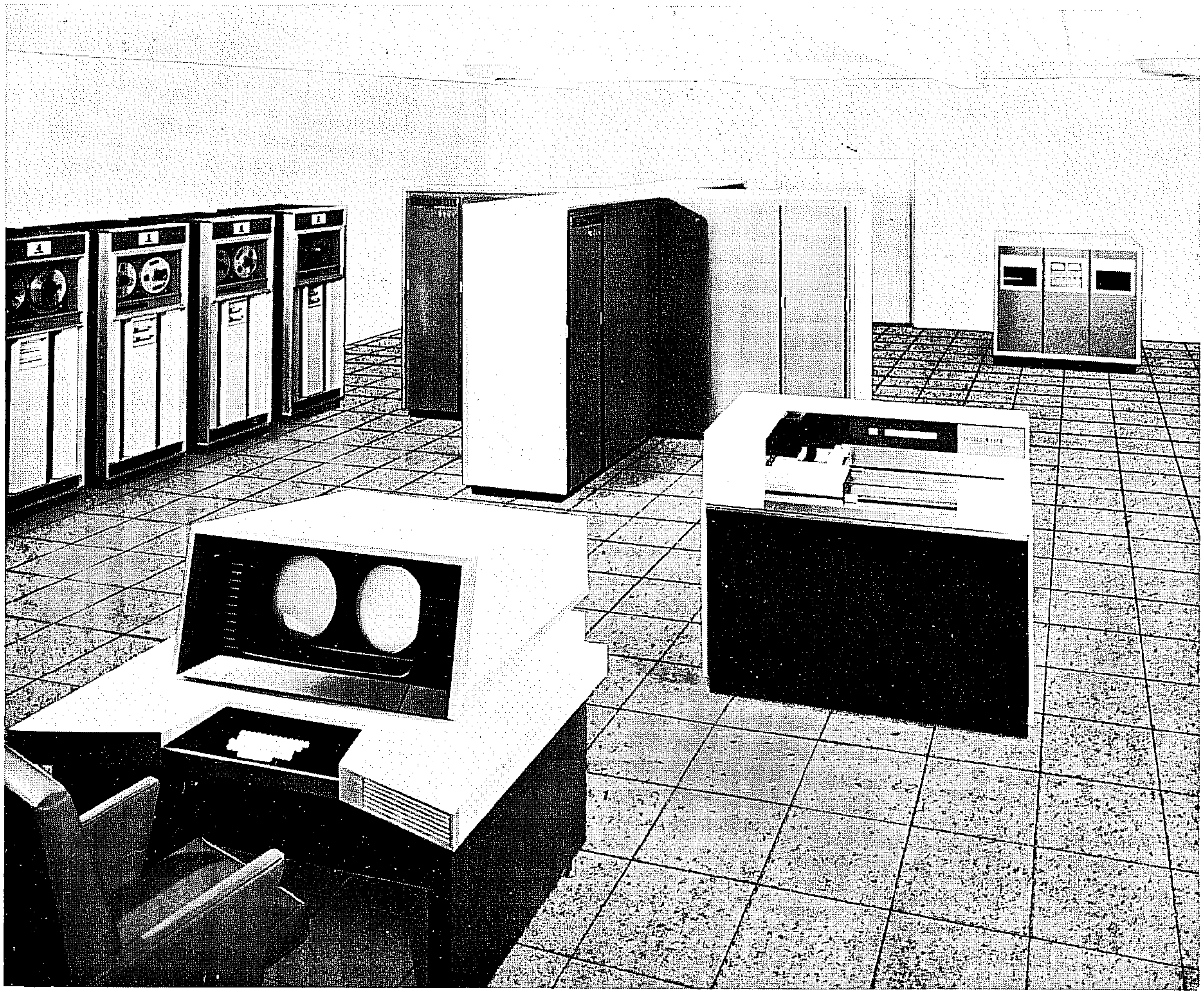
The CONTROL DATA 6600 Programming System is comprised of three major sections, FORTRAN, ASCENT, and ASPER language processing programs. Each step of an object program is capable of switching control between the FORTRAN and ASCENT programs and from either of them to the ASPER program. To preserve processing efficiency, each subsystem has a direct, although not exclusive, path for its own type of instructions. Only when a switch between languages occurs do parts other than the direct path act.

This manual is devoted to a description of the direct path for the 6600 Central Processor assembly language programs, ASCENT. Part 1 gives a general orientation to the 6600 hardware and system concepts as related to ASCENT programming. Part 2 defines specific entities of the language. Part 3 and 4 give the instruction forms; Part 5, the pseudo operations; Parts 6 and 7, the system macros; and Part 8, the assembler diagnostics. Part 9 describes how subroutines are used, while Parts 10 and 11 explain Program Segmentation and Organization, respectively.

TABLE OF CONTENTS

FIGURES

APPENDIX



6600 COMPUTING SYSTEM

Main frame *(center)—contains 10 peripheral and control processors, central processor, central memory, some I/O synchronizers.*

Display console *(foreground)—includes a keyboard for manual input and operator control, and two 10-inch display tubes for display of problem status and operator directives.*

CONTROL DATA 607 tapes *(left front)—½ inch magnetic tape units for supplementary storage; binary or BCD data handled at 200, 556, or 800 bpi.*

CONTROL DATA 626 tapes *(left rear)—1-inch magnetic tape units for supplementary storage; binary data handled at 800 bpi.*

Disc file *(right rear)—Supplementary mass storage device holds 500 million bits of information.*

CONTROL DATA 405 card reader *(right front)—reads binary or BCD cards at 1200 card per minute rate.*

SYSTEM ORGANIZATION

The CONTROL DATA® 6600 is a large-scale, solid-state, general-purpose digital computing system. The advanced design techniques incorporated in the system provide for extremely fast solutions to data processing, scientific and control center problems.

Within the 6600 are eleven independent computers (Fig. 1). Ten of these are constructed with the peripheral and operating system in mind. These ten have separate memory and can execute programs independently of each other or the central processor. The eleventh computer, the central processor, is a very high-speed arithmetic device. The common element between these computers is the large central memory.

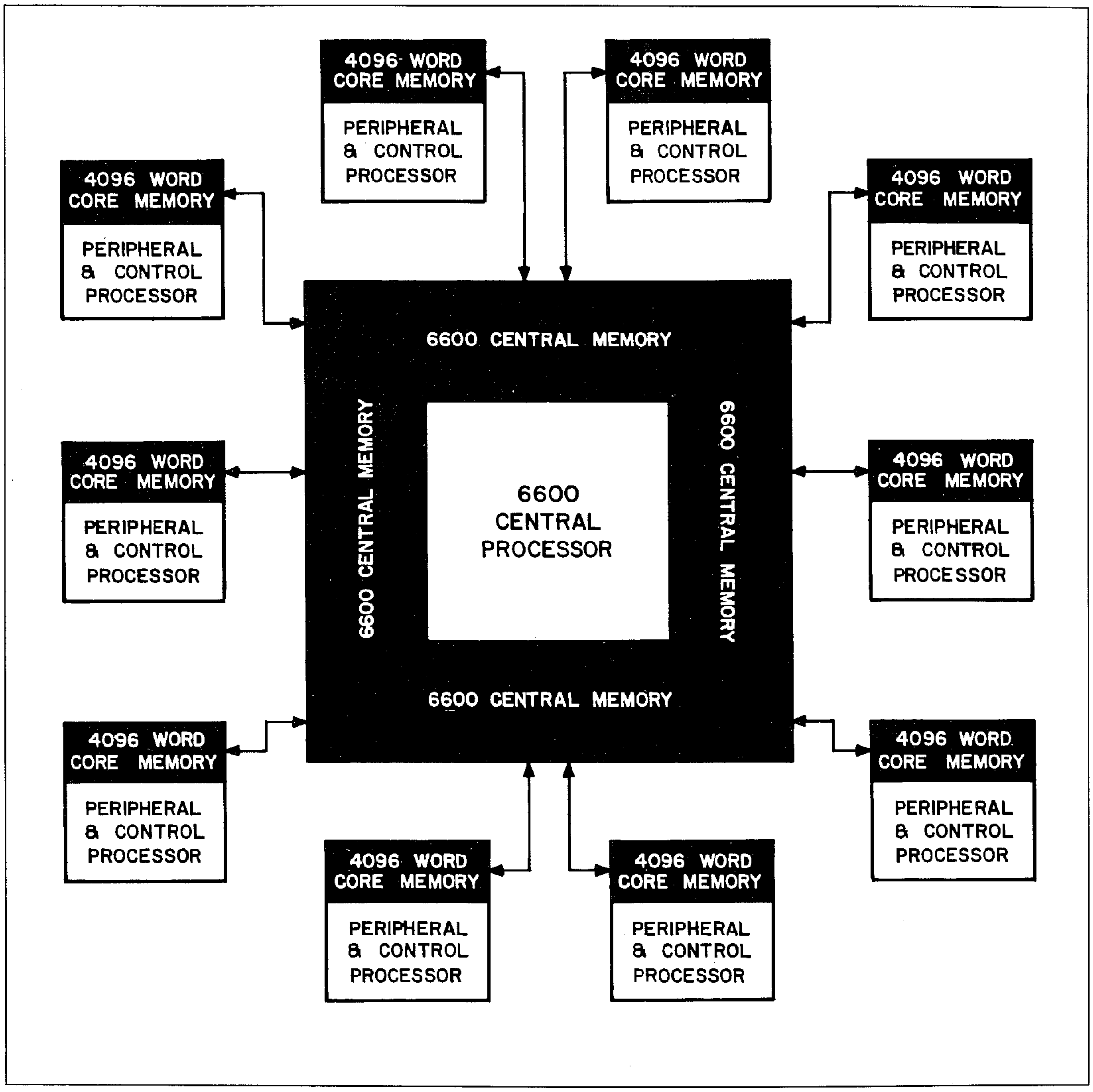
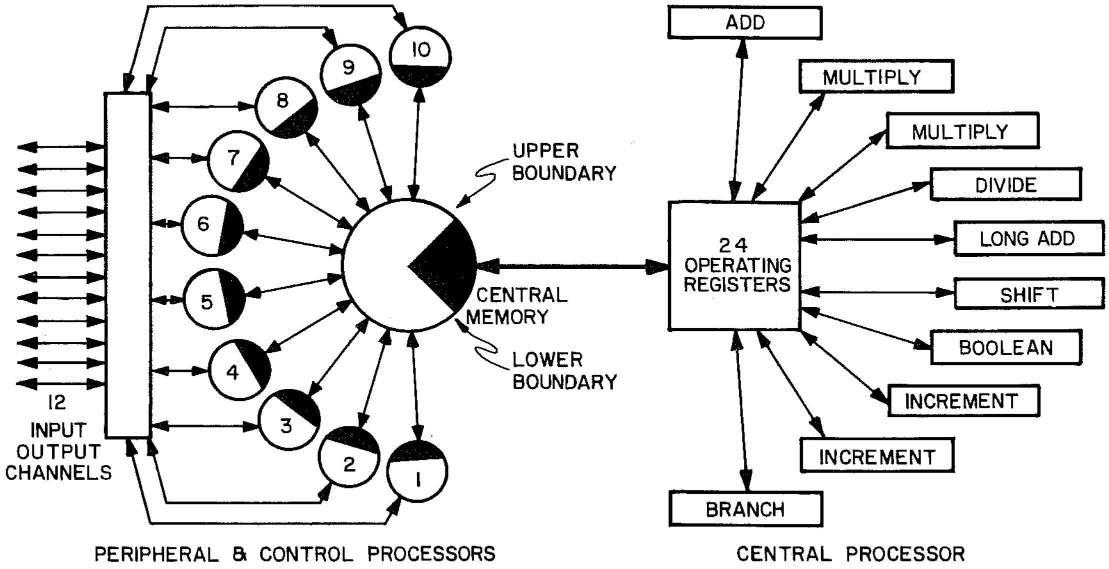


Figure 1 CONTROL DATA 6600



CENTRAL MEMORY

– 131,072 words

– 60-bit words

– Memory organized in 32 logically independent banks of 4096 words with corresponding multi-phasing of banks

– Random access, coincident-current, magnetic core

– One major cycle for read-write

– Maximum memory reference rate to all banks – one address/minor cycle

– Maximum rate of data flow to/from memory – one word/minor cycle

DISPLAY CONSOLE

– Two display tubes

– Modes
Character
Dot

– Character size
Large – 16 characters/line
Medium – 32 characters/line
Small – 64 characters/line

– Characters
26 alphabetic
10 numeric
11 special

Figure 2 BLOCK DIAGRAM OF 6600

1. ASCENT SYSTEMS CONCEPTS

1.1 SYSTEM CONFIGURATION

The basic 6600 computing system is comprised of a central processor with 131,072 words (60-bits each) of magnetic core memory, ten peripheral processors with 4096 12-bit words of magnetic core memory each and joint control of 12 input/output data channels with a minimum of:

One disk unit with 8 million 60-bit words

One display console

One 1200 card/minute reader

One 1000 line/minute printer

One 250 card/minute punch

Bank of two 607 or 626 magnetic tape units

The central processor and its operation are under the direction of the peripheral processors. At any given moment, a peripheral processor has access to all, or only one subset, of central memory at the discretion of the directing peripheral processor. To the central processor the relevant subset is always addressed as locations 0 to n, where n is its size regardless of the location of the subset within the total capacity of central memory. The primary function of the central processor is to handle the computational load, while central memory stores operational and system programs together with the data they require.

The peripheral processors as a class have autonomy over the input/output channels and have the capability of directly addressing a word in central memory. It is a function of these processors to transfer into central memory, from peripheral input equipment, the programs to be executed by the central processor, as well as all input data required at execute time. Similarly, they must transfer output data, generated by the central programs, from central memory and place it on the proper peripheral equipment.

1.2 OPERATING SYSTEM CONCEPTS

The timely and orderly exchange of programs and data between central memory and peripheral equipments is the function of the standard operating system, SIPROS. Complete writeups on SIPROS are contained in the manual by that name. However, certain items of particular importance to ASCENT programmers are given here.

1.2.1 GENERAL

While computational programs are operating in central memory, parts of SIPROS reside in different portions of the 6600 system. The PP controlling the system performs all the executive and monitoring functions. This Executive program is responsible for the control and management of all other parts of the system, including allocation of central memory, tasks assigned to the other PP's, and allocation of and communication with peripheral equipment. In addition, it monitors the status of the current job and checks regularly for changes in status.

An important feature of the system is the use of the disk. To optimize such use, a Disk Executive routine, upon receiving assignments from the Executive program, processes all requests involving reading or writing the disk. Two disk "slave" PP's, under the control of the disk executive, cooperate alternately in reading (or writing) information into central memory and in writing (or reading) data from consecutive sectors of the disk. This two-processor approach to disk reading and writing maximizes the disk transfer rate.

Residing in the same PP as the Disk Executive is the Console Display program. It is through the use of the console that man-machine communications are maintained. It supplies the operator with information such as the status of jobs in central memory and when tapes require mounting or dismounting. It also allows the operator to communicate with the system such information as changes in priorities or a request for a display of the jobs in central memory. The remaining peripheral processors (Pool PP's) are assigned a variety of tasks. These include such operations as:

Job Loader

Card Reader

Printer

Punch

Tape

Job Termination

1.2.2 COMMUNICATIONS CONVENTIONS

To accomplish the various tasks mentioned in the preceding section, the monitor function of the

Executive program watches the central processor programs for status changes and for I/O requests. The links between the executing program and the operating system are defined and implemented through the conventions of System Macro* operations.

A system macro produces a calling sequence to a communication subroutine stored in a standard resident section of all central processor programs. Parameters of the macro supply the subroutine with values and their addresses which designate the operation to be performed. The subroutine places a sequence of values obtained directly and indirectly from the parameters into designated core communication words. One of the values is a flag to the monitor function of the Executive program and another is the operation code which defines the request.

A *ready* or *not ready* status exists after the execution of the system macro, which may be checked by the object program. It is possible, with consecutive system macros, to have a second request ready before the first one has been initiated by the Executive program. If this occurs, the communication subroutine holds the request until the communication words become ready, thereby preventing any requests from being unfulfilled.

If a *not ready* status exists, the communication subroutine will select one of the following conditions depending upon the request mode:

> (1) If the request includes an indication that no processing may take place until the requested operation is completed, the system is notified and control is transferred to another central processor program during the wait period.
>
> (2) If control is returned to the object program prior to the completion of the requested operation, or if no other program is ready for the central processor, the communication subroutine waits, within itself, for the request to be completed. If another program achieves *ready* status, the second program has precedence and the first program waits for a control transfer.

If the buffering mode (2) is indicated, the request is initiated and control is returned to the object program. The program must then interrogate for the completion of the requested operation, checking the status word which the program has listed as a parameter of the macro.

* The term macro is used only because the specification format is consistent with true macro forms (as defined later) rather than the way it is implemented for system communication. Object code, in this case, is subroutine form.

2. LANGUAGE DEFINITIONS

2.1 CHARACTERS

ASCENT uses the following character set:

The alphabet — letters A through Z

The arabic numerals — numbers 0 through 9

The special characters — + - / * = ( ) . , $ space

2.2 SYMBOLS

A symbol is any arrangement of letters and numbers which starts with a letter and contains no more than 8 total characters.

Examples:

T, PROG, ZIZ, ABCD1234

Exceptions:

1. Character arrangements A0, A1, . . . A7, B0, B1, . . . B7, and X0, X1, . . . X7 are excluded.
2. The special character * has momentary properties of a symbol under certain usage as defined under 3.3.2.

Register Definitions:

A0, A1, . . . A7 are used for address registers (18 bits)

B0, B1, . . . B7 are used for index registers (18 bits)

X0, X1, . . . X7 are for arithmetic and operand registers (60 bits)

2.3 CONSTANTS

Constants may be any of the following forms.

2.3.1 INTEGER

An integer is any arrangement of 18 or less decimal digits from $-(2^{59}-1)$ to $(2^{59}-1)$.

Examples:

3, 8125, 1234567891011121

2.3.2 OCTAL

An octal constant is any arrangement of 20 or less octal digits 0 through 7 appended with the letter B.

Examples:

47B, 770077B, 52525252525252525252B

2.3.3 SYMBOLIC

A symbolic constant meets the specifications for a symbol but is equated to a constant or to the difference of two symbols.

Examples: TAM EQU 3677B – 150B

GAT EQU 64+99

MAG EQU 20

SAG EQU TAG – SAM

if TAG and SAM are assigned memory locations 120_8 and 100_8, respectively, then SAG is equated to 20_8.

2.3.4 SINGLE PRECISION FLOATING POINT

A single precision floating point constant is expressed by one of two forms:

1. An arrangement of 15 or less decimal digits and a single decimal point.

Examples:

1., .1, 0.1, 1.0, 5248.6153

2. An arrangement of 15 or less decimal digits with or without a single decimal point, followed by a power of 10 representation as follows:

   E ± n or En

   where: E specifies that an exponent follows

   n is any arrangement of 3 or less decimal digits and is the power of 10 to be applied to the constant

   ± is the sign of n. It may be omitted in the case of +.

Examples:

1E+5, 1.0E+250, .1E–30, 5248.6153E7, 14E51

2.3.5 DOUBLE PRECISION FLOATING POINT

A double precision floating point constant is expressed by one of two forms:

1. An arrangement of 29 or less decimal digits with or without a decimal point followed by the letter D.

Examples:

1.D, .1D, 0.1D, 5248.6153D, 10D

2. An arrangement of 29 or less decimal digits with or without a decimal point followed by a power of 10 representation as follows:

   D± n or Dn

   where: D is a required letter

   n is any arrangement of 3 or less decimal digits and is the power of 10 to be applied to the constant

   ± is the sign of n. It may be omitted in the case of +.

Examples:

1D+5, 1.0D+200, .1D−77, 5248.6153D7, 14D51

2.3.6 COMPLEX

A complex constant is any pair of single precision floating point constants separated by a comma and enclosed in parentheses.

Examples:

(1.0,−2.2), (1E+5, .001E−15)

2.4 OPERATORS

In certain cases, operators join an abbreviated mnemonic code in defining the numerical operation code of an instruction. Operators used are the special characters:

+ addition

− subtraction

* multiplication

/ division

The + and − are also used in address manipulation specifications.

Examples:

SYMBOL + 2, SYMBOL − 1

2.5 LITERALS

Literals are used for addressing a core location whose contents are specified by the value within parentheses. Literals may be any of the following forms:

(Constant)

(Symbol)

(Symbol ± I)

(Symbol − Symbol)

where: I is an integer, octal or symbolic constant.

When a two-word form such as

(Floating Double Precision Constant)

(Complex Constant)

is defined, the first word only is addressed.

Examples:

(3.2), (SAM), (SAM + 5), ((700., 5.1E31)), (3.4D70)

2.6 SEPARATORS

Separators are used to indicate the end of distinct entities of an instruction; the six characters used are:

$, + space . =

Other characters assume the role of separators depending upon usage. The four characters are:

+ − * /

2.7 OPERANDS

Operands are combinations of symbols, literals, the operators + and −, and certain types of constants.

