All SYSTEM/360 *functional characteristics having programming significance are completely and concisely described.*

The description, which is formal rather than verbal, is accomplished by a set of programs, interacting through common variables, used in conjunction with auxiliary tables.

The language used in the programs involves operators and notation selected from mathematics and logic, together with additional operators and conventions defined to facilitate system description.

Although the formal description is complete and self-contained, text is provided as an aid to initial study.

Examples to illustrate the application of the formal description are given in an appendix.

A formal description of SYSTEM/360

by A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth

This paper presents a precise formal description of a complete computer system, the IBM SYSTEM/360. The description is functional: it describes the behavior of the machine as seen by the programmer, irrespective of any particular physical implemention, and expressly specifies the state of every register or facility accessible to the programmer for every moment of system operation at which this information is actually available.

The work is based on the SYSTEM/360 manual[1] and on interpretations and revisions furnished by the system architects, some of whom have assisted in a thorough audit of the present description.[2]

The formal description comprises a set of programs and auxiliary tables, all of which are grouped for easy cross-reference beginning on page 240.[3] These provide a complete, self-contained description of the system which, after some familiarity with the notation and programs is gained, will be found more convenient for reference than verbal description. An appendix furnishes examples of reference use of the material. The remainder of the text is designed primarily as a guide and aid in the initial reading of the programs.

The second and third sections describe the central processing unit and the input/output system, respectively. The first section defines the notation employed and illustrates its use. Further illustrations may be found in the references 4, 5, 6.

The notation

A system is described by a collection of interacting programs, each program consisting of a list of statements executed in an alterable, but specified, sequence. The interaction between programs occurs through shared variables and through direct alteration of the sequence in one program by another.

Statements are of two major types, called *specification* and *branch*. A specification statement incorporates a left-pointing arrow and implies that its execution respecifies the value of the variable to the left of the arrow by the value of the expression to the right of the arrow. Thus, if x and y have values 3 and 4, respectively, the execution of the specification statement

$$z \leftarrow x + y$$

sets the variable z to the value 7.

The statements of a program are numbered serially from zero and are executed in serial order except that the execution of a branch statement may interrupt the sequence. One type of branch has the form

$$\rightarrow \alpha, n$$

and implies that Program α immediately executes its statement numbered n (to be referred to hereafter as *line n*) and proceeds from there. If the foregoing branch statement occurs in Program β (where $\beta \neq \alpha$), the sequence in Program β is not affected. For example, the execution of line 1 of *IPL*, the initial program load program (page 259) resets *CPU*, the central processing unit program, to its line 0 but does not affect the sequence in *IPL*, which continues to line 2.

In the more familiar case the branch statement $\rightarrow \alpha, n$ occurs in Program α, and the program name is elided to yield the form $\rightarrow n$. Line 11 of the *CPU* program furnishes an example, causing a branch to one of lines 12, 14, 13, 17, or 19 according to which component of the vector (12, 14, 13, 17, 19) is selected by the index n_2. The statement itself completely specifies the branch, and the broken arrows from line 11 to each of its potential successors are provided merely as a graphic aid to comprehension.

Solid arrows from line to line are, however, used as an alternative specification of branching within a given program. An unlabelled arrow denotes an unconditional branch; thus line 12 of the *CPU* program is invariably followed by line 16. A solid arrow labelled with a relation $\mathcal{R}$ and emanating from a line containing a statement of the form

$$x : y$$

implies that the branch arrow is followed if and only if the relation $x\mathcal{R}y$ holds. Thus line 40 of the *CPU* is followed by line 1 if $p_{14} = 0$, and by line 25 if $p_{14} \neq 0$. In following branch arrows that cross, the path does not change direction at a crossing.

Table 1 Notation

	Operation	*Notation*①	*Definition*②
OPERANDS	Scalar	x	
	Vector	x	$x \equiv x_0, x_1, \cdots, x_{(\nu x)-1}$ $\quad \nu x \equiv$ number of components
	Matrix	X	$X \equiv \begin{bmatrix} X_0^0 & \cdots & X_{(\nu X)-1}^0 \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ X_0^{(\mu X)-1} & \cdots & X_{(\nu X)-1}^{(\mu X)-1} \end{bmatrix}$ $\begin{cases} X^i \equiv i\text{th row vector} \\ X_j \equiv j\text{th column vector} \\ \mu X \equiv \text{number of rows} \\ \nu X \equiv \text{number of columns} \end{cases}$
LOGICAL and NUMERICAL	Arithmetic	$+ \; - \; \times \; \div$	Usual definitions
	Absolute value	$z \leftarrow \lvert x$	$z \equiv$ maximum of x and $-x$
	Floor	$k \leftarrow \lfloor x$	$k \leq x < k+1$
	Ceiling	$k \leftarrow \lceil x$	$k-1 < x \leq k$; k, m, n, q integers
	Residue modulo m	$k \leftarrow m \mid n$	$n \equiv (m \times q) + k$, $0 \leq k < m$
	And	$w \leftarrow u \wedge v$	$w \equiv 1$ if and only if $u \equiv 1$ and $v \equiv 1$
	Or	$w \leftarrow u \vee v$	$w \equiv 1$ if and only if $u \equiv 1$ or $v \equiv 1$
	Negation	$w \leftarrow \tilde{u}$ or $w \leftarrow \sim u$	$w \equiv 1$ if and only if $u \equiv 0$
	Relation	$w \leftarrow x \mathcal{R} y$	$w \equiv 1$ if and only if $x \mathcal{R} y$ is true
			All operations are extended component-by-component to dimensionally compatible vectors and matrices. If one of the operands is a scalar, it is treated as a vector or matrix of appropriate dimension whose components are all equal. Examples: $z \leftarrow x + y$; $z \leftarrow x \times y$; $W \leftarrow U \wedge V$; $w \leftarrow x \neq y$; $w \leftarrow x < y$
REDUCTION	Reduction	$z \leftarrow \odot/x$	$z \equiv x_0 \odot x_1 \odot \cdots \odot x_{(\nu x)-1}$
	Row reduction	$z \leftarrow \odot/X$	$z_i \equiv \odot/X^i$
	Column reduction	$z \leftarrow \odot//X$	$z_j \equiv \odot/X_j$
	Matrix product	$Z \leftarrow X \overset{\odot_1}{\odot_2} Y$	$Z_j^i \equiv \odot_1/X^i \odot_2 Y_j$
			$\odot$ is any binary operator or relation. The case $X \overset{+}{\times} Y$ is the ordinary matrix product. The expressions $X \overset{\odot_1}{\odot_2} y$, $x \overset{\odot_1}{\odot_2} Y$, and $x \overset{\odot_1}{\odot_2} y$ are treated as in matrix algebra. Thus $x \overset{+}{\times} y$ is the scalar product.
BASE VALUE	Base 10 value	$z \leftarrow 10 \perp x$	z is the base-10 value of the vector x
	Base 2 value	$z \leftarrow \perp u$ or $z \leftarrow \perp U$	z is the base-2 value of the vector u; $z_i \equiv \perp U^i$
	Representation		
	base 10	$z \leftarrow 10(n) \top j$	$\nu z \equiv n$ and $10 \perp z \equiv 10^n \mid j$
	base 2	$u \leftarrow (n) \top j$	$\nu u \equiv n$ and $\perp u \equiv 2^n \mid j$
SELECTION	Catenation	$z \leftarrow x, y$	$z \equiv x_0, x_1, \cdots, x_{(\nu x)-1}, y_0, y_1, \cdots, y_{(\nu y)-1}$
	Row catenation	$Z \leftarrow x \oplus y$	$Z_0 \equiv x$; $Z_1 \equiv y$
	Compression		
	vector	$z \leftarrow u/x$	z obtained by suppressing from x each x_i for which $u_i \equiv 0$
	row	$Z \leftarrow u/X$	$Z^i \equiv u/X^i$
	column	$Z \leftarrow u//X$	$Z_j \equiv u/X_j$
	row list	$z \leftarrow \mathcal{E}/X$	$z \equiv X^0, X^1, \cdots, X^{(\mu X)-1}$
	Row list expansion	$X \leftarrow \mathcal{E}(m, n)\backslash z$	$\mu X \equiv m$, $\nu X \equiv n$, and $\mathcal{E}/X \equiv z$. Thus $X \equiv \begin{bmatrix} z_0 & \cdots & z_{n-1} \\ z_n & \cdots & \cdots \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdots & & z_{(m \times n)-1} \end{bmatrix}$
	Mask	$z \leftarrow /x; u; y/$	$\bar{u}/z \equiv \bar{u}/x$; $\quad u/z \equiv u/y$.
	Indexing	$z \leftarrow x_m$	$z_i \equiv z_{m_i}$
		$Z \leftarrow X^m$	$Z^i \equiv X^{m_i}$
		$Z \leftarrow X_m$	$Z_j \equiv X_{m_i}$
	Maximum prefix	$w \leftarrow \alpha/u$	$w \equiv \alpha^j(\nu u)$ and j is maximum for which $\wedge/w/u \equiv 1$
SHIFTING	Left rotation	$z \leftarrow k \uparrow x$	$z_i \equiv x_j$; $j \equiv (\nu x) \mid i + k$
	Right rotation	$z \leftarrow k \downarrow x$	$z_i \equiv x_j$; $j \equiv (\nu x) \mid i - k$; cyclic left (right) rotation of x by k places.
	Left shift	$z \leftarrow k \overset{\circ}{\uparrow} x$	$z \equiv \bar{\omega}^k \wedge k \uparrow x$
	Right shift	$z \leftarrow k \overset{\circ}{\downarrow} x$	$z \equiv \bar{\omega}^k \wedge k \downarrow x$; left (right) shift bringing zeros into evacuated positions
SPECIAL VECTORS	Full	$w \leftarrow \epsilon(n)$	$w_i \equiv 1$
	Characteristic	$w \leftarrow \epsilon^j(n)$	$w_i \equiv (\vee/i = j)$
	Prefix	$w \leftarrow \alpha^j(n)$	$w_i \equiv (i < j)$
	Suffix	$w \leftarrow \omega^j(n)$	$w_i \equiv ((n-i) \leq j)$
	Random	$w \leftarrow ?$	$w \equiv 0$ or 1 (arbitrary)
		$w \leftarrow ?(n)$	$w_i \equiv 0$ or 1
		$w \leftarrow ?^j(n)$	$w_i \equiv 0$ or 1 but $+/w \equiv j$
	Interval	$z \leftarrow \iota^j(n)$	$z \equiv j, j+1, \cdots, j+n-1$
			Dimension of w is n. The n may be omitted if it is clear from context.

Table 1 (Continued)

① The notation for each operation is only the portion to the right of the specification arrow; the variable to the left facilitates definition.

② Throughout this paper elementary operations occurring in compound expressions are, except as indicated by parentheses, executed in strict order from right to left. In this table the symbol ≡ is used instead of = to denote equality, since the latter denotes an operator of the class $\mathfrak{R}$.

③ The following arguments are used in the examples:

$a \equiv 7, -6, 5, -4, 3$
$b \equiv 3, 2, 1$
$c \equiv 0, 0, 1, 1, 0, 1, 1, 1$
$n \equiv$ any logical vector ($n_i \equiv 0$ or 1)
$p \equiv 1, 0, 1, 0, 1$
$q \equiv 1, 0, 1$
$r \equiv +, -, \times, \div$

$$A \equiv \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix} \qquad P \equiv \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

*Examples*③

$a_1 \equiv -6$
$r_0 \equiv +$
$2 \times b \equiv b \times 2 \equiv 6, 4, 2$
$|-3.6 \equiv 3.6$
$\lfloor 3.6 \equiv 3$
$\lceil 3.6 \equiv 4$
$7 \mid 19 \equiv 5$
$p_0 \wedge q_0 \equiv 1$
$p_0 \vee q_0 \equiv 1$
$\tilde{q}_0 \equiv \sim q_0 \equiv 0$
$a_4 \leq b_0 \equiv 1$

$b + q \equiv 4, 2, 2$
$b\, r_0\, q \equiv 4, 2, 2$
$b \times b \equiv 9, 4, 1$
$|q - 3 \equiv 2, 3, 2$
$\lfloor -3.6 \equiv -4$
$\lceil -3.6 \equiv -3$
$7 \mid 21 \equiv 0$
$q \wedge P^0 \equiv 0, 0, 1$
$q \vee P^0 \equiv 1, 1, 1$
$\sim q \wedge P^0 \equiv \tilde{q} \vee \tilde{P}^0 \equiv 1, 1, 0$
$b < A_1 \equiv 0, 0, 1$
$2 < A_1 \equiv 0, 0, 1$

$A_0 \equiv 0, 1, 2$
$P^1 \equiv q$
$a - A^0 \equiv 7, -7, 3, -7, -1$
$|a - A^0 \equiv 7, 7, 3, 7, 1$
$\lfloor a \div 2 \equiv 3, -3, 2, -2, 1$
$\lceil a \div 2 \equiv 4, -3, 3, -2, 2$
$2 \mid a \equiv 2 \mid a \times a \equiv 1, 0, 1, 0, 1$
$q \wedge 2 \mid b \equiv q \equiv A_1^0 \wedge q$
$q \vee 2 \mid b \equiv q$
$q_0 \neq p \equiv \tilde{p} \equiv \sim q_0 = p$

$+/a \equiv 5$
$+/A \equiv 10, 15, 20$
$+//A \equiv 3, 6, 9, 12, 15$
$r_2/a \equiv 2520$
$\neq/P \equiv 0, 0$
$\neq//P \equiv 1, 1, 0$
$\wedge/n \equiv \sim\vee/\sim n$
$\wedge/p \equiv \sim \vee/\sim p$
$\neq/p \equiv 2 \mid +/p \equiv \sim =/\tilde{p}$
$\times/A \equiv 0, 120, 720$
$\neq/n \equiv \sim = /\sim n$

$$P \overset{+}{\times} A \equiv \begin{bmatrix} 3 & 5 & 7 & 9 & 11 \\ 2 & 4 & 6 & 8 & 10 \end{bmatrix}$$

$P \triangleq q \equiv \sim P \overset{\vee}{\neq} q \equiv 0, 1$

$10 \perp b \equiv 321$
$\perp P \equiv 3, 5$
$10(3) \top 144 \equiv 1, 4, 4$
$(4) \top 13 \equiv 1, 1, 0, 1$

$\perp q \equiv 5$
$\perp 2 \mid A \equiv 10, 21, 10$
$10(2) \top 144 \equiv 4, 4$
$(3) \top 13 \equiv 1, 0, 1$

$c_{\perp \omega^3/c} \equiv c_7 \equiv 1$
$10(4) \top 144 \equiv 0, 1, 4, 4$
$(5) \top 13 \equiv 0, 1, 1, 0, 1$

$P \equiv (0, 1) \oplus (1, 0) \oplus (1, 1) \equiv \epsilon^1(2) \oplus \epsilon^0(2) \oplus \epsilon^{0,1}(2)$
$A \equiv \iota^0(3) \oplus \iota^1(3) \oplus \iota^2(3) \oplus \iota^3(3) \oplus \iota^4(3)$
$p, b \equiv 1, 0, 1, 0, 1, 3, 2, 1$
$q/b \equiv 3, 1$
$\nu q/b \equiv +/q \equiv 2$
$(2 \mid A^0)/A^1 \equiv 2, 4$
$(a > 0)/a \equiv 7, 5, 3$

$$p/A \equiv \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \equiv A_{0,2,4} \qquad q//A \equiv \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix} \equiv A^{0,2}$$

$E/A \equiv 0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 2, 3, 4, 5, 6$

$$E(4, 2)\backslash a, b \equiv \begin{bmatrix} 7 & -6 \\ 5 & -4 \\ 3 & 3 \\ 2 & 1 \end{bmatrix}$$

$/a; p; A^0/ \equiv 0, -6, 2, -4, 4$
$/b; p_0; q/ \equiv q$
$a_{0,3,1} \equiv 7, -4, -6$
$b_q \equiv 2, 3, 2$
$a_{1,2(2)} \equiv 5, -4$
$\alpha/\tilde{c} \equiv 1, 1, 0, 0, 0, 0, 0, 0$
$\alpha/p \equiv 1, 0, 0, 0, 0$
$+/\alpha/\bar{n} \equiv$ number of leading zeros in $n \equiv \nu n$ or index of leading 1 in n

$2 \uparrow A^1 \equiv 3, 4, 5, 1, 2$
$3 \downarrow A^1 \equiv 3, 4, 5, 1, 2$
$2 \overset{\circ}{\uparrow} A^1 \equiv 3, 4, 5, 0, 0$
$2 \overset{\circ}{\downarrow} A^1 \equiv 0, 0, 0, 1, 2$

$2 \uparrow p \equiv 1, 0, 1, 1, 0$
$(2 \downarrow \alpha^3(8))/c \equiv 1, 1, 0$

$\epsilon(4) \equiv 1, 1, 1, 1$
$\epsilon^{1,2,4}(5) \equiv 0, 1, 1, 0, 1$
$\alpha^2(5) \equiv 1, 1, 0, 0, 0$
$\omega^2(5) \equiv 0, 0, 0, 1, 1$
$\iota^2(5) \equiv 2, 3, 4, 5, 6$
$?(2) \equiv 0, 0$ or $0, 1$ or $1, 0$ or $1, 1$
$?^2(3) \equiv 0, 1, 1$ or $1, 0, 1$ or $1, 1, 0$

$\tilde{\epsilon}(4) \equiv 0, 0, 0, 0$
$\tilde{\epsilon}^{A^2}(8) \equiv 0, 0, 1, 1, 1, 1, 1, 0$
$\alpha^2(5)/a \equiv 7, -6$
$\omega^3/\tilde{\omega}^2/c \equiv 1, 0, 1$
$c_{\iota^2(5)} \equiv 1, 1, 0, 1, 1$
$(0 \leq \perp ?(k)) \wedge ((\perp ?(k)) < 2^k) \equiv 1$

$\tilde{\epsilon}^2(4) \equiv 1, 1, 0, 1$
$(1 \downarrow \alpha^2(5))/a \equiv -6, 5$

For brevity, branch and specification statements are sometimes combined in the form

$$z : y \leftarrow x$$

implying that y is specified by x and is then compared with z to determine a branch.

The operand types and elementary operators employed are defined in Table 1, together with examples. These examples permit Table 1 to be studied by itself before proceeding to the more complex expressions in the programs.

In a compound expression such as

$$(x \times y + z) - q \wedge \sim r$$

the order of execution of the elementary operations is determined by parentheses in the usual way and any remaining ambiguity is resolved by proceeding from right to left, *no priorities* being accorded to multiplication or any other operator. This convention applies, in particular, to Table 1 itself.

The convenience of the right-to-left order of execution is indicated by the paucity of parentheses in the programs. As an example, consider line 5 of the *CPU* program:

$$\omega^{24}/\boldsymbol{p} \leftarrow (24)\top 2 + \bot\omega^{24}/\boldsymbol{p}.$$

Proceeding from right to left, the last 24 bits of the vector $\boldsymbol{p}$ are selected, the base-2 value of the resulting vector is taken and added to 2, and finally the 24-bit representation of the sum respecifies the last 24 bits of $\boldsymbol{p}$.

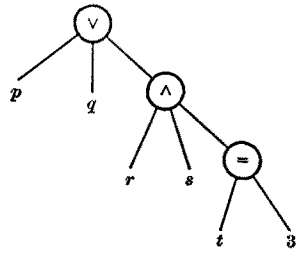
In complex logical expressions, the right-to-left convention permits convenient interpretation from left to right; for example, the expression

$$p \vee q \vee r \wedge s \wedge t = 3$$

is interpreted as indicated by the tree shown in Figure 1.

No elision of operator symbols is permitted; consequently, the names of variables can, without ambiguity, consist of any number of alphabetic characters, including spaces. For brevity, single characters are used for all variables except for those which occur infrequently, such as the panel switches occurring in *CP*, the control panel program. All variable names are in italics: lightface lower case for scalars, boldface lower case for vectors, and boldface upper case for matrices. Literal alphabetic values (e.g., "stop" and "operate" on lines 12 and 13 of the *CP* program) are denoted by roman type.