The acceptable forms are:

Symbol

Symbol ± I

Symbol − Symbol

± I

Literal

Literal + 1

where: I is an integer, octal or symbolic constant

Examples:

SAM, SAM + 3, SAM − 15, SAM1 − SAM2, 14, (1.25)

2.8 FIELDS

An instruction is a combination of the following fields.

LOCATION: Provides a symbol for referencing by other instructions.

OPCODE: Defines the instruction.

ADDRESS: Supplies the instruction with appropriate operands.

REMARKS: Programmer notes only. This field has no effect on the assembly process and must begin with a period in or after column 11.

3. LANGUAGE SPECIFICATIONS

3.1 FORMATS

ASCENT has one basic instruction format:

LOCATION OPCODE ADDRESS .REMARKS

Examples of various central processor instructions are given below:

LOCATION	INSTRUCTION	REMARKS
	BX6 X1	.X1 TO X6
START	BX4 −X3	.−X3 TO X4
	FX7 X6*X4	.FLOATING X6*X4 to X7
	BX3 −X4+X1	.X1+COMP.X4 TO X3
	EQ B5 B2 AB	.IF B5=B2, GO TO AB
	SA7 B2+DATA	.STORE X7 TO DATA +B2
	SA7 DATA	.STORE X7 to B0+DATA
	NZ X1 ABC	.IF X1 NOT ZERO, GO TO ABC
	RJ SUB	.RETURN JUMP TO SUB
START1	RDC 1,ST,(BA),(BA+8),8,2	.LIST
	SB1 1 $ SA2 DATA+1	.PACKED CARD
START2	LX1 6 $ MX2 48 $ JP AB+2	.MAXIMUM 6 PER CARD
	SB6 −8 $ SA5 B6+DATA $ SB7 B5−B6	.BEGIN REMARKS WITH PERIOD
	JP B2+BETA	.JUMP TO B2+BETA

The Location field is a fixed length field and occupies columns 2-9.

The Opcode field is variable length and starts in or after column 11 and must be terminated by at least one separator.

The Address field is variable length and has any of the following formats:

REGISTER

−REGISTER

REGISTER OPERATOR REGISTER

−REGISTER OPERATOR REGISTER

REGISTER REGISTER OPERAND

REGISTER OPERATOR OPERAND

REGISTER OPERAND

OPERAND

LIST

(LIST is a sequence of registers and operands as specified for the operation code. Adjacent operands must be separated by a comma. The LIST form in an address field is used in certain pseudo and macro codes.)

The Remarks field* is either blank or starts with the special character, period, in any column 11-72.

ASCENT considers only card columns 2 through 72. Column 1 is reserved for the exclusive use of the Programming System Control Package. Column 10 is blank and serves as a separator between the Location field and the Opcode field. ASCENT ignores column 10.

RULES and EXTENSIONS:

Up to six instructions may be placed on one input card. The special character $ is used to denote the beginning of a new instruction. The following rules apply:

1. Only one location field may be used on a card regardless of the number of instructions it contains. When one is used, it applies to the first instruction on the card.

* Certain instructions exclude the Remarks field. The exclusion is noted with the definitions of each relevant instruction in Table 2.

2. The $ acts as the recurrence of column 10 on the card. The next expected item is an opcode.
3. All instructions on the card must be completed prior to column 73.

3.2 FIELDS

3.2.1 LOCATION FIELD

The Location field may be blank or contain a plus, minus, or symbol starting in any column 2-5 and ending before colomn 10.

A symbol or plus causes the assembler to assign the first instruction on the card to the leftmost position in a 60-bit word. Any partially filled object code word is filled with *no operation* instructions, and the new instruction is forced into the leftmost position of a new object code word. A minus, normally used to override assembly forcing assumptions, causes the first instruction on the card to be placed in the next available position in the current object code word if space is available.

Rules:

1. Any given symbol may appear in the location field only once within a program or subroutine.
2. Symbols defined as formal parameters of a subroutine may not appear in the location field within the subroutine. (See Section 9.2.)
3. Symbols defined in COMMON or DIMENSION statements may not be used in the location field. (See Section 9.2.)
4. Plus and minus signs may be used repeatedly in the location field. However, an instruction may not be referenced by using either of the special characters.
5. Register names may not appear in the location field.
6. The special character * may not appear in the location field.

3.2.2 OPCODE FIELD

The Opcode field may contain any of the following items.

1. The 6600 central processor mnemonic codes as given in Table 1.
2. ASCENT pseudo codes as given in Table 2.
3. System macro codes as given in Table 3.
4. Name of any programmer-defined macro.
5. An integer or octal constant $\leq 2^{12} - 1$.

Mnemonic codes are evaluated to determine either the octal equivalent of the code or its class. Pseudo operations are interpreted and used in assembler control. Macro instructions produce predefined sequences of object code with parametric values changed to actual values and, for system macros, with the generated calling sequence used by the communication subroutine. The numerical opcode is converted if necessary and placed in the upper 12-bit position (Opcode, i, j) of a 30-bit instruction format.

A separator terminates the field.

3.2.3 ADDRESS FIELD

The content of the address field varies with the instruction. Therefore, several types of instructions are important in its specification.

1. No address required.

   PS

   NO

2. Numeric

LXi	jk
AXi	jk
MXi	jk

   The address field may contain blank, or an integer or octal constant $< 63_{10}$.

3. Registers and Operators

CXi	Xk	FXi	Xj + Xk
BXi	Xj	FXi	Xj −Xk
BXi	−Xk	DXi	Xj + Xk
BXi	Xj * Xk	DXi	Xj −Xk
BXi	Xj + Xk	RXi	Xj + Xk
BXi	Xj − Xk	RXi	Xj −Xk
BXi	−Xk * Xi	FXi	Xj * Xk
BXi	−Xk + Xj	DXi	Xj * Xk
BXi	−Xk −Xj	RXi	Xj * Xk
IXi	Xj + Xk	FXi	Xj/Xk
IXi	Xj − Xk	RXi	Xj/Xk

The address field is used not only to specify the registers but also to define the exact operation code within the class defined by the opcode. Except for instructions containing minus signs or slash signs, the registers may be ordered arbitrarily. However, when a minus or slash sign is used, the order given above must be maintained.

4. Registers only

NXi	Bj	Xk
ZXi	Bj	Xk
UXi	Bj	Xk
PXi	Bj	Xk
LXi	Bj	Xk
AXi	Bj	Xk

The register designation in the address field may be in any order. If the Bj term is equal to B0, it may be omitted.

5. Opcode complete, order independent

Constant		K	ZR	Xj	K
RJ		K	NZ	Xj	K
JP	Bi +	K	PL	Xj	K
ZR	Bi	K	NG	Xj	K
NZ	Bi	K	IR	Xj	K
PL	Bi	K	OR	Xj	K
NG	Bi	K	DF	Xj	K
			ID	Xj	K

The register designation and the operand may be in any order. A B0 in the jump instruction may be omitted.

6. Opcode complete, order dependent

EQ	Bi	Bj	K
NE	Bi	Bj	K
GE	Bi	Bj	K
LT	Bi	Bj	K

The registers and the operand should be written in the order designated. If only one B term is specified, the Bj is asumed to be B0.

7. 18-bit arithmetic with registers and operand

SAi	Aj ± operand
SAi	Bj ± operand
SAi	Xj ± operand
SBi	Aj ± operand
SBi	Bj ± operand
SBi	Xj ± operand
SXi	Aj ± operand
SXi	Bj ± operand
SXi	Xj ± operand

The registers and operands may be written in either order when the separating sign is plus. However, when the sign is minus, the register must be first. A B0 may be omitted.

8. 18-bit arithmetic for sum of two registers

SAi	Xj + Bk	SBi	Xj + Bk	SXi	Xj + Bk
SAi	Aj + Bk	SBi	Aj + Bk	SXi	Aj + Bk
SAi	Bj + Bk	SBi	Bj + Bk	SXi	Bj + Bk

The registers may be written in any order. A B0 may be omitted.

9. 18-bit arithmetic for difference of two registers

SAi	Aj − Bk	SBi	Aj − Bk	SXi	Aj − Bk
SAi	Bj − Bk	SBi	Bj − Bk	SXi	Bj − Bk

The register order must be maintained.

3.3 SPECIAL USAGE

3.3.1 ASCENT FORCING CONVENTION

ASCENT forces the next instruction (NI) to the upper portion of the 60-bit word following a PS, JP, or RJ instruction. A minus sign in the location field of these instructions will override this feature of ASCENT.

3.3.2 ASTERISK

The special character *, when used as an operand or part of an operand, assumes the value of the current object code address. The legal forms are:

*

* ± Constant

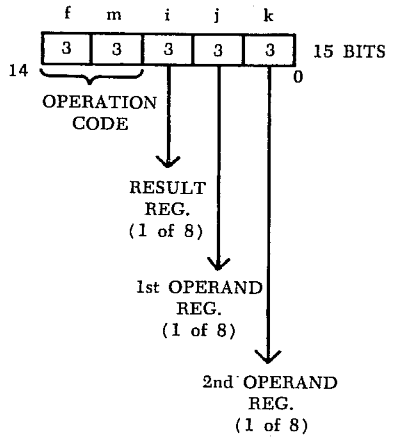
Examples:

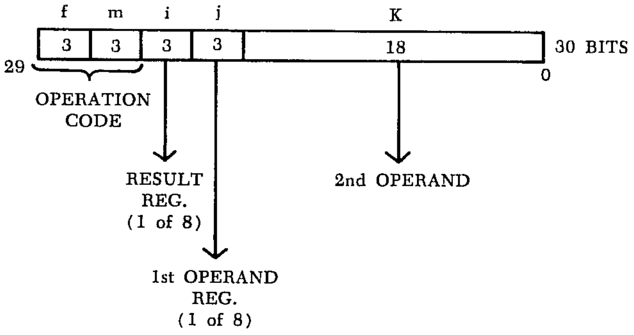
SAi	*	. Load current object code word
JP	* + 3	. Jump forward 3 60-bit words

4. CENTRAL PROCESSOR INSTRUCTIONS

4.1 INSTRUCTION FORMAT

An instruction may have a 15-bit or a 30-bit format. Either format uses a 6-bit operation code. The result register requires 3 bits and the number of bits used for the operand varies with the instruction.





4.2 DEFINITIONS FOR CENTRAL PROCESSOR INSTRUCTIONS

fm Operation code (6 bits)

i Specifies which of eight designated registers is the result register (3 bits)

j Specifies which of eight designated registers is the first operand register (3 bits)

k Specifies which of eight designated registers is the second operand register (3 bits)

jk Constant, indicating number of shifts to be taken (6 bits)

K Constant, indicating branch destination or second operand (18 bits)

A One of eight address registers (18 bits)

B One of eight index registers (18 bits) B0 is permanently set equal to zero

X One of eight operand registers (60 bits)

4.3 OPERATING REGISTERS

There are 24 operating registers identified by letters and numbers. These registers are labeled:

A Address register (A0,A1 . . . A7)

B Increment register (B0,B1 . . . B7)

X Operand register (X0,X1 . . . X7)

4.3.1 A REGISTER

The execution of a SAi (i = 1-5) instruction produces an immediate memory reference to the address contained in Ai and reads the contents at that location into the corresponding operand register Xi (i = 1-5). When a SAi (i = 6, 7) instruction is executed, the contents of the corresponding X register is stored at the address specified by Ai. The address register A0 is used for temporary storage; i.e., the execution of a SA0 instruction does not affect a load of X0.

SA3 = A4 + 10B

This example adds 10_8 to the address in A4 and sets the A3 register to this sum. The X3 register is set to the contents of the location specified by the new A3.

SA6 = A2 − 15B

This example stores the contents of X6 into the location obtained by subtracting 15_8 from the address in A2.

4.3.2 B REGISTER

The increment register B0 is set permanently to an 18-bit plus zero which may be used in testing for zero or as an unconditional jump modifier. B1-B7 are used as modifiers and for program indexing. For example, B4 may be used to control the number of passes of a program loop, terminating when a given condition is reached.

SB3 = B5 + B4

This example adds the values contained in the two increment registers, B5 and B4, and places the result in B3.

4.3.3 X REGISTER

Any of the registers X0-X7 may be used as a result or operand register of an instruction. The registers X1-X5 hold read operands from central memory, while X6 and X7 hold results sent to central memory. The operand registers may be used and changed without causing a change in the corresponding address register.

BX2 X2 + X4

This example performs the logical addition of X2 and X4 and places the resulant sum in X2.

SX6 A2 − B5

This example subtracts the contents of B5 from the contents of A2 and stores the result in X6. This operation produces no change in memory.

4.4 DESCRIPTION OF OPERATION CODES

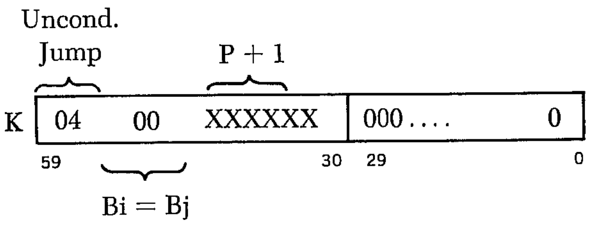
Following is a list of the instructions for the central processor. They are ordered by octal code which in turn separates the instructions by functional unit.

00 PS Program Stop

Stops the central processor at the current step in the program. An exchange jump instruction is necessary to restart the central processor.

01 RJ K Return Jump to K

Stores an unconditional jump (04) and the current program address plus one in the upper 30 bits of K and then branches to K+1 for the next instruction step. The contents of K after the instruction appear as follows:



A jump to K at the end of the branch routine returns to the original programming sequence.

02 JP Bi + K Jump to Bi + K

Adds the contents of Bi to K and branches to the address specified by the resultant sum. When Bi = B0, the branch address is K. Addition is performed modulus $2^{18} - 1$.

030 ZR Xj K Jump to K if Xj = 0

Branches to K if Xj is equal to zero. If the condition is not met, the next consecutive instruction step is executed. The test is made in the long add unit.

031 NZ Xj K Jump to K if Xj ≠ 0

Branches to K if Xj is not equal to zero. If the condition is not met, the next consecutive instruction step is executed. The test is made in the long add unit.

032 PL Xj K Jump to K if Xj is Plus

Branches to K if Xj is positive. If the condition is not met, the next consecutive instruction step is executed. The test is made in the long add unit.

033 NG Xj K Jump to K if Xj is Negative

Branches to K if Xj is negative. If the condition is not met, the next consecutive instruction step is executed. The test is made in the long add unit.

034 IR Xj K Jump to K if Xj is In Range

Branches to K if Xj is in range. The range test is a comparison with infinity ($377700 \ldots 0_8$) and is made in the long add unit.

035 OR Xj K Jump to K if Xj is Out of Range

Branches to K if Xj is out of range. The range test is a comparison with infinity ($377700 \ldots 0_8$) and is made in the long add unit.

036 DF Xj K Jump to K if Xj is Definite

Branches to K if Xj is definite. The test is a comparison against an indefinite quantity ($177700 \ldots 0_8$) and is made in the long add unit.

037 ID Xj K Jump to K if Xj is Indefinite

Branches to K if Xj is indefinite. The test is a comparison against an indefinite quantity (177700 . . . 0_8) and is made in the long add unit.

04 EQ Bi Bj K Jump to K if Bi = Bj

Compares Bi with Bj and branches to K if Bi is equal to Bj. The test is made in the increment unit.

04 ZR Bi K Jump to K if Bi = B0

Compares Bi with B0 and branches to K if Bi is zero. The test is made in the increment unit.

05 NE Bi Bj K Jump to K if Bi ≠ Bj

Compares Bi with Bj and branches to K if Bi is not equal to Bj. The test is made in the increment unit.

05 NZ Bi K

Compares Bi with B0 and branches to K if Bi is not zero. The test is made in the increment unit.

06 GE Bi Bj K Jump to K if Bi ≧ Bj

Compares Bi with Bj and branches to K if Bi is greater than or equal to Bj. The test is made in the increment unit.

06 PL Bi K Jump to K if Bi ≧ B0

Compares Bi with B0 and branches to K if the result is positive. The test is made in the increment unit.

07 LT Bi Bj K Jump to K if Bi < Bj

Compares Bi with Bj and branches to K if Bi is less than Bj. The test is made in the increment unit.

07 NG Bi K Jump to K if Bi < B0

Compares Bi with B0 and branches to K if Bi is negative. The test is made in the increment unit.

10 BXi Xj Transmit Xj to Xi

Transfers the 60-bit word in operand register Xj to Xi.

11 BXi Xj*Xk Logical Product of Xj and Xk to Xi

Forms the logical product (AND function) of the 60-bit words in operand registers Xj and Xk and places the result in Xi.

Xj = 0101
Xk = 1100
Xi = 0100

12 BXi Xj + Xk Logical Sum of Xj and Xk to Xi

Forms the logical sum (inclusive OR) of the 60-bit words in operand registers Xj and Xk and places the result in Xi.

Xj = 0101
Xk = 1100
Xi = 1101

13 BXi Xj − Xk Logical Difference of Xj and Xk to X0

Forms the logical difference (exclusive OR) of the 60-bit words in operand registers Xj and Xk and places the result in Xi.

Xj = 0101
Xk = 1100
Xi = 1001

14 BXi −Xk Transmit Xk Complement to Xi

Transmits the complement of the 60-bit word in operand register Xk to Xi. The contents of Xk are not changed.

15 BXi −Xk*Xj Logical Product of Xj and Xk Complement to Xi

Forms in Xi the logical product (AND function) of Xj and the complement of Xk. The contents of Xk and Xj are not changed.

Step 1	Xj = 0101	Step 2	Xj = 0101
	Xk = 1100		Xk = 0011
			Xi = 0001

16 BXi −Xk + Xj Logical Sum of Xj and Xk Complement to Xi

Complements the 60-bit word in Xk, then forms the logical sum (inclusive OR) of this quantity and Xj, and places the result in Xi. The contents of Xk and Xj are not changed.

Step 1	Xj = 0101	Step 2	Xj = 0101
	Xk = 1100		Xk = 0011
			0111

17 BXi −Xk − Xj Logical Difference of Xj and Xk Complement to Xi

Complements the 60-bit word in Xk. Then forms the logical difference (exclusive OR) of this quantity and Xj, and places the result in Xi. The contents of Xk and Xj are not changed.

Step 1 Xj = 0101
Xk = 1100

Step 2 Xj = 0101
Xk = 0011
Xi = 0110

20 LXi jk Shift Xi Left jk Places

Shifts the 60-bit word in Xi left circular jk places. The shift moves the leftmost bits of Xi through the lower bits of Xi.

The 2-digit shift count jk many be an octal or decimal number and allows a complete circular shift of Xi.

21 AXi jk Shift Xi Right jk Places

Shifts the 60-bit word in Xi right jk places. The rightmost bits of Xi are discarded and the sign bit is extended. The 2-digit shift count jk may be an octal or decimal number.

22 LXi Bj Xk Left Shift Xk Nominally Bj Places to Xi

Shifts the 60-bit word in Xk the number of places specified by the low-order 6 bits of Bj and places the result in Xi.

If Bj is positive, Xk is shifted left circular.
If Bj is negative, Xk is shifted right (end off with sign extension).
When Bj is negative, the complement of the low-order 6 bits of Bj constitutes the shift count.

23 AXi Bj Xk Arithmetic Right Shift Xk Nominally Bj Places to Xi

Shifts the 60-bit word in Xk the number of places specified by the low-order 6 bits of Bj and places the result in Xi.

If Bj is positive, Xk is shifted right (end off with sign extension).
If Bj is negative, Xk is shifted left circular.
When Bj is negative, the complement of the low-order 6 bits of Bj constitutes the shift count.

24 NXi Bj Xk Normalize Xk in Xi and Bj

Normalizes the floating point quantity in Xk and places it in Xi. The number of left shifts required to normalize the quantity is placed in Bj durihg the operation. Normalizing a zero coefficient reduces the exponent by 48. If the size of the exponent is less than the number of leading zeros in the coefficient, underflow occurs during normalizing and the exponent and coefficient are both cleared.

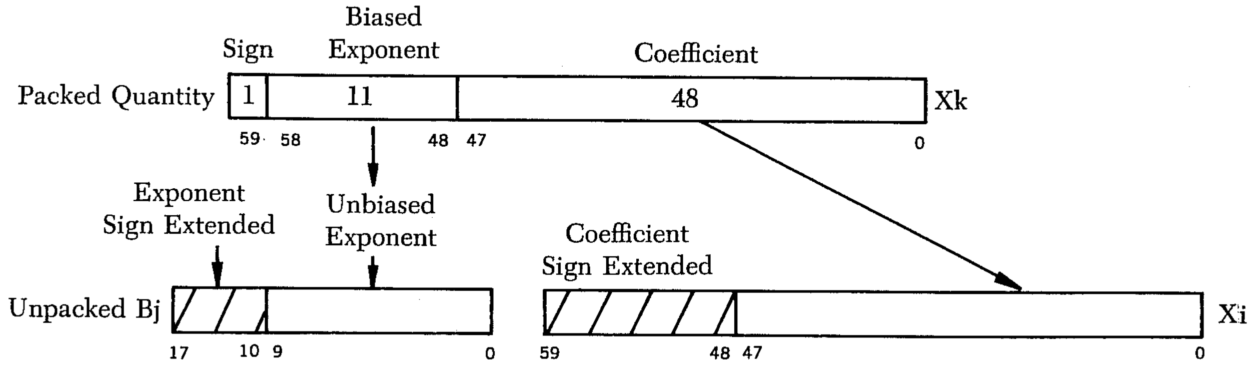
25 ZXi Bj Xk Round and Normalize Xk in Xi and Bj

Performs the same operation as NXi (24) except that the quantity in Xk is rounded before it is normalized. Normalizing a zero coefficient places the round bit in bit 47 and reduces the exponent by 48.