Figure 1 Tree representation of a compound statement

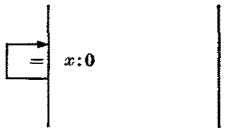


In order to reduce the number of variable symbols assigned to local counters and other intermediate variables, certain symbols (including i, j, k, $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$) will be used only as local variables, i.e., the value of any such variable will be relevant only in the program in which it occurs and will not affect or be affected by any variable of the same name occurring in another program. A family of matrices will be denoted by a pre-superscript as, for

example, ^{i}J, in line 20 of $MAC^{i}(j; k)$, the memory access program.

The system description comprises a set of tables and programs. Programs include *system programs* and *defined operations*. All system programs operate concurrently and continuously, with precisely one line active in each program at all times. This active line will often be a *dwell* of the form shown in Figure 2. The program breaks from the dwell only when the variable x is set to a nonzero value by some other program in the system.

Figure 2 Dwell



A defined operation is a program which operates only when invoked by some other program. It can be distinguished from a system program by the presence of exit arrows. When called, a defined operation constitutes the active line of the program it is serving, and will itself have precisely one line active at a time until an exit is reached.

Variables occurring in the name of a defined operation (for example, j and k in the case of $MAC^{i}(j; k)$) are dummy variables whose values are determined by the values of the variables occurring in any particular use of the defined operation. Thus, the performance of line d3 of *EXC* (that is, MAC^{9} $(a_2, 2, f, d; \omega^{16}/R^{a_1})$) executes MAC^{9} with $j = (a_2, 2, f, d)$ and with $k = \omega^{16}/R^{a_1}$. A study of *MAC* will show that this causes 2 bytes of *d*ata, *f*etched from the memory location starting at a_2, to be transferred to the last sixteen positions of the general register whose index is a_1.

All components of the formal system are listed in Table 2. The text and figures, and the tables not listed, are intended for exposition only.

Table 2 Components of formal description

System programs

		page			page
BMT	Burst-mode timer	261	*HFC*^{e}	Hardware failure in channels	261
CH^{e}	Channels	261	*IOIE*	I/O interruption entry	261
CP	Control panel	259	*IPL*	Initial program load	259
CPU	Central processing unit	259	*MCIE*	Machine check interruption entry	259
EIE	External interruption entry	259	*T*	Timer	259
EP	Emergency pull	259	*TOL*^{e}	Time-out limiters	261
ES	External signals	259	*TU*	Timer update	259

Defined operations

		page		page
DELAY		*	*MODEL-DEPENDENT RESET*	*
DIAGNOSE		*	*POWER-OFF SEQUENCE*	*
EXC	Instruction execution	240	*POWER-ON SEQUENCE*	*
$MAC^{i}(j; k)$	Memory access	259	*RESET*	259
MALFUNCTION RESET		*	*SYSTEM STOP*	*

* These operations are not detailed in this description.

Tables

		page
Table 3	System reference table for programs and variables: symbol and dimension columns only	250
Table 5	Navigation matrix N and reference table: first 11 columns (navigation matrix) only	256
Table 6	Operation decoding matrix O: numerical entries only	258

The central processing unit

The central processing system comprises the nine system programs shown on page 259, and the defined operations listed in Table 2. An overall view of the system can be gained from Table 3, which lists all variables occurring in the programs, indicates the meaning or significance of each symbol and the dimensions of each vector or matrix variable, and provides reference to every program statement (or entire program) in which each variable is specified or used. Since the range of each variable is readily deduced from its use in the programs, ranges are not explicitly specified in the table. Most of the variables are *logical* (i.e., have the range 0, 1), but some, such as the components of a, m, and N, are not. The ranges of the components of some arrays (e.g., N) are not all alike.

Table 4 Central processing unit program segments

Functions	*CPU lines*
Initial-program-load dwell	0
Instruction fetch	1–7
Instruction interpretation	8–10
Effective address calculation	11–19
Instruction execution	20–21
Entry of program interruption	22–24
Interruption service	25–33
Single-step, stop, and wait tests	34–36

Although Table 3 is designed for reference, it will also repay careful initial study. It shows, for example, that the memory M behaves as a 9-bit wide memory of at most 2^{24} 8-bit *bytes* with parity, that each block of 2^{11} bytes of M may be protected by a 4-bit protection key K^i, and that 16 single-word general registers R and 4 double-word floating-point registers F are provided. All of the more detailed information on the treatment of the variables (such as the formats used in instruction addresses) is immediately available in the programs.

CPU program

The core of the system is the *CPU* program, which describes the sequencing and execution of instructions and the servicing of interruptions. The functional segments of this program are listed in Table 4.

instruction fetch (lines 1—7)

An instruction may be 1, 2, or 3 half-words long and is fetched by lines 3–7 (using the *MAC* operation detailed on page 259), two bytes at a time, from memory locations specified by the *instruction counter* represented by ω^{24}/p, the last 24 bits of the *program status word* p. For each half-word fetched, the *instruction length code* ($p_{32,33}$) is augmented by 1 (line 4) and the instruction address by 2. The sum of the first 2 bits of the first half-word plus 1 determines (on line 7) the number of half-words fetched.

Line 2 enters a specification exception if the protection feature is not installed ($m_0 = 0$) and any of positions 8–11 of p are nonzero. Addressing and specification exceptions are detected by lines 7–8 of the *MAC* operation and are entered (*MAC* line 15) into t_5 and t_6, respectively. Either of these errors subsequently causes a branch from *CPU* line 6 to line 24, skipping the instruction execution phase and any remaining portion of the fetch.

instruction interpretation (lines 8—10)

If the fetch concludes without error, lines 8–9 interpret the instruction by selecting from the navigation matrix N (Table 5) a row N^i to specify the vector n used in the subsequent control of the instruction execution phase. The row of N selected is determined by the particular element of the operation decoding matrix O (Table 6) selected by the 8-bit operation code in the first byte of the instruction. Table 6 displays the mnemonics

used for the instructions as well as the index to N, which is the only formal part of the matrix O. Similarly, Table 5 includes much informal information in addition to the formal specification of the matrix N.

Except for its first two components, the *navigation vector* n is of formal interest only, since the sequences it determines in the *CPU* program (line 11) and in the *EXC* operation are also indicated informally by broken-line arrows and labels. If the 8-bit operation code corresponds to no installed operation, then $n_0 = 0$ and an operation exception exists (line 10); if the operation is privileged ($n_1 = 1$) and if $p_{15} = 1$ (that is, the processor is in the *problem state* as opposed to the *supervisor state*), a privileged operation exception exists.

effective address calculation (lines 11—19)

The manner of specifying the operands of an instruction is a function of the *format* of that instruction. Most instructions specify two operands; the *effective addresses* of the first and second operands are computed as a_1 and a_2, respectively. Normally, the addresses are used to select two general registers (RR format), a register and storage in memory (RS), a register and storage with a second register for indexing (RX), two areas in storage (SS) or storage and *immediate data* from the instruction itself (SI), all as indicated in Table 7. However, since many exceptions exist, Table 7 should be considered as a guide only, all operands being explicitly defined in *EXC*. Some instructions in the RS format use a third address a_3. A terminal R and a terminal I in instruction mnemonics usually indicate the RR and SI formats, respectively.

For instructions in the SS format, the lengths of the operands are defined by line 17; $l_1 + 1$ and $l_2 + 1$ are the lengths (in bytes) of the first and second operands, and $l_0 + 1$ is the length for some instructions in which a single length is required. Again, the matrix N (and hence n) provides (via *CPU* line 11) the formal specification of the format used by each instruction.

The calculation of the effective addresses is straightforward and will be discussed only for the RX and RS cases. In the latter case, the second operand address a_2 is determined by the second half-word of the instruction and is formed by adding to the value of its last 12 bits ($\perp\omega^{12}/I^1$) the value of the general register selected by its first 4 bits ($\perp R^{\perp\alpha^4/I^1}$), unless the zeroth register is selected ($0 = \perp\alpha^4/I^1$), in which case zero is added. Finally, the residue of this sum modulo 2^{24} specifies a_2. The first and third operand addresses are determined by the values of groups of 4 bits in the first half-word, $\perp\alpha^4/\omega^4/I^0$ and $\perp\omega^4/I^0$, respectively. The RX format differs only in that the *index* quantity contained in the general register selected by the last 4 bits of the first half-word is (again, unless the zeroth register is selected) also added to the sum used to determine a_2, and in that the third address is not applicable.

instruction execution (lines 20—21)

The calculation of the effective address is immediately followed by a use of the defined operation *EXC* on line 20. This operation begins with a branch from line 0 to line n_3, that is, to the segment

of the *EXC* operation appropriate to the particular instruction being executed. For example, if the instruction is LR, then n_3 is specified by N_3^{71}, and n_3 = d0. This is also indicated informally by the "LR" written to the left of line d0.

To continue the example, line d0 specifies the value of general register R^{a_1} by the value of R^{a_2}, and the exit arrow on the same line indicates that *EXC* is complete.

The execute instruction (EX) beginning on line b0, furnishes a somewhat more complex example of the execution phase. Except that the length code $p_{32,33}$ and the instruction counter ω^{24}/p are not disturbed, lines b0–3 are like the instruction fetch (*CPU* 3–7) and therefore load the instruction register with the data beginning at byte address a_2 in memory. Since n_3 = b0 in this case, line 21 of *CPU* subsequently causes (unless a program interruption has been entered in t, either on line b5 or by use of *MAC* on line b1) a branch to line 8, thus interpreting and executing the instruction just fetched by the EX instruction, without disturbing the instruction counter. If this *subject instruction* is itself an EX, it has the operation code 0100 0100, and line b5 therefore enters an execute exception and line 21 does not cause a branch, thus aborting the execution of the subject EX instruction. The phrase $\bar{t}_5 \wedge \bar{t}_6$ in line b5 prevents a spurious execute exception if addressing or specification exceptions have already occurred.

If the first operand address of EX is nonzero, then line b4 will *or* the last byte of general register R^{a_1} with the last byte of the first half-word of the subject instruction. This permits a programmer to specify such parameters as length, index, or immediate data in the subject instruction indirectly via R^{a_1}.

entry of program interruption (lines 22—24)

If no program interruption has been generated by the preceding fetch and execution, each component of t will be zero, and lines 23 and 24 will be skipped. Otherwise line 24 presents the interruption by setting h_1 to 1 and entering an appropriate code in positions 16–31 of p. Thus, if the interruption is occasioned by an addressing exception, t_5 = 1, the expression t/ι^0 yields 5, and its base 2 representation (namely, 0000 0000 0000 0101) is entered in p. If multiple causes of program interruption occur, then one of the appropriate codes will be selected at random and stored in p. For example, if $t_5 = t_6 = 1$, then $t/\iota^0 = (5, 6)$ and $?^1/(5, 6)$ selects one of 5 and 6.

If the cause of the program interruption entry is neither t_4 nor t_5 (that is, neither a protection nor an addressing exception), then the instruction length code $p_{32,33}$ is unchanged by line 23; otherwise line 23 *may* set the length code to zero.

Other types of interruptions are entered in other components of h by other programs (listed in the reference column of Table 3) whose details will be considered later.

interruption service (lines 25—33)

An interruption service places an appropriate code in bits 16–31 of p, stores p in memory at one of five fixed locations determined by the type of interruption (line 31), and finally respecifies p (line 32) from one of five other fixed memory locations. Since

this respecifies the instruction counter ω^{24}/p, it occasions an alteration in the normal sequence of instructions.

Interruption servicing is skipped if the expression

$$\vee/h \wedge (0, 1, 1, p_7, (\alpha^7/p) \times B_8)$$

(line 25) has the value zero. The vector $(0, 1, 1, p_7, (\alpha^7/p) \times B_8)$ is therefore a mask which causes certain components of h to be ignored. Thus h_0 (machine check interruption) is always ignored at this point, h_1 and h_2 (program and supervisor call interruptions) never are, h_3 (external) is ignored if $p_7 = 0$, and h_4 (I/O) is ignored unless an interruption is being presented (in B_8) by one of the channels for which the corresponding mask bit in α^7/p is set to 1.

If there is an acceptable interruption, then line 26 determines h as the index (in h) of the interruption to be serviced. The queue discipline in h is not first-in first-out but rather strict priority (according to position in h) of the unmasked components of h presented.

If $h = 0$, 3, or 4 (machine check, external, or I/O), then g_0 is set to signal the appropriate program (*MCIE*, *EIE*, or *IOIE*) to enter the appropriate data in bits 16–31 of p, while the *CPU* program dwells on line 28. Then the accepted interruption indication is reset on line 29. In certain situations, an I/O channel is unable to present an indicated interruption at the time it is accepted by the *CPU*; this is signalled by the *IOIE* program by setting g_1 to 1 before resetting g_0 to 0, thus aborting the interruption by the branch on line 30. In any event, line 25 is executed again so that all outstanding unmasked interruptions are serviced in turn before continuing to line 34.

Whenever the program status word p is loaded from memory, bits 16–33 (the interruption code and instruction length code) are indeterminate as shown by *CPU* 33, *IPL* 8, and *EXC* a26.

single-step, stop and wait tests (lines 34—36)

If the *operating state* is set to "stop", then the *CPU* dwells on line 35 with the *manual light* "on". Events which set the *operating state* are listed in Table 3. These include line 34 of the *CPU* which sets it to "stop" if the rate switch is not set to "process", e.g., if set to "single-step".

Subsequent to the dwell on line 35, the behavior is determined by the wait bit p_{14}; if $p_{14} = 0$, the *wait light* is turned "off" and the next instruction fetch is begun (branch from line 36 to line 1); if $p_{14} = 1$, the wait light is turned "on" and the interruption service phase is entered at line 25. Hence if $p_{14} = 1$, the *CPU* "waits", executing no further instructions until an external or I/O interruption or initial program load replaces p with a value such that $p_{14} = 0$.

initial-program-load dwell (line 0)

The computer memory is initially loaded by the *IPL* program which resets (line 1) the *CPU* to the dwell at line 0, where it remains until the loading (effected by an I/O channel) is complete, as signalled by the setting of the variable *ipl* to zero. The branch to line 25 shows that when loading is complete, operation is resumed with interruption service rather than with the instruction fetch.

memory access operation

The MAC operation, which occurs in the instruction fetch and throughout the EXC operation, serves to fetch from or store in memory a specified number of bytes beginning at a specified address. Because it incorporates certain tests and other functions, it warrants a detailed scrutiny.

The general form of the MAC operation is MAC^i $(j; k)$, where i specifies one of nine identical but independent MAC programs, $i = 9$ for the CPU memory-access, $i = 8$ for the interval timer, and $i = 0$–6 for the channels; where k is the vector involved in the transfer to or from memory; and where j is a four-component vector specifying the address in memory (j_0), the number of bytes transferred (j_1), the performance of a fetch (j_2 = f) or store (j_2 = s), and the type of address being treated (j_3 = d for data address, i for instruction address, g for a machine-generated [i.e., fixed] address, and h for hold, which prevents any of the other MAC^i programs from operating until the current MAC^i program has been used again with $j_3 \neq$ h).

memory access priority

Since the several MAC^i programs all use the same memory, they must observe a queue discipline. It is controlled by the queue vector q and the request vector r. When MAC^i is invoked by any system program (e.g., MAC^9 in CPU line 3), then a request for service is entered (line 0) by setting r_i to 1. If the queue is empty, the request is also entered in q_i. In any case, the MAC^i program dwells on line 1 until i is recognized as the "first" nonzero entry in the queue. The queue discipline is not first-in first-out, but is in order by position in a permutation of q specified by the vector *rank*, which gives priority according to the index i, except that $i = 0$ (the multiplexor channel) may be assigned out of order. This implies that the channels have priority over the interval timer which has priority over the CPU.

Any request for service which is not entered directly in q on line 0 is entered from r by line 24. If $j_3 \neq$ h, then q is respecified by r (with r_i already set to zero by line 2) except that a CPU request is entered in q_9 only if $w = 0$. The variable w is controlled (line 2) only by MAC^8, the timer update memory-access, and it remains at 0 or 1 according to whether the last use of MAC^8 was for a store or for a fetch. This excludes the CPU from memory during the updating of the interval counter (TU lines 1, 3) and prevents the inadvertent overwriting (by TU line 3) of a new setting of the interval timer counter by a CPU "store" instruction. However, the use of memory by I/O is not excluded by w.

If j_3 = h, line 24 leaves q unchanged and therefore prevents any other MAC from being executed until after the next use of the same MAC with $j_3 \neq$ h. The use of j_3 = h occurs only in the instruction TS (test and set) as follows (EXC lines a29, 30)

$MAC^9(a_1, 1, f, h; u)$
$MAC^9(a_1, 1, s, d; \epsilon(8))$.

Thus, the addressed byte is set immediately following a test of its value, before any other access to memory can occur.

address modification and comparison

The first 2^{12} bytes of memory are normally used for special purposes by the machine (e.g., 48, 40, 32, 24, and 56 are used to store $\boldsymbol{p}$ on line 31 of the *CPU* program) and by the supervisory program. Any address j_0 in this region is automatically prefixed (*MAC* lines 3,4) by a 12-bit prefix selected from either the *main* or *alternate* prefix (having wired-in values) according to the setting of the *prefix trigger*, which is set during the initial program loading (*IPL* line 3). It must be emphasized that *every* memory address (including those used in I/O) below 2^{12} is modified in this way, although this fact will not be referred to again in the text.

The effect of line 6 is to set the *operating state* to "stop" if all but the last bits of the ***address switch*** on the console agree with the specified address j_0, but line 6 is skipped if the *address compare switch* is on "normal" or if it is set to "instruction" and j_3 does not specify that j_0 is an instruction address.

exceptions

Lines 7–11 specify three types of exception conditions. A specification exception is indicated (line 7) if the address j_0 is not divisible by the number of bytes j_1. An addressing exception is indicated (line 8) if the address j_0 exceeds the size of memory, and in this event the address j_0 is respecified for all subsequent purposes by an arbitrary value within the range of the memory (line 9). A protection exception is indicated (lines 10,11) if the machine has the protection feature ($m_0 = 1$) and the memory-access is of the store data type ($j_{2,3}$ = s, d) and the keys u_0 and u_1 differ and neither is zero, where u_0 is the protection key in the memory bank addressed by j_0, and u_1 is either the protection key in $\boldsymbol{p}$ (in the case of a *CPU* memory-access) or the protection key in the appropriate channel address word (in an I/O memory-access).

If there are multiple exceptions, line 13 chooses some one of them to enter into $\boldsymbol{s}$ and thence into the program exceptions vector $\boldsymbol{t}$ (in the case of *CPU* or timer memory-access) or in the channel status word $\boldsymbol{S}^{\iota}$ (in the case of a channel). If the error is a specification exception ($s_0 = 1$) or if the operation is a store, then line 17 skips all further steps except to respecify the queue (line 24). Otherwise, i.e., on a fetch operation with an addressing exception, in which the address j_0 has been respecified arbitrarily (line 9), the (meaningless) fetch from memory proceeds.

If no exception occurs, or if a machine-generated address is being treated, line 12 skips the entry of exceptions, branching directly to line 18 which chooses line 23 for store and 19 for fetch. Line 23 stores the specified vector $\boldsymbol{k}$ together with the appropriate parity bits in the specified rows of $\boldsymbol{M}$. Line 19 fetches the appropriate rows of $\boldsymbol{M}$ to specify the matrix ${}^{\iota}\boldsymbol{J}$. The row list of the last 8 columns of ${}^{\iota}\boldsymbol{J}$ specifies $\boldsymbol{k}$, and the parity bits are available (e.g. to the channel) in ${}^{\iota}J_0$. In the case of a *CPU* or a *TU* memory-access, line 22 checks the parity of the data fetched from memory and signals an error by setting component 9 or 8 of the machine failure vector $\boldsymbol{f}$.

other central processing system programs

The details of the control panel, the interruption entries, the interval timer, the initial program load, and the *EXC* operation will now be treated in that order. At this point, however, the reader should be equally prepared to approach them in any other desired order. It must be re-emphasized that all system programs, including the *CPU* program, run concurrently.

control panel and emergency pull programs

The normal dwell of the *CP* program (lines 4,5) is broken by depression of the *power-off key* (line 4) or by depression of a console button b_i or by a pulse on one of the IPL in-lines e_4 or e_5. Line 6 determines b as the index (with respect to e_4, e_5, b) of the signal to be serviced, line 7 dwells until the signal returns to zero, and line 8 branches to the appropriate program segment.