26 UXi Bj Xk Unpack Xk to Xi and Bj

Unpacks the floating point quantity in Xk and sends the 48-bit coefficient to Xi and the 11-bit exponent to Bj. The exponent bias is removed during unpack so that Bj is the true 1's complement representation of the exponent. Xk may be a unnormalized number.

The exponent and coefficient are sent to the low-order bits of the respective registers as shown in the following diagram.



27 PXi Bj Xk — Pack Xi from Xk and Bj

Packs a floating point number in Xi. The coefficient of the number is obtained from Xk and the exponent from Bj. A bias of 2^{10} (2000_8) is added to the exponent during the pack operation. The coefficient is not normalized.

Exponent and coefficient are obtained from the proper low-order bits of the respective register and packed as shown in the diagram for the unpack (26) instruction. Overflow is produced during pack when Bj is a positive number of more than 10 bits; the overflow exit is optional. Underflow is produced (no exit) when Bj is a negative number of more than 10 bits.

30 FXi Xj + Xk — Floating Sum of Xj and Xk to Xi

Forms the sum of the floating point quantities in Xj and Xk and packs the result in Xi. The packed result is the *upper half* of a double precision sum.

At the start both arguments are unpacked, and the coefficient of the argument with the smaller exponent is entered into the upper half of a 96-bit accumulator. The coefficient is shifted right by the difference of the exponents. The other coefficient is then added into the upper half of the accumulator. If overflow occurs, the sum is shifted right one place, and the exponent of the result is increased by one. The upper half of the accumulator holds the coefficient of the sum, which is not necessarily in normalized form. The exponent and upper coefficient are then repacked in Xi.

If both exponents are zero and no overflow occurs, the instruction effect an ordinary integer addition.

31 FXi Xj − Xk — Floating Difference of Xj and Xk to Xi

Forms the difference of the floating point quantities in Xj and Xk and packs the result in Xi. Alignment and overflow operations are similar to the floating sum (30) instruction, and the difference is not necesarily normalized. The packed result is the *upper half* of a double precision difference.

An ordinary integer subtraction is performed when the exponents are equal.

32 DXi Xj + Xk — Floating DP Sum of Xj and Xk to Xi

Forms the sum of two floating point numbers as in the floating sum (30) instruction, but packs the *lower half* of the double precision sum with an exponent 48 less than the upper sum.

33 DXi Xj − Xk — Floating DP Difference of Xj and Xk to Xi

Forms the difference of two floating point numbers as in the floating difference (31) instruction, but packs the *lower half* of the double precision difference with an exponent of 48 less than the upper difference.

34 RXi Xj + Xk — Round Floating Sum of Xj and Xk to Xi

Forms the round sum of the floating point quantities in Xj and Xk and packs the *upper sum* of the double precision result in Xi. The sum is formed in the same manner as the floating sum (30) instruction except that the operands are rounded before the addition, as explained below, to produce a round sum.

1. A round bit is attached at the right end of both operands if:

   a. both operands are normalized, or

   b. the operands have *unlike* signs.

2. For all other cases, a round bit is attached at the right end of the operand with the larger exponent.

35 RXi Xj − Xk — Round Floating Difference of Xj and Xk to Xi

Forms the round difference of the floating point quantities in Xj and Xk and packs the *upper difference* of the double precision result in Xi. The difference is formed in the same manner as the floating difference (31) instruction except that the operands are rounded before the subtraction, as explained below, to produce a round difference.

1. A round bit is attached at the right end of both operands if:

   a. both operands are normalized, or

   b. the operands have *like* signs.

2. For all other cases, a round bit is attached at the right end of the operand with the larger exponent.

36 IXi Xj + Xk Integer Sum of Xj and Xk to Xi

Forms a 60-bit one's complement sum of the quantities in Xj and Xk and stores the result in Xi. An overflow condition is ignored.

37 IXi Xj − Xk Integer Difference of Xj and Xk to Xi

Forms the 60-bit one's complement difference of the quantities in Xj (minuend) and Xk (subtrahend) and stores the result in Xi.

40 FXi Xj * Xk Floating Product of Xj and Xk to Xi

Multiplies the floating point quantities in Xj (multiplier) and Xk (multiplicand) and packs the *upper product* result in Xi.

The result is a normalized quantity only when both operands are normalized; the exponent is then the sum of the exponents plus 47 (or 48).

The result is unnormalized when either or both operands are unnormalized; the exponent is then the sum of the exponents plus 48.

41 RXi Xj * Xk Round Floating Product of Xj and Xk to Xi

Attaches a round bit to the floating point number in Xk (multiplicand), multiplies this number by the floating point number in Xj, and packs the *upper product* result in Xi. (No lower product is available.)

The result is a normalized quantity only when both operands are normalized, the exponent is then the sum of the exponents plus 47 (or 48).

The result is unnormalized when either or both operands are unnormalized; the exponent is then the sum of the exponents plus 48.

42 DXi Xj * Xk Floating DP Product of Xj and Xk to Xi

Multiplies the floating point quantities in Xj and Xk and packs the *lower product* in Xi. The result is not necessarily a normalized quantity.

43 MXi jk Form Mask in Xi, jk bits

Forms a mask in Xi. The 6-bit quantity jk defines the number of one's in the mask as counted from the highest order bit in Xi.

44 FXi Xj/Xk Floating Divide Xj by Xk to Xi

Divides the floating point quantities in Xj (dividend) and Xk (divisor) and packs the quotient in Xi.

The exponent of the result in a no-overflow case is the difference of Xj and Xk exponents minus 48.

A one-bit overflow is compensated for by shifting the coefficient right one place and increasing the exponent by one. The exponent is then the difference of Xj and Xk exponents minus 47.

The result is a normalized quantity when both Xj and Xk are normalized.

45 RXi Xj/Xk Round Floating Divide Xj by Xk to Xi

Divides the floating point quantity in Xj (dividend) by Xk (divisor) and packs the round quotient in Xi. A ⅓ round bit is added to the least significant bit of the dividend (Xj) before division starts.

The result exponent in a no-overflow case is the difference of Xj and Xk exponents minus 48.

A one-bit overflow is compensated for by shifting the coefficient right one place and increasing the exponent by one. The exponent is then the difference of Xj and Xk exponents minus 47.

The result is a normalized quantity when both Xj and Xk are normalized.

46 NO Pass

No operation.

47 CXi Xk Count the Number of 1's in Xk to Xi

Counts the number of one's in Xk and stores the count in Xi.

50	SAi	Aj + K	(or complement of K)	Set Ai
51	SAi	Bj + K	(or complement of K)	
52	SAi	Xj + K	(or complement of K)	
53	SAi	Xj + Bk		
54	SAi	Aj + Bk		
55	SAi	Aj − Bk		
56	SAi	Bj + Bk		
57	SAi	Bj − Bk		

These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result in Ai.

Operands are obtained from address (A), increment (B), and operand (X) registers as well as the K portion of the instruction. K is an 18-bit signed constant. As used in instructions 50, 51, and 52, if the sign of K is minus, ASCENT places the 18-bit one's complement of K in the K portion of the instruction word. Operands obtained from an X register are the truncated lower 18 bits of the 60-bit register.

An immediate memory reference to the address specified by the final contents of address register Ai is effected by the execution of a SAi (i = 1-7) instruction. The operand read from memory address specified by A1-A5 is sent to the corresponding operand register X1-X5. The operand from X6 or X7 is stored at the address specified by the corresponding A6 or A7.

60	SBi	Aj + K	(or complement of K)	Set Bi
61	SBi	Bj + K	(or complement of K)	
62	SBi	Xj + K	(or complement of K)	
63	SBi	Xj + Bk		
64	SBi	Aj + Bk		
65	SBi	Aj − Bk		
66	SBi	Bj + Bk		
67	SBi	Bj − Bk		

These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result in Bi.

Operands are obtained from address (A), increment (B), and operand (X) registers as well as the K portion of the instruction. K is an 18-bit signed constant. As used in instructions 60, 61, and 62, if the sign of K is minus, ASCENT places the 18-bit one's complement of K in the K portion of the instruction word. Operands obtained from an X register are the truncated lower 18 bits of the 60-bit register.

70	SXi	Aj + K	(or complement of K)	Set Xi
71	SXi	Bj + K	(or complement of K)	
72	SXi	Xj + K	(or complement of K)	
73	SXi	Xj + Bk		
74	SXi	Aj + Bk		
75	SXi	Aj − Bk		
76	SXi	Bj + Bk		
77	SXi	Bj − Bk		

These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result in Xi.

Operands are obtained from address (A), increment (B), and operand (X) registers as well as the K portion of the instruction. K is an 18-bit signed constant. As used in instructions 70, 71, and 72, if the sign of K is minus, ASCENT places the 18-bit one's complement of K in the K portion of the instruction word. Operands obtained from an X register are the truncated lower 18 bits of the 60-bit register.

Operands obtained from an Xj register are the truncated lower 18 bits of the 60-bit register. Conversely, an 18-bit result placed in Xi carries the sign bit extended to the remaining bits of the 60-bit register.

5. PSEUDO OPERATION CODES

Pseudo operations provide the means for directing the assembler to carry out certain functions. The instruction format is the same as the basic format given in 3.1. As used here, LOC indicates that the particular operation may have a symbolic identifier in the location field. Where none is shown, these columns are ignored by the assembler. Any FORTRAN statement may also be used as additional pseudo operation codes. The CALL statement in particular is used to reference subroutines defined by SUBROUTINE cards. Normal mixing rules apply. Following are ASCENT pseudo operations and meanings.

ASCENT SYMBOL

Defines the beginning of a program and its name, SYMBOL. This must be the first instruction of an ASCENT routine.

END

Terminate the assembly process for this program or subprogram. When punched into a card, END must be the only entry in the card and must start in column 11.

ASPER SYMBOL

Defines the end of the current ASCENT routine and/or the beginning of a peripheral processor routine. SYMBOL is the name of the ASPER routine.

SUBROUTINE SYMBOL (LIST)

Defines the beginning of a subroutine, its name, SYMBOL, and the formal parameters given by (LIST). See Section 9.2 for more detail.

BSSD N, L, NAME, R

Defines on logical disk unit N the file identified by NAME which has L number of 60-bit words in its longest record. R specifies the maximum number of logical records into which the file may be segmented. The parameters N, L, and R must be numbers, where $N \leqq 16_{10}$, $L \leqq 2^{17}$, and $R \leqq 4000_{10}$. NAME must be unique within the routine.

LOC BSS OPERAND

Reserve the number of 60-bit locations specified by OPERAND beginning at LOC. The contents of the locations reserved are not set to a particular condition. The LOC symbol is equated to the address of the first word of the area. Any symbolic constant appearing in the address field must be previously defined.

LOC BSSZ OPERAND

Same as BSS except the contents of the locations reserved are set to zero in the object code.

LOC EQU OPERAND

The symbol in the LOC field is assigned the value of the address field. Any symbol appearing in the address field must be previously defined.

LOC DPC $*C_1 C_2 - - - C_n*$

Convert the characters enclosed by the asterisks to display code, ten characters per word beginning at LOC. Incomplete words are padded out with DPC blanks. The LOC symbol is equated to the address of the first word of the area.

LOC DPC $nnC_1 C_2 - - - C_{nn}$

Convert the nn characters, $C_1, C_2, - - - C_{nn}$, to display code, ten characters per word beginning at LOC. The number of characters, nn, must be a two-digit decimal number. Incomplete words are padded out with DPC blanks. The LOC symbol is equated to the address of the first word of the area.

LOC BCD $*C_1 C_2 - - - C_n*$

Convert the characters enclosed by the asterisks to BCD code, ten characters per word beginning at LOC. Incomplete words are padded out with BCD blanks. The LOC symbol is equated to the address of the first word of the area.

LOC BCD $nnC_1 C_2 - - - C_{nn}$

Converts the nn characters $C_1, C_2, - - - C_{nn}$, to BCD code, ten characters per word beginning at LOC. The number of characters, nn, must be a two-digit decimal number. Incomplete words are padded out with BCD blanks. The LOC symbol is equated to the address of the first word of the area.

LOC CON V_0, - - - , V_n

Convert each V_i term to a 60 or 120-bit constant. If more than one is defined per a CON pseudo code, each V_i must be separated by commas. The V_i may be:

a. ± octal integer

b. ± decimal integer

c. symbol

d. symbol ± integer

e. symbol − symbol

f. ± single precision floating point number

g. ± double precision floating point number

h. ± complex number

The LOC symbol is equated to the address assigned to the V_0 term. Remarks are not permitted on a CON operation.

LIST P

Controls the listing of the side-by-side so that sections of coding may be omitted from the listing. At the beginning of each ASCENT program the assembler assumes the list case of $P = 0$, unless otherwise specified by the LIST pseudo opcode.

If $P = 0$, list the side-by-side that follows.

If $P \neq 0$, suppress the side-by-side listing.

SPACE nn

Space nn lines on the listing. The integer, nn, is evaluated $\leq 63._{10}$.

EJECT

Eject the listing to the top of the next page.

6. SYSTEM MACROS

6.1 GENERAL INFORMATION

System macro instructions provide communication links between a program in central memory and the system peripheral processors. While most of these macros direct the operating system to perform input/output operations, others request equipment assignment, check the status of external operations, produce program overlays, and utilize system peripheral processors in conjunction with the central processor program.

The communication link provided by the system macros allows a two-way information transfer. The central memory program not only sends the peripheral processor request information but also reserves a location in central memory in which the system peripheral processor enters the status of the requested operation, reporting its success back to the central memory program. Each system macro must have a status response word which is set by the operating system in performing the function of the individual macro request.

Further, to facilitate multiprocessing, each system macro provides a buffered and non-buffered mode. In the buffered mode, the macro used without the appended "W," it is up to the CP program to determine when an operation is completed or to execute another macro to wait for completion at a later time. In the non-buffered mode, with the "W" appended to the macro code, the macro turns central processor control back to the operating system for assignment to another CP program. Control will not be returned to the next object code step of the CP program in question until the macro request is completed or aborted. Both modes return full status information to the CP program relative to the success of the peripheral processor in carrying out the request.

ASCENT generates a sequence of code from the system macro which initiates the requested system function. The Return Jump generated is followed by the parameters in line with the object code. The parameters may be any of the following forms:

Constant (integer or octal) – specifies the parameter itself, such as a unit number, the record length, or the conversion mode.

Symbolic Location – specifies the location which contains the parameters.

Literal – specifies the parameter itself. ASCENT places in a location at the end of the object code the parameter specified by the literal.

NAME – certain macros require a file or program name. The NAME is converted to Display Code and becomes the parameter.

An example of a macro follows:

RDCW 1, (S), (BA), (BA + 8), 8, 2

Assume: S = 1001

BA = 2500

Then: ASCENT generates a location for each literal specified by the macro. If the end of the object code is location 4200, then:

4201 = 0 . . . 01001

4202 = 0 . . . 02500

4203 = 0 . . . 02510

The communication link in the object code to the Central Resident program becomes:

	59 … 30	29 … 0	
P	RJ SUB		
P + 1	EQ B0 B0 P + 5	OP N	
P + 2	00 . . . 01	40 . . . 04201	Parameters
P + 3	40 . . . 4202	40 . . . 04203	
P + 4	00 . . . 0010	00 . . . 00002	
P + 5	OBJECT CODE		

SUB – A routine that forms the parameters in locations 000002 through (N + 1) for communication with the operating system. Location 000001 contains the operation code of the macro requested.

OP – Operation code assigned by the system to each macro

N – Number of parameters

The following is an explanation of certain letters, terms, and phrases used in connection with macros.

A — Symbolic address in central memory which contains the address of the first word of the requested block assigned by the system or which contains the address, as specified by the programmer, of the first word of the block in memory to be released to the system.

BA — Symbolic address in central memory which contains the beginning address of the buffer area.

C — Conversion mode

Card operations:

C = blank or 0 — no conversion (binary image)

1 — Hollerith to display code for read; display code to Hollerith for punch

2 — Hollerith to BCD for read; BCD to Hollerith for punch

Magnetic tape:

C = blank or 0 — no conversion

1 — BCD to display code

2 — display code to BCD

Printer:

C = blank or 0 — no conversion

2 — display code to BCD

EA — Symbolic address in central memory which contains the ending address + 1 of the buffer area.

K — Number (or CM symbolic address of number) of logical tape records.

L — Number (or CM symbolic address of number) of 60-bit words in the longest record in the file identified by NAME.

N — Equipment logical number (or CM symbolic address of number), i.e., 1,2, . . . M for M total units of equipment type in the system.

NAME — Symbolic name uniquely identifying the disk logical file being referenced.

NW — Total number (or CM symbolic address of number) of central memory words requested or released.

P — Logical record number (or CM symbolic address of number) in disk file to start read or write.

R — Maximum number (a CM symbolic address of number) of logical records into which the disk file may be segmented.

RL — Card operations: total number (or CM symbolic address of number) of leftmost 5 columns (binary image) or 10-character fields (coded mode) of the card. For BCD or DPC conversion mode, each central memory word contains ten 6-bit characters. For binary image, each central memory word contains 5 columns.

Console operations: total number (or CM symbolic address of number) of characters in the message to be transmitted.

Magnetic tape: number (or CM symbolic address of number) of 60-bit words per tape record.

Printer: number (or CM symbolic address of number) of 10-character words per line to print.

S — Symbolic address in central memory which contains the address for the STATUS RESPONSE WORD from the PP I/O routine. The PP I/O routine handling the request requires that a location in central memory be reserved and identified for each macro request.

The PP I/O routine reports to this location the status of the requested operation.

SYMBOL Program overlay: name of overlay region to be loaded.

System action: name of PP program defined by ASPER pseudo operation.

Wait check: name of transfer location if abort is indicated by the status response word.

T Display character size:

T = blank or 0 – 64 char./line

1 – 32 char./line

2 – 16 char./line

3 – plot mode

TAG Identification number $\leq$ 18 bits (or CM symbolic address of number) of message to be displayed.

W A W appended to the opcode of a macro indicates a "wait for reply." If the W is not used (buffered mode), the CP program may continue processing while the requested I/O operation is being performed. However, the program must do its own checking on the progress of the request by means of the WAI (Wait Check) macro. If the request is in process, the status response word is positive and nonzero; if the request is completed, the word is zero; if the request is aborted, the word is negative.

When the W is appended to the macro (non-buffered mode) and the requested operation can be performed, control is turned over to the operating system and the CP program delays until the status response word is zero (completed) or negative (aborted), at which time control is given back to the program.

In both modes if the requested operation is successful, the next in-line instruction is executed.

6.2 MACRO FORMATS

6.2.1 MAGNETIC TAPE OPERATIONS

OPCODE	ADDRESS FIELD	REMARKS	
RQTW	N, S	Request tape assignment from system.	Wait if W used.
DRTW	N, S	Release tape back to system.	Wait if W used.
SFFW	N, S	Search file mark forward.	Wait if W used.
SFBW	N, S	Search file mark backward.	Wait if W used.
WFMW	N, S	Write file mark.	Wait if W used.
RWLW	N, S	Rewind tape to load point.	Wait if W used.
RWUW	N, S	Rewind tape for unload.	Wait if W used.
FSPW	N, S, K	Forespace	Wait if W used.
BSPW	N, S, K	Backspace	Wait if W used.
RFCW	N, S, BA, EA, RL, C	Read tape forward coded mode.	Wait if W used.
RFBW	N, S, BA, EA, RL, C	Read tape forward binary mode.	Wait if W used.
WRCW	N, S, BA, EA, RL, C	Write tape coded mode.	Wait if W used.
WRBW	N, S, BA, EA, RL, C	Write tape binary mode.	Wait if W used.

N = Magnetic tape logical unit number; 1, 2, . . . M for M tape units in the system.

S = Location containing the central memory address for status response code from System PP I/O routine.

K = Number of logical tape records.

BA = Location containing the beginning address of buffer area in central memory.

EA = Location containing the ending address + 1 of buffer area in central memory.

RL = Number of 60-bit words per tape record.

C = Conversion mode.