The last four buttons are ineffective if the machine is in the "operate" state (line 13), but otherwise perform the straightforward functions detailed in lines 15–25. The stop key (line 12) causes the *CPU* to dwell the next time it reaches line 35; the interrupt key sets the *console interrupt* which (as indicated in Table 3) is used in the *EIE* program to enter an external interruption; the IPL in-lines, the load key and the reset key all reset the system (line 9), and all but the latter set *ipl*, releasing the *IPL* program from its normal dwell.

The *EP* program serves to stop the entire system until the *emergency pull switch* is restored, whereupon it sets the *CP* program to line 1, to dwell until the *power-on key* is depressed.

external interruption entry program

The timing signal in-lines ω^6/E^3 are momentary pulses ($\frac{1}{2}$ to 1 microsecond) supplied from some data transmission line, perhaps the output signals E^2 of some co-operating computer. The *ES* program shows the latches ***external signals*** being set by ω^6/E^3. Similarly, *EIE* shows the *timer alarm*, *console interrupt*, and ***external signals*** being entered as interruptions. If any of these signals appear, *EIE* line 0 sets h_3. The *CPU* program will eventually recognize h_3 and set $h = 3$ and $g_0 = 1$ (lines 26,27), thus breaking the dwell on lines 0 and 1 of the *EIE* program. Line 2 then enters the eight external interruptions in p and line 3 resets those signals entered, but does *not* reset any which may have come on during the brief interval since the entry operation on the preceding line. The reset of g_0 on line 4 frees the *CPU* from its dwell on line 28.

machine check interruption entry program

The entry of a machine check (i.e., a hardware failure) is radically different, since, after setting h_0, it preempts the *CPU* by causing a branch to *CPU* line 26. The *MCIE* program also sends a momentary signal on the machine-check out-line e_3 (which may be connected as an input to some co-operating computer), enters zeros as the interruption code in p, performs a diagnosis, stores certain machine registers (referred to as ***cpu status***) in memory beginning at byte 128, and finally resets g_0 (to free the *CPU*), the failure vector f, and any outstanding program and supervisor call interruptions (but not external or I/O interruptions). The machine failure is serviced (line 0) only when the machine check mask $p_{13} = 1$.

Indications of machine failure are entered in *f* from various sources, but only two are shown in this description. One occurs on line 4 of *HFC* for hardware failures in certain I/O channels. The second occurs on line 22 of *MAC*, and detects parity errors in data fetched from memory for the *CPU* or the interval timer. Parity checks are also made on ***R***, ***F***, and ***p***, and at other points, but none of these are shown explicitly in the programs. Their inclusion would merely encumber the description.

timer and timer update programs

The *TU* program is activated by the signal *tick* from the *T* program, but only if the *CPU* is not stopped at the dwell on lines 36–37, the *rate switch* is set to "process", and an RDD (read direct) instruction is not awaiting a hold-in signal on e_2. When activated, line 1 fetches the four bytes representing the interval timer counter, line 2 decreases it (modulo 2^{32}) by an amount dependent on the timer frequency which drives the *T* program, line 3 restores the new count to memory, and line 4 sets the *timer alarm* if the count has passed through zero. The execution of MAC^8 on lines 1 and 3 will, of course, be deferred until all I/O requests for memory access have been serviced. Moreover, no *CPU* memory-access can intervene between *TU* lines 1 and 3 (see *MAC* lines 2, 24).

initial-program-load program

The *IPL* program dwells at line 0 until *ipl* (set only by *CP* line 10) becomes 1, whereupon the *CPU* is forced to its dwell at line 0, and the *load light* is turned "on". The further behavior depends upon which agency (IPL in-lines or load key) initiated the action. If it were the load key, the program would dwell (line 5) awaiting a satisfactory error-free channel-end signal from the I/O unit designated by the setting of the ***load unit switch***, store ***load unit switch*** in memory at address 2, and load ***p*** from memory at address 0. If the initiating agency were one of the IPL in-lines, only the loading of ***p*** would be performed.

If a machine failure occurs in the loading, the *IPL* program dwells on line 9, regardless of the value of the machine check mask; otherwise, line 10 turns the *load light* to "off", the *operating state* to "operate", and *ipl* to zero. The role of I/O in the initial program load will be clarified in the third section; here it will suffice to remark that the I/O channels are reset by the *RESET* occurring on line 9 of the *CP* program, and the I/O unit designated by the ***load unit switch*** then begins to perform a read.

instruction execution operation

The 143 machine instructions described by the defined operation *EXC* are grouped in twelve families. Because there is little interaction with other programs, the interpretation is straightforward, and textual comment will therefore be limited to the more difficult cases.

Table 5 can facilitate reference in many ways. The final columns indicate the effect of each type of interruption on each instruction; it may *suppress* the instruction so that none of the result variables are affected, *terminate* it after some but not necessarily all of the results are respecified (and all results must be

considered unreliable), or allow it to *complete*. Since a protection exception t_4 is occasioned only by storing in memory, the corresponding column of Table 5 can be used to identify all "store" type instructions; similarly, column t_5 identifies all instructions which refer to memory. Other columns of Table 5 identify those instructions which set the condition code, general registers, and floating point registers.

status switching, read/write direct, diagnose (lines a0—34)

SSM (set system mask) is a privileged operation executable only in the supervisor mode ($p_{15} = 0$) as shown by the exit on line a1 and the setting of t_2 on *CPU* line 10. ISK (insert storage key) and SSK (set storage key) are also privileged and are also suppressed (line a4) if the protection feature is not installed (that is, $m_0 = 0$ and hence $t_1 = \bar{n}_0 = \overline{m}_0 = 1$), if the address $\perp\omega^{24}/R^{a_2}$ is outside the range of the memory, or if any of the last 4 bits of R^{a_2} are nonzero.

These last 4 bits of R^{a_2} are not otherwise relevant to ISK and SSK (see lines a6, 7) and the specification error test is intended to prevent the programmer from using them for other purposes, and hence to reserve them for use in possible modifications of SYSTEM/360, such as an extension of the length of the protection keys. The devious programmer can, of course, use a nonzero final half-byte in R^{a_2} to force a specification exception, but the prudent programmer will not. Similar tests will be found elsewhere (e.g., on the format of channel commands) and serve a similar function.

Since SSM, SSK, and ISK are privileged instructions, only the supervisor program can set the system mask or set or refer to the memory protection keys K^i. SPM (set program mask) is not privileged.

WRD (write direct) transfers one byte from memory to the direct control out-lines E^0 and the immediate data byte from the instruction register to the timing signal out-lines E^2, and sets the write-out signal e_0 to 1. The signals E^2 and e_0 are momentary; E^0 remains unchanged until another WRD is executed. The test for suppression of the instruction (line a9) includes the term t_1 because the direct control feature is optional and may not be installed, and the term t_2 because WRD is privileged. Except for the dwell on line a17, the behavior of RDD (read direct) is similar. Normally, the outputs E^0 and E^2 of one computer are connected into the inputs E^1 and E^3 of a co-operating computer. Programs *ES* and *EIE* enter any nonzero signal on E^3 as an interruption.

The diagnose instruction is privileged (as indicated by the presence of t_2 on line a20) and may also be suppressed by an unsatisfactory address a_1. It performs a certain diagnosis of the hardware and then, for certain models, loads p using 8 bytes beginning at 112. LPSW (load program status word), which is also privileged, performs a similar load from memory at address a_1.

SVC (supervisor call) forces an interruption by setting h_2; the interruption code is the immediate data byte ω^8/I^0, prefixed by 8 zeros.

TS (test and set) simply fetches one byte u from memory (a29), sets the condition code to (0, u_0), and sets the same byte in memory to all 1's. The significant characteristic of this instruction is that (because of the "hold" h in the first use of *MAC*) it sets the tested byte in memory before any other reference to memory can occur.

branches, execute, test under mask (lines b0—24)

EX (execute) was discussed in conjunction with the *CPU* program. TM (test under mask) sets the condition code as shown on line b7. Thus, the code is (0, 0) if the immediate data byte (ω^8/I^0) is entirely zero; otherwise, the first bit is set to 1 if *all* bits of $(\omega^8/I^0)/u$ are 1, whereas the second bit is set if *any* bit of $(\omega^8/I^0)/u$ is 1. The expression $(\omega^8/I^0)/u$ denotes the components of the byte u from memory which are extracted by the nonzero bits of the immediate data byte.

The branch instructions set the instruction counter ω^{24}/p conditionally as shown, and can therefore be used to alter the normal sequence of instruction execution. A full appreciation of BXH (branch on index high) and BXLE depends upon a knowledge of the 2's complement representation of signed numbers, which is also used in the fixed-point arithmetic instructions.

A logical vector (register) u of dimension d represents any integer n in the range -2^{d-1} to $2^{d-1} - 1$ in the form

$$n = (\perp u) - u_0 \times 2^d.$$

For example, if $d = 3$ the representation scheme is given in Table 8.

Table 8

n	u
3	0 1 1
2	0 1 0
1	0 0 1
0	0 0 0
−1	1 1 1
−2	1 1 0
−3	1 0 1
−4	1 0 0

It is easily verified that the representation of a number n in the appropriate range is given by the statement

$$u \leftarrow /(d)\top n;\ (n < 0);\ \sim(d)\top |n + 1/.$$

Arithmetic operations (such as the summation occurring in BXH (line b20)) may, however, produce a result outside the representable range. The representation shown above is, however, used even in this case. The matter is illustrated in Table 9 by examples computed for a dimension $d = 4$.

Table 9

n	*Representation*	*Interpretation*
$2^3 + 1$	1 0 0 1	−6
2^3	1 0 0 0	−7
$2^3 - 1$	0 1 1 1	+7
-2^3	1 0 0 0	−8
$-(2^3 + 1)$	0 1 1 1	+7
$-(2^3 + 2)$	0 1 1 0	+6

For the BXH instruction, lines b17–19 show the specification of k_0, k_1, and k_2 as the signed numbers represented in general registers a_1, a_3, and either $a_3 + 1$ (if a_3 is even) or a_3 (if a_3 is odd). Register a_1 is then respecified by the 2's complement representation of the sum $k_0 + k_1$. The value of this result is compared with k_2 (line b23) to determine whether to branch to a_2.

input/output (lines c0—15)

The four I/O instructions are all privileged (line c0), determine a channel address from the first three of the last eleven bits of the first operand address, test whether the indicated channel is operational (line c2), and conclude by setting the condition code to (1, 1) if it is not. If the channel selected is the multiplexor channel ($i = 0$) and a "burst mode" timer has not run out, then c5 dwells for a maximum of about 100 microseconds before testing (line c6) whether the channel is busy.

If the channel is busy, the condition code is set to (1, 0) on

line c7, and all instructions but HIO (halt I/O) conclude; HIO proceeds to signal the channel to stop by setting B^i_{10} and then dwells until the channel resets it to zero.

If the channel is not busy, line c11 branches to conclude on line c12 for TCH (test channel), and otherwise to set B^i_{10} and dwell (c14, 15) awaiting a response from the channel. The dwell endures until either B^i_{10} is reset by the channel or until (in the case of the multiplexor channel only) the channel becomes busy ($B^i_9 = 1$). The latter case is followed by a repetition of the sequence from line c4. This repetition could recur many times due to short bursts of activity in the multiplexor channel instigated by I/O units already in operation. Further discussion of the I/O instructions will be deferred to the third section.

load and store general registers (lines d0—24)

LH (load halfword) loads the last two bytes of R^{a_1} from memory and then, unless a specification exception has occurred in the fetch from memory, extends the sign bit $R^{a_1}_{16}$ to the left to give the correct 2's complement representation in the entire register.

LPR (load positive), LNR (load negative), LTR (load and test) and LCR (load complement) illustrate the use of the 2's complement representation. Since line d11 sets the condition code to (1, 1) only in the event of overflow, line d12 indicates a fixed point overflow only if the overflow mask $p_{36} = 1$.

STM (store multiple) stores a number of general registers beginning with a_1 and continuing in cyclic order through register a_3. A specification exception suppresses the instruction since it occurs on the first execution of line d14 before any data has been transferred, but in the event of either a protection or addressing exception *all* of the result field becomes unreliable (d19–24). LM (load multiple) behaves similarly except that a protection exception cannot occur.

shifts (lines e0—15)

All single-length shifts operate on R^{a_1}; all double-length shifts operate on the combined quantity R^{a_1}, R^{a_1+1} and cause a specification error (which suppresses the instruction) if a_1 is odd. The amount of shift is the residue modulo 64 of a_2; zeros are introduced in the evacuated positions for all except the arithmetic right shifts.

All of the arithmetic shifts shift the entire quantity except the first bit (e8, 10) and set the condition code to (0, 1), (0, 0), or (1, 0) according as the result is $<$, $=$, or $>$ zero, except that the left shifts set it to (1, 1) if a significant digit (i.e., in the 2's complement representation, one which differs from the leading bit) is lost in the shift (e7). In this event the left shifts also set a fixed point overflow exception (e9) if mask $p_{36} = 1$. The right shifts extend the sign bit (e11) to fill evacuated positions to give the correct representation of the result in 2's complement form.

logical operations, compare logical (lines f0—32)

This family comprises four operations (*compare logical, or, and,* and *exclusive-or*) in four formats (RX, RR, SI, and SS). The first three formats are treated in lines f0–13. CL, CLR, and CLI merely set the condition code (f7); the rest perform the appropriate logical operation (f8) and set the condition code as shown on line f13, except that in the SI format the setting of the condition code

is suppressed (f12) by a protection or addressing exception.

The SS format for logical operations (f14–32) operates byte-by-byte from left to right. As in the LM and STM instructions, a protection or addressing exception will make the entire result field and the condition code unreliable (f27–32).

move, pack, translate (lines g0—62)

The instructions in this group treat the operands byte-by-byte, the first three (MVO, PACK, UNPK) in right-to-left order and the rest left-to-right. The single-byte uses of *MAC* involved cannot produce a specification exception; protection or addressing exceptions make unreliable the entire result field in memory, as well as the condition code when it is a result.

MVO (move with offset) moves the second field to the first field, off-setting it one-half byte to the left to leave the rightmost half-byte of the result field unchanged. Zero fill occurs (g9, 10) when the source field is exhausted, and conclusion occurs (g6) when the result field is exhausted.

PACK converts a decimal number in *zoned format* (one digit per byte with $\alpha^4/\boldsymbol{byte}$ as the zone bits, except for the low-order byte in which $\alpha^4/\boldsymbol{byte}$ represents the sign) into *packed format* (two digits per byte, namely, $\alpha^4/\boldsymbol{byte}$ and $\omega^4/\boldsymbol{byte}$, except for the low-order byte in which $\omega^4/\boldsymbol{byte}$ represents the sign). Five operations, selected by the variable i (line g15), are used in constructing the packed bytes as follows:

0 the sign and digit of the low-order byte are interchanged;
1 the digit is placed in the right half of the byte;
2 the digit is placed in the left half of the byte;
3 a zero digit is placed in the left half of the byte;
4 zero digits are placed in both halves of the byte.

The variable i respecifies itself (line g16) by the ith component of (4, 3, 4, 4, 4) if the source field is exhausted, or by $(1, 2, 1, -, -)_i$ if it is not. The process ends (g21) when the result field is completed.

UNPK (unpack) performs the operation converse to PACK. The program is analogous but simpler, since i has only four states.

The remaining instructions in this group scan the fields from left to right. TRT (translate and test) uses successive bytes from the first field as relative addresses in the second field until the byte fetched therefrom is nonzero; this byte and the current address to the first field are then stored in $\boldsymbol{R}^2$ and $\boldsymbol{R}^1$, respectively (g43), and the condition code is set. TR (translate) replaces each byte of the first field by its correspondent selected from the second field. MVZ (for which $\boldsymbol{I}_6^0 = 1$) moves only the zone portion (i.e., the left half) of each byte; MVN moves the numeric portion.

edit (lines h0—37)

ED uses the first field both as result and as a *pattern* to select successive bytes from the second *source* field and from a *fill* byte specified (h13) by the first byte of the pattern field. Each pattern byte is classified (h12) as "digit select", "significance start", "field separator", or "other" (*class* = 0, 1, 2, or 3); *class* then controls the subsequent assembly of the result byte.

Except for a byte with a sign (i.e., a non-numeric right half), bytes from the source field are used one half-byte at a time, each half being prefixed by a standard zone j selected by p_{12} (lines h2, 17–22). The source byte thus constructed is used (h24–26) only if the numeric part is nonzero or if the current portion of the field is already significant ($s = 1$); otherwise the fill byte is used. The byte is constructed, and the source field is advanced, only when the class of the pattern byte is either a "digit select" or a "significance start" (h14). If the class is "other" and $s = 1$, the pattern byte is left unchanged (h15); otherwise the fill byte is inserted (h16).

The significance trigger s is set only on line h8. It is set to zero if $j = 0$, that is, at the outset (h1) and also if the last byte used was from the source field and the numeric part was a plus sign (h9, 23). If $j = 1$, then s is set according to the class of the preceding pattern byte, s changing from 1 to 0 only in the case of a "field separator" (*class* = 2), and from 0 to 1 only in the case of "significance start" (*class* = 1) or in the case of "digit select" if the last byte was chosen from the source field and was nonzero, as seen from the setting of u on lines h10 and h21.

The final setting of the condition code is determined (h3) by s and i. The latter is set by a nonzero digit (h27) and reset (h7) by a "field separator." A data exception (t_7) occurs if the left half of a source byte is non-numeric (h20), and terminates (h4) in the manner of protection and addressing exceptions. Since ED is an optional (decimal) feature, it is aborted (h0) if $t_1 = 1$, and is treated as an "undefined operation" by branching to line l0. EDMK (edit and mark) differs only in that the byte address of the first nonzero digit in the last nonzero field is entered (h30) in R^1.

fixed-point arithmetic (lines i0—30)

In this group of instructions, the first operand is taken from one or two general registers, determined by a_1. The second operand is chosen as R^{a_2} for the RR format instructions, as 2 bytes from memory at a_2 for the half-word instructions, or as 4 bytes from memory for all others.

The arguments k_1 and k_2 are derived from the operands in two ways: as the base-2 value (i5, 16) in the "logical" group AL, ALR, SL, SLR, and as the 2's complement value (i4, 10) for all others.

A specification exception suppresses the instruction (i10). An addressing exception makes the result k questionable (i11, 12); otherwise k is the true result of the appropriate arithmetic operation on k_1 and k_2 (line i17).

The specification of the final result is illustrated by line i19. The treatment of results outside the representable range has already been discussed in the section devoted to branch instructions. The setting of the condition code and the exception conditions t_8 and t_9 is straightforward.

decimal arithmetic

The intermediate result k is obtained (j22) by applying the appropriate arithmetic operation to the arguments k_1 and k_2 or,

in the case of ZAP (zero and add) and CVB (convert to binary), by using the single argument k_2 directly. CVD also involves a single argument only (j1).

convert (lines j0—55

The argument k_i is determined (j19) as the signed base-10 value of the vector $\bar{\omega}^1/v$ of decimal digits, v being assembled (j15) by catenating the base-2 values of half-bytes from field i. The last half-byte determines the sign (j19), a negative sign being represented by 13 (that is, 1101) in the extended BCD code ($p_{12} = 0$), and by 11 in the American Standard code. For example, if $p_{12} = 0$, $l_1 = 1$ and the bytes at M^{a_1} and M^{a_1+1} are 0101 0000 and 1001 1101, respectively, then $v = 5, 0, 9, 13$, and $k_1 = -509$.

The result k is converted to the decimal representation and stored (j34–42) and, except for DP (divide decimal), the operations end on line j43. Because of the setting of j on line j33, the quotient k is stored only in the first $(l_1 + 1) - (l_2 + 1)$ bytes of the first field, leaving space of length $(l_2 + 1)$ for the remainder, which is computed on line j47 and converted and stored by repeating from line j35.

CVD produces an 8-byte result (j2), and CVD and CVB produce specification exceptions (j39, 3) if the argument address is not at an 8-byte boundary. For MP and DP, a specification exception is occasioned (j9) if the length of the second field exceeds either 8 bytes or the length of the first (i.e., the result) field. All specification exceptions suppress the instruction; except for CVD, in which the result field remains unchanged, all other errors make the relevant result fields unreliable (j48–55). A data exception t_7 occurs in ZAP (j8) if the fields overlap such that the right-hand end of the second field is to the right of the right-hand end of the first. Data exceptions caused by improper overlapping in the other instructions are detected and entered on line j18.

Two floating-point representations are used; the *short* (one-word) and *long* (two-word) representations utilize logical vectors u of dimensions 32 and 64, respectively. A number n, represented by u, is evaluated as follows:

floating-point arithmetic (lines k0—73)

u_0 is the *sign* (0 for +, and 1 for −);
$\perp \bar{\alpha}^1/\alpha^8/u$ is the *characteristic* c;
$2^{-\nu\bar{\alpha}^8/u} \times \perp \bar{\alpha}^8/u$ is the *fraction* f;
$c - 64$ is the *exponent* e; and
$|n$ is equal to $f \times 16^e$.

The program comprises three major segments, the fetching of operands u and v (lines k0–16), the computation of the results (k17–65), and the storing of results and the setting of the condition code and of (lost) significance (t_{14}) and exponent underflow (t_{13}) exceptions (k66–73).

Floating point register i is selected by the address $2 \times i$, and the relevant addresses (both a_1 and a_2 in the RR format) are therefore subject to a specification exception check (k1) which suppresses the instruction. Specification exceptions occasioned by *MAC* also suppress the instruction (k14).

The behavior of the computation phase will be illustrated by AD (add normalized (long)). The argument with the smaller characteristic is shifted right (k21–23) by four times the difference in the characteristics, since the characteristic is taken to be the exponent of 16, and one hexadecimal digit comprises four bits.

The appropriate common characteristic determines j on line k24. Lines k28–30 show the determination of the arguments k_1 and k_2 as the signed values of the fractional parts of u and v, and the determination of the result k therefrom. The sign of the result is determined on line k32, and the fraction and possible fraction overflow i on line k33.

If fraction overflow occurs, line k37 shifts the fractional part of the result right one hexadecimal digit and prefixes it with hexadecimal 1 (thus restoring the overflow), and the next line increases the characteristic accordingly. If exponent overflow results (k39), the entire vector u is questionable (k40); otherwise, the characteristic is specified by j on line k48. If fraction overflow does not occur, the program continues with line k41.

The treatment of a zero result fraction depends on the (lost) significance mask p_{39}; if $p_{39} = 1$, the fraction is combined with the normal characteristic j (lines k41, 48); if $p_{39} = 0$, it is combined with a zero characteristic (i.e., the most negative possible exponent) and a positive sign (k42, 47).

A nonzero fraction is normalized (k44, 45), the amount of shift i (in hexadecimal digits) being determined as the integral part of one-fourth of the number of leading zeros. The characteristic is reduced accordingly; if it becomes negative, the entire result field is set to zeros (k47) and the exponent underflow exception (t_{13}) is set conditionally on line k73.

undefined operation codes (lines l0—5)

Because the tests for the various exception conditions appropriate to a given class of instructions (including the check on the legitimacy of the operation code) may proceed concurrently, addressing and specification exceptions may be presented even for an undefined operation code. The undefined operation exception t_1 will, of course, be presented (*CPU* line 10), but in the event of multiple exceptions *any one* of the exceptional conditions may actually be recognized (*CPU* line 24).

If the first half of the operation code byte has any of the values 0, 1, 4, etc., listed on line l0, no spurious exception conditions can occur. Any other codes *may* (as indicated by the question marks) set t_5 and t_6 as shown by lines 14–5. For example, the first half-byte of the illegitimate code 0011 0101 has the value 3 and is treated, for error check purposes, the same as the floating-point instructions occurring in row 3 of the operation decoding matrix O. Thus t_5 is set to zero and t_6 may be set to the value $i_4 \vee i_5$. In other words, this case may present a specification exception if either of the effective addresses do not designate a valid index to a floating-point register. The format used for the calculation of the effective addresses is determined by $\perp I^0_{0,1}$ as shown by N^0_2.

Input/output

The SYSTEM/360 input/output included in this formal description comprises five system programs: CH^c (channel c), *IOIE* (I/O interruption entry), HFC^c (hardware failure in channel c), TOL^c (time-out limiter for channel c), and *BMT* (burst-mode timer). The description extends as far as the interface between a channel, which communicates directly with the *CPU* and the system memory $\boldsymbol{M}$, and control units, which are closely associated with input/output devices.

Channels are of two major types, *multiplexor* and *selector*, and a system has at most one of the former and six of the latter. Both types are encompassed in the general program CH^c, where c is the channel number, the number zero being reserved for a multiplexor channel. In a formal reading of the program, the value of c is fixed, and the single program CH^c represents several independent programs, one for each channel in the system. The behavior of a multiplexor channel ($c = 0$) differs significantly from that of a selector channel; the differences appear wherever a statement involves the terms ($c = 0$) or ($c \neq 0$) or involves a branch based on the variable c. For example, because of the branch on *CH* line 46, lines 47 and 48 apply only to the multiplexor channel, a point emphasized by the use of the superscript 0 rather than c.

The programs TOL^c and HFC^c also represent a multiplicity of similar, but not necessarily identical, programs for different values of c. However, operational differences in these programs depend not upon distinctions between multiplexor and selector channels, but rather upon model-dependent factors such as the degree to which a particular channel shares physical facilities with the *CPU*. These factors are reflected in the "channel model" matrix $\boldsymbol{CM}$.

At this point, Table 3 merits further study for the information it yields with regard to the logical structure of the input/output system. It will be found that most of the channel variables are formally matrices, with the rows indexed by the channel number. The rows are therefore completely independent in their behavior, and a column is of interest *per se* only in the case of pending interruptions ($\boldsymbol{B}_8$), as shown in *CPU* line 25 and *IOIE* line 1. Analogously, matrix $\boldsymbol{V}$ possesses one row for each control unit on the system, and *CH* line 37 uses columns $\boldsymbol{V}_8$ and $\boldsymbol{V}_{12}$.

A channel resembles an independent computer insofar as it executes a sequence of special fixed-length (8-byte) instructions, called *commands*, stored in the system memory $\boldsymbol{M}$. To this end it possesses a sequence counter $\omega^{24}/\boldsymbol{CAW}^c$, (initially set by an I/O instruction of the *CPU*) and a command register $\boldsymbol{C}^c$. The commands themselves are limited, so far as the channel is concerned, to the transfer of information to or from $\boldsymbol{M}$, known respectively as *read* and *write*, and a respecification of the next command address ($\omega^{24}/\boldsymbol{CAW}^c$), or *transfer in channel* (tic). Special

cases of read and write are, respectively, *sense* and *control*, but these are distinguished as such only at the control unit or device level.

The execution of a sequence of commands (of length greater than one) is called *chaining*, and is signalled by the flags C^c_{32} and C^c_{33}. *Data-chaining* causes a respecification of the data address ($\omega^{24}/\alpha^{32}/C^c$), the flags ($C^c_{32,33,34,35,36}$), and the byte count ($\omega^{16}/C^c$), but does not disturb the command code (α^8/C^c) and hence merely continues the operation in progress. *Command-chaining*, which takes place only when an operation is completed, causes all of C^c to be replaced and thus may initiate a different kind of operation. A tic may occur in either type of chaining, but this affects C^c only indirectly. Any chained sequence of commands relates to a fixed device address, that initially set by the SIO instruction.

Direct communication between the *CPU* and a channel is initiated either by the *CPU* through an I/O instruction, or by the channel through an I/O interruption. This communication with the *CPU* involves the variable B^c, and in particular the four central bits, $B^c_{8,9,10,11}$. The suffix (ω^8/B^c) and prefix (α^8/B^c) bytes hold, respectively, a device address and *device status* information for the indicated device.

B^c is formally identified with channel c, as distinguished from *subchannel c*, which is formally identified with the variables necessary for sustaining an I/O operation. These are C^c and CAW^c, introduced above, and S^c, which has a structure similar to that of B^c. The address (ω^8/S^c) in this case is the address of the device currently being serviced, and the byte α^8/S^c holds *channel status* information associated with this device. The variables C^c, CAW^c, and S^c comprise the only active subchannel associated with a channel B^c; in particular, the *interruption-pending* and *working* states of the channel and of the subchannel are given, respectively, by B^c_8, B^c_9, S^c_8, and S^c_9.

A multiplexor channel has a number of facilities T^i, each of which can store the active subchannel variables and respecify them when required. It is, therefore, said to have μT subchannels, and it can sustain μT simultaneous data transfer operations by time-sharing its active facilities. At any instant only one device can be in contact with a multiplexor channel and the operational information associated with this device will at that time preempt C^0, CAW^0, and S^0 long enough to transfer a characteristic number of bytes or to perform an initiation or termination sequence.

A selector channel differs from a multiplexor channel primarily in having no subchannel storage other than C^c, CAW^c, and S^c, so that only one data transfer operation can be in progress at any time. Once an operation with a particular device is initiated on a selector channel, that device will stay connected at least until all data called for by the operation have been transferred, or until the operation is countermanded by an HIO instruction. There is no interleaving of data transfers from different devices, but between the termination of a sequence of commands with

one device and the initiation of a new sequence, the channel will service requests from devices that have outstanding status information to transmit.

A selector channel always works in the "burst" mode, and the characteristic operation of a multiplexor channel is the "multiplex" mode. However, a multiplexor channel is *said* to operate in the burst mode if a particular device monopolizes its facilities for more than approximately 100 microseconds. This does not imply a difference in channel operation, but relates only to the question of availability, since efficient operation requires that the *CPU* be able to distinguish between a current operation that is likely to keep the channel busy for a relatively long time and one that will soon be over. In terms of the variables used here, it is a question of how long B_9^0 remains equal to 1.

The *interface* between a channel and its attached control units or devices is represented by the variables U^c and P^c. The suffix ω^9/U^c is a *bus* that carries a byte of information (ω^8/B^c) and a bit (U_7^c) for odd parity. The prefix α^3/U^c comprises three tag bits which specify the type of information on the bus. When the interplay bit $U_3^c = 1$, information on the bus is outgoing (from channel to control units) and the tags are, in order, *command-out service-out*, and *address-out*; when $U_3^c = 0$, information is ingoing and the tags are *status-in, service-in*, and *address-in*. In operation only one tag may be set at a time, and its significance depends, in part, on the state of the channel. The remaining elements of U^c are concerned with establishing and holding a logical connection between the channel and a particular control unit. They are *suppress-out* (U_4^c) and *operational-out* (U_5^c) which are set only by the channel, and *operational-in* (U_6^c), which is set only by the control unit. The *polling line* P^c is a vector that has a position for each control unit on the interface, in the order in which they are connected. P_0^c is called *select-out* and $P_{w_c}^c$, where w_c has a value equal to the number of control units connected, is called *select-in*.[7]

Control units may be physically separate from or integral with their associated devices; a multiplicity of similar devices may be connected to a single control unit or, conversely, a device may communicate with more than one control unit. Although the present formal description does not include details of this side of the interface, it will sometimes be necessary to refer to it in the text. The term "device" will usually be used when speaking of specific tasks that devices perform once an active connection has been established between device and channel, and "control unit" will usually be used when emphasis is on the establishment of such a connection.

The formal description shows the generation and processing of all results of channel operation to which a programmer has access, including the data transferred to or from memory, a condition code setting ($p_{34,35}$) for I/O operations, an interruption code setting ($p_{1^{*6}(16)}$), and a *channel status word* (CSW) which comprises CAW^c, a device status byte α^8/B^c, a channel status

byte α^8/S^c, and a byte count ω^{16}/C^c.

In the discussion that follows, major phases of the *CH* program will be outlined, certain critical portions will be examined in detail, and the operation of the four auxiliary programs will be summarized. As before, the discussion is intended as a guide and introduction only, the complete description of channel operation being embodied in the programs.

channel program

Functional segments of *CH* are listed in Table 10 in the order in which they will be considered. The table gives the extent of each segment, its entry and exit points, and the channel and subchannel states that predominate when the segment is active.

data transfer (lines 7—31, 59—65)

It is assumed initially that the channel is actively engaged in data transfer, i.e., a byte has been transferred either to memory (line 18) or from memory (line 11). If an invalid address has been indicated in S_2^c by the use of *MAC* on line 11, then line 12 branches to line 7 to place a signal on the interface that will be interpreted by the device as an order to stop data transmission. If instead the normal branch to line 13 is taken, a parity failure in the byte from memory will be recorded as a *channel data check* (S_4^c) in the channel status byte, but it will not cause termination of the operation. In line 14 the data byte is placed on the interface, together with the (possibly incorrect) parity bit ${}^cJ_0^0$ and the service-out tag U_1^c. The interplay bit U_3^c is set to 1, indicating that this is an outgoing transmission. The unconditional branch to line 19 updates the memory address, and line 20 reduces the byte count. Had the operation been a read ($C_7^c = 0$) rather than a write ($C_7^c = 1$), a parity error on the interface would have been noted in line 15 and the receipt of the byte acknowledged to the device on line 16 before the attempt to store the byte on line 18. Following this *MAC*, however, read and write are treated identically until they separate again on line 10 during the next byte cycle.

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
mpx	?	1	?	1
sel	?	1	0	1

Certain actions are peculiar to the read operation. First, there is a conditional setting of U_4^c on line 16 which will be discussed in connection with data chaining. Second, certain options are available during read: the *skip* flag C_{35}^c determines (line 17) whether the received byte should actually be stored; and if the last four bits of the command code are 1100, indicating a *backward read*, the address will be decreased by one, rather than increased (line 19).

If data chaining is not indicated by the test on line 21, the action moves to line 59 to test for a *program controlled interruption* (pci). If the need for a pci is indicated by S_0^c, and if no prior interruption request is pending ($B_8^c = 0$), then an interruption request is initiated on line 60 by setting B_8^c to 1 and loading into the appropriate parts of B^c an all-zero device status byte and the address of the working device (ω^8/S^c).

It is noteworthy that in a selector channel a prior interruption pending at this point can only be a pci entered at an earlier time in the execution of the current sequence of commands, but in a

multiplexor channel it could be any kind of interruption. However, if the device for which the interruption is pending in a multiplexor channel is the device that is involved in the current sequence ($(\omega^8/B^0) = \omega^8/S^0$), then the interruption is necessarily a pci.

The foregoing remarks can be confirmed by a study of the general topology of the *CH* program: if $c \neq 0$ the entry to line 59 from outside the data transfer segment can come only from lines 69 or 130; moreover, the entry from line 69 occurs only during command chaining and therefore belongs to the execution of the current sequence of commands. The entry from line 130, however, represents the initiation of a new command, and if this is not part of a sequence of chained commands, its source can be traced back through lines 94 and 96, which together assure that B_8^c will be zero upon entry into a new command sequence in a selector channel. On the other hand, for a multiplexor channel entering through line 130, line 96 is skipped and a sequence of tests and settings in lines 97–100 may allow B_8^c to remain set. More broadly, however, for $c = 0$ the entry to line 59 may have come from lines 50 or 32, in which case nothing can be said about B_8^0 except that it retains its previous setting, whatever the source.

Treatment of the pci is followed by a dwell at lines 61 and 62. A multiplexor channel at this point examines the state of U_6^c (operational-in) to determine whether the presently connected device wishes to extend the current burst of information transfer. If $U_6^c = 0$, exit is made from line 61 to line 58, storing the state of the active subchannel, making the channel not busy ($B_9^0 \leftarrow 0$), and resetting the burst timer control ($g_2 \leftarrow 0$). In a selector channel, or in a multiplexor channel where the device is maintaining the connection, line 62 controls the dwell. Here the channel waits for a response from the interface, indicated by $U_3^c = 0$; for an order from the *CPU* to halt the I/O operation, indicated by $B_{10}^c \wedge \tilde{g}_0$; or for a signal that the device response time has been excessive and there may be trouble on the interface, indicated by an *interface control check* (S_6^c) set, for example, by a *TOL* program.

If the escape from line 62 was not for HIO or an interface control check, and no improprieties were found on line 63, then the subsequent decision at line 65 depends on whether the device desires data transfer ($U_1^c = 1$) or has sent in status information ($U_1^c = 0$ and $U_0^c = 1$). Data transfer takes the program to line 1 where an *incorrect length* indication (S_1^c may be generated, and then to line 9 where the channel decides to terminate or continue the operation. Termination at this point—where continued data transfer has been requested by the device—may be caused by a zero byte count (regardless of the setting of S_1^c), or by a previously recognized but unfulfilled HIO, signalled by S_8^c (possible only on a multiplexor channel), or by the presence of a nonzero channel status bit other than S_0^c (pci) or S_4^c (data check). A program or protection check generated by *MAC* during the previous

byte cycle on a read operation first takes effect at this point. If termination is not required, data transfer is effected as before, beginning with the branch from line 10 for read or write.

At different moments in the discourse across the interface between a channel and a device only certain responses from the device are valid. Thus, upon escaping from the dwell on line 62 because of a response on the interface, the channel expects either status-in (U_0^c) or service-in (U_1^c), and a properly operating control unit will return only one of these tags and no others. Any other response must be considered an interface error, and is so recorded on line 63, which also includes a test for parity failure on a status byte. The action to be taken in case of *interface control check* and other error conditions will be discussed more fully in the treatment of *HFC*, but it may be remarked here that the branch to the HIO sequence (line 78) is the mildest action appropriate under the circumstances.

The data transfer cycle has been described here as a strictly byte-by-byte operation, although in most implementations a channel will buffer a certain number of bytes in order to use the central memory more efficiently. While this may have a noticeable effect on the timing of I/O operations, the only observable effect of such buffering on the static results is found in the case of termination due to a *program* or *protection check*. In this case the byte count subsequently stored as part of a channel status word will not necessarily reflect the actual amount of data transferred, a state of affairs indicated by the expression ?(16) appearing in lines 136 and 154, and in *IOIE*, line 24.

data chaining (lines 22—31)

The number of bytes specified in a channel command word will always be transferred to or from contiguous memory locations, as shown in line 19. When the count becomes zero, however, the operation may be continued by fetching a new channel command word with a new count and a new memory address. Such a continuation is signalled on line 21, which tests for a zero count and the presence of the *chain data* flag C_{32}^c.

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
both	?	1	0	1

Line 23 fetches the double word indicated by the address in CAW^c, and the address is updated on line 24. The four-bit pattern in the command code detected by line 25 signals a tic (transfer in channel) which, if present, causes the replacement of the address in CAW^c (line 31) and a repetition of the fetch, line 23. If this produces another tic, a program check is recorded on line 30 because of the prior setting of j on line 31, and the process is stopped by a branch to line 29. Line 30 also checks for a specification or addressing error in the address field of the tic command. This test, which would normally occur in the succeeding *MAC* on line 23, is done explicitly at this point in order to preserve the address of a faulty tic for diagnostic purposes, since the tic address would not be recoverable after the execution of line 31.

If the command code field of the double word fetched on line 23 does not specify a tic, its overall format is checked on line 26, and if satisfactory it is used to respecify all but the command

code in C^c (line 27). The pci bit is set if called for by C^c_{36}, and *channel overrun* (S^c_7) is recorded if a read operation is in progress and the device has already signalled on the interface.

The setting of U^c_4 on line 16 is a signal that data chaining is about to take place during a read operation. Certain devices can respond to this signal by delaying the transmission of the next byte until U^c_4 becomes zero (line 29), thus avoiding the possibility of channel overrun. Otherwise, overrun is a function of the relative speeds of device, channel, and memory, as well as of the current memory activity.

termination
(lines 66—81, 56—58)

Channel type	*State* B^c_8	B^c_9	S^c_8	S^c_9
both	1	0	1	0

A channel cannot break out of the data transfer phase on its own initiative unless it recognizes an interface control check (line 63). In normal operation it must wait for a stop order (HIO) from the *CPU* or a status byte from the device, even though it may have initiated the termination by issuing the command-out tag on line 7. A multiplexor channel will leave from line 61 any time during data transfer if the connected device sets $U^c_6 = 0$, but this is not a termination of the operation; the channel will return to line 59 and continue in the data transfer phase the next time the device requests service and is reconnected.

The first action in a normal termination (line 66) is a setting of the incorrect length indication for a long count (i.e. when termination occurs with a non-zero count). A prior setting, possibly incurred on line 8 for a short count, will be preserved. In both cases the setting depends on the state of the control flags in C^c, if data chaining is indicated ($C^c_{32} = 1$), the wrong length indication cannot be suppressed by C^c_{34}. However, if a program, specification, or channel-overrun check has occurred, the indication of wrong length may fail to appear (as shown by the conjunction with $? \vee \sim \vee / S^c_{2,3,7}$).