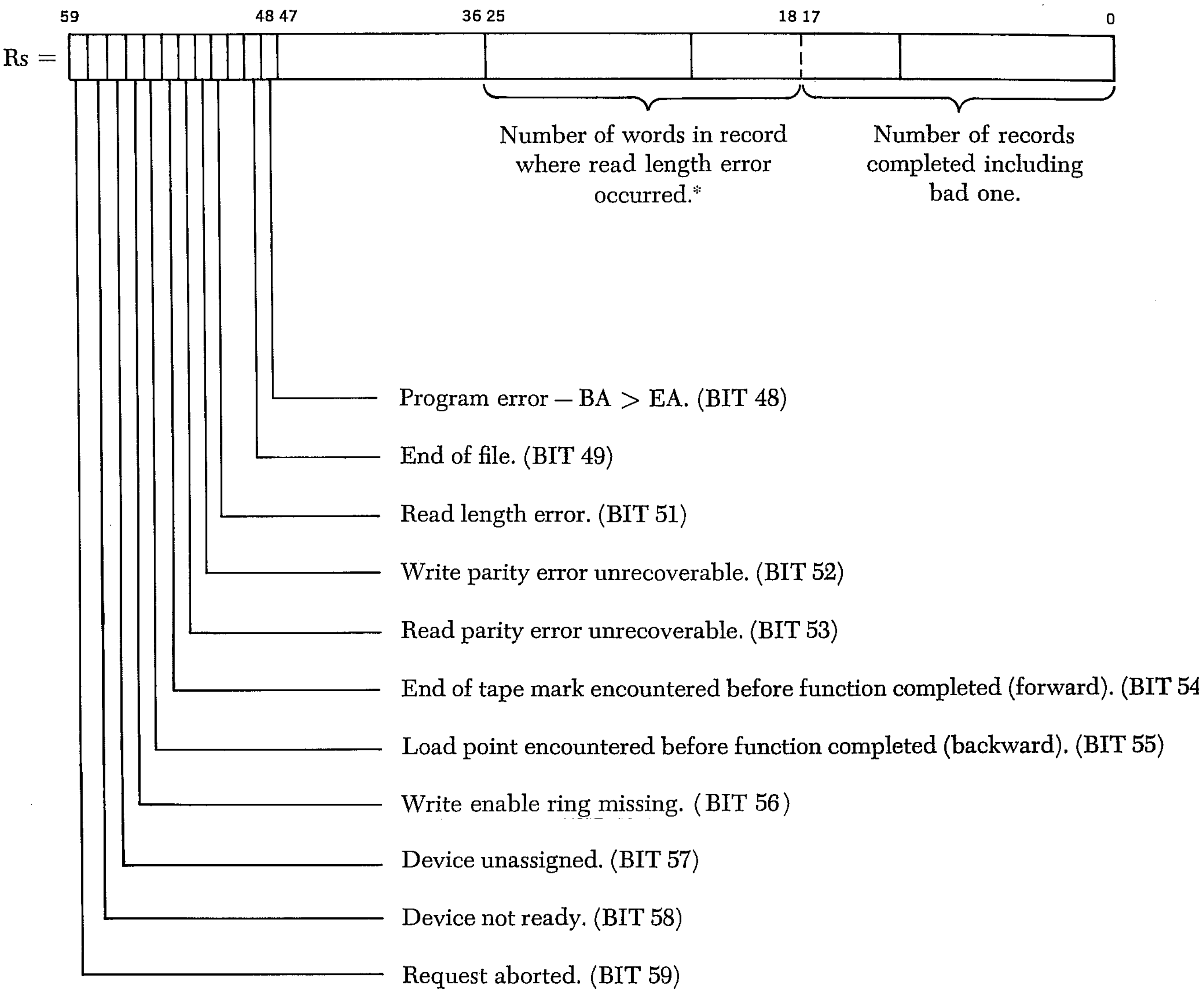
Blank or 0 — No conversion.

1 — BCD to Display Code.

2 — Display Code to BCD.

STATUS RESPONSE CODES — positioned as per address S.

Rs = 0	Request completed with no trouble.
Rs = 1	Request in process.
Rs < 0	Request aborted. Reason give in bits 58-48.


59
48 47
36 25
18 17
0
Rs =
Number of words in record where read length error occurred.*
Number of records completed including bad one.
Program error — BA > EA. (BIT 48)
End of file. (BIT 49)
Read length error. (BIT 51)
Write parity error unrecoverable. (BIT 52)
Read parity error unrecoverable. (BIT 53)
End of tape mark encountered before function completed (forward). (BIT 54
Load point encountered before function completed (backward). (BIT 55)
Write enable ring missing. (BIT 56)
Device unassigned. (BIT 57)
Device not ready. (BIT 58)
Request aborted. (BIT 59)

where: 1 implies the condition exists.
0 implies the condition does not exist.

*Refers to peripheral processor words. Also, an attempt to write when the file protect ring is out will cause bit 58 to be set.

6.2.2 DISK TRANSFERS

Provision is made in the operating system for the programmer to read and write scratch data to and from disk storage units. Data are usually broken up into related blocks called *files*. The files, in turn, are segmented into the blocks of data that are transmitted at one time. These are called *logical records*. For most efficient utilization of disk storage, logical records contain a minimum of 512 central memory words. A file is defined by the ASCENT pseudo operation, BSSD, which specified the number of 60-bit words in the longest record, the maximum number of logical records into which the file is to be segmented, and the symbolic name by which to identify the file. The actual data transmission is accomplished through the use of the following macro operators.

OPCODE	ADDRESS FIELD	REMARKS	
RDHW	N, S, BA, EA, NAME, P	Read record and hold data on disk.	Wait if W used.
RDRW	N, S, BA, EA, NAME, P	Read record and release data on disk.	Wait if W used.
WRDW	N, S, BA, EA, NAME, P	Write record on disk.	Wait if W used.

N = Disk logical unit number; 1, 2, . . . M for M disk units in the system.

S = Location containing the central memory address for status response code from System PP I/O routine.

BA = Location containing the beginning address of buffer area in central memory.

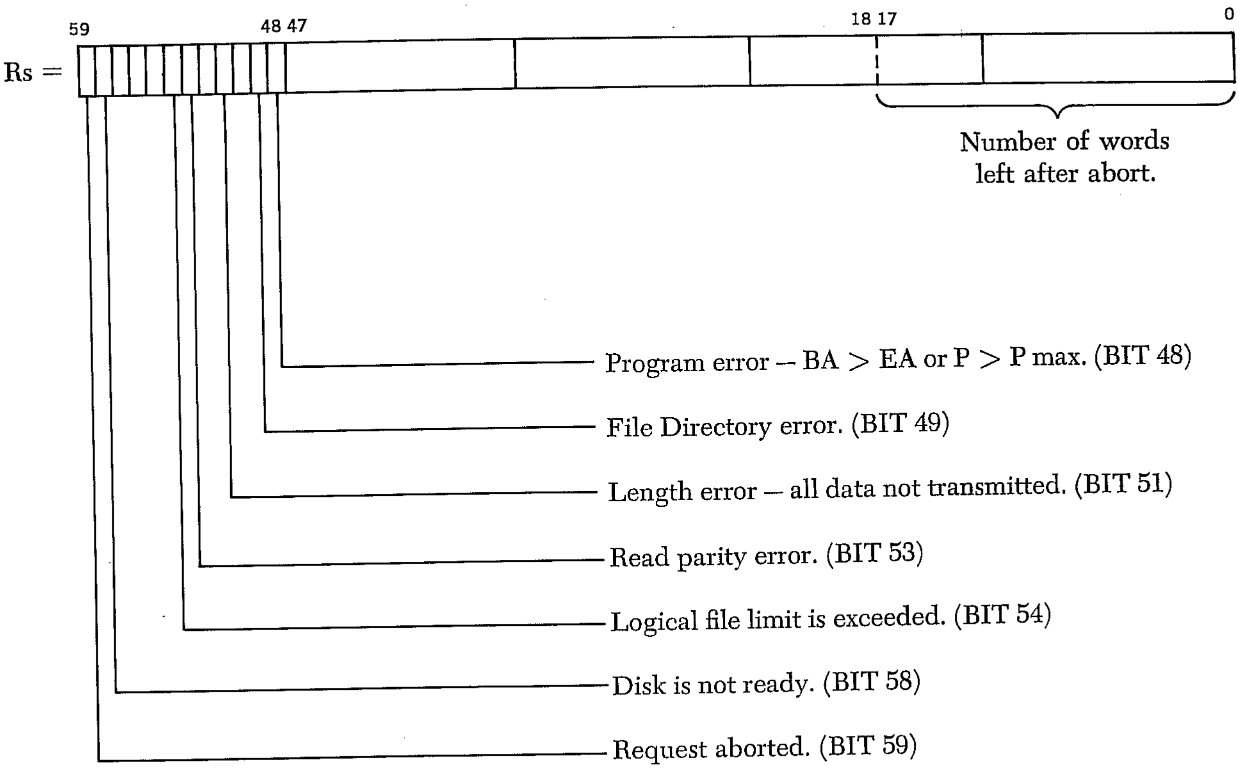
EA = Location containing the ending address + 1 of buffer area in central memory.

NAME = Symbolic name to identify disk logical file to be referenced.

P = Logical record number used to identify record read from disk or written onto disk.

STATUS RESPONSE CODES—positioned as per address S.

Rs = 0	Request is completed with no trouble.
Rs = 1	Request is in process.
Rs < 0	Request aborted. Reason given in bits 58-48.



Number of words left after abort.

Program error – BA > EA or P > P max. (BIT 48)

File Directory error. (BIT 49)

Length error – all data not transmitted. (BIT 51)

Read parity error. (BIT 53)

Logical file limit is exceeded. (BIT 54)

Disk is not ready. (BIT 58)

Request aborted. (BIT 59)

where: 1 implies the condition exists.
0 implies the condition does not exist.

6.2.3 PRINTER OPERATIONS

OPCODE	ADDRESS FIELD	REMARKS	
SSPW	N, S	Single space printer.	Wait if W is used.
DSPW	N, S	Double space printer.	Wait if W is used.
FC7W	N, S	Select Format Channel 7.	Wait if W is used.
FC8W	N, S	Select Format Channel 8.	Wait if W is used.
MC1W	N, S	Select Monitor Channel 1.	Wait if W is used.
MC2W	N, S	Select Monitor Channel 2.	Wait if W is used.
MC3W	N, S	Select Monitor Channel 3.	Wait if W is used.
MC4W	N, S	Select Monitor Channel 4.	Wait if W is used.
MC5W	N, S	Select Monitor Channel 5.	Wait if W is used.
MC6W	N, S	Select Monitor Channel 6.	Wait if W is used.
CMCW	N, S	Clear Monitor Channels 1 - 6.	Wait if W is used.
SPAW	N, S	Suppress space after next print.	Wait if W is used.
PRNW	N, S, BA, EA, RL, C	Print single line or multiple lines.*	Wait if W is used.

*If SPA is given preceding a multiple line print, it applies only to the first line.

N = Printer logical unit number; 1, 2, . . . M for M printers in the system.

S = Location containing the central memory address for status response code from System PP I/O routine.

BA = Location containing the beginning address of buffer area in central memory.

EA = Location containing the ending address + 1 of buffer area in central memory.

RL = Number of 10 character words per line to print.

C = Conversion mode.

Blank or 0 – No conversion.

2 – Display Code to BCD.

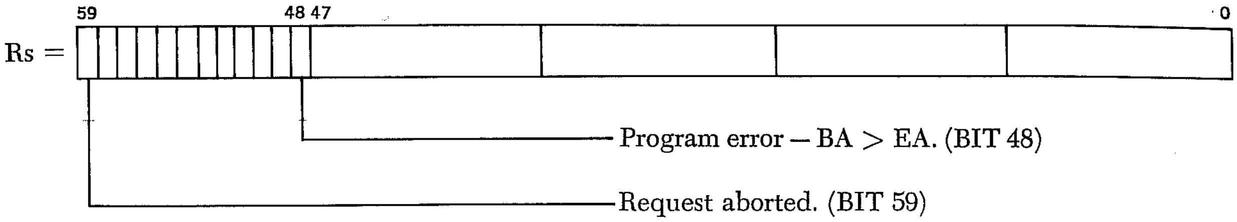
Printer character codes are given in Table 4 of the Appendix.

STATUS RESPONSE CODES – positioned as per address S.

Rs = 0 Request is completed with no trouble.

Rs = 1 Request is in process.

Rs < 0 Request aborted. Reason given in bits 58-48.



where: 1 implies the condition exists.

0 implies the condition does not exist.

6.2.4 CARD OPERATIONS

OPCODE	ADDRESS FIELD	REMARKS	
PCHW	N, S, BA, EA, RL, C	Punch cards.	Wait if W is used.
RDCW	N, S, BA, EA, RL, C	Read cards.	Wait if W is used.

N = Card reader or punch logical unit number; 1,2, . . . M for M readers or punches in the system.

S = Location containing the central memory address for status response code from System PP I/O routine.

BA = Location containing the beginning address of buffer area in central memory.

EA = Location containing the ending address + 1 of buffer area in central memory.

RL = Number of leftmost 10-character fields or 5 columns of the card.

C = Conversion mode.

Blank or 0 – No conversion; i.e., binary image input/output.

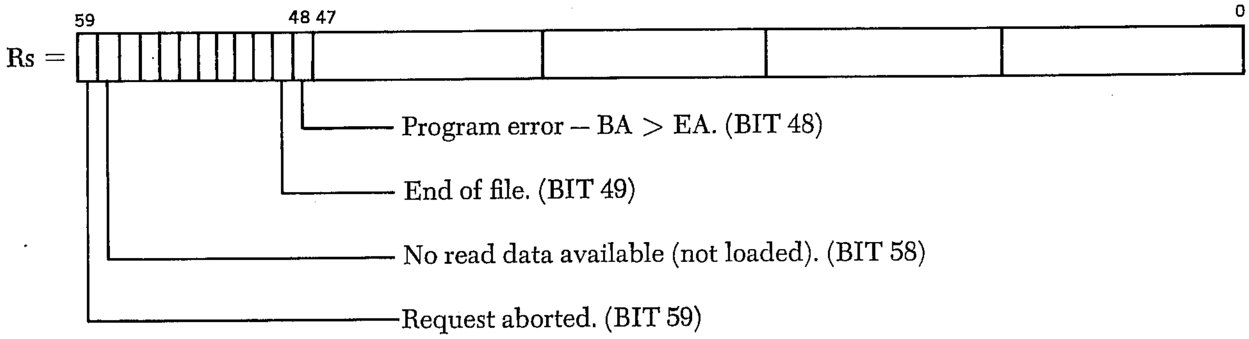
1 – Hollerith to Display Code for read; Display Code to Hollerith for punch.

2 – Hollerith to BCD for read; BCD to Hollerith for punch.

Display character codes are given in Table 4 of the Appendix.

STATUS RESPONSE CODES – positioned as per address S.

Rs = 0	Request is completed with no trouble.
Rs = 1	Request is in process.
Rs < 0	Request aborted. Reason given in bits 58-48.


59
48 47
0
Rs =
Program error – BA > EA. (BIT 48)
End of file. (BIT 49)
No read data available (not loaded). (BIT 58)
Request aborted. (BIT 59)

where: 1 implies the condition exists.

0 implies the condition does not exist.

6.2.5 CONSOLE OPERATIONS

Request procedures are provided for ASCENT routines to display messages on the primary console right scope or either of the scopes on other consoles. The system provides a timing service for removal of displays after a certain exposure. However, the request procedure gives an option to override the system time limit on display. In this mode, it is assumed that the ASCENT routine will request a removal of the display as a result of console acknowledgment or internal decision.

OPCODE	ADDRESS FIELD	REMARKS
DSRW	N, S, BA, EA, RL, TAG, T	Display on Right Scope for system time limit. Wait if W is used.
DSLW	N, S, BA, EA, RL, TAG, T	Display on Left Scope for system time limit. Wait if W is used.
DHRW	N, S, BA, EA, RL, TAG, T	Display on Right Scope and hold indefinitely. Wait if W is used.
DHLW	N, S, BA, EA, RL, TAG, T	Display on Left Scope and hold indefinitely. Wait if W is used.
RDPW	N, S, TAG	Remove display. Wait if W is used.
RTYW	N, S, BA, EA, RL, TAG	Read console typewriter. Wait if W is used.

N = Console logical unit number; 1, 2, . . . M for M consoles in the system.

S = Location containing the central memory address for status response code from System PP I/O routine.

BA = Location containing the beginning address of buffer area in central memory.

EA = Location containing the ending address + 1 of buffer area in central memory.

RL = Total number of characters in the message to be transmitted.

TAG = Identification number $\leq$ 18 bits for display message.

T = Display character size.

Blank or 0 – 64 characters/line.

1 – 32 characters/line.

2 – 16 characters/line.

3 – plot mode.

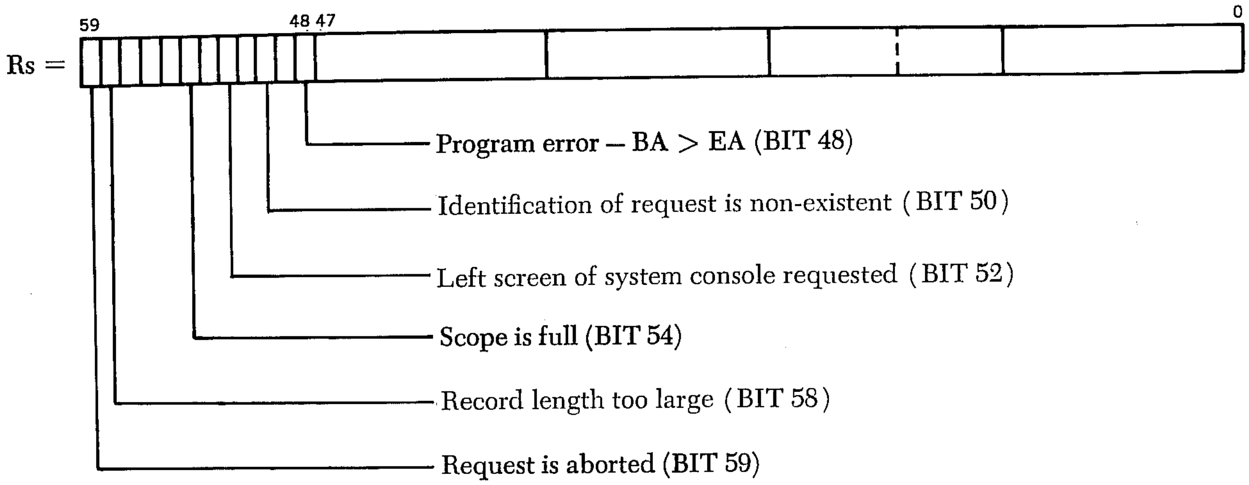
Display character codes are given in Table 4 of the Appendix.

STATUS RESPONSE CODES – positioned as per address S.

Rs = 0 Request is completed with no trouble.

Rs = 1 Request is in process.

Rs < 0 Request aborted. Reason given in bits 58-48.



Rs =
59
48 47
0
Program error – BA > EA (BIT 48)
Identification of request is non-existent (BIT 50)
Left screen of system console requested (BIT 52)
Scope is full (BIT 54)
Record length too large (BIT 58)
Request is aborted (BIT 59)

where: 1 implies the condition exists.
0 implies the condition does not exist.

6.2.6 SYSTEM ACTION

OPCODE	ADDRESS FIELD	REMARKS
TPPW	N, S, SYMBOL	Transfer program SYMBOL from CM to PP memory and begin execution with first ASPER instruction. Wait if W is used.
RQMW	NW, S, A	Request memory. Wait if W is used.
DRMW	NW, S, A	Release memory. Wait if W is used.
RQDW	N, S, L, NAME, R	Request disk space. Wait if W is used.
DRDW	N, S, NAME	Release disk space. Wait if W is used.

N = Logical number of PP or disk unit.

S = Location containing the central memory address for status response code from System PP I/O routine.

R = Maximum number of logical records into which the file may be segmented.

NW = Total number of words.

L = Number of 60-bit words in longest record.

A = Location containing the central memory address of the first word of block assigned by the system or released by the programmer.

NAME = Symbolic name uniquely identifying the disk logical file being referenced.

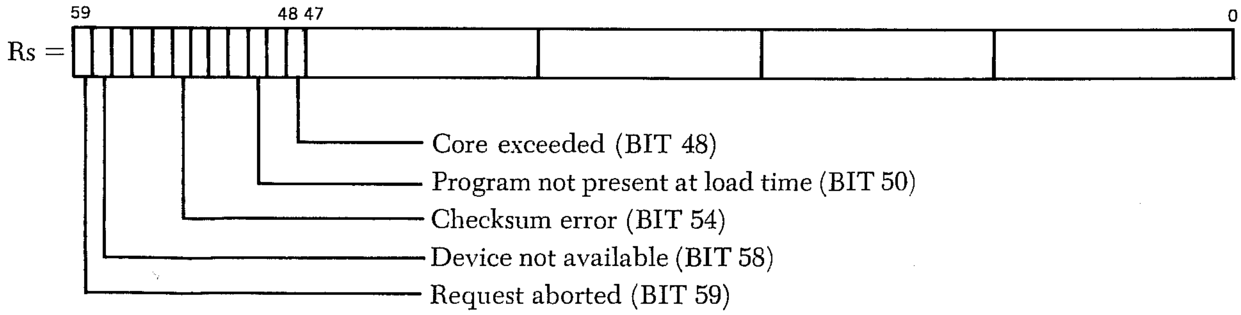
SYMBOL = Name of PP program defined by ASPER pseudo operation.