If the possibility of command chaining is ruled out on line 67, termination of device operation proceeds with the setting of the subchannel state $S^c_{8,9}$ to "interruption pending" and "not working" on line 71. The interruption condition must now be entered in the channel, if possible, and to this end it is necessary to ensure that a previously pending interruption is not at this instant being serviced by the *CPU* through *IOIE*. Line 72 is an interlock for this purpose.

The test on line 73 is vacuous for a selector channel, which always proceeds through the next two lines, setting up the channel interruption and branching to line 57 and then to line 56, where it waits for service from the *CPU*. A multiplexor channel, however, may be unable to accept the status byte and consequent interruption at this time. If so, it skips lines 74 and 75 and, because of the setting of j, returns command-out rather than service-out on the interface (line 76). The former is interpreted by the device as an order to *stack* (i.e., save) the information just transmitted; the latter, as permission to *clear* (i.e., destroy) the status and go about other business which may, in fact, be the completion of a phase of the current operation that does not require channel

facilities. In either case, the multiplexor places the device address in the field of the no-longer-needed data address (line 77), returns to line 59, passes through line 61 as soon as the device disconnects ($\boldsymbol{U}_6^0 = 0$), stores the subchannel information and makes the channel not-busy (line 58), and goes into the idle phase (lines 35–55).

It will be observed that a selector channel will not have signalled on the interface at this time, thus keeping the device connected. The channel itself is not available for anything but a clearance of the interruption just set. This can be seen by tracing *IOIE* and the I/O instructions of the *CPU* with $\boldsymbol{B}_8^i$, $\boldsymbol{B}_9^i$, $\boldsymbol{S}_8^i$, $\boldsymbol{S}_9^i$ set to 1, 0, 1, 0, respectively. A TIO addressed to the proper device, or an interruption service, will cause program *CH* to follow the path $\cdots$, 56, 82, 83, 87, 88, 89, 92, 153, 154, 155, 160, $\cdots$, which clears the interruption condition and releases the device, and then goes on to make the channel available for new work. A TIO for another device, or an SIO, is rejected by the sequence $\cdots$, 56, 82, 83, 87, 88, 89, 92, 93, 138, 134, 135, 32, 33, 56, $\cdots$, and leaves the channel state unchanged.

Termination due to the instruction HIO is indicated when the *CPU* tries to contact a channel through $\boldsymbol{B}_{10}^i$ when the channel is busy (*EXC* c6–9). This is sensed in line 62 and causes a branch from line 64 to line 78, where a signal to the device to disconnect is transmitted on the interface ($\boldsymbol{U}_2^c = 1$ when $\boldsymbol{P}^c$ is all zeros). The possibility of command chaining is then erased and the *CPU* is released (line 79). A multiplexor channel will then return to line 35 through line 59, as described for normal termination, and contact with the device will eventually be reestablished when the device has come to a stopping point and generated a status byte and a request for service. To the multiplexor channel this request will be indistinguishable from a service request for additional data transfer or normal termination. In contrast, a selector channel at line 80 proceeds very much as if termination had been signalled by the device, combining the operations of lines 66, 71 and 74 in line 81, and returning to lines 57 and 56. The significant difference is the specification of zeros as the device status byte ($\boldsymbol{\alpha}^8/\boldsymbol{B}^c$).

There is a two-level hierarchy for device status bytes: those associated with a termination (evidenced by the state of $\boldsymbol{S}_8^c$) have an irrevocable hold on the channel interrupt signal ($\boldsymbol{B}_8^c$) until cleared; others, externally generated or arising after the device is disconnected from the channel, may be displaced, without being accepted by the *CPU*, to allow a new operation to start (in which case the information must be saved by the device). Non-displaceable status bytes are usually characterized by the presence of *channel end* ($(\boldsymbol{\alpha}^8/\boldsymbol{B}^c)_4$), but exceptions occur when a sequence is terminated during command chaining and when, as just noted, HIO is issued to a busy selector channel. In the first case channel end may never appear; in the second case, the channel-end byte, when it is received, will have the same significance for

channel operation as any other status byte submitted after termination.

Each device status byte implies the formation of a channel status word (CSW), either by means of an interruption, a TIO, or an SIO. The execution of HIO by a busy selector channel will, therefore, cause an extra CSW to be generated, for a possible total of four associated with the subject command sequence. The three nonzero device status bytes which are always possible would contain, respectively, bits designating *channel end*, *control unit end*, and *device end*, together with whatever other conditions happened to be present. Any two, or all three, end conditions may appear in the same byte, but in any case an operation is not actually completed until device end for it has been submitted and cleared at the device. Control unit end is supplied only under special circumstances but, apart from the exceptions possible during command chaining, channel end and device end are made available at the termination of every sequence, and will appear in a *CSW*.

command chaining
(lines 68—70)

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
both	?	1	0	1

Command chaining (the execution of a sequence of operations by the same device) is initiated (line 68) if the tests in line 67 are satisfied: the flag settings $C^c_{32,33}$ must be 01, no bit other than pci (S^c_0) may be present in the channel status byte, and the device status byte on U^c must conform to one of the allowed patterns listed. (In the absence of hardware failures it is possible, coming from line 66, for any but the *busy* bit $(\omega^8/U^c)_3$ to be present in addition to channel end $(\omega^8/U^c)_4$.) The inception of command chaining is signalled to the device on line 68 using the same signal as was used to signify data chaining (line 16), but this time it is in response to status-in rather than service-in. In response to this signal the device will clear the status, but the channel must note the presence of device-end to decide the branch on line 69. If device end is present ($j = 1$), initiation of the next command takes place immediately, starting with line 70, where the address in CAW^c of the next command word is increased one double word if the *status modifier* bit $(\omega^8/U^c)_1$ had been sent with device end. The program then branches to line 104 where it joins the SIO sequence described below under "*CPU* service". If device end has not been received, line 69 causes a branch to line 59, after which a selector channel dwells at lines 61 and 62, and a multiplexor channel exits at line 61 when the device disconnects by setting U^c_6 to zero.

channel idle
(lines 35—55)

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
mpx	?	0	?	?
sel	0	0	0	0

When a device disconnects from a channel (for any one of a variety of reasons on a multiplexor channel, but only following the acceptance of a terminating device status byte on a selector channel), the channel is free to respond to calls from the *CPU* or to service requests from its devices. Line 37 shows a dwell on the two possibilities, and line 38 shows that requests from devices take precedence over *CPU* requests. The expression $\vee/V_8 \wedge V_{12} = c$ corresponds to *request-in*, and rests upon the following formulations. Each control unit on a system is asso-

ciated with a row of a matrix V. Control unit u, desiring service for one of its attached devices, sets $V_8^u = 1$ and at the same time indicates the channel wanted by holding the channel number in V_{12}^u. If $V_8^u = 1$ and $V_{12}^u = c$, this clearly represents a call for channel c, and all such possibilities are monitored by the *or* over the conjunction. The numbers available to control unit u for V_{12}^u must, of course, be restricted to those of the channels on whose interfaces the control unit is indeed connected.[8]

In response to request-in the channel attempts to establish a working connection with a device by setting select-out (P_0^c) to 1 with the tags α^3/U^c set to zeros (line 39). Control units are connected serially on the interface with respect to select-out. Only the first control unit will sense P_0^c, and it will pass the signal on (setting $P_1^c = 1$ and $P_0^c = 0$) if it does not require service. Successive control units will pass the signal on in a similar way until either a control unit responds with address-in or the last control unit, passing the signal on in turn, sets select-in ($P_{w_c}^c$) to 1. A tag other than address-in, or wrong parity on an incoming address, will cause an interface control check (line 41).

For a multiplexor channel, two major types of request for service from a device are possible:

1. from a device still in the data transfer state (S_9^c was 1 last time it disconnected at line 61);
2. a) from a device for which a termination status byte was stacked,
   b) from a device for which device end is due, or
   c) from a device presenting an externally generated signal such as *attention* or a change from *not-ready* to *ready* (which will be indicated as device-end).

For a selector channel, only types 2b and 2c are possible.

A multiplexor channel always honors requests of the first type, but attempts to suppress others if the interruption buffer (α^8/B^0 and ω^8/B^0) is loaded, as indicated by $B_8^0 = 1$. This is shown in lines 36 and 39 where suppress-out (U_4^c) is set to the value of B_8^c. When $U_4^c = 1$, a control unit must either suppress requests of the second kind by not activating V_8^u and V_{12}^u or else pass along select-out when the channel tries to establish a connection under these circumstances. On a selector channel, B_8^c is zero during the idle phase and all requests are, therefore, honored.

If a connection is established (U_3^c becomes 0 and U_2^c becomes 1, lines 40, 42), a multiplexor channel will immediately set select-out to zero (line 44), giving the device control over the connection. Operational-in (U_6^c) will have become 1 with U_2^c, and now the channel (always passing through lines 59 and 61 before returning to idle) will be constrained to hold the connection as long as $U_6^c = 1$.

Line 44 has no effect on a selector channel, which simply continues by placing the device address in the working address

register ω^s/S^c (line 45), acknowledging its receipt with command-out (line 49), waiting for the next response (line 51), which this time must be status-in (line 52), and loading both status byte and address into the channel registers as an interruption request (line 53). It then dwells at line 56 waiting for a call from the *CPU*.

A multiplexor channel has a slightly more complex sequence after line 45. It first uses the device address just received to generate an index to the subchannel storage facilities (line 47). All eight bits of the address are used to generate this index if the zeroth bit is a zero. If not, only bits one, two, and three are used, thus allowing several devices to share a sub-channel, i.e., a storage location in T. The stored operational information is then loaded into the active subchannel facilities, the channel is put into the working state by setting B^0_9 to 1, and BMT (burst mode timer) is released from its dwell by setting g_2. The device is signalled to proceed (line 49), and the state of S^0_9 is sensed (line 50) to determine which phase of operation the device is in.

If $S^0_9 = 0$, the device is requesting service of the second kind and the program proceeds much as in the case of a selector channel. An additional malfunction is recognized in this case if the control unit overlooks suppress-out and accepts service when $B^0_8 = 1$ (line 52). If no malfunction is recognized, an interruption request is entered on line 53 and receipt of the status is acknowledged on line 55. The earlier remarks concerning line 76 are relevant here, since the device may still be required to stack the information in spite of the fact that the interruption facilities are now available. The criterion this time is whether the byte just received is termination status stacked at the time of generation. The state of S^0_8 determines this, and so is used to choose between command-out (U^0_0) and service-out (U^0_1), allowing the device to clear the status if $S^0_8 = 1$. As noted before, the need to stack arises because non-termination status information can be displaced from B^c without entry into a CSW. Finally, the branch to line 59 returns the channel to idle status through lines 61 and 58.

If $S^0_9 = 1$ in line 50, there is an immediate branch to line 59, and the dwell at lines 61 and 62 would await a response on the interface, just as if the device had never disconnected from the channel. The device may be at any stage in its operation and it is possible that it will transmit status information rather than data at this time. This would, of course, lead into the termination sequence, as previously outlined. If data remains to be transferred, the device will transmit or accept its characteristic number of bytes and then set U^0_6 to zero, allowing the channel to return to idle.

Certain channel models, when idling, continuously scan their interfaces for service requests rather than waiting for request-in to rise. This difference is shown by the branch at line 35. The continuous-scan type follows the loop $\cdots$, 39, 40, 41, 42, 43, 35, 39, $\cdots$, skipping the dwell at line 37 and responding (line 43) to the *CPU* interlock B^c_{10} after each non-response from the interface ($U^c_2 = 0$, line 42). The other type, largely described above,

would mainly dwell at line 37, and normally proceeds through $\cdots$, 38, 39, 40, 41, 42, 44, $\cdots$ for a device request, or leaves at line 38 for a *CPU* request. For certain control units, however, a non-response even after request-in has risen is a normal possibility, so the loop via line 43 may be followed. Interface control checks arising in this phase of channel operation have again been shown in their weakest form, simply causing a return to line 35, repetition of interface scan, and testing of B^c_{10}.

CPU service
(lines 82—165, 32—34)

Channel type	*State* B^c_8	B^c_9	S^c_8	S^c_9
both	?	?	?	?

The channel responds to the *CPU* in executing the SIO, TIO or HIO instructions and in processing interruptions. Provision is made in *IOIE* for servicing certain interruptions, but otherwise interruption processing is very like TIO. Although the initiation of a new command by command chaining is not strictly part of *CPU* service, it is almost identical with SIO and they will be treated together, where appropriate.

Except for command chaining, it is assumed in what follows that B^i_{10} has been set to 1 in *EXC* line c13 or *IOIE* line 8, and has caused CH^c to branch to line 82 from line 38, 43, or 56.

state analysis
(lines 82—101)

Channel type	*State* B^c_8	B^c_9	S^c_8	S^c_9
mpx	?	0	?	?
sel	?	0	?	0

The first phase of *CPU* service starts by setting a working device address either from the effective address a_i supplied by the *CPU* or from the interruption register ω^s/B^c (line 82). A selector channel passes over tests in lines 83, 87 and 88 (and line 89 if not HIO), and continues to line 92. The branch to line 153 is taken for interruption service ($g_0 = 1$) or for TIO with termination status ($S^c_8 = 1$) available for the addressed device. (Note that if $g_0 = 1$, then ω^s/S^c would have been set by ω^s/B^c on line 82. The requirement that this be termination status is not imposed on interruption service, because the desideratum here is the availability of status information at the channel level, a condition that is always satisified when $B^c_8 = 1$ in a selector channel.) Failing the branch to line 153, line 93 merely checks, for $c \neq 0$, the state of the subchannel, and if it is holding termination status, the branch to line 138 will be taken. Otherwise, information is not available at the channel level and the path through line 94 is followed, leading ultimately to the selection of a device on the interface.

A multiplexor channel leaving line 82 must first determine the index to its subchannel storage (line 84), and an invalid index causes a branch to line 140. Otherwise, the active subchannel facilities are loaded and a series of inquiries are undertaken to determine, as in the selector channel, whether the necessary information is available at the channel level.

For interruption service a totally inactive subchannel (line 87) implies that it is necessary to go to the interface, but $S^0_8 = 1$ or $S^0_9 = 1$ imply, respectively, that termination status or a pci is available at the channel level, and the branch from line 88 to line 136 is followed. For TIO to follow the same path requires, as in a selector channel, that the channel be holding termination status for the addressed device. In this case $S^0_8 = 1$ alone does not distinguish the termination state, since the case $S^0_8 = 1$

and $S_9^0 = 1$ is used to denote a pending order to stop, set on line 145 or 148 by a previously issued HIO, to be honored (line 2) the next time the device requests service. If ω^8/S^0 and ω^8/B^0 do not agree, it is still possible for a termination status condition to be cleared, but only by going to the interface and selecting the device. This situation is treated on line 93, which is reached by a multiplexor channel if the sequence to line 89 is followed and the case is not HIO. For both SIO and TIO at line 93, a multiplexor will proceed to line 94 if the subchannel is both not working and not holding an interruption. For TIO the additional possibility of clearing an interruption is signalled by the same criteria as in line 88, except that ω^8/S^0 must now match the address previously stored in the address field of C^c in line 77.

If there is an HIO at line 89 and the subchannel has an interruption pending (line 90), no action is taken in either type of channel, and the branch to line 161 sets the condition code. If no interruption is pending, the possibility of command chaining (in a multiplexor channel) is cancelled (line 91) and a branch to the interface selection sequence (line 115) occurs. Ultimately, an order to stop will be issued on the interface (line 151) or, for a multiplexor, the subchannel will be set (line 145) to stop the operation the next time the device requests service.

Lines 94–100 cancel interruption conditions arising from non-termination status, as discussed in regard to termination. A selector channel must order the device to stack the information at this point (line 96), but a multiplexor channel does not, since it accepted non-termination status only provisionally in the first instance (line 55). Line 100 indicates that under some (indeterminate) circumstances, a multiplexor channel may not cancel a non-termination interruption condition.

At line 101, SIO and TIO are the only possible cases. TIO causes a branch to line 115, but SIO proceeds through lines 102 and 103, to join command chaining at line 104. A channel address word is fetched in line 102 from a fixed location in main storage. It is checked for protection key and format (line 103) and any nonconformity is recorded as a program check (S_2^c) in the channel status byte. In line 104, the local variable j, set to zero for command chaining ($B_9^c = 1$), will be used as in data chaining to prevent two successive tic's. Setting j to 1 for SIO, (when $B_9^c = 0$), precludes the possibility of a tic as the first or only command of a sequence. The pci bit is set, if necessary, on line 114.

Lines 105–114 correspond closely to the data chaining segment, lines 23–31. The differences are largely in error indications, including an added aspect of the format check (compare lines 26 and 109), but the most significant difference occurs in line 113 (as compared to line 27) where k now respecifies all of C^c including the command code portion.

All program checks, including a faulty CAW^c, cause a branch to line 112, which is entered only if there *is* a program check. From there, command chaining will cause a branch to line 115,

start I/O
(lines 102—114)

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
mpx	?	0	0	0
sel	0	0	0	0

command
chaining
(lines 104—114)

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
both	?	1	0	1

whereas SIO may or may not, depending upon the channel model and immediate circumstances. The alternative for SIO is the branch to line 139 which stores the status byte portion of the CSW.

At line 115, the possible cases are command chaining, SIO, TIO, or HIO for any type of channel, and interruption service for a multiplexor channel. Selection is started by setting select-out, as in line 39, but the working address is now on the interface bus (ω^s/U^c) and address-out (U_2^c) is set. The dwell on line 116 differs from line 40 in the added term S_6^c, which will be set by the *TOL* (time-out limiter) program (line 0) if the response from the device is delayed beyond a maximum interval specified for the particular channel model. The interface control check S_6^c may also be set (line 117) if this is an interruption service ($g_0 = 1$) or command chaining ($B_9^c = 1$) and no control unit acknowledges the address or address-in does not rise in response; if any combination of incoming tags appears other than U_0^c or U_2^c alone; if there is a parity error on the incoming interface; or if an address received in response does not match the address sent out.

channel-initiated selection (lines 115—130)

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
mpx	?	?	?	?
sel	?	?	0	?

CSW and condition code setting (lines 136—165)

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
mpx	?	0	?	?
sel	?	0	?	0

For command chaining, the minimal response to an interface control check at this point is the branch from line 118 to the disconnect sequence starting at line 78, just as for an interface control check in the data transfer phase on line 63. The corresponding response for *CPU* service is the branch to line 143 and storage of a complete CSW for interruption service or TIO (142), or of the status portion only for SIO or HIO (144). Line 142 also may be entered by the sequence $\cdots$, 85, 140, 141, 142, $\cdots$ in the case of an invalid subchannel index during interruption service.

The branch to line 119 for $S_6^c = 0$ implies that all subsequent tests are for normal possibilities. Thus, if $P_{w_c}^c = 1$ in line 119 or $U_0^c = 1$ in line 120, it must be that SIO, TIO, or HIO are under consideration, for only in these cases is a "not-operational" response or a "control-unit-busy" response possible without error. The consequent action in each case is shown by the respective branches to lines 145 and 147.

At line 145, which is also entered by the sequence $\cdots$, 85, 140, 145, $\cdots$, HIO on a multiplexor channel sets the subchannel to terminate as soon as it requests service, and in line 146 the condition code is set to indicate "not-operational" in response to whichever of HIO, SIO or TIO may be current.

At line 147, which is entered only in the case of a control-unit-busy response, there is a division (like that on line 143) between interruption service and TIO on one hand, and SIO and HIO on the other. For the former, the entire CSW is again specified (line 150), although useful information will appear only in the status portion, which is the only part stored (line 149) by the SIO and HIO. For a multiplexor channel HIO again sets the subchannel to terminate (line 148). The device status byte stored at this point contains the busy bit and status modifier only ($\epsilon^{1,3}$). In the case of SIO, the channel status byte is zero, except for a possible program check (S_2^c).

In line 121 a multiplexor channel sets P^c to zero. If HIO has lasted to this point, it implies that the device is able to accept the order to disconnect, and line 122 branches to line 151 to issue this order. The terms B_9^c and $\bar{g}_0$ in line 122 ensure that this is truly HIO and not a fortuitous value of I^0 during command chaining or interruption service.

The command byte is specified in line 123 in preparation for its transmission to the device in line 124. The last terms of line 123 ensure that, regardless of the command code in C^c, a programming error ($S_2^c = 1$) carried over from line 112 will cause an all-zero byte (the interface command code for TIO or interruption service) to be transmitted. The only legitimate response to a command code is a status byte with correct parity (line 126) and, when this response arrives (within the allowed time-out limit), the program branches (line 127) to line 153 for *CPU* service.

At line 153 the possibility of an interruption pending in the subchannel is checked. This could arise either because line 153 was entered directly from line 92 by a selector channel, as noted earlier, or because the special condition for a multiplexor channel in line 92 had been satisfied. In both cases a TIO must be in progress and a full CSW is stored in line 154. For a selector channel the interruption pending in the subchannel must have also been pending in the channel, and so both are cleared in line 155. For a multiplexor channel, however, the subchannel interruption could not have been in the channel, or the branch from line 88 to line 136 would have been taken before reaching line 93, so in this case ($c = 0$) the state of B_8^c is left untouched. Line 155 is followed by line 160, which releases the device. The need for an interface signal is obvious for a multiplexor channel because the status has just been transmitted; and, for the selector channel, it will be recalled that the interruption was entered in line 74 and simply held by the channel with no return signal to the device at that time.

The sequence just described, and the sequence $\cdots$ 88, 136, 137, $\cdots$ both relate to the formation of a CSW with termination status. The latter sequence, which obtains only in a multiplexor channel, may also be followed for a pci while the subchannel is still working. In this case the combination $S_8^0 = S_9^0 = 1$ would, as usual, be the result of a prior HIO, and it would be improper to reset S_8^0 in line 137. But B_8^0 is unconditionally reset, providing a double contrast with the action in line 155. In both line 136 and line 154 the count field of the CSW is indeterminate if there has been a program or protection check, and in line 136 this is also true for a pci.

If $S_8^c = 0$ in line 153, further possibilities are checked in line 156. If this is not interruption service ($g_0 = 0$), and there has been no programming error ($S_2^c = 0$), and either the status byte on the interface (ω^8/U^c) is all zero or this is SIO with command chaining indicated, and the status byte conforms to one of three allowable patterns, then the branch is taken to line 161. Otherwise the case is either interruption service to clear a non-termina-

Table 11 Device status bytes stored in CSW

device status bits								*channel activity and CH line on which stored*			
attention	status modifier	control unit end	busy	channel end	device end	unit check	unit exception	interruption	TIO	SIO	HIO
0	0	?	0	1	0	?	?	136	158	136	
?	?	?	0	?	1	?	?	154		154	
0	0	1	0	0	0	?	?	158		159	
0	?	0	0	0	0	?	?	IOIE 17			
1	0	0	1	0	0	?	?				
1	0	0	0	0	0	?	?	159	158	159	
0	0	0	1	0	0	0	0				
0	1	0	1	0	0	0	0		149	150	149

tion status from a device, SIO or TIO to a device that is working or holding status information, or a rejected SIO. The status bytes subsequently stored in line 158 or line 159 will indicate the situation: no device status at this point should contain channel end; a busy device will return the busy bit to SIO or TIO; a device holding status information will include the busy bit in response to SIO but not TIO (in both cases the device will be cleared, as in interruption service); and the status for a rejected SIO will contain a unit check or unit exception.

All possible configurations of device status bytes that may appear in a CSW are shown in Table 11. As usual, the symbol ? denotes the possibility of a zero or one in that position; the first line, for example, represents 8 possible status bytes. Altogether, 94 possibilities are represented.

At line 161 an HIO from line 90 may be present as well as SIO or TIO from line 156. In all cases a condition code of zero is specified. HIO leaves immediately at line 162, TIO leaves at line 164 after signalling on the interface. If this is SIO, a command has been successfully initiated and the channel and subchannel state variables are set in line 165 to reflect this. For a multiplexor channel the timer is started. If the status byte is all zero (line 129), service-out is signalled on the interface (line 130) and the data transfer phase is entered at line 59. A non-zero status byte for SIO at line 129 indicates an *immediate* command, usually a control operation that can be executed without help from the channel. It also means that command chaining was indicated by the bits in C^c. Both of these factors were checked in line 156, which strongly resembles line 67, and the branch from line 129 therefore goes to line 68 to prepare for command chaining.

If command chaining had been in progress at line 127, the branch to line 128 would have been followed. Here, either a non-zero status byte or a program check will cause a branch to line 67 where the choice is made between termination and further com-

mand chaining. Alternatively, the branch from line 128 to 130 leads to the data transfer phase, as in SIO.

A program check during command chaining may produce one of the odd situations alluded to in the section on termination. The device involved in the sequence of commands is available to the channel at this point and it will ordinarily respond to TIO (the command actually issued—see line 123) with zero status. But S_2^c will be detected on line 67 and the termination sequence starting at line 71 will be entered. The termination status byte may now be zero, and both channel end and device end will have been lost.

end of CPU service (lines 131—135, 32—34)

All exits in the segment 137–164 go to line 131 if a CSW has been stored, otherwise to line 134. At line 131 the channel status byte is cleared, and except in interruption service, the condition code is set to (0, 1) in line 133 to signify that CSW information is available. Except for interruption service, then, all entries to line 134 occur immediately after a setting of the condition code. Line 135 is an interlock with *IOIE*, followed by a return to line 32. As usual, a multiplexor channel returns to idle through line 59, whereas a selector channel either returns to the dwell at line 56 or sets P^c to zero and goes on to idle.

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
mpx	?	0	?	?
sel	?	0	?	0

system reset and IPL (lines 0—6)

A channel that is not operational is shown as dwelling at line 1. If it is on a system, it can be moved from this dwell only by system reset or some other external agency. A forced branch to line 2 (by *RESET* line 4) clears the various channel and subchannel facilities and places on the interface a system reset signal ($U_{4,5}^c = 0, 0$) which is recognized as such by all control units attached to the channel.[9] A multiplexor channel also resets the state bits and status bytes in subchannel storage (line 4).

Channel type	*State* B_8^c	B_9^c	S_8^c	S_9^c
both	0	0	0	0

The channel selected for initial program loading recognizes its number in the ***load unit switch*** (line 5) and sets its working registers with the canonical information specified in line 6. It then branches to the selection sequence starting at line 115, from which point on it cannot be distinguished from any other SIO. If the loading fails for any reason, a new attempt must be mounted at the control panel, causing another system reset and returning the channel to line 2.

other I/O system programs

input/output interruption entry

The dwell at lines 1 and 2 of *IOIE* responds to the *CPU* program. When the dwell is broken, the interruption retraction bit g_1 is set to 0 and the unmasked channel of highest priority with an interruption pending is selected for service (line 4). B_{10}^i is set to 1 on line 8 if the channel is not working, with consequences that have been explored above. If the channel is working it will not be able to respond to B_{10}^i directly. This bit can therefore be used as an interlock, and is set on line 11 for this purpose (see *CH* line 72).

A selector channel, or a busy multiplexor channel with working and interruption addresses that match, could only be holding a pci at this point. This is cleared in line 14 following the formation of a CSW in line 13. The return to line 0 then sets B_8^i to zero,

loads the condition code in p, and signals the *CPU* by setting g_0 to zero.

A busy multiplexor channel with non-matching addresses must check the subchannel storage location indicated by the interruption address and, if the interruption is for a pci or termination (line 15), generate a CSW and clear the subchannel (lines 18 and 19) much like *CH* lines 136 and 137. If the subchannel test (line 16) results in the branch to line 17, the bit g_1 is set to 1 and the program returns to line 0 as in the other cases. However, the *CPU* will recognize g_1 (line 30) as an indication that information for the proffered interruption was not available and it will not execute the usual interruption procedure.

burst mode timer and time-out limiter

BMT is a time-limiting clock with two ways of stopping. It is started by setting g_2 in CH^0, line 48 or 165, and it runs until stopped by $B_9^0 = 0$ (line 5) or by its counter running out (line 4).

Where performance requirements permit, channels make use of *CPU* facilities and controls to varying degrees. The effect of this on the logical behavior of the channels is confined to the recognition of hardware failures and consequent corrective action. Program TOL^c is relevant where *CPU* controls are preempted for channel interface operations, so that an independent means for preventing indefinite delays is required. Two such levels are distinguished in TOL^c. If $CM_2^c = 1$, channel c uses *CPU* controls for all interface operations other than polling in the channel idle phase, and if $CM_3^c = 1$, it uses these controls for polling as well. Thus if $CM_2^c = 1$, the dwell on line 1 is broken each time $U_3^c = 1$ and either $CM_3^c = 1$ or one of the interface tag lines is nonzero. (The polling during channel idle is distinguished from all other interface operations by $U_3^c = 1$ and $(\vee/\alpha^3/U^c) = 0$.)

In TOL^c there are two time-limiting clocks in series. The first clock, which is started in line 2, times either the establishment of a connection ($U_6^c = 1$), or the return of select-in ($P_{w_c}^c = 1$), or any response at all ($U_3^c = 0$). The maximum time for the dwell encompassing lines 3–5 is of the order of 32 microseconds. When a connection is established ($U_6^c = 1$) (possibly, in the case of data transfer, even before the first clock is started), the maximum time is of the order of 500 milliseconds. This clock is stopped either by $U_3^c = 0$, indicating a response of some kind from the device, or by $U_6^c = 0$, indicating that the device wishes to disconnect. If either clock runs out before it is stopped, an interface control check is set in line 0 and detected in CH^c in one of lines 63, 117, or 126, or in *HFC* line 1.

hardware failure in channel

Program HFC^c distinguishes between the case where the channel shares both data paths and controls with the *CPU* ($CM_1^c = 1$) on the one hand, and all other degrees of sharing and independence on the other. Thus, in line 1, the dwell is broken in all cases for a *channel control check*[10] (S_5^c), but is not disturbed for either a channel data check (S_4^c) or interface control check (S_6^c) unless $CM_1^c = 1$. Whereas an interface control check will be acted upon by program CH^c in any case, a channel data check will not, and hence

it has no effect on a current operation unless *CPU* hardware is involved. If the dwell is broken and $CM_1^c = 1$ (line 2) the branch to line 3 is taken, stopping CH^c abruptly by a forced branch to its line 0. A machine check is then entered (line 4), which will be recognized by program *MCIE*. The subsequent branch to line 0 invokes the defined operation *MALFUNCTION RESET* (not detailed in this description) which will carry out the (model-dependent) recovery procedure called for by the prevalent circumstances, taking into account the fact that *MCIE* has been alerted by $f_c = 1$. Whatever else it does, line 0 must ultimately cause CH^c to leave the dwell on line 1 with S^c reset to zero.

Channels that do not use *CPU* hardware for data transfer disregard S_4^c and, if they leave line 1, branch to line 5, where a malfunction reset signal ($U_{4,5}^c = 1, 0$) may be issued on the interface.[11]

If the channel is not working directly with the *CPU* at this moment (line 6), the program returns to line 0. Otherwise, CH^c is immobilized (line 7) and *HFC* generates a CSW (line 8) in which any field may be set to zero if there happens to be a parity error in the associated register. After storing either all or part of the CSW (lines 10 or 11) the model-dependent reset is executed on line 1 before returning to the normal dwell.

Appendix

This appendix furnishes a number of examples to illustrate the use of the programs and Tables 3 and 5 for reference in answering specific questions concerning the operation of SYSTEM/360.

What events can cause the *CPU* to enter the stopped state; in particular, can the stopped state be entered with any interruptions pending? Table 3 shows that *operating state* can be set by *CPU* line 34 (to "stop" if the console *rate switch* is not at "process"), by *CP* lines 12 and 15 (of which line 12 sets it to "stop" when the *stop key* is depressed), by *MAC* line 6 (if the current address to memory agrees with the setting of the address switch and other conditions (line 5) are met), by *IPL* line 10 (during initial program load), and by *RESET* line 1. The stopped state (*CPU* line 35) is actually entered only by a branch from line 25 or by a forced branch (*RESET* line 2). In the former case the branch is taken only after all pending interruptions are exhausted, while in the latter case all pending interruptions are cancelled by the reset of h on *RESET* line 0.

Can any of the effective addresses constructed in the instruction fetch phase be captured and stored? A scan of the *EXC* program (limited to the references to a_1 and a_2 indicated in Table 3) shows that LA places the second effective address (prefixed by zeros) in a general register. LA also provides a convenient means of setting any register R^{a_1} to zero.

By what instructions can the system mask α^8/p be set? Table 3 shows that all of p is set on *EXC* line a26 (that is, by LPSW) and that α^8/p is set on *EXC* line a2 (that is, by SSM). Since both

instructions are suppressed by t_2 (lines a1 and a25), both are privileged and could, in a normal operating system, be executed only in the supervisor program.

What instructions are included in the floating-point feature? Table 3 shows that the feature options are specified by the machine characteristics vector m, and that m_2 identifies the floating-point option. The occurrences of m_2 in column N_0 of Table 5 therefore identify the floating-point instructions.

Can data be transferred directly (i.e., not via memory) between a general register R^i and a floating-point register F^i? Comparing the "Results" column of Table 5 with column N_0 shows that F is set only by floating-point instructions and R is never set by floating-point instructions. Since R is set only by non-floating-point instructions and since Table 3 shows that F is referred to only in floating-point instructions (segment k of *EXC*), direct transfer from F to R is impossible. Similarly, since F is set only by segment k of *EXC*, it remains only to scan the argument fetch portion (k0–16) to see that R does not occur.

Under what circumstances does the interval timer fail to record elapsed time? The entry for *timer alarm* in Table 3 refers to the *TU* program whose dwell on line 0 contains the conditions of interest. In particular, the last term prevents normal decrementation during the dwell on line a17 of the read direct instruction. Moreover, normal decrementation is delayed by the use of MAC^8 on *TU* line 1 if the channels (which have a higher priority) keep the memory-access facility occupied.

Can the console operator display the contents of the interval timer and tell if it is decreasing appropriately by watching the display lights flicker? Line 18 of the *CP* program shows that any memory location (in this case 80) selected by the ***address switch*** can be displayed. However, this segment of the program (15–25) can, because of line 13, be reached only if the *CPU* is first stopped. Any displayed value is therefore fixed. In particular, the timer is not updated when the *CPU* is stopped (*TU* line 0).

How are program interruptions caused by shift instructions; what determines the amount of shift? Line e1 sets t_6 (which the footnote to Table 5 shows to be the "specification exception") for an odd first address in instructions SLDA, SRDA, SLDL, and SRDL. Line e9 sets t_8 (fixed point overflow) for instructions SLA and SLDA if the mask p_{36} is on and a significant (differing from the sign) bit has been lost in the shift. The shifting is performed on lines e5, e6, e8 and e10, and the amount of shift is determined on line e3 as the residue modulo 64 of the second effective address.

What instructions employ three specifiable addresses? Table 3 refers to *EXC* b18, 19, d17, 23 for effective address a_3; the instructions involved are therefore BXH, BXLE, LM, and STM.

ACKNOWLEDGMENT

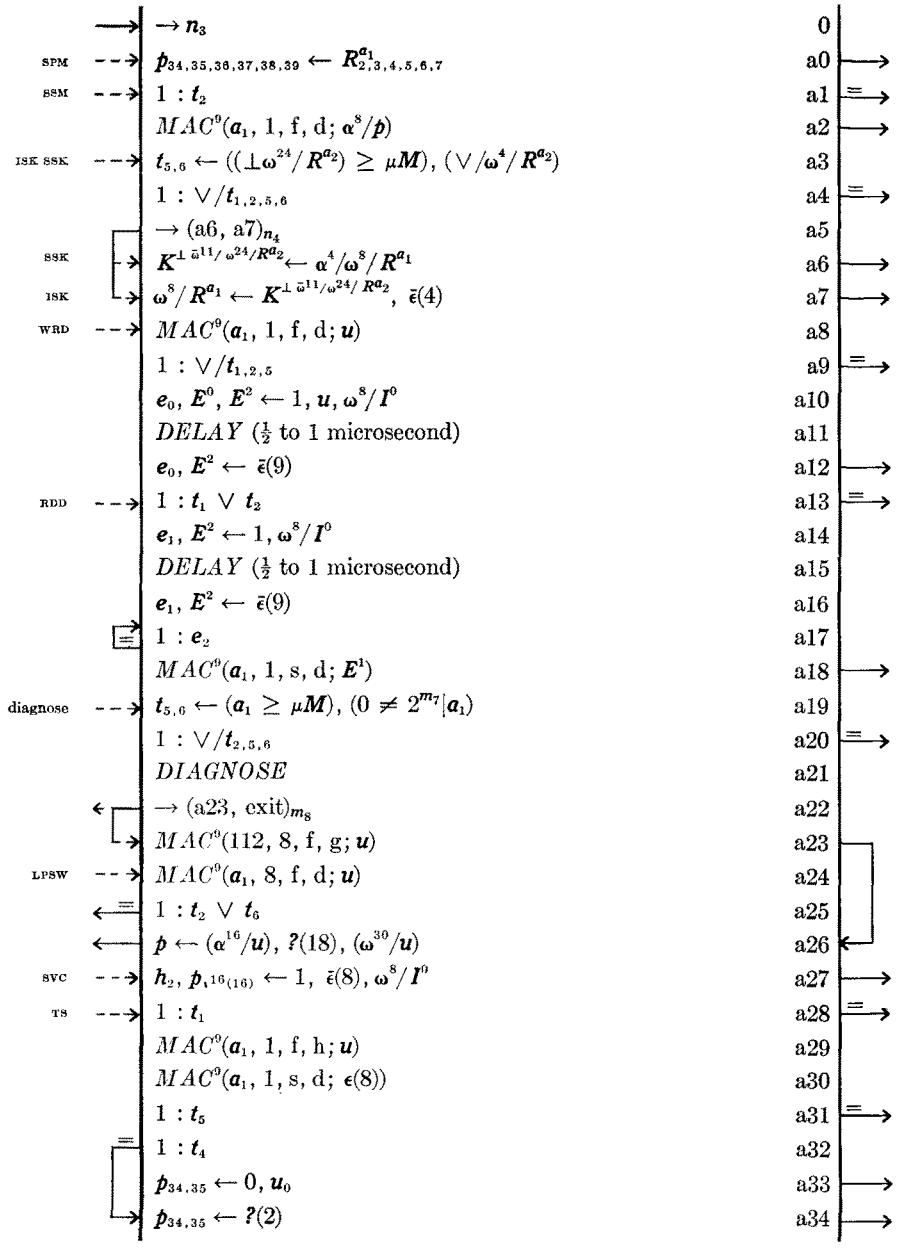
We are greatly indebted to many of our colleagues: to W. C. Carter of the Data Systems Division for suggesting the work; to C. H. Liu

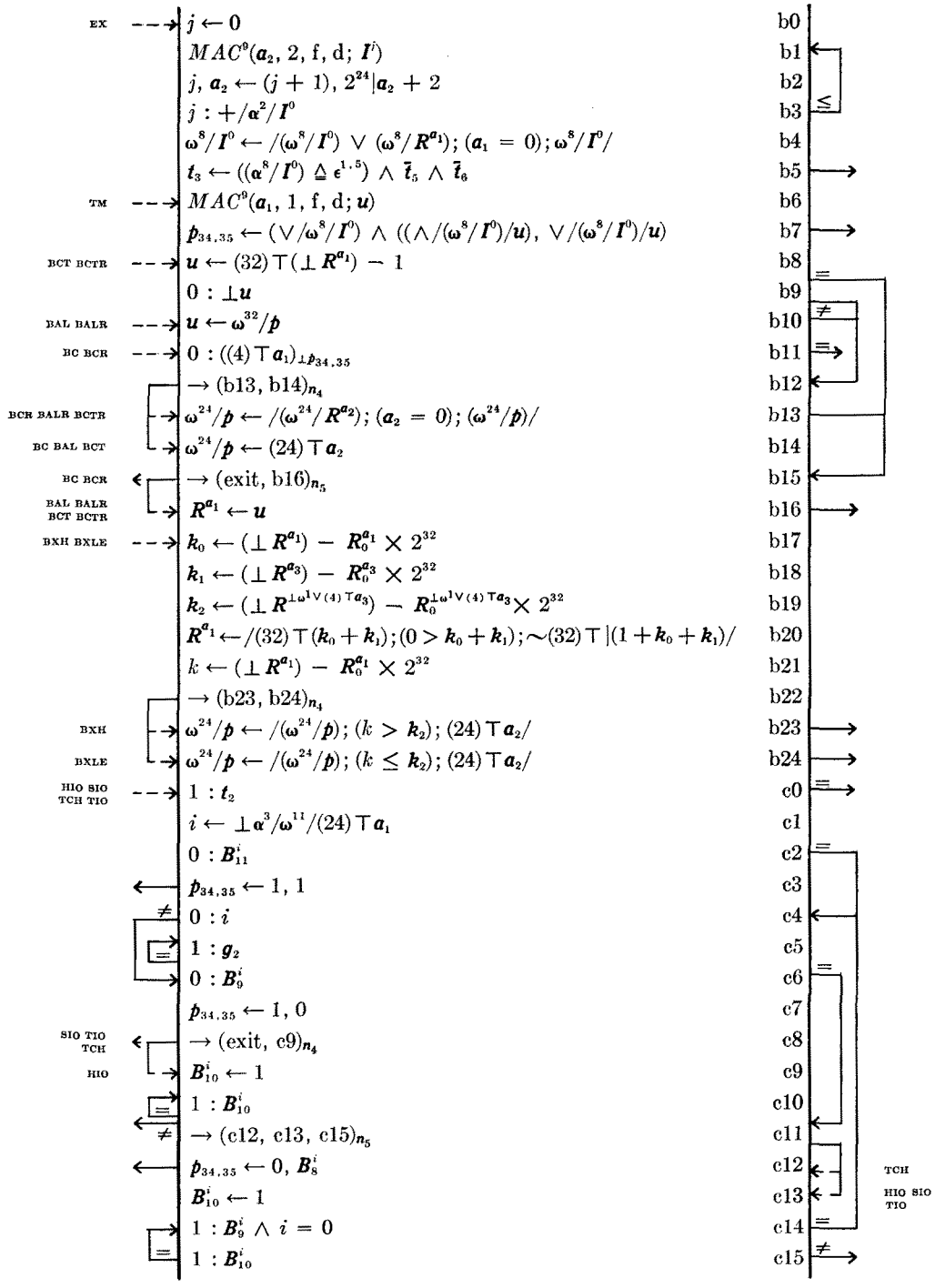
for assistance in the early stages; to IBM SYSTEM/360 architects G. A. Blaauw and A. Padegs for the patient exposition and checking of countless points; to J. C. McPherson, Director of the IBM Systems Research Institute for an opportunity to present the material in a course; and to members of the Institute class for helpful comments and corrections.