STATUS RESPONSE CODES – positioned as per address S.

Rs = 0 Request completed with no trouble.

Rs = 1 Request in process.

Rs < 0 Request aborted. Reason given in bits 58-48.



where: 1 implies the condition exists.
0 implies the condition does not exist.

6.2.7 LOAD SEGMENT

During initial loading, segmentation control cards are matched against subroutines present to assure overlay capability when called. Therefore, during the load process control is taken from the CP program and is returned when the load is successful. No status response is required since success is necessary for the CP program to regain control.

OPCODE	ADDRESS FIELD	REMARKS
LOAD	SYMBOL	Load segment SYMBOL
LOAD	*SYMBOL*	Load segment SYMBOL and transfer control to indicated routine.

6.2.8 WAIT CHECK

When a buffered operation is initiated, a Wait Check macro may be used to check status and exit if an operation abort occurred on the request. Also, the macro has provision for turning control over to the system if the request is not completed.

OPCODE	ADDRESS FIELD	REMARKS
WAIW	S,SYMBOL	Check status of S. Exit to SYMBOL if abort. Wait for reply if not ready and W is used.

S = Location containing the central memory address for status response code from System PP I/O routine.

SYMBOL = Transfer location if an abort is indicated by the status response code.

6.3 PROGRAMMER DEFINED MACROS

The macro instructions provided by the system may be expanded through a feature of ASCENT that permits the programmer to define new macros at assembly time.

Programmer macros must be defined prior to the executable code in the program or subroutine. As the programmer macro is encountered, it is stored in memory, which serves as a skeleton, ready to be called and inserted into the body of the program. A programmer macro is local to the program or subroutine in which it is defined.

A programmer macro may be defined by using the pseudo operation code MACRO in the operation code field. The macro must be assigned a unique name followed by a list of formal parameters.

```
LOC      OP        FIELD                                   REMARKS
       MACRO       SYMBOL, LIST                          .MACRO DEFINITION
```

where: MACRO is the pseudo operation code,

SYMBOL is the name of the macro that is used to call the macro.

LIST is a sequence of formal parameters which specify the items that may be substituted each time the macro is called. Each parameter of the LIST may be a symbol, constant, or a register. The limit on the number of parameters in the LIST is undefined, but all parameters must appear on one card.

The programmer macro code follows the definition MACRO card. The programmer macro code does not differ from coding in other parts of the program, except it is restricted to ASCENT code. For example, the programmer macro code may *not* contain FORTRAN statements, ASPER programs, or may not call a programmer-defined macro.

The end of the programmer macro is denoted by the ENDM pseudo operation code appearing in the operation field.

Programmer-defined macros may be used in the body of the program or subroutine in which they are defined. Calls to the macros are made by entering the name of the macro in the operation code field, and as actual parameters (a list of operands and registers). The code of the macro is inserted in line with the actual parameters substituted for the formal parameters.

```
  LOC       OP       FIELD                                  REMARKS
LOCATE    SYMBOL     LIST                          .CALL OF PROGRAMMER MACRO
```

where: LOCATE a symbol in the location field of a call to a programmer macro is optional. If a location symbol appears, the first instruction of the macro is forced to the upper portion of a word with the symbol assigned the address of that instruction.

SYMBOL the name of the programmer-defined macro.

LIST a sequence of symbols, constants, and/or registers which serve as the actual parameters that are substituted in the skeleton code as defined by the formal parameters. The parameters in the LIST must be in the same order as the formal parameters for proper substitution.

Rules:

1. The definition of the programmer macro must precede the first executable instruction in the program or subroutine in which it is defined.

2. The name of the macro must not be identical to a machine mnemonic code, pseudo code, a system macro, or any other programmer-defined macro in the same routine. Programmer-defined macros are local only to the program or subroutine in which the definition appears. Therefore, the names of programmer-defined macros may appear in other subroutines.

3. The order of actual and formal parameters must be the same.

4. Neither FORTRAN statements nor ASPER subroutines may be used within the definition of a macro.

5. A programmer-defined macro may not call a programmer-defined macro.

6. Location symbols which appear within the body of the programmer-defined macro must appear as formal parameters in the definition, and each call must specify a unique location symbol as the actual parameter.

Therefore, programmer-defined macros define a sequence of code with formal parameters which serve as a skeleton of instructions; as each call is made to the macro, the code is inserted into the main body of the program, with the actual parameters substituted for the formal parameters. ASCENT assembles the macros in line with the program as if the macro code had been a part of the original code.

EXAMPLE 1

Suppose a programmer needs to transfer the contents of one core location to another several times within a program and the relative inefficiency of the coding sequence is not important.

```
        SA1        A                  . load first value into X6
        BX6        X1                 . move to storage register
        SA6        B                  . store in B the value in X6
```

Then he can define the macro, TRANS.

```
        MACRO      TRANS, A, B        . define MACRO, TRANS
        SA1        A                  . load first value
        BX6        X1                 . move to storage register
        SA6        B                  . store to second word
        ENDM                          . end of MACRO definition
```

Each time he needs to make the transfer, he can write a macro call for TRANS.

```
        TRANS      X, Y               . transfer x to y
```

where: x and y are the appropriate addreses.

The coding of the macro would be moved in line with parameter substitutions as a replacement for the call. The in-line code would be:

```
        SA1        X                  . load first value
        BX6        X1                 . move to storage register
        SA6        Y                  . store to second word
```

EXAMPLE 2

Suppose the need is the same as Example 1 except register conflicts exist from time to time with X1 and X6. In this case the macro can be defined as:

```
        MACRO      TRANS, TAG, A, B, A1, X1, A6, X6
        SA1        A
TAG     BX6        X1
        SA6        B
        ENDM
```

If at the time the transfer is needed, registers A2, X2, A7, and X7 are free, the call can be written as:

```
          CALL      TRANS          LOC, X, Y, A2, X2, A7, X7
```

The resulting object code would be:

```
                    SA2            X
          LOC       BX7            X2
                    SA7            Y
```

7. MACRO INSTRUCTIONS

7.1 DESCRIPTION

Backspace

BSP N, S, K

Backspaces K number of records on logical tape unit N.

Clear Monitor Channels 1-6

CMC N, S

Deselects monitor channels 1-6 on line printer N. This macro must be used before selecting another channel.

Display on Left Scope and Hold Indefinitely

DHL N, S, BA, EA, RL, TAG, T

Displays a message on the left scope of the console and holds the display indefinitely or until an RDP request is received. When displayed the message is accompanied by the 18-bit identifier, TAG. BA and EA contain the locations for the beginning and ending addresses of the buffer area storing the message to be displayed. Each CM word contains 10 consecutive display-coded characters of the message ordered from left to right in the word. The display character size is determined by T. RL specifies the number of characters to be displayed on each line on the scope and is limited by the character size chosen. The logical console number, N, indicates which console is to be used. See Example 1.

Display on Right Scope and Hold Indefinitely

DHR N, S, BA, EA, RL, TAG, T

Displays a message on the right scope of the console and holds the display indefinitely or until an RDP request is received. See macro DHL for further explanation of parameters.

Release Disk Space Back to System

DRD N, S, NAME

Releases the file indentified by NAME on the logical disk unit N.

Release Memory

DRM NW, S, A

Releases from the block of central memory words which the PP has reserved the total number of words specified by NW beginning with the CM address given in A.

Release Tape Back to System

DRT N, S

Releases the logical tape unit specified by N for general system usage.

Display on Left Scope for System Time Limit

DSL N, S, BA, EA, RL, TAG, T

Displays a message on the left scope of the console for the length of time set by the system. See macro DHL for further explanation of parameters.

Double Space Printer

DSP N, S

Advances logical printer N two lines.

Display on Right Scope for System Time Limit

DSR N, S, BA, EA, RL, TAG, T

Displays a message on the right scope of the console for the length of time set by the system. See macro DHL for further explanation of parameters.

Select Format Channel 7

FC7 N, S

Selects format channel 7 on logical printer unit N. This format channel advances the paper to a selected line.

Select Format Channel 8

FC8 N, S

Selects format channel 8 on logical printer unit N. This format channel ejects the page to the top of the form.

Forespace

FSP N, S, K

Spaces forward K number of records on logical tape unit N.

Select Monitor Channel 1

MC1 N, S

Selects monitor channel 1 on logical printer unit N. The monitor channels contain predesigned line-space formats.

Select Monitor Channel 2

MC2 N, S

Select monitor channel 2 on logical printer unit N.

Select Monitor Channel 3

MC3 N, S

Select monitor channel 3 on logical printer unit N.

Select Monitor Channel 4

MC4 N, S

Select monitor channel 4 on logical printer unit N.

Select Monitor Channel 5

MC5 N, S

Select monitor channel 5 on logical printer unit N.

Select Monitor Channel 6

MC6 N, S

Select monitor channel 6 on logical printer unit N.

Punch Cards

PCH N, S, BA, EA, RL, C

Punches cards on logical unit N for the number of leftmost 5 columns (binary output, no conversion) or 10-character fields (coded mode) as given by RL. The conversion mode is specified by C. The card images are read from central memory beginning at the address contained in location BA and ending at the address contained in location EA. See Example 2.

Print Single Line or Multiple Lines

PRN N, S, BA, EA, RL, C

Prints on logical unit N the number of 10-character words per line as given by RL in the conversion mode specified by C. RL may specify up to 12 or 14* words per line. The print image is stored in central memory beginning at the address contained in location BA and ending at the address contained in location EA. See Example 2.

Read Card

RDC N, S, BA, EA, RL, C

Reads cards on logical unit N for the number of leftmost 5 columns (binary input, no conversion) or 10-character fields (coded mode) as given by RL. The conversion mode is specified by C. The cards are read into central memory beginning at the address contained in location BA and ending at the address contained in location EA. See Example 2.

*For the 120 character/line 1612 printer and the 136 character/line 501 printer, respectively.

Read Record and Hold Data on Disk

RDH N, S, BA, EA, NAME, P

Reads into the buffer area in central memory the logical record specified by P of the file identified by NAME onto logical disk N. The words are read, without code translation, into the buffer area beginning at the address contained in location BA and ending at the address contained in location EA. The data are held on disk for subsequent re-use.

Remove Display

RDP N, S, TAG

Erases from the scope at console N the display identified by TAG.

Read Record and Release Data on Disk

RDR N, S, BA, EA, NAME, P

Reads into the buffer area in central memory the logical record specified by P of the file identified by NAME onto logical disk N. The words are read, without code translation, into the buffer area beginning at the address contained in location BA and ending at the address contained in location EA. Once the data are in memory, the disk space is released for use by other programs.

Read Tape Forward, Binary Mode

RFB N, S, BA, EA, RL, C

Reads, in binary parity, the number of 60-bit words per tape record, RL, from logical tape unit N. Each 6-bit character is converted as specified by the conversion mode C. BA and EA contain the location for the beginning and ending addresses of the buffer area into which the words are read. See Example 2.

Read Tape Forward, Coded Mode

RFC N, S, BA, EA, RL, C

Reads, in BCD parity, the number of 60-bit words per tape record, RL, from logical tape unit N. Each 6-bit character is converted as specified by the conversion mode C. BA and EA contain the location for the beginning and ending addresses of the buffer area into which the words are read. See Example 2.

Request Disk Space

RQD N, S, L, NAME, R

Reserves on logical disk unit N the file identified by NAME which has L number of 60-bit words in its longest record. R specifies the maximum number of logical records into which the file may be segmented. The parameters N, L, and R must be numbers, where $N \leqq 16_{10}$, $L \leqq 2^{17}$, and $R \leqq 4000_{10}$. NAME must be unique within the routine.

Request Memory Space

RQM NW, S, A

Reserves in central memory the total number or words specified by NW. The system sets A to the location containing the address of the first word of the assigned block in central memory.

Request Tape Assignment from System

RQT N, S

Requests logical tape unit N for the exclusive use of a program.

Read Console Typewriter

RTY N, S, BA, EA, RL, TAG

Reads and identifies a message with the identification number, TAG, typed on the typewriter at logical console unit N. Transmits RL number of characters to a buffer area in central memory beginning at the address contained in location BA and ending at the address contained in location EA.

Rewind Tape to Load Point

RWL N, S

Rewinds logical tape unit N to the physical load point on the tape.

Rewind Tape for Unload

RWU N, S

Rewinds logical tape unit N so that the tape may be dismounted.

Search File Mark Backward

SFB N, S

Searches the tape on logical unit N one record at a time back towards the load point until a file mark is passed over. When the mark is found, the tape is positioned on the load-point side of the file mark. If none is found, the macro is equivalent to RWL.

Search File Mark Forward

SFF N, S

Searches the tape on logical unit N one record at a time from the current position forward until a file mark is passed over. When the mark is found, the tape is positioned on the side of the file mark away from the load point. If no mark is found, the end of tape marker stops the search.

Suppress Space After Next Print

SPA N, S

Suppresses on logical printer N the automatic advance after the next line printed with a PRN macro.

Single-Space Printer

SSP N, S

Advances logical printer N one line.

Transfer PP Program and Begin Execution

TPP N, S, SYMBOL

Produces a calling sequence to the PP loader which, during execution, transfers PP program SYMBOL from central memory to logical peripheral processor N and begins execution with the first ASPER instruction. This macro is used to load an ASPER program into a PP from CM at execute time. The load begins at the first binary card and continues until the loader encounters another ASPER header card, a SUBP header card, or a terminate card. Execution begins at the first ASPER instruction defined under an ORGR pseudo code.

The TPP call from a CM program can load any PP in the system. However, the TPP call by a PP program can load any other PP in the system but cannot load itself.

Wait Check

WAI S, SYMBOL

Checks the status response word of other macros during a buffered operation. If the operation has been aborted, the WAI macro exits to the address specified by SYMBOL. If not, the next instruction, in line, is executed.

Write File Mark

WFM N, S

Writes an end of file mark on the tape on logical unit N.

Write Tape, Binary Mode

WRB N, S, BA, EA, RL, C

Writes, in binary parity, the data between BA and EA in records of RL 60-bit words each onto logical tape unit N. Each 6-bit character transferred is converted as requested by the conversion mode C. The words are written from a buffer area in central memory beginning at the address contained in location BA and ending at the address contained in location EA. If the conversion mode is 0, a straight binary output is expected. If one of the other conversion modes is used, Example 2 applies.

Write Tape, Coded Mode

```
WRC    N, S, BA, EA, RL, C
```

Writes, in BCD parity, the data between BA and EA in records of RL 60-bit words each onto logical tape unit N. Each 6-bit character transferred is converted as requested by the conversion mode C. The words are written from a buffer area in central memory beginning at the address contained in location BA and ending at the address contained in location EA. See Example 2.

Write Record on Disk

```
WRD    N, S, BA, EA, NAME, P
```

Writes from the buffer area in central memory the logical record specified by P of the file identified by NAME onto logical disk N. The words are written, without code translation, from the buffer area beginning at the address contained in location BA and ending at the address contained in EA.

Load Segment

```
LOAD   SYMBOL
```

Loads the subroutine SYMBOL into PP memory. SYMBOL is a subroutine defined by the pseudo opcode SUBP. When asterisks enclose SYMBOL, then control is transferred to the indicated routine.

7.2 EXAMPLES

EXAMPLE 1

DISPLAY AND TYPEWRITER INPUT/OUTPUT

Suppose a program needs to display a request for control information which requires a reply from the operator. The message might be:

REQUEST SWITCH SETTING 1-5

Either the DHL or DHR macro may be used. Both require that (1) the message data be organized and ready for display before the macro itself is executed, and (2) a set of parameters define the message organization to the operating system.

(1) Data Organization:

Status — a word may be reserved for the status response from the system by use of the BSS pseudo code.

```
S     BSS'    1
```

Data — the message data may be entered into the ASCENT program by use of the DPC pseudo code.

```
DATA     DPC
*REQUEST SWITCH
 SETTING 1-5*
```

The output data block may be reserved by

```
DIS     BSS     3
```

The one-word message input area may be reserved by

```
DISIN     BSS     1
```

Record Length — The length of the record to be displayed is 26 characters.

(2) Parameters:

Unit Number — The logical unit number is used only to indicate, relatively, a different console between different macros in the same program. For instance, logical unit number 2 may be any console that is available except one which has been previously referenced as logical unit number 1, 3, 4, etc.

Status — The central memory address, S, may be designated by use of a literal (S).

BA — The beginning address of the message in central memory, DIS, may be designated, as was S, with a literal (DIS).

EA — The ending address in central memory, DIS+3 may be similarly written (DIS+3).

RL — The record length may be given explicitly as 26 or as a symbolic CM address, Z, which contains 26.

```
Z   CON   26
```

TAG — A program may put up more than one request which requires a reply from the operator. Therefore an identifier "TAG" is provided. This tag is then appended to the program account number by the system to provide total uniqueness to all requests from the same and/or different programs. Let us suppose the account number is 3512, and TAG is 1.

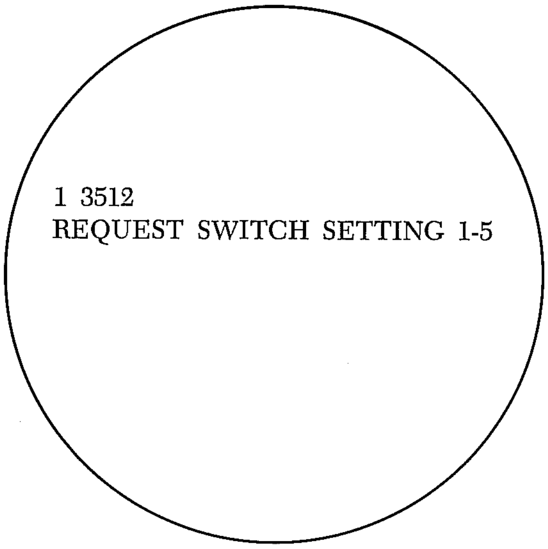
SIZE — The message character size may be chosen as 64, 32, or 16 characters per line. Let us suppose 32 characters per line is chosen.

The macro is then written:

DHL 1, (S), (DIS), (DIS+3), 26, 1, 1

In this example, logical console number 1 is used. The literal notation is used for the address specification of the status response word and data locations and the RL is given numerically.

The result of executing the macro would be a display positioned somewhere on the left scope of logical console number 1 as follows:



```
1 3512
REQUEST SWITCH SETTING 1-5
```

Implication from the message is that the program expects the operator to type a reply. Acceptance of the reply requires another macro, RTY. The parameters for this are N, S, BA, EA, RL, TAG.

N might be 1 to specify the typewriter on logical console 1

S is a CM word and in this case may be the same one as before

BA is the beginning address of the input message area, DISIN

EA need be only one larger than BA since reply is less than 10 characters

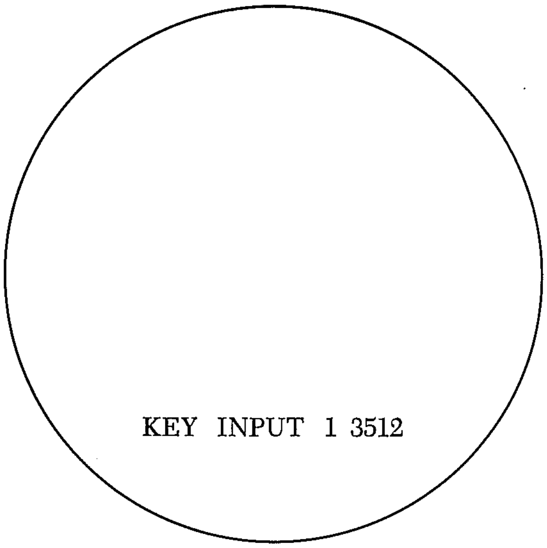
RL is 1 since the response is a single digit

TAG is the identifier that the operator must respond to in order to associate his typing with the request being made, namely 1 3512.

The macro issued would be:

RTY 1, (S), (DISIN), (DISIN+1), 1, 1

When the system indicates a ready with the following display:



```
KEY INPUT 1 3512
```

the operator must type a number, say 3, which is the switch setting:

3 carriage return

to satisfy both the RTY and DHL macros. The system places this response, 3, at the bottom of the scope.

EXAMPLE 2

PUNCH READER PRINTER AND TAPE INPUT/OUTPUT

Prior to execution of coded data output macros, it is necessary that the data to be outputed exist in central memory in BCD or display coded form. The coded data are assumed, by the macro, to be packed 10 characters/word from left to right for all words between the addresses contained in locations BA and EA.

Execution of the macro produces r cards, print lines or tape records of 10*(RL) characters each, where r is the number of records required to output all the data between BA and EA. In the process of transfer, each character is translated from the internal code to the output code according to the code conversion mode C.

Suppose data to be punched are:

$$1.0_{\Delta\Delta\Delta\Delta}3.925_{\Delta\Delta\Delta}1.3124 \qquad (1)$$

$$2.0_{\Delta\Delta\Delta\Delta}4.177_{\Delta\Delta\Delta}3.2127$$

then the internal storage in display code would be:

```
BA      34573300000000365744
        35400000003457363435
        37000000000000000000          (2)
        35573300000000375734
        42420000003657353435
        42000000000000000000
```

The data are to be punched and therefore must be converted to Hollerith which calls for a conversion mode of 1 for display to Hollerith.

The Punch macro is:

```
PCH  1,  (S),  (BA),  (BA+6),  3,  1
```

Execution of the macro would produce the two cards of output left justified from Column 1 as given in (1) above.

For input the same conventions hold except in this case the data are external and will be placed into memory as given above. If the data example above were left justified on two consecutive cards, and the read card macro

```
RDC  1,  (S),  (BA),  (BA+6),  3,  1
```

were executed, the data would come into central memory as shown in (2) above beginning at BA.

Compatibility exists between formats for tape I/O and cards and between card and tape output and printer output. The conversion mode differs due to the introduction of Hollerith code for cards. To print the data in (2) above, the macro used would be:

```
PRN  1,  (S),  (BA),  (BA+6),  3,  2
```

and write tape would be

```
WRC  1,  (S),  (BA),  (BA+6),  3,  2
```

To read the data from the output tape a

```
RFC  1,  (S),  (BA),  (BA+6),  3,  1
```

produces the same internal form as given in (2) above.

For binary data transfers, the conversion mode C = 0 is used. This mode produces a bit-by-bit transfer without conversion to the output device from memory or from the device to memory. In the case of card input and output, one column on the card corresponds to one of 5 12-bit bytes of each CM word. That is, the leftmost 5 columns are inserted from left to right into the first CM word specified, the next five into the next CM word, etc. RL is the number of consecutive 5 column fields to be considered on each card. Examples of binary and coded inputs and their conversion to card image in central memory are given in Figures 3 and 4, respectively.

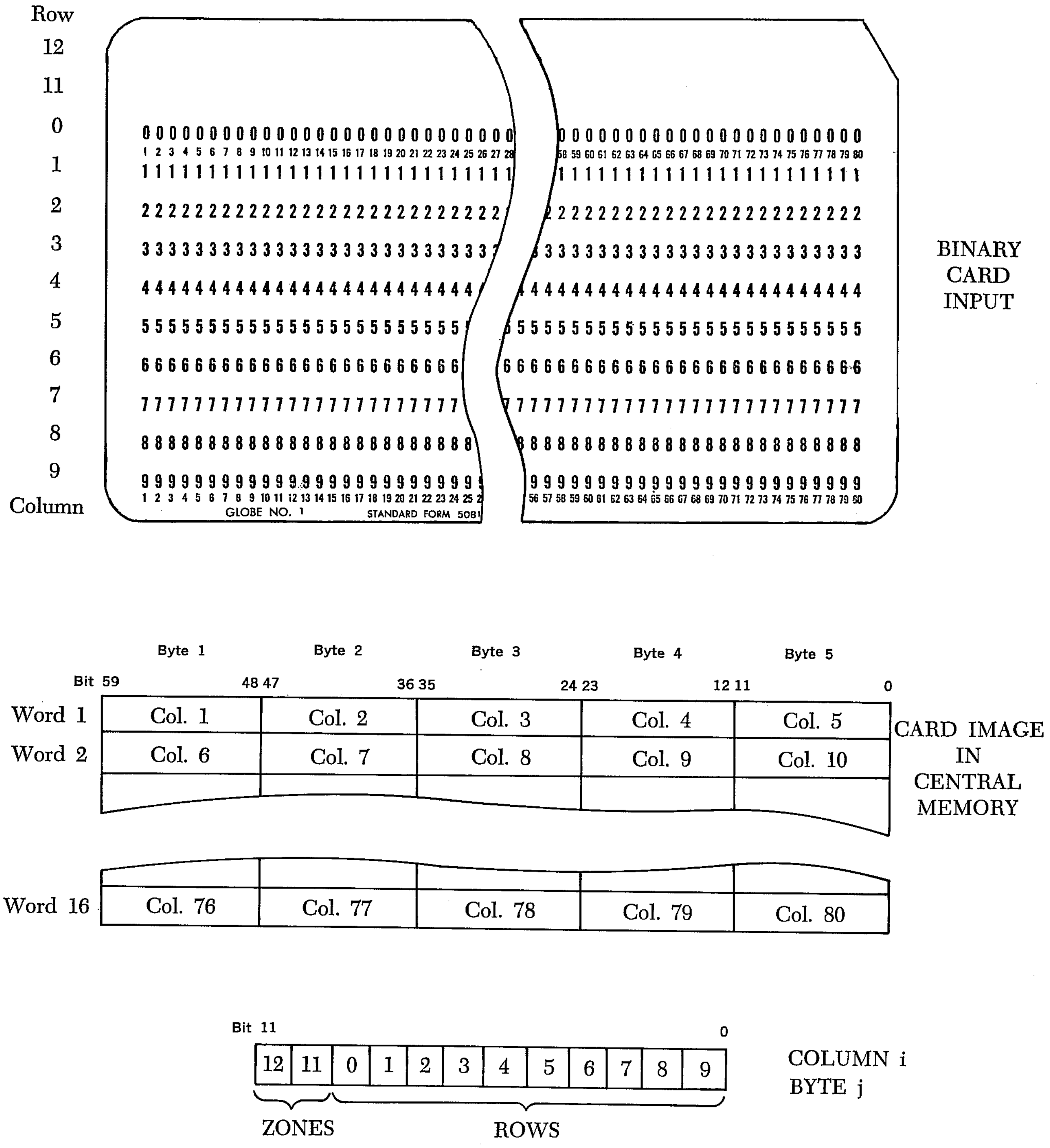
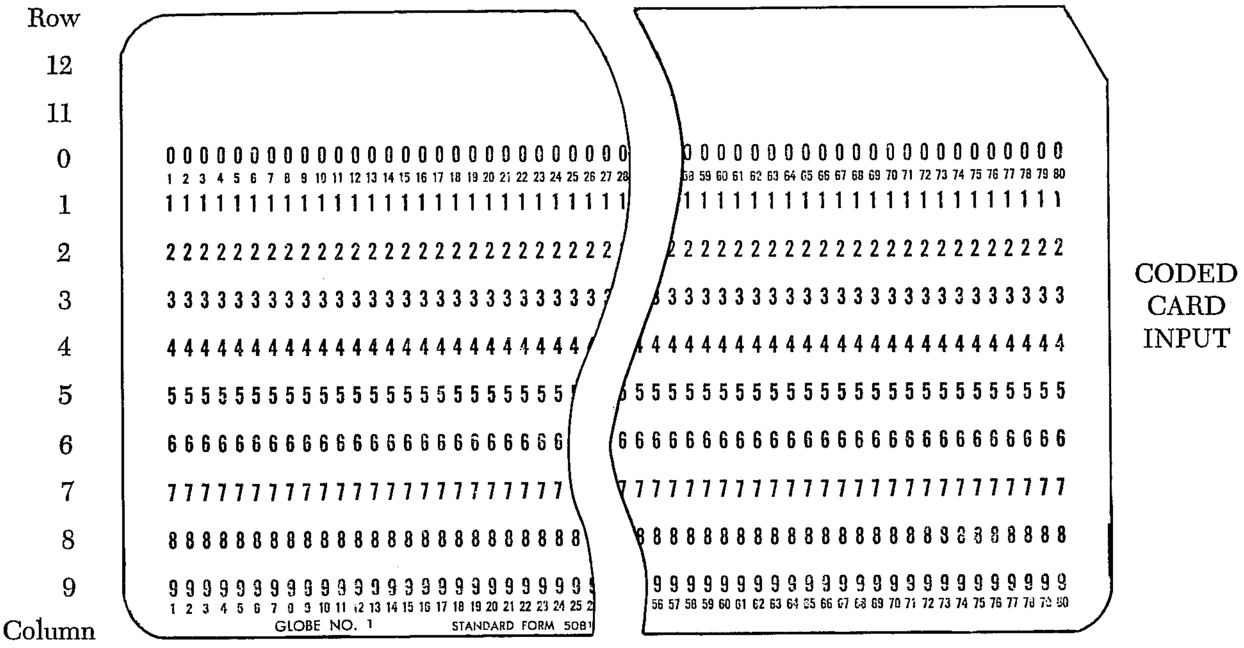


Figure 3. BINARY CARD INPUT



	Byte 1 Bit 59 … 48	Byte 2 47 … 36	Byte 3 35 … 24	Byte 4 23 … 12	Byte 5 11 … 0
Word 1	Cols. 1-2	Cols. 3-4	Cols. 5-6	Cols. 7-8	Cols. 9-10
Word 2	Cols. 11-12	Cols. 13-14	Cols. 15-16	Cols. 17-18	Cols. 19-20
Word 8	Cols. 71-72	Cols. 73-74	Cols. 75-76	Cols. 77-78	Cols. 79-80

CARD IMAGE IN CENTRAL MEMORY

Bit 11 … 6	5 … 0
Col. i,Coded	Col. i+1,Coded

COLUMN i, i+1
BYTE j

Figure 4. CODED CARD INPUT

8. DIAGNOSTICS AND ASSEMBLER OUTPUT

8.1 ASCENT ERROR PRINTOUTS

A Literal Table Full. The literal is not assigned a location.

B Symbol Table Full. The symbol is not assigned a location.

D Duplicate Symbol. The symbol in the location field has been previously defined. A list of all duplicate symbols is printed at the end of the side-by-side listing.

E Instruction Error. There are more than six instructions on the card.

F Format Error. An error is detected in the format of an instruction.

I Integer Error. An error is detected in a decimal or octal number.

K K-Field Error (address field). The address portion of the instruction does not meet program specifications or is out of range.

L Literal Error. An error is detected in the evaluation or conversion of the literal.

M Multiple Defined Reference. A reference is made to a symbol that appears more than once in the location field.

O Operation Code Error. The operation code cannot be evaluated. ASCENT assumes an operation code of zero and processes the instruction accordingly.

P Parameter List Error. The parameter list does not satisfy ASCENT specifications. The list may contain too few or too many parameters.

R Register Error. An error is detected in the format of a register name or its improper usage.

S Sign Error. A sign is incorrect or out of order.

T Tag Error. A symbol in the location or address field does not meet ASCENT specifications.

U Undefined Symbol. A reference is made to a symbol that does not appear in the location field. ASCENT assigns a location at the end of the object program to each unique undefined symbol. In certain pseudo codes (EQU, BSS, BSSZ), a symbol used in the address field must be defined prior to the pseudo code. A list of undefined symbols appears at the end of the side-by-side listing.

8.2 SAMPLE PROGRAM PRINTOUT

```
020000  6110000001             START    SB1     1                              .1 TO B1
                  6170001000            SB7     1000B                          .INDEX TO B7
020001  5157020043             START1   SA5     B7+SYMBOL                      .CONTENTS SYMBOL+B7 TO X5
                  0100020017            RJ      SUB                            .RETURN JUMP TO SUB
020002  0307020004                      ZR      X7 START2                      .TEST X7=0
                  67771                 SB7     B7-B1                          .DECREMENT INDEX
020003  0570020001                      NZ      B7 START1                      .TEST B7
                                        .COMPUTE F=(A*B)/E+C*D
020004  5110020037             START2   SA1     A                              .LOAD A TO X1
                  5120020040            SA2     B                              .LOAD B TO X2
020005  41712                           RX7     X1*X2                          .MULTIPLY X1*X2
             5140020041                 SA4     C                              .LOAD C TO X4
02006   5150021043                      SA5     D                              .LOAD D TO X5
                  41645                 RX6     X4*X5                          .MULTIPLY X4*X5
020007  5130021044                      SA3     E                              .LOAD E TO X3
                   45073                RX0     X7/X3                          .DIVIDE (A*B) BY E
                        30706           FX7     X0+X6                          .ADD (A*B)/E TO C*D
020010  25707                           ZX7     X7                             .ROUND AND NORMALIZE RESULT
             5170020042                 SA7     F                              .STORE RESULT IN F
020011  01000000130000000000            PRN     1,(STATUS),(MES),(MES+2),2,2   .PRINT MESSAGE

020016  0000000000                      PS                                     .STOP
020017  0200020017             SUB      JP      *                              .FORCE N I UPPER
020020  61100000016140777774            SB1     1         $ SB4  -3            .EXAMPLE OF MORE THAN
020021  43760205145130021050            MX7     48        $ LX5  12 $ SA3 NS   .ONE INST.PER CARD
020022  514002104763330                 SA4     ATS       $ SB3  X3
020023  043002001753340                 ZR      B3 SUB    $ SA3  X4
020024  11673                  SUB1     BX6     X7*X3                          .BOOLEAN X3*X7
             54434                      SA4     A3+B4
                  55331                 SA3     A3-B1                          .CONTENTS A3-B1 TO X3
                        37665           IX6     X6-X5                          .X6-X5 TO X6
020025  0306020031                      ZR      X6 SUB2                        .TEST X6 FOR 0
                  11673                 BX6     X7*X3                          .BOOLEAN X7*X3
                        20424           LX4     20                             .X4 LEFT 20
```

```
020026  55331                                SA3      A3-B1                             . CONTENT A3-B1 TO X3
             37665                           IX6      X6-X5                             . X6-X5 TO X6
                  0306020034                 ZR       X6 SUB4                           . TEST X6 FOR 0
020027  0631020024                           GE       B3 B1 SUB1
                  76710                      SX7      B1                                . B1 TO X7
                       20560                 LX5      48                                . LEFT X5 BY 48
020030  0400020017                           EQ       SUB                               . JUMP TO EXIT
020031  54331717000777721450    SUB2         SA3      A3+B1$ SX7   7777B$ AX4 40        . EXAMPLE OF
020032  116732056043700                      BX6      X7*X3$ LX5   48    $ MX7 0        . MORE THAN
020033  0400020017                           EQ       SUB                               . ONE INSTRUCTION
020034  543310400020031         SUB4         SA3      A3+B1$ EQ   SUB2                  . PER CARD
020035  00241011230011230001    MESSAGE      DPC      * THIS IS AN EXAMPLE *
020037  17205000000000000000    A            CON      1.25
020040  17206000000000000000    B            CON      1.5
020041  17204000000000000000    C            CON      17204000000000000000B
020042  00000000000000000001    F            BSSZ     1
                                TAG          EQU      SUB2
020043  00000000000000001000    SYMBOL       BSS      1000B
021043  17244000000000000000    D            CON      1.6E+1
021044  17174000000000000000    E            CON      .5
021045  77777777777777776773    LOC          CON      A-D
021046  00000000000000000001    STATUS       BSSZ     1
021047  00000000000000000001    ATS          BSSZ     1
021050  00000000000000002000    NS           CON      2000B
                                             END

ERRORS        000000
SYMBOLS       000024
ASCENT        021054
ASPER-PP      000000
ASPER-CM      000000
020000   N    START 020031 N      TAG 021045   N      LOC
```

8.3 SUMMARY PAGE DIAGNOSTICS

At the end of each ASCENT assembly a summary page is printed that includes the number of errors detected, number of symbols assigned, length of ASCENT program, length of ASPER program, amount of central memory storage defined by the ASPER program, and a list of symbols that are undefined, duplicated or not referenced. An example follows:

```
ERRORS        00005
SYMBOLS       00234
ASCENT        02011
ASPER-PP      03121
ASPER-CM      01000
000100  N  ABCDE      000205  N  TAGA      000500  D  AB      002006  U  ST
002007  U      TA     002010  U  SYMB
```

Explanation

ERRORS - Total number of lines with at least one error.

SYMBOLS - The number of symbols assigned a location.

ASCENT - Address of the next central memory location that is available after the central memory program.

ASPER-PP - Address of the next peripheral processor memory location that is available.

ASPER-CM - Number of central memory locations defined by the ASPER program.

aaaaaa NUD TAG

a = location assigned to TAG

N = TAG is not referenced by the program. (NULL)

U = TAG is undefined.

D = TAG is a duplicate symbol.

All numbers are octal.

9. SUBROUTINES

9.1 SYSTEM LIBRARY SUBROUTINES

A set of subroutines are provided in the system library for general use by both ASCENT and FORTRAN-66. In many cases, the library routines are referenced as Function Subroutines by FORTRAN-66 and as subroutines in ASCENT coding. Therefore, a compatible format is used in the definition of the routines, the FORTRAN code generators, and the ASCENT calling sequences.

The general form is:

CALL   NAME   (LIST)

where: CALL is a FORTRAN statement.

NAME is the name of the routine.

LIST contains a sequence of operands* which define actual parameters.

Table 5 of the Appendix gives a list of library subroutines and their respective calling sequences.

9.2 PROGRAMMER DEFINED SUBROUTINES

In addition to the library functions, a programmer may define new subroutines in the process of writing a program. These also provide separate assembly and/or debugging operations from the main program and other subroutines since over-all program linkages are made at load time, rather than at compile time. Symbols within a subroutine are local to that subroutine.

A compatible definition and calling format are used. To define a subroutine, a header card is needed:

SUBROUTINE   SYMBOL   (LIST)

where: SUBROUTINE is the pseudo operation code.

SYMBOL is the identification name for the subroutine.

LIST is a sequence of symbols, called *Formal Parameters,* separated by commas, which represent input and output variables to the subroutine.

References to the subroutine are made with the statement:

CALL   SYMBOL   (LIST)

where: SYMBOL is the same combination of letters used in the subroutine identification name.

LIST contains the names and values# of the input or output parameters for the subroutine in the same order as given in the subroutine definition list.

In addition to the parameter list, other communications are provided between the subroutine and the calling programs. These include alternate entry points, common data blocks, and variable subroutine and function names for calls made within the subroutine. For a full discussion of these features, see the following items in the FORTRAN-66 manual:

ENTRY

COMMON

EXTERNAL

The generation of code as a result of the CALL statement produces a fixed sequence as follows:

RJ   SUB   . $K^* 2^{18} + L$
JP   [ENTRY]   . 0 A(P1)
JP   [ERROR]   . 0 A(P2)
        . 0 A(P3)
        .
        .
        .
0   0   . 0 A(P)n
[ENTRY]

where:

SUB   is the address of the first word of the subroutine.

* Operands may be generalized to arithmetic statement forms due to FORTRAN-ASCENT language mix properties. See CALL statements in FORTRAN-66 manual.

Values may be generalized to arithmetic statement forms due to FORTRAN-ASCENT language mix properties. Also, continuation cards may be used. See CALL statements in the FORTRAN-66 manual.

K — is the number of parameters given in the list.

L — is an 18-bit linkage value (used by the loader).

[ENTRY] — is next line of normal coding after the generated code.

[ERROR] — is an error exit address.

A(Pi) — is the address of parameter Pi; i = 1, 2, . . . K.

Execution of the instruction

```
    RJ      SUB
```

sends the address of the reentry point to the first location in SUB. This reentry point is also the address of the first parameter.

In assembly language, this information may be used any way the programmer chooses. If a formal parameter list is used which matches the list of actual parameters in the CALL statement, as in FORTRAN, he may reference them in his code. However, it must be realized that each formal parameter symbol is assigned a value by the assembler which corresponds to its position in the list. Therefore, references to it yield an additive which can be applied to the address stored in the first word of the subroutine and used to locate the desired parameter address.

Examples:

```
    SUBROUTINE      A(P1, P2, P3)
```

P1 is assigned value 0.

P2 is assigned value 1.

P3 is assigned value 2.

The first word of SUBROUTINE A contains, in the address portion of the left 30-bit instruction, the address of the parameter list. As an example, then, the address of the third actual parameter is

$$(\text{First word}) * 2^{-30} + \text{P3}$$

The FORTRAN convention is as follows:

One of the B registers, say Bk, is set permanently to the address from relative zero. Then parameter values of simple variables are loaded using:

```
    SAi = Bk + Pi    . address of Pi→ Xi
    -     -     -
    -     -     -    . other coding to buffer
    -     -     -    . memory access
    SAi = Xi         . value of Pi to Xi
```

If an index is applied to the parameter, the second instruction is biased by the B register containing the index, Bj, for example.

```
    SAi = Xi + Bj    . value of (Pi)j to Xi
```

10. PROGRAM SEGMENTATION

In general, the complex for a central processor program is assumed to be made up of control programs, subroutines, and common data blocks. The initiating control program, any common subroutines and data blocks comprise a permanent segment in core. Any subsequent control programs, subroutines and their data blocks are arranged dynamically in core in segments according to requests encountered during execution, and as defined by segmentation control cards.

The compiling process handles all programs and subroutines individually. The routines are compiled separately and, in binary form, are put together with segmentation specifications at execute time. Linkage betwen segments and between routines is handled at load time. Although a routine may appear in any number of segments, only one copy need be compiled and placed with the job.