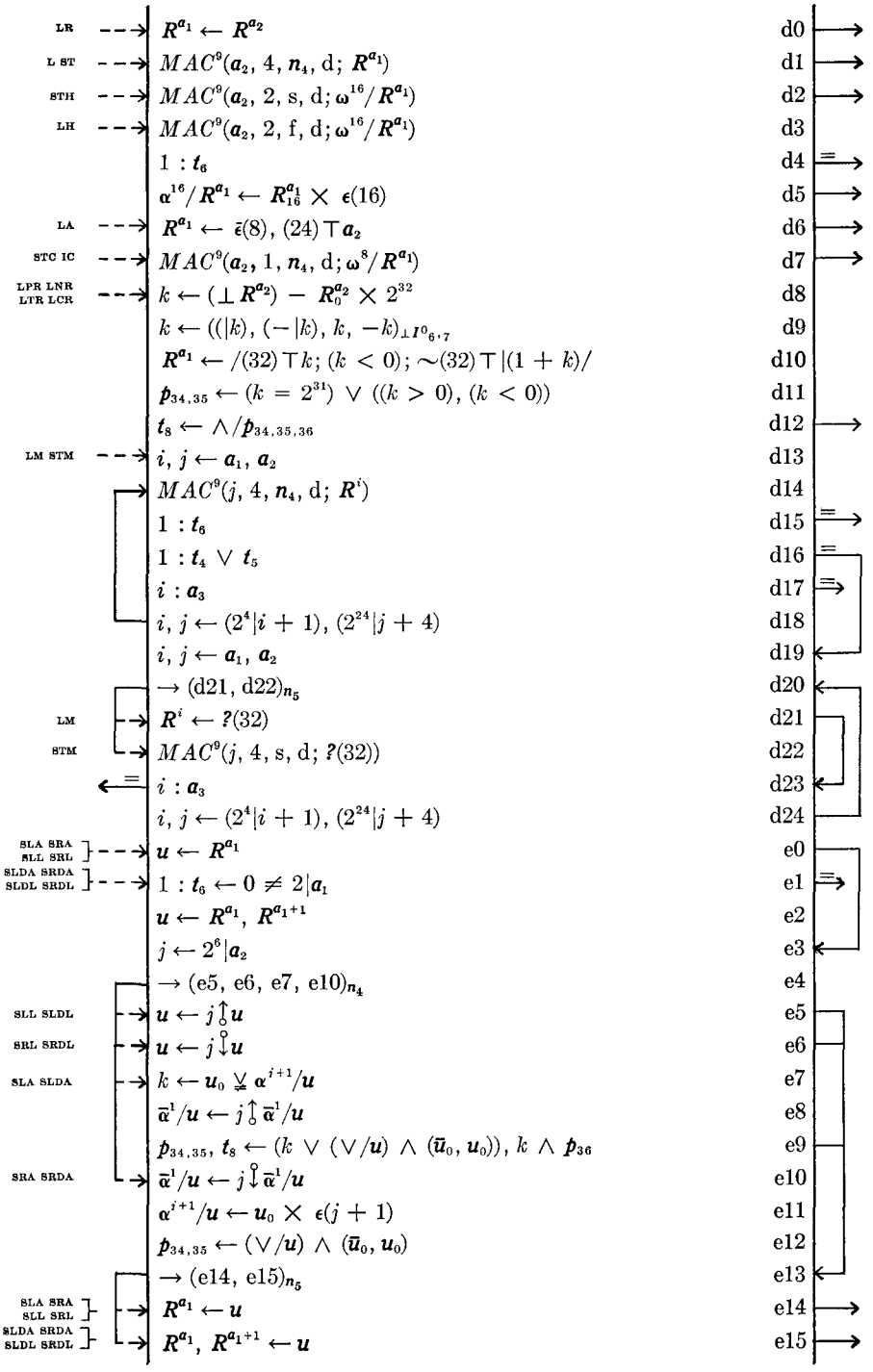
CITED REFERENCES AND FOOTNOTES

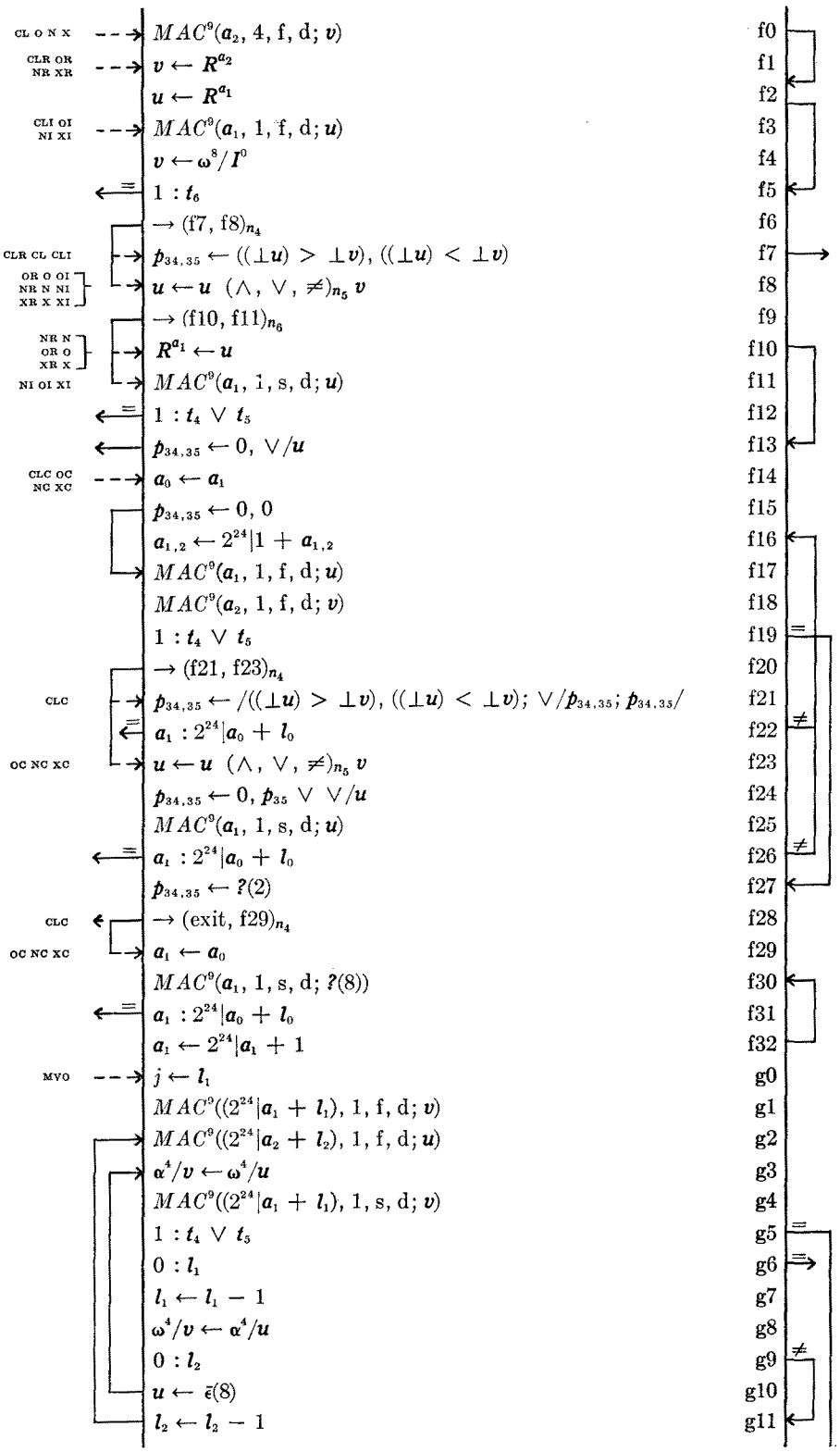
1. IBM SYSTEM/360 Principles of Operation, Form A22-6821-0, IBM CORP., 1964.
2. The present description is carefully checked against the specifications of IBM SYSTEM/360 available in August, 1964. It is, however, not official; because of normal revisions of the system and possible errors in the present paper, the machine manuals issued by IBM will, as usual, be the final arbiters of system specifications.
3. The programs and tables have been reprinted on large display sheets to facilitate study. Copies may be obtained on request to the *IBM Systems Journal.*
4. K. E. Iverson, *A Programming Language*, John Wiley and Sons, 1962.
5. A. D. Falkoff, "Algorithms for Parallel-Search Memories", *Journal of the ACM* 9, No. 4, 488, (October, 1962).
6. K. E. Iverson, "Programming notation for systems design", *IBM Systems Journal* 2, 117, (June, 1963).
7. To provide complete freedom in the attachment of devices in SYSTEM/360, the hardware for the interface has been standardized with respect to both electrical characteristics and logical behavior. The correspondence between this hardware and the representation used here is as follows. The hardware has a separate bus and separate tags for each direction of data flow, and therefore has no need for U_3^c. There is an exact correspondence between physical control lines and each of $U_{4,5,6}^c$, and P^c. The hardware has, in addition, two lines not represented by U^c or P^c. A line called "hold-out" is not needed here because its function has been incorporated in the treatment of P^c, which differs in detail from the hardware select-out; and a line called "request-in" appears in lines 37 and 38 of *CH* only as a logical function of the variables V_8 and V_{12} which are set by the control units.
8. The hardware analog of this formulation is derived by identifying with each channel number available to a control unit a wire from it to the appropriate channel. Within each channel all such wires are *or*ed together. Putting a number in V_{12}^u and setting V_8^u then corresponds to selecting an outgoing wire in the control unit and signalling on it.
9. "If [a] device is currently communicating over the I/O interface, the device immediately disconnects from the channel. Data transfer and any operation using the facilities of the control unit are immediately terminated, and the I/O device is not necessarily positioned at the beginning of a block. Mechanical motion not involving the use of the control unit, such as rewinding magnetic tape or positioning a disk access mechanism, proceeds to the normal stopping point, if possible. The device remains unavailable until the termination of mechanical motion or the inherent cycle of operation, if any, whereupon it becomes available. Status information in the device and control unit is reset, and no interruption condition is generated upon completing the operation." (Ref. 1, p. 90).
10. Like machine checks in other parts of the system, which depend upon sensing the parity in various machine registers and data paths, the generation of S_5^c is not shown in this description.
11. This reset signal is detected only by a device that is connected to the channel at the time, and will cause this device and its control unit to react exactly as for a system reset (see note 9), except that an interruption condition *may* be generated when a mechanical operation is completed.

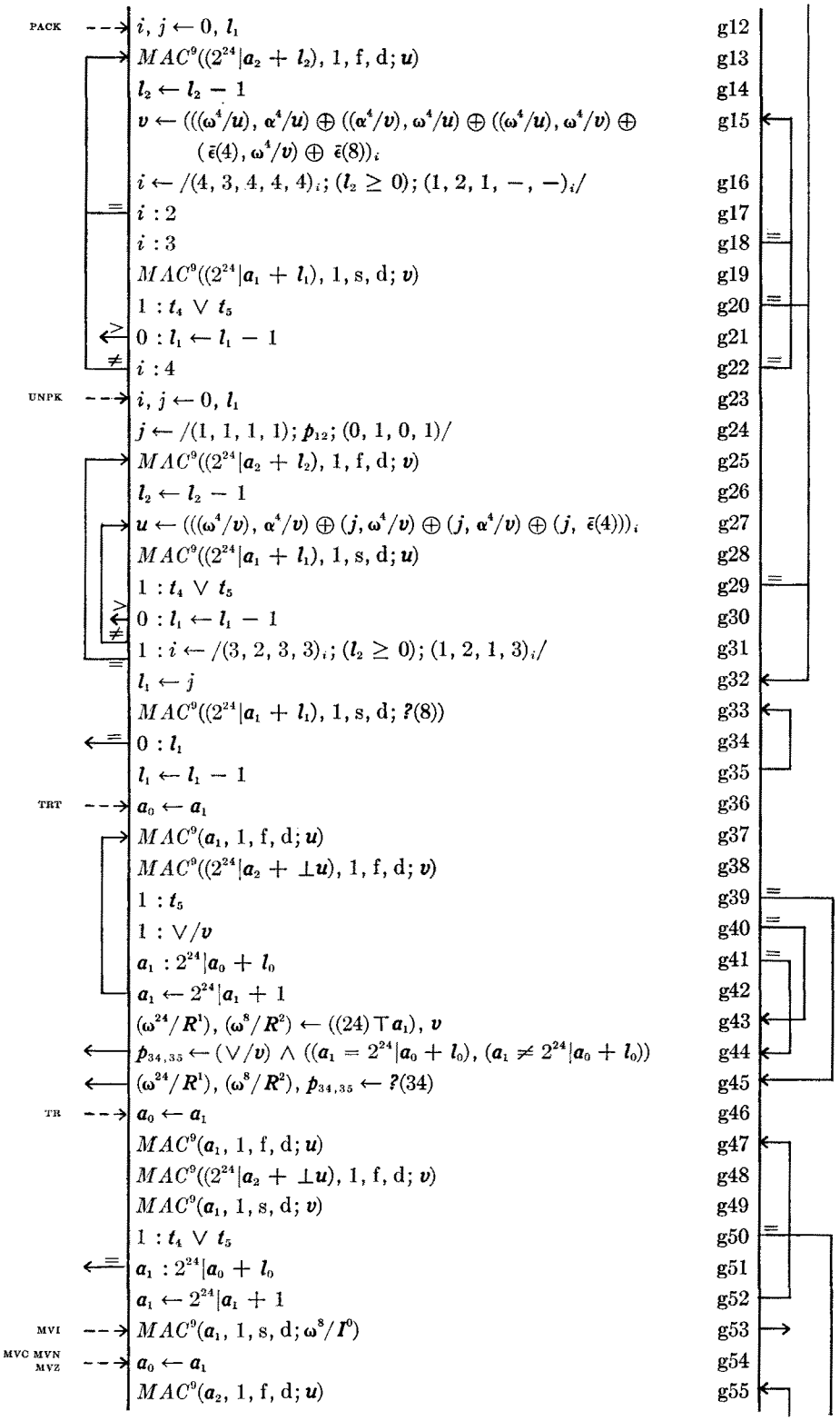
EXC, instruction execution *defined operation*