Definition of Terms

Basic Segment – a fixed arrangement of a control program, subroutines, and common data blocks.

Normal Segment – an arrangement of a control program, subroutines, and common data read into central memory dynamically as required. It is defined by a segment control card and loaded by means of the LOAD statement. Normal segments may be overlayed.

Control Program – defined with a PROGRAM card and is the only executable program within a basic segment or a normal segment.

Subroutine – a routine defined by a SUBROUTINE card and executed by means of a CALL statement.

LOAD – is a macro defined as the overlay segment request.

The contents of a segment are provided by segmentation control cards at load time and are specified as a combination of control programs, subroutines, and other segments. The overlay operation when executed does not disturb the contents of the basic segment. However, it does destroy all other requested segments operated prior to the loading of the overlay. Overlay requests may occur without restriction in any segment and are coded in line where the decision is reached that an overlay is required.

Two independent segmentation concepts are provided under a single programming mechanism. The first method allows a basic segment, which resides permanently in core, to initiate loading, and to control routing to the various subroutines in the job. Under this concept, control flows back and forth between the basic segment and any routine in the other segments, but always returns to the basic segment prior to the initiation of an overlay. When an overlay is initiated, control continues to the next instruction in line (no control transfer occurs).

The second method provides a program chaining operation. Under this method each successive overlay has its own control program and provides control routing through the various routines in that overlay. Each new overlay destroys the preceding one. Since loading is initiated by the overlay, it is necessary that a control transfer be made in conjunction with the load request.

An overlay request may occur in any segment. However, a transfer address must be provided if the request is made from other than the basic segment. The transfer address, if required, is taken to be that of the control program which has a trace of asterisks all the way through to the LOAD statement. An example of a segmented program and the tracing of control through the segments with asterisks follows:

EXAMPLE: (Control in Normal Segments)

Assume the basic segment is loaded consisting of non-executable statements defining common data and storage areas and a load request:

```
LOAD     * SEG1 *
```

SEG1 is defined as:

```
SEG1 = *PROGRAM A*, SUBROUTINE
          A1, SUBROUTINE  A2
```

After the load is accomplished, control is shifted to Program A as a result of asterisks around SEG1 and Program A in the definition of SEG1.

Assume a load request is given in SEG1:

LOAD   * SEG2 *

SEG2 = *PROGRAM B*, SUBROUTINE B1, SUBROUTINE B2

The load is then performed and control turned over to Program B.

Suppose an overlay request exists within SEG2 which includes both segments SEG1 and SEG2, plus Program C. In addition, suppose control is to be shifted back to Program A after SEG3 has been loaded. The specification of the third segment may take one of several forms:

1. SEG3 = *PROGRAM A*, SUBROUTINE A1, SUBROUTINEA2, PROGRAM B, SUBROUTINE B1, SUBROUTINE B2, PROGRAM C
2. SEG3 = /*SEG1*/, PROGRAM B, SUBROUTINE B1, SUBROUTINE B2, PROGRAM C
3. SEG3 = /*SEG1*/, /SEG2/, PROGRAM C

where slashes indicate the enclosed is a segment name.

Upon a LOAD *SEG3* statement, each of these three forms will produce the same core arrangement and transfer control to Program A since it is the only routine in this segment specification which has an asterisk trace through the segment definitions to the LOAD statement. Program A is asterisked (definition 1 above contains asterisks around Program A, definitions 2 and 3 above have asterisks around SEG1, which contains Program A enclosed with asterisks), SEG3 is asterisked in the LOAD statement.

However, if control is desired for Program C, SEG3 may then be defined as:

SEG3 = /SEG1/, /SEG2/, *PROGRAM C*

or either of the other two forms may be used with Program C enclosed with the asterisks.

A transfer address is not necessary when the basic segment maintains control throughout the execution of the object program. In this instance, segments consist of subroutines which are executed by means of a CALL statement. An example of a segmented program with control residing in the basic segment follows:

EXAMPLE: (Control in Basic Segment)

Assume the basic segment is loaded and executing. At some point within the basic segment, a load request is coded:

LOAD   SEG4

SEG4 is defined as:

SEG4 = SUBROUTINE A, SUBROUTINE A1, SUBROUTINE A2

After the load is accomplished, all programming conventions relative to subroutine calls and communications may be followed. However, when the tasks of the routines in the normal segment are completed, control always reverts to the calling program in the basic segment which may then request the loading of another segment.

With reference to these examples, general characteristics of segment specifications are stated as follows:

1. Segments may be defined as a combination of routines and other segments. There is no limit on the depth used in the specification. Segment hierarchies may be conveniently defined as specifying large and highly overlapping segments.
2. There is no requirement that a one-to-one correspondence exist between segment definitions and segments referenced in LOAD statements. Extra segments may be defined which contain common groupings of routines and these segments may then be used to define the larger segments. Each segment referenced by a LOAD statement must be defined by a segmentation card; if not, the entire job is aborted.
3. With the chaining method, control is passed to a program in the loaded segment by an unbroken chain of asterisks from the LOAD statement through the segment specifications to the program name. This provides the necessary control routine capability.

4. There is no requirement that program or subroutine names appear uniquely in the combination of segments that define a segment. The union of the segments designates the routines to be loaded. This assures that there is no loss in core utilization due to name redundancies.

5. Segment identifiers used in the definition of other segments are surrounded by slash marks (/) to distinguish them from program and subroutine names. If a segment name is enclosed in asterisks, the asterisks are placed inside the slash marks.

The following restrictions are made:

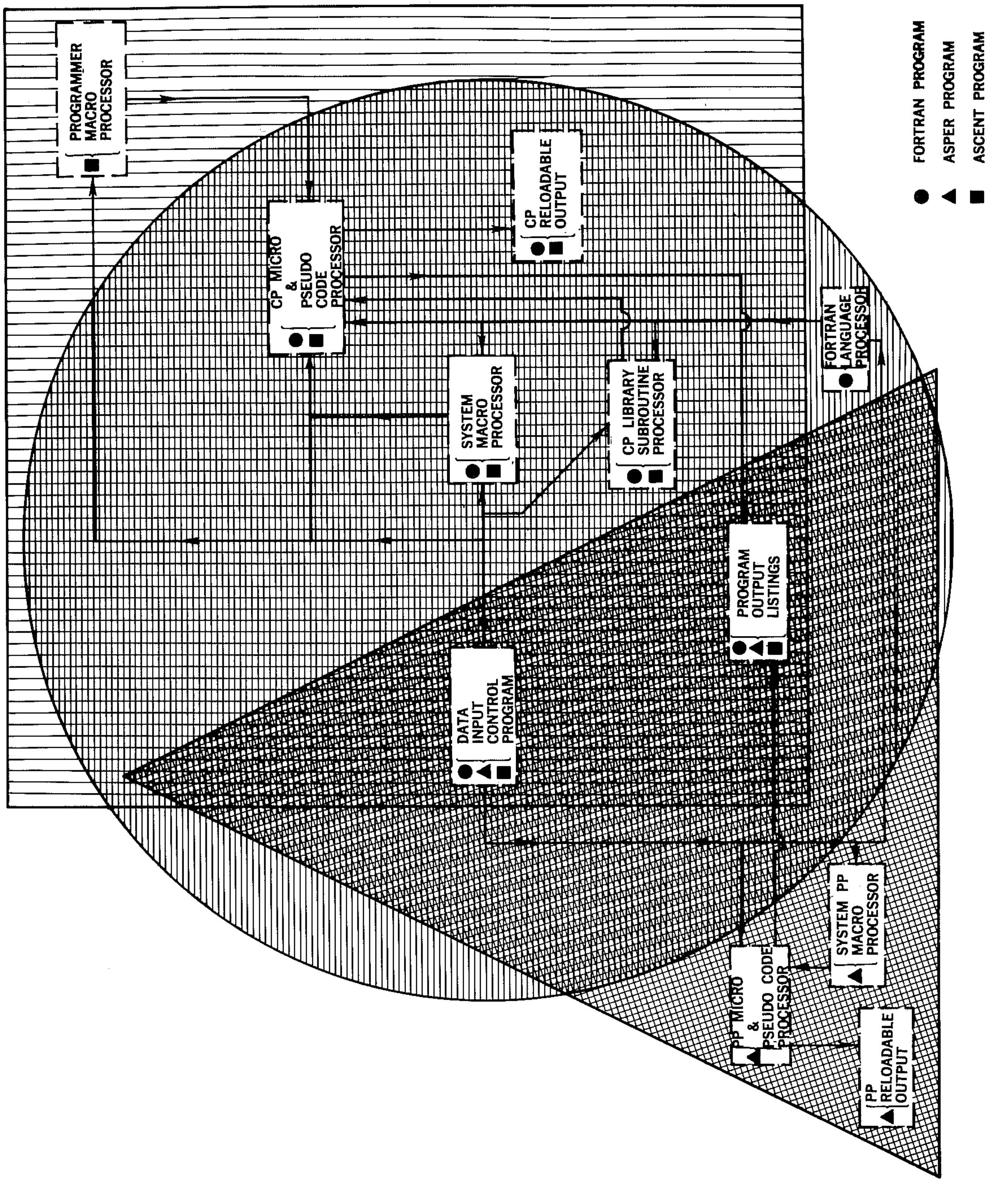
1. If the LOAD statement specifies a transfer of control, one and only one program in the loaded segment must be designated as recipient of the control by meeting characteristic number 3.

2. No mixing of the two methods is permissible although a one-time switch may be made from the basic segment concept to the chaining concept. Once the chaining process is used, there is no way to return control to the PROGRAM in the basic segment.

A core map taken at the completion of the loading of the basic segment would show the control programs, the programmer subroutines in the basic segment, one copy of common data blocks, and one copy of library subroutines and functions referenced by these basic segment routines. The block of memory required for this segment is permanently reserved. The next location after this block is the initial address for all normal segments called by a LOAD statement.

If there are additional segments specified to accompany the basic segment in the initial load or when an overlay request occurs, these segments are brought in from the disk and relocated at the initial overlay address. Addresses for common blocks referenced in the segments are determined from their location within the basic segment, or they are assigned and inserted where appropriate. Also one copy of any library subroutine, not previously required or loaded, is added to the program and linkages for all routines are then made.

When a transfer is indicated by the segment specification (asterisks around NAME), a similar name form is sought from the segment name list and control is routed to its single entry point.



■ PROGRAMMER MACRO PROCESSOR
● ■ CP MICRO & PSEUDO CODE PROCESSOR
● ■ CP RELOADABLE OUTPUT
● ■ SYSTEM MACRO PROCESSOR
● ■ CP LIBRARY SUBROUTINE PROCESSOR
● FORTRAN LANGUAGE PROCESSOR
● ▲ ■ PROGRAM OUTPUT LISTINGS
● ▲ ■ DATA INPUT CONTROL PROGRAM
▲ SYSTEM PP MACRO PROCESSOR
▲ PP MICRO & PSEUDO CODE PROCESSOR
▲ PP RELOADABLE OUTPUT
● FORTRAN PROGRAM
▲ ASPER PROGRAM
■ ASCENT PROGRAM

Figure 5. 6600 PROGRAMMING SYSTEM

11. PROGRAM ORGANIZATION

In the most general case, a program may have a combination of ASCENT or FORTRAN main programs, FORTRAN subroutines, assembly language subroutines, and peripheral processor programs. The program may be logically broken into any number of segments or overlays to be called during execution.

Any one of the central processor programs or subroutines, whether ASCENT or FORTRAN, may contain both languages mixed on a line-for-line basis. Also, any one of the central processor programs or subroutines, without regard to the segment in which it lies, may have its own set of peripheral processor programs. The individual peripheral processor programs may have overlays that are called during their own execution. They may also contain calls for the loading of other peripheral processor programs. A program with these operations is illustrated in skeleton form in Table 6. Several items which are not illustrated but can exist are:

1. Each of the CP programs or subroutines may contain COMMON block definitions and references.

2. Each PP program may define its own private data blocks in central memory.

3. Any of the routines may contain calls for system library subroutines and functions.

4. Any of the routines, CP or PP, may contain macro calls for system I/O operations.

5. Any one or all of the PP routines may contain requests to the system for I/O channels and its own I/O operations.

The general composite program in Table 6 shows those cases where unusual situations exist such as real time applications, very large problems, or problems which lend themselves to joint effort by more than one processor. Other programs, more conventional in nature, are handled in a normal manner by appropriate parts of the programming system. Setup procedures are standardized so that it is not necessary to put special control information in a program to indicate its nature. Rather, the various operations inherent in the unusual program are used to determine control requirements.

11.1 PROGRAM SETUP

11.1.1 HOMOGENEOUS PROGRAMS

The setup of a conventional program requires only the problem related instructions as defined for the language used – FORTRAN programs contain only compatible FORTRAN statements, etc. Precise specifications and examples of single language programs appear in the ASCENT, ASPER, and FORTRAN-66 manuals. Figure 5 illustrates the relationship of these programs.

11.1.2 MIXED FORTRAN-66/ASCENT LANGUAGE PROGRAMS

At any given moment during compilation, the programming system is in one of three modes, two of which are of interest here – FORTRAN mode and ASCENT mode. Initial mode is established by the header card used:

```
PROGRAM NAME       }
SUBROUTINE NAME    } → FORTRAN mode
           (LIST)  }

ASCENT NAME          → ASCENT mode
```

Once a mode is established, processing proceeds as in a homogeneous program until the mode is switched by the contents of one of the statement cards as follows:

```
F in column 1            }
Numeric statement tag    } → FORTRAN Mode
  in columns 2 - 5       }

A in column 1            }
Non-numeric statement    }
  tag beginning in       } → ASCENT Mode
  columns 2 - 5          }
```

It should be noted that only the first statement of a sequence of like code need contain the mode information, although redundant mode information on the cards does not affect the compilation.

The internal procedure, upon the occurrence of an END card, depends on the type of header card used. If a PROGRAM card is used, a stop instruction is generated; if a SUBROUTINE card is used, an exit return is generated; and if an ASCENT card is used, no generation takes place.

11.1.3 MIXED CENTRAL PROCESSOR AND ASPER PROGRAMS

In addition to the mix capability provided for central processor languages, it is also permissible to insert peripheral processor programs into central processor programs. This gives the programmer the capability of writing routines that share the processing load. Any number of ASPER routines may be defined as part of one central processor routine. The only limit exists relative to the number of routines which can be expected for simultaneous execution in peripheral processors. It should be noted that the assignments of ASPER routines are dynamically arranged during execution, and are assigned to a peripheral processor only when requested and remain there only as long as required. This means that, during the process of a run, a single peripheral processor might execute, by request, several different programs or the same one several times.

ASPER routines are inserted into the central processor program decks immediately prior to the END card and are, within themselves, homogeneous ASPER language routines. They are headed with an:

ASPER NAME

card. Each is followed by either the END card for the central processor routine or another ASPER routine similarly headed by the ASPER header card. A central processor program deck, with one ASPER routine inserted, would be made up as follows:

```
PROGRAM   NAME 1
—
.
.         Central processor coding
.
—
ASPER     NAME 2
—
.
.         Peripheral processor coding
.
—
END }     Normal END card for CP program
```

Similarly, a central processor program deck, with two ASPER routines inserted would be made up as follows:

```
PROGRAM   NAME 1
—
.
.         Central processor coding
.
—
ASPER     NAME 2
—
.
.         Peripheral processor coding
.
—
ASPER     NAME 3
—
.
.         Peripheral processor coding
.
—
END }     Normal END card for CP program
```

APPENDIX

TABLE 1
CENTRAL PROCESSOR OPERATION CODES

Octal Opcode	Mnemonic	Address	Comments
			. **BRANCH UNIT**
00	PS		. Program stop
01	RJ	K	. Return jump to K
02	JP	Bi + K	. Jump to Bi + K
030	ZR	Xj K	. Jump to K if Xj = 0
031	NZ	Xj K	. Jump to K if Xj ≠ 0
032	PL	Xj K	. Jump to K if Xj = plus (positive)
033	NG	Xj K	. Jump to K if Xj = negative
034	IR	Xj K	. Jump to K if Xj is in range
035	OR	Xj K	. Jump to K if Xj is out of range
036	DF	Xj K	. Jump to K if Xj is definite
037	ID	Xj K	. Jump to K if Xj is indefinite
04	EQ	Bi Bj K	. Jump to K if Bi = Bj
04	ZR	Bi K	. Jump to K if Bi = B0
05	NE	Bi Bj K	. Jump to K if Bi ≠ Bj
05	NZ	Bi K	. Jump to K if Bi ≠ B0
06	GE	Bi Bj K	. Jump to K if Bi ≧ Bj
06	PL	Bi K	. Jump to K if Bi ≧ B0
07	LT	Bi Bj K	. Jump to K if Bi < Bj
07	NG	Bi K	. Jump to K if Bi < B0
			. **BOOLEAN UNIT**
10	BXi	Xj	. Transmit Xj to Xi
11	BXi	Xj*Xk	. Logical Product of Xj & Xk to Xi
12	BXi	Xj + Xk	. Logical sum of Xj & Xk to Xi
13	BXi	Xj − Xk	. Logical difference of Xj & Xk to Xi
14	BXi	− Xk	. Transmit the comp. of Xk to Xi
15	BXi	− Xk*Xj	. Logical product of Xj & Xk comp. to Xi
16	BXi	− Xk + Xj	. Logical sum of Xj & Xk comp. to Xi
17	BXi	− Xk − Xj	. Logical difference of Xj & Xk comp. to Xi
			. **SHIFT UNIT**
20	LXi	jk	. Left shift Xi, jk places
21	AXi	jk	. Arithmetic right shift Xi, jk places
22	LXi	Bj Xk	. Left shift Xk nominally Bj places to Xi
23	AXi	Bj Xk	. Arithmetic right shift Xk nominally Bj places to Xi
24	NXi	Bj Xk	. Normalize Xk in Xi and Bj
25	ZXi	Bj Xk	. Round and normalize Xk in Xi and Bj
26	UXi	Bj Xk	. Unpack Xk to Xi and Bj
27	PXi	Bj Xk	. Pack Xi from Xk and Bj
43	MXi	jk	. Form mask in Xi, jk bits
			. **ADD UNIT**
30	FXi	Xj + Xk	. Floating sum of Xj and Xk to Xi
31	FXi	Xj − Xk	. Floating difference Xj and Xk to Xi
32	DXi	Xj + Xk	. Floating DP sum of Xj and Xk to Xi

Octal Opcode	Mnemonic	Address	Comments
33	DXi	Xj − Xk	. Floating DP difference of Xj and Xk to Xi
34	RXi	Xj + Xk	. Round floating sum of Xj and Xk to Xi
35	RXi	Xj − Xk	. Round floating difference of Xj and Xk to Xi
			. **LONG ADD UNIT**
36	IXi	Xj + Xk	. Integer sum of Xj and Xk to Xi
37	IXi	Xj − Xk	. Integer difference of Xj and Xk to Xi
			. **MULTIPLY UNIT**
40	FXi	Xj * Xk	. Floating product of Xj and Xk to Xi
41	RXi	Xj * Xk	. Round floating product of Xj & Xk to Xi
42	DXi	Xj * Xk	. Floating DP product of Xj & Xk to Xi
			. **DIVIDE UNIT**
44	FXi	Xj / Xk	. Floating divide Xj by Xk to Xi
45	RXi	Xj / Xk	. Round floating divide Xj by Xk to Xi
46	NO		. No operation
47	CXi	Xk	. Count the number of 1's in Xk to Xi
			. **INCREMENT UNIT**
50	SAi	Aj + K	. Set Ai to Aj + K
50	SAi	Aj − K	. Set Ai to Aj + comp. of K
51	SAi	Bj + K	. Set Ai to Bj + K
51	SAi	Bj − K	. Set Ai to Bj + comp. of K
52	SAi	Xj + K	. Set Ai to Xj + K
52	SAi	Xj − K	. Set Ai to Xj + comp. of K
53	SAi	Xj + Bk	. Set Ai to Xj + Bk
54	SAi	Aj + Bk	. Set Ai to Aj + Bk
55	SAi	Aj − Bk	. Set Ai to Aj − Bk
56	SAi	Bj + Bk	. Set Ai to Bj + Bk
57	SAi	Bj − Bk	. Set Ai to Bj − Bk
60	SBi	Aj + K	. Set Bi to Aj + K
60	SBi	Aj − K	. Set Bi to Aj + comp. of K
61	SBi	Bj + K	. Set Bi to Bj + K
61	SBi	Bj − K	. Set Bi to Bj + comp. of K
62	SBi	Xj + K	. Set Bi to Xj + K
62	SBi	Xj − K	. Set Bi to Xj + comp. of K
63	SBi	Xj + Bk	. Set Bi to Xj + Bk
64	SBi	Aj + Bk	. Set Bi to Aj + Bk
65	SBi	Aj − Bk	. Set Bi to Aj − Bk
66	SBi	Bj + Bk	. Set Bi to Bj + Bk
67	SBi	Bj − Bk	. Set Bi to Bj − Bk
70	SXi	Aj + K	. Set Xi to Aj + K
70	SXi	Aj − K	. Set Xi to Aj + comp. of K
71	SXi	Bj + K	. Set Xi to Bj + K
71	SXi	Bj − K	. Set Xi to Bj + comp. of K
72	SXi	Xj + K	. Set Xi to Xj + K
72	SXi	Xj − K	. Set Xi to Xj + comp. of K
73	SXi	Xj + Bk	. Set Xi to Xj + Bk
74	SXi	Aj + Bk	. Set Xi to Aj + Bk
75	SXi	Aj − Bk	. Set Xi to Aj − Bk
76	SXi	Bj + Bk	. Set Xi to Bj + Bk
77	SXi	Bj − Bk	. Set Xi to Bj − Bk

TABLE 2
PSEUDO OPERATION CODES

OPCODE	*MEANING*
ASCENT	Defines CP program
END	Defines end of CP program
ASPER	Defines PP routine
SUBROUTINE	Defines subroutine name
BSSD	Reserves disk space
BSS	Reserves central memory region
BSSZ	Reserves central memory region and presets it to zero
EQU	Equates a symbol to a value
DPC	Inserts display-coded characters into program
BCD	Inserts BCD characters into program
CON	Defines constants in program
	Remarks field excluded
LIST	Controls side-by-side listing
SPACE	Spaces side-by-side listing
EJECT	Ejects page on side-by-side listing

TABLE 3

SYSTEM MACROS

RQTW	Request tape assignment from system.
DRTW	Release tape back to system.
SFFW	Search file mark forward.
SFBW	Search file mark backward.
WFMW	Write file mark.
RWLW	Rewind tape to load point.
RWUW	Rewind tape for unload.
FSPW	Forespace.
BSPW	Backspace.
RFCW	Read tape forward coded mode.
RFBW	Read tape forward binary mode.
WRCW	Write tape coded mode.
WRBW	Write tape binary mode.
RDHW	Read record and hold data on disk.
RDRW	Read record and release data on disk.
WRDW	Write record bn disk.
SSPW	Single space printer.
DSPW	Double space printer.
FC7W	Select Format Channel 7.
FC8W	Select Format Channel 8.
MC1W	Select Monitor Channel 1.
MC2W	Select Monitor Channel 2.
MC3W	Select Monitor Channel 3.
MC4W	Select Monitor Channel 4.
MC5W	Select Monitor Channel 5.
MC6W	Select Monitor Channel 6.
CMCW	Clear Monitor Channels 1 – 6.
SPAW	Suppress space after next print.
PRNW	Print single line or multiple lines.
PCHW	Punch cards.
RDCW	Read cards.
DSRW	Display on right scope for system time limit.
DSLW	Display on left scope for system time limit.
DHRW	Display on right scope and hold indefinitely.
DHLW	Display on left scope and hold indefinitely.
RDPW	Remove display.
RTYW	Read console typewriter.
WAIW	Check status word.
TPPW	Transfer program SYMBOL from CM to PP memory and begin execution with first ASPER instruction.
RQMW	Request memory.
DRMW	Release memory.
RQDW	Request disk space.
DRDW	Release disk space.
LOAD	Load segment SYMBOL.