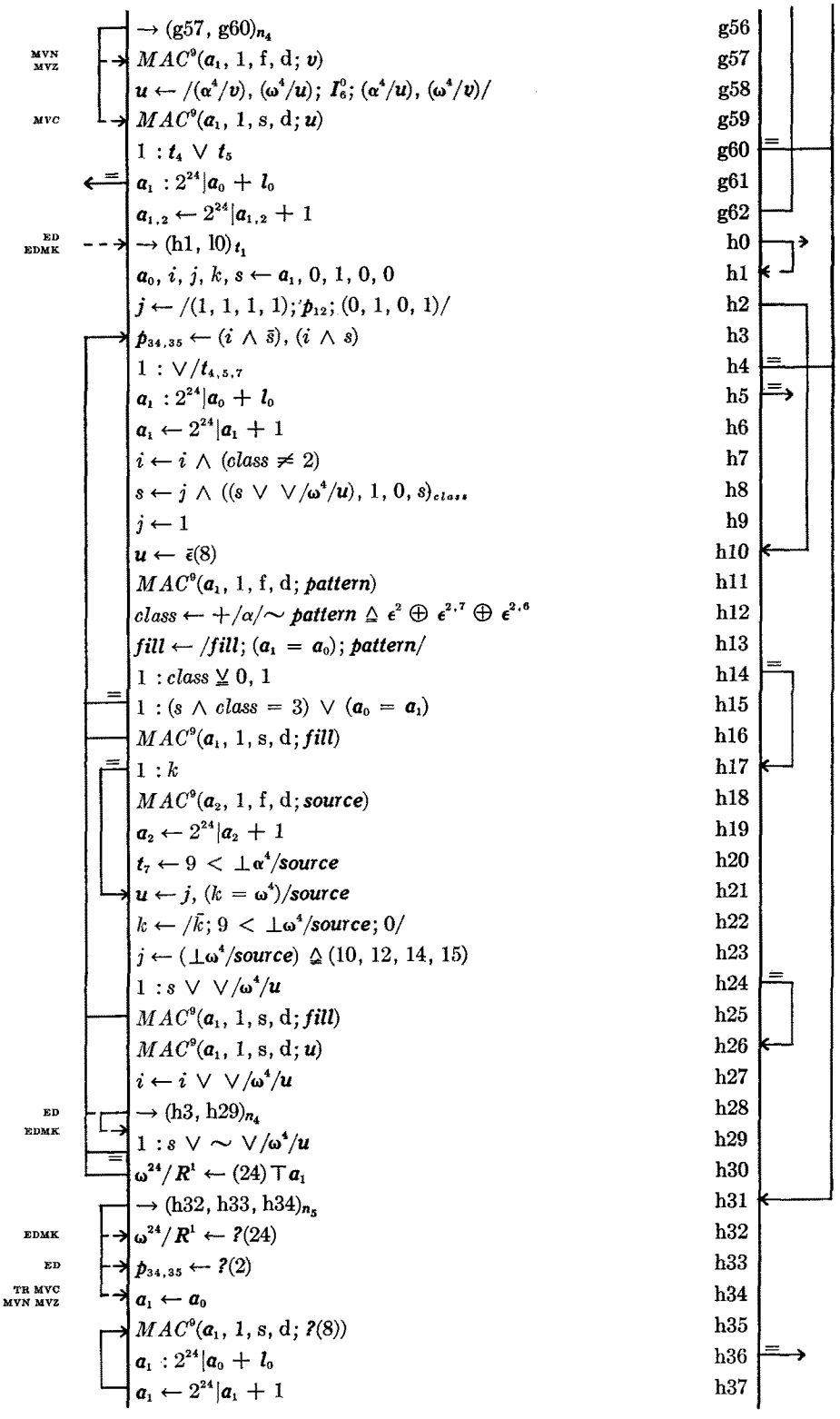


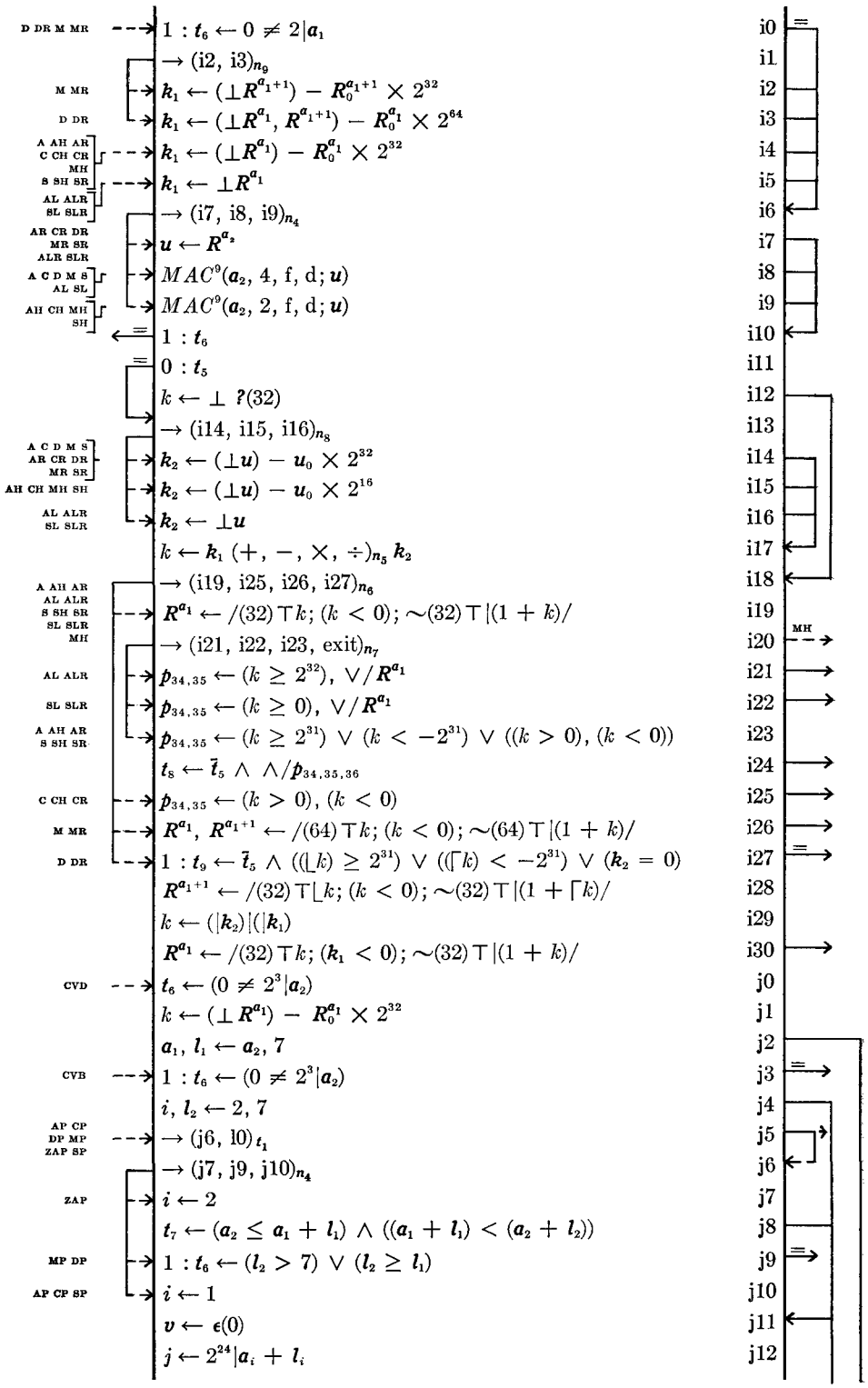

EX
TM
BCT BCTR
BAL BALR
BC BCR
BCR BALR BCTR
BC BAL BCT
BC BCR
BAL BALR BCT BCTR
BXH BXLE
BXH
BXLE
HIO SIO TCH TIO
SIO TIO TCH
HIO
TCH
HIO SIO TIO
j ← 0
MAC⁹(a₂, 2, f, d; I⁰)
j, a₂ ← (j + 1), 2²⁴|a₂ + 2
j : +/α²/I⁰
ω⁸/I⁰ ← /(ω⁸/I⁰) ∨ (ω⁸/R^a1); (a₁ = 0); ω⁸/I⁰/
t₃ ← ((α⁸/I⁰) ≙ ε¹·⁵) ∧ t̄₅ ∧ t̄₆
MAC⁹(a₁, 1, f, d; u)
p₃₄,₃₅ ← (∨/ω⁸/I⁰) ∧ ((∧/(ω⁸/I⁰)/u), ∨/(ω⁸/I⁰)/u)
u ← (32)⊤(⊥R^a1) − 1
0 : ⊥u
u ← ω³²/p
0 : ((4)⊤a₁)⊥p₃₄,₃₅
→ (b13, b14)n4
ω²⁴/p ← /(ω²⁴/R^a2); (a₂ = 0); (ω²⁴/p)/
ω²⁴/p ← (24)⊤a₂
→ (exit, b16)n5
R^a1 ← u
k₀ ← (⊥R^a1) − R₀^a1 × 2³²
k₁ ← (⊥R^a3) − R₀^a3 × 2³²
k₂ ← (⊥R^⊥ω¹∨(4)⊤a3) − R₀^⊥ω¹∨(4)⊤a3 × 2³²
R^a1 ← /(32)⊤(k₀ + k₁); (0 > k₀ + k₁); ∼(32)⊤|(1 + k₀ + k₁)/
k ← (⊥R^a1) − R₀^a1 × 2³²
→ (b23, b24)n4
ω²⁴/p ← /(ω²⁴/p); (k > k₂); (24)⊤a₂/
ω²⁴/p ← /(ω²⁴/p); (k ≤ k₂); (24)⊤a₂/
1 : t₂
i ← ⊥α³/ω¹¹/(24)⊤a₁
0 : B^i_11
p₃₄,₃₅ ← 1, 1
0 : i
1 : g₂
0 : B^i_9
p₃₄,₃₅ ← 1, 0
→ (exit, c9)n4
B^i_10 ← 1
1 : B^i_10
→ (c12, c13, c15)n5
p₃₄,₃₅ ← 0, B^i_8
B^i_10 ← 1
1 : B^i_9 ∧ i = 0
1 : B^i_10
b0
b1
b2
b3
b4
b5
b6
b7
b8
b9
b10
b11
b12
b13
b14
b15
b16
b17
b18
b19
b20
b21
b22
b23
b24
c0
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12
c13
c14
c15

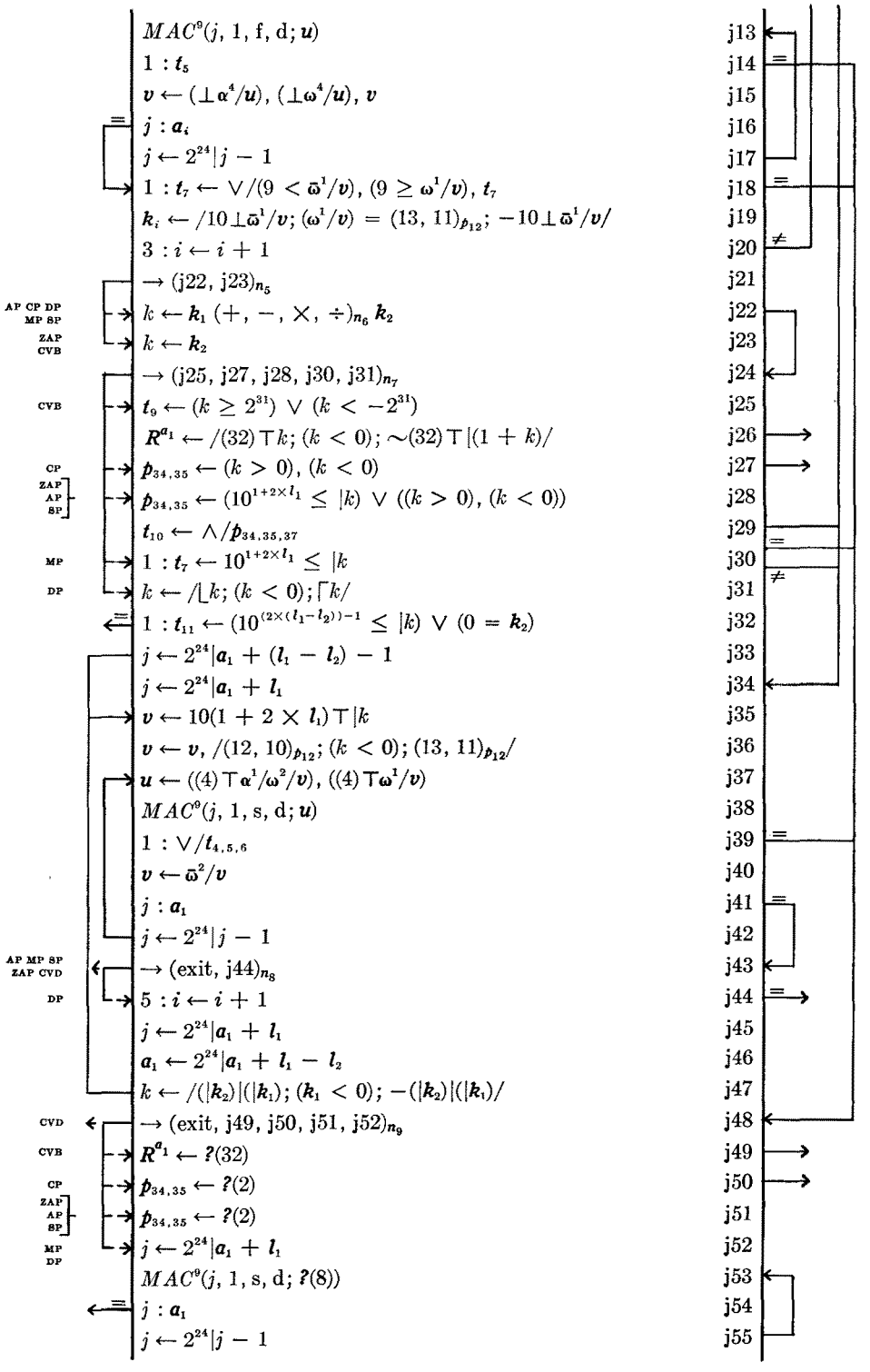

LR
L ST
STH
LH
LA
STC IC
LPR LNR
LTR LCR
LM STM
LM
STM
SLA SRA
SLL SRL
SLDA SRDA
SLDL SRDL
SLL SLDL
SRL SRDL
SLA SLDA
SRA SRDA
SLA SRA
SLL SRL
SLDA SRDA
SLDL SRDL
$R^{a_1} \leftarrow R^{a_2}$
$MAC^9(a_2, 4, n_4, \mathrm{d}; R^{a_1})$
$MAC^9(a_2, 2, \mathrm{s}, \mathrm{d}; \omega^{16}/R^{a_1})$
$MAC^9(a_2, 2, \mathrm{f}, \mathrm{d}; \omega^{16}/R^{a_1})$
$1 : t_6$
$\alpha^{16}/R^{a_1} \leftarrow R^{a_1}_{16} \times \epsilon(16)$
$R^{a_1} \leftarrow \bar{\epsilon}(8), (24) \top a_2$
$MAC^9(a_2, 1, n_4, \mathrm{d}; \omega^8/R^{a_1})$
$k \leftarrow (\perp R^{a_2}) - R^{a_2}_0 \times 2^{32}$
$k \leftarrow ((|k), (-|k), k, -k)_{\perp I 0_{6,7}}$
$R^{a_1} \leftarrow /(32) \top k; (k < 0); \sim(32) \top |(1 + k)/$
$p_{34,35} \leftarrow (k = 2^{31}) \vee ((k > 0), (k < 0))$
$t_8 \leftarrow \wedge/p_{34,35,36}$
$i, j \leftarrow a_1, a_2$
$MAC^9(j, 4, n_4, \mathrm{d}; R^i)$
$1 : t_6$
$1 : t_4 \vee t_5$
$i : a_3$
$i, j \leftarrow (2^4|i + 1), (2^{24}|j + 4)$
$i, j \leftarrow a_1, a_2$
$\rightarrow (\mathrm{d21}, \mathrm{d22})_{n_5}$
$R^i \leftarrow ?(32)$
$MAC^9(j, 4, \mathrm{s}, \mathrm{d}; ?(32))$
$i : a_3$
$i, j \leftarrow (2^4|i + 1), (2^{24}|j + 4)$
$u \leftarrow R^{a_1}$
$1 : t_6 \leftarrow 0 \neq 2|a_1$
$u \leftarrow R^{a_1}, R^{a_1+1}$
$j \leftarrow 2^6|a_2$
$\rightarrow (\mathrm{e5}, \mathrm{e6}, \mathrm{e7}, \mathrm{e10})_{n_4}$
$u \leftarrow j \updownarrow_0 u$
$u \leftarrow j \updownarrow_0 u$
$k \leftarrow u_0 \veebar \alpha^{i+1}/u$
$\bar{\alpha}^1/u \leftarrow j \updownarrow_0 \bar{\alpha}^1/u$
$p_{34,35}, t_8 \leftarrow (k \vee (\vee/u) \wedge (\bar{u}_0, u_0)), k \wedge p_{36}$
$\bar{\alpha}^1/u \leftarrow j \updownarrow_0 \bar{\alpha}^1/u$
$\alpha^{i+1}/u \leftarrow u_0 \times \epsilon(j + 1)$
$p_{34,35} \leftarrow (\vee/u) \wedge (\bar{u}_0, u_0)$
$\rightarrow (\mathrm{e14}, \mathrm{e15})_{n_5}$
$R^{a_1} \leftarrow u$
$R^{a_1}, R^{a_1+1} \leftarrow u$
d0
d1
d2
d3
d4
d5
d6
d7
d8
d9
d10
d11
d12
d13
d14
d15
d16
d17
d18
d19
d20
d21
d22
d23
d24
e0
e1
e2
e3
e4
e5
e6
e7
e8
e9
e10
e11
e12
e13
e14
e15

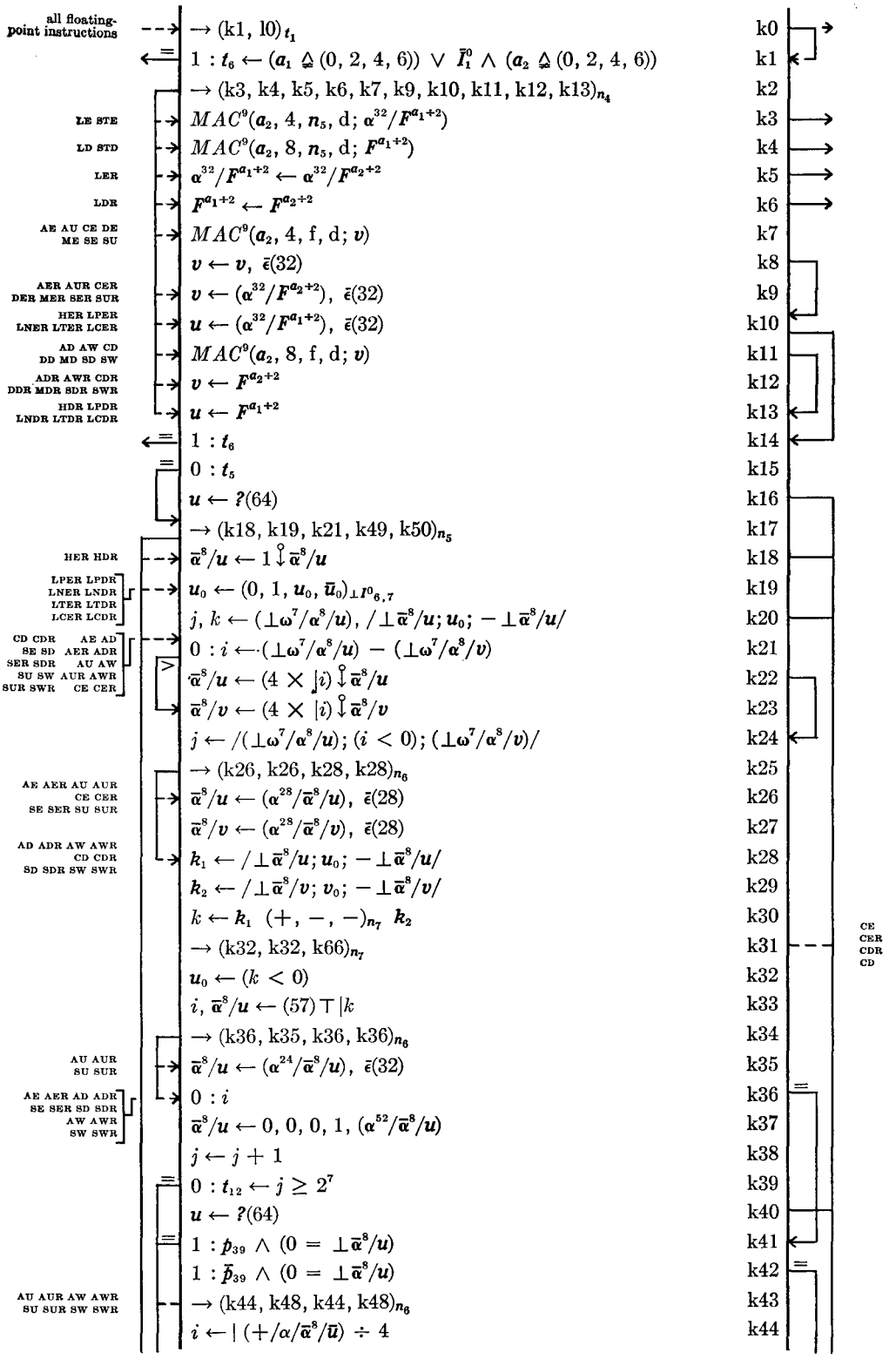

CL O N X
CLR OR NR XR
CLI OI NI XI
CLR CL CLI
OR O OI NR N NI XR X XI
NR N OR O XR X
NI OI XI
CLC OC NC XC
CLC
OC NC XC
CLC
OC NC XC
MVO
$MAC^9(a_2, 4, f, d; v)$
$v \leftarrow R^{a_2}$
$u \leftarrow R^{a_1}$
$MAC^9(a_1, 1, f, d; u)$
$v \leftarrow \omega^8/I^0$
$1 : t_6$
$\rightarrow (f7, f8)_{n_4}$
$p_{34,35} \leftarrow ((\perp u) > \perp v), ((\perp u) < \perp v)$
$u \leftarrow u \ (\wedge, \vee, \not\equiv)_{n_5} v$
$\rightarrow (f10, f11)_{n_6}$
$R^{a_1} \leftarrow u$
$MAC^9(a_1, 1, s, d; u)$
$1 : t_4 \vee t_5$
$p_{34,35} \leftarrow 0, \vee/u$
$a_0 \leftarrow a_1$
$p_{34,35} \leftarrow 0, 0$
$a_{1,2} \leftarrow 2^{24}|1 + a_{1,2}$
$MAC^9(a_1, 1, f, d; u)$
$MAC^9(a_2, 1, f, d; v)$
$1 : t_4 \vee t_5$
$\rightarrow (f21, f23)_{n_4}$
$p_{34,35} \leftarrow /((\perp u) > \perp v), ((\perp u) < \perp v); \vee/p_{34,35}; p_{34,35}/$
$a_1 : 2^{24}|a_0 + l_0$
$u \leftarrow u \ (\wedge, \vee, \not\equiv)_{n_5} v$
$p_{34,35} \leftarrow 0, p_{35} \vee \vee/u$
$MAC^9(a_1, 1, s, d; u)$
$a_1 : 2^{24}|a_0 + l_0$
$p_{34,35} \leftarrow ?(2)$
$\rightarrow (exit, f29)_{n_4}$
$a_1 \leftarrow a_0$
$MAC^9(a_1, 1, s, d; ?(8))$
$a_1 : 2^{24}|a_0 + l_0$
$a_1 \leftarrow 2^{24}|a_1 + 1$
$j \leftarrow l_1$
$MAC^9((2^{24}|a_1 + l_1), 1, f, d; v)$
$MAC^9((2^{24}|a_2 + l_2), 1, f, d; u)$
$\alpha^4/v \leftarrow \omega^4/u$
$MAC^9((2^{24}|a_1 + l_1), 1, s, d; v)$
$1 : t_4 \vee t_5$
$0 : l_1$
$l_1 \leftarrow l_1 - 1$
$\omega^4/v \leftarrow \alpha