TABLE 4
6600 COMPUTER
CHARACTER CODES

Character	*Display Code*	*Printer Code*	*Hollerith Punch Positions*
A	01	61	12-1
B	02	62	12-2
C	03	63	12-3
D	04	64	12-4
E	05	65	12-5
F	06	66	12-6
G	07	67	12-7
H	10	70	12-8
I	11	71	12-9
J	12	41	11-1
K	13	42	11-2
L	14	43	11-3
M	15	44	11-4
N	16	45	11-5
O	17	46	11-6
P	20	47	11-7
Q	21	50	11-8
R	22	51	11-9
S	23	22	0-2
T	24	23	0-3
U	25	24	0-4
V	26	25	0-5
W	27	26	0-6
X	30	27	0-7
Y	31	30	0-8
Z	32	31	0-9
0	33	12	0
1	34	01	1
2	35	02	2
3	36	03	3
4	37	04	4
5	40	05	5
6	41	06	6
7	42	07	7
8	43	10	8
9	44	11	9
blank	00	20	space
+	45	60	12
−	46	40	11
*	47	54	11-8-4
/	50	21	0-1
(	51	34	0-8-4

Character		*Display Code*	*Printer Code*	*Hollerith Punch Positions*
)		52	74	12-8-4
=		54	13	8-3
≠		55	14	8-4
,		56	33	0-8-3
.		57	73	12-8-3
$		63	53	11-8-3
:	*internal codes only*	53	00	illegal
≦		60	15	5-8
%		61	16	6-8
[		76	17	7-8
]		77	32	0-2-8
→		62	35	0-5-8
≡		64	36	0-6-8
∧		65	37	0-7-8
∨		66	52	11-2-8
↑		67	55	11-5-8
↓		70	56	11-6-8
>		71	57	11-7-8
<		72	72	12-2-8
≧		73	75	12-5-8
¬		74	76	12-6-8
;		75	77	12-7-8

TABLE 5
LIBRARY SUBROUTINES

Any library subroutine may be called by name, either with or without the terminal F.

Let: Si be the ith symbol
I represent integer
F represent floating single precision

NAME	*Calling Sequence*	*Input Parameters*	*Mode*	*Output Parameters*	*Mode*	*Remarks*
ABSF	Call ABSF (S_1, S_2)	S_1	I or F	S_2	same as input	Form absolute value
INTF	Call INTF (S_1, S_2)	S_1	F	S_2	F	Truncate fraction
XINTF	Call XINTF (S_1, S_2)	S_1	F	S_2	I	Truncate fraction
XFIXF	Call XFIXF (S_1, S_2)	S_1	F	S_2	I	Truncate fraction
MODF	Call MODF (S_1, S_2, S_3)	S_1, S_2	F	S_3	F	$S_3 = S_1$ modulo S_2
XMODF	Call XMODF (S_1, S_2, S_3)	S_1, S_2	I	S_3	I	$S_3 = S_1$ modulo S_2
MAXOF	Call MAXOF (S_1, S_2, . . . Sn)	S_1, S_2, . . . Sn − 1	I	Sn	F	Sn = maximum (S_1, S_2 . . . Sn − 1)
XMAXOF	Call XMAXOF (S_1, S_2, . . . Sn)	S_1, S_2, . . . Sn − 1	I	Sn	I	Sn = maximum (S_1, S_2 . . . Sn − 1)
MAX1F	Call MAX1F (S_1, S_2, . . . Sn)	S_1, S_2, . . . Sn − 1	F	Sn	F	Sn = maximum (S_1, S_2 . . . Sn − 1)
XMAX1F	Call XMAX1F (S_1, S_2, . . . Sn)	S_1, S_2, . . . Sn − 1	F	Sn	I	Sn = maximum (S_1, S_2 . . . Sn − 1)
MINOF	Call MINOF (S_1, S_2, . . . Sn)	S_1, S_2, . . . Sn − 1	I	Sn	F	Sn = minimum (S_1, S_2 . . . Sn − 1)
XMINOF	Call XMINOF (S_1, S_2, . . . Sn)	S_1, S_2, . . . Sn − 1	I	Sn	I	Sn = minimum (S_1, S_2 . . . Sn − 1)
MIN1F	Call MIN1F (S_1, S_2, . . . Sn)	S_1, S_2, . . . Sn − 1	F	Sn	F	Sn = minimum (S_1, S_2 . . . Sn − 1)
XMIN1F	Call XMIN1F (S_1, S_2, . . . Sn)	S_1, S_2, . . . Sn − 1	F	Sn	I	Sn = minimum (S_1, S_2 . . . Sn − 1)
SINF	Call SINF (S_1, S_2)	S_1, radians	F	S_2	F	$S_2 = \sin S_1$
COSF	Call COSF (S_1, S_2)	S_1, radians	F	S_2	F	$S_2 = \cos S_1$
TANF	Call TANF (S_1, S_2)	S_1, radians	F	S_2	F	$S_2 = \tan S_1$
ASINF	Call ASINF (S_1, S_2)	S_1	F	S_2, radians	F	$S_2 = \sin^{-1} S_1$
ACOSF	Call ACOSF (S_1, S_2)	S_1	F	S_2, radians	F	$S_2 = \cos^{-1} S_1$
ATANF	Call ATANF (S_1, S_2)	S_1	F	S_2, radians	F	$S_2 = \tan^{-1} S_1$
TANHF	Call TANHF (S_1, S_2)	S_1, radians	F	S_2	F	$S_2 = \tanh S_1$
SQRTF	Call SQRTF (S_1, S_2)	S_1	F	S_2	F	$S_2 = \sqrt{S_1}$
LOGF	Call LOGF (S_1, S_2)	S_1	F	S_2	F	$S_2 = \log_e S_2$
EXPF	Call EXPF (S_1, S_2)	S_1	F	S_2	F	$S_2 = e^{S_1}$
SIGNF	Call SIGNF (S_1, S_2, S_3)	S_1, S_2	F or I	S_3	F or I	$S_3 =$ Sign S_2 times S_1

NAME	*Calling Sequence*	*Input Parameters*	*Mode*	*Output Parameters*	*Mode*	*Remarks*
DIMF	Call DIMF (S_1, S_2, S_3)	S_1, S_2	F	S_3	F	If $S_1 > S_2$ then $S_3 = S_1 - S_2$ If $S_1 \leqq S_2$ then $S_3 = 0$
XDIMF	Call XDIMF (S_1, S_2, S_3)	S_1, S_2	I	S_3	I	If $S_1 > S_2$ then $S_3 = S_1 - S_2$ If $S_1 \leqq S_2$ then $S_3 = 0$
CUBERTF	Call CUBERTF (S_1, S_2)	S_1	F	S_2	F	$S_2 = (S_1)^{1/3}$
FLOATF	Call FLOATF (S_1, S_2)	S_1	I	S_2	F	$S_2 = S_1$ in floating single precision form
RANF	Call RANF (S_1, S_2)	S_1	I or F	S_2	I or F*	$S_2 =$ number, repeated usage gives uniformally distributed set
POWERF	Call POWERF (S_1, S_2, S_3)	S_1, S_2	F	S_3	F	$S_3 = S_1^{S_2}$
ITOJ	Call ITOJ (S_1, S_2, S_3)	S_1, S_2	I	S_3	I	$S_3 = S_1^{S_2}$
XTOI	Call XTOI (S_1, S_2, S_3)	S_1, S_2	F, I respectively	S_3	F	$S_3 = S_1^{S_2}$
ITOX	Call ITOX (S_1, S_2, S_3)	S_1, S_2	I, F respectively	S_3	F	$S_3 = S_1^{S_2}$

*Sign of S_1 defines result mode, + is I, − is F.

TABLE 6

COMPOSITE PROGRAM

Program Item	*Remarks*
PROGRAM COMPOSITE	. First card of normal program deck.
—	
.	
.	. Conventional FORTRAN and/or ASCENT operations.
.	
—	
LOAD S1	. Load segment S1 which contains subroutines “B” and “C” immediately after subroutine “A.”
CALL B (X1, X2)	. Normal subroutine calls for overlay subroutines.
—	
.	
.	. Conventional FORTRAN and/or ASCENT operations.
.	
—	
TPP 1, S, PPA	. Transfer PP routine “PPA” to any available PP. Label that PP logical 1, start execution.
—	. Continuation of FORTRAN and ASCENT language steps.
.	. References to any identifiers of “COMPOSITE,” including COMMON block data, are permissible.
.	
.	
—	
LOAD S2	
ASPER PPA	. Define beginning of PP routine “PPA.”
—	
.	
.	
.	
—	
TAG	. .Tagged line to be used as overlay point.
—	. Continuation of program for first segment.
.	. References to any identifiers of “COMPOSITE,” including COMMON block data, are permissible.
.	
.	
—	
SUBP PPA1, TAG	. Header card for overlay section of PP code. “TAG” is overlay point.

Program Item	*Remarks*
— . . . —	. Continuation of code. References to identifiers in either "COMPOSITE" or "PPA" prior to line "TAG."
END	. End card for program "COMPOSITE."
SUBROUTINE A (P1, P2, P3)	. Header card for subroutine "A."
— . . . —	. Conventional FORTRAN and/or ASCENT operations.
END	
SUBROUTINE B (P1, P2)	. Header card for subroutine "B."
— . . . —	
END	
SUBROUTINE C (P1, P2, P3, P4)	. Header card for subroutine "C."
— . . . —	
TPP 2, S, PPB	. Transfer PP routine "PPB" to any available PP. Label this PP logical 2, start execution. (Logical 1 could be specified without conflict since the previous logical 1 was defined in another subroutine.)
— . . . —	. Conventional FORTRAN and/or ASCENT operations.
TPP 3, S, PPC	. Transfer PP routine "PPC" to any available PP other than logical 2. . Label it logical 3, start execution.
— . . . —	. Conventional FORTRAN and/or ASCENT operations.

Program Item	*Remarks*
TPP 2, S, PPD	. Transfer PP routine “PPD” to any available PP other than logical 3, if logical 2 is available. If not, hold transfer until it is. Label it logical 2. Start execution.
—	
.	
.	. Conventional FORTRAN and/or ASCENT operations.
.	
—	
ASPER PPB	. Header card for peripheral processor routine. The name could be PPA without conflict since the other PPA was local to “COMPOSITE.”
—	
.	. Normal ASPER Steps.
.	. References to identifiers of subroutine “C” including COMMON block data are permissible.
.	
—	
ASPER PPC	. Header card for PP routine “PPC.”
—	
.	. Normal ASPER Steps. References to identifiers of subroutine “C.”
.	
.	
—	
ASPER PPD	. Header card for PP routine “PPD.”
—	
.	. Normal ASPER steps.
.	. References to identifiers of subroutine “C.”
.	
—	
TPP 1, S, PPE	. Transfer PP routine “PPE” to any available PP other than logical 2 and 3.
—	
.	. Normal ASPER steps.
.	. References to identifiers of subroutine “C.”
.	
—	
ASPER PPE	. Header card for PP routine “PPE.”
—	
.	. Other ASPER steps.
.	. References to identifiers in subroutine “C.”
.	
—	
END	. End of subroutine “C” and related PP routines.
PROGRAM D	. Header card for alternate control program after overlay.

Program Item	*Remarks*
—	
.	
.	. Conventional FORTRAN and/or ASCENT coding.
.	
—	
END	. End of Program D.
END	. End of overall program including related subroutines.

Central Processor
Instruction Execution Times

Mnemonic & Octal Code		Name	Time (Minor Cycles)
		BRANCH UNIT	
PS	00	STOP	–
RJ	01	RETURN JUMP to K	13
JP	02	GO TO K + Bi	8*
ZR	030	GO TO K if Xj = zero	8*
NZ	031	GO TO K if Xj ≠ zero	8*
PL	032	GO TO K if Xj = positive	8*
NG	033	GO TO K if Xj = negative	8*
IR	034	GO TO K if Xj is in range	8*
OR	035	GO TO K if Xj is out of range	8*
DF	036	GO TO K if Xj is definite	8*
ID	037	GO TO K if Xj is indefinite	8*
EQ ZR	04 04	GO TO K if Bi = Bj	8*
NE NZ	05 05	GO TO K if Bi ≠ Bj	8*
GE PL	06 06	GO TO K if Bi ≧ Bj	8*
LT NG	07 07	GO TO K if Bi < Bj	8*
		BOOLEAN UNIT	
BXi	10	TRANSMIT Xj to Xi	3
BXi	11	LOGICAL PRODUCT of Xj and Xk to Xi	3
BXi	12	LOGICAL SUM of Xj and Xk to Xi	3
BXi	13	LOGICAL DIFFERENCE of Xj and Xk to Xi	3
BXi	14	TRANSMIT Xk COMP. to Xi	3
BXi	15	LOGICAL PRODUCT of Xj and Xk COMP. to XI	3
BXi	16	LOGICAL SUM of Xj and Xk COMP. to Xi	3
BXi	17	LOGICAL DIFFERENCE of Xj and Xk COMP. to Xi	3
		SHIFT UNIT	
LXi	20	SHIFT Xi LEFT jk places	3
AXi	21	SHIFT Xi RIGHT jk places	3
LXi	22	SHIFT Xk NOMINALLY LEFT Bj places to Xi RXi	3
AXi	23	SHIFT Xk NOMINALLY RIGHT Bj places to Xi	3
NXi	24	NORMALIZE Xk in Xi and Bj	4
ZXi	25	ROUND AND NORMALIZE Xk in Xi and Bj	4
UXi	26	UNPACK Xk to Xi and Bj	3
PXi	27	PACK Xi from Xk and Bj	3
MXi	43	FORM jk MASK in Xi	3
		ADD UNIT	
FXi	30	FLOATING SUM of Xj and Xk to Xi	4
FXi	31	FLOATING DIFFERENCE of Xj and Xk to Xi	4
DXi	32	FLOATING DP SUM of Xj and Xk to Xi	4
DXi	33	FLOATING DP DIFFERENCE of Xj and Xk to Xi	4
RXi	34	ROUND FLOATING SUM of Xj and Xk to Xi	4
RXi	35	ROUND FLOATING DIFFERENCE of Xj and Xk to Xi	4
		LONG ADD UNIT	
IXi	36	INTEGER SUM of Xj and Xk to Xi	3
IXi	37	INTEGER DIFFERENCE of Xj and Xk to Xi	3
		MULTIPLY UNIT	
FXi	40	FLOATING PRODUCT of Xj and Xk to Xi	10
RXi	41	ROUND FLOATING PRODUCT of Xj and Xk to Xi	10
DXi	42	FLOATING DP PRODUCT of Xj and Xk to Xi	10
		DIVIDE UNIT	
FXi	44	FLOATING DIVIDE Xj by Xk to Xi	29
RXi	45	ROUND FLOATING DIVIDE Xj by Xk to Xi	29
NO	46	PASS	–
CXi	47	SUM of 1's in Xk to Xi	8
		INCREMENT UNIT	
SAi	50	SUM of Aj and K to Ai	3
SAi	51	SUM of Bj and K to Ai	3
SAi	52	SUM of Xj and K to Ai	3
SAi	53	SUM of Xj and Bk to Ai	3
SAi	54	SUM of Aj and Bk to Ai	3
SAi	55	DIFFERENCE of Aj and Bk to Ai	3
SAi	56	SUM of Bj and Bk to Ai	3
SAi	57	DIFFERENCE of Bj and Bk to Ai	3
SBi	60	SUM of Aj and K to Bi	3
SBi	61	SUM of Bj and K to Bi	3
SBi	62	SUM of Xj and K to Bi	3
SBi	63	SUM of Xj and Bk to Bi	3
SBi	64	SUM of Aj and Bk to Bi	3
SBi	65	DIFFERENCE of Aj and Bk to Bi	3
SBi	66	SUM of Bj and Bk to Bi	3
SBi	67	DIFFERENCE of Bj and Bk to Bi	3
SXi	70	SUM of Aj and K to Xi	3
SXi	71	SUM of Bj and K to Xi	3
SXi	72	SUM of Xj and K to Xi	3
SXi	73	SUM of Xj and Bk to Xi	3
SXi	74	SUM of Aj and Bk to Xi	3
SXi	75	DIFFERENCE of Aj and Bk to Xi	3
SXi	76	SUM of Bj and Bk to Xi	3
SXi	77	DIFFERENCE of Bj and Bk to Xi	3

Comp.—Complement

DP—Double Precision

*Add 5 minor cycles to branch time for a branch to an instruction which is out of the stack (no memory conflict considered)

CONTROL DATA SALES OFFICES

ALAMOGORDO • ALBUQUERQUE • ATLANTA • BILLINGS • BOSTON • CAPE CANAVERAL • CHICAGO • CINCINNATI • CLEVELAND • COLORADO SPRINGS DALLAS • DAYTON • DENVER • DETROIT • DOWNEY, CALIFORNIA • HONOLULU HOUSTON • HUNTSVILLE • ITHACA • KANSAS CITY, KANSAS • LOS ANGELES MADISON, WISCONSIN • MINNEAPOLIS • NEWARK • NEW ORLEANS • NEW YORK CITY • OAKLAND • OMAHA • PALO ALTO • PHILADELPHIA • PHOENIX PITTSBURGH • SACRAMENTO • SALT LAKE CITY • SAN BERNARDINO • SAN DIEGO • SEATTLE • ST. LOUIS • WASHINGTON, D.C.

ATHENS • CANBERRA • DUSSELDORF • FRANKFURT • THE HAGUE • HAMBURG JOHANNESBURG • LONDON • MELBOURNE • MEXICO CITY (REGAL ELECTRONICA DE MEXICO, S.A.) • MILAN • MONTREAL • MUNICH • OSLO • OTTAWA PARIS • SAVYON • STOCKHOLM • STUTTGART • SYDNEY • TOKYO (C. ITOH ELECTRONIC COMPUTING SERVICE CO., LTD.) • TORONTO • ZURICH

8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440

Pub. No. 60101600B

tho in U.S.A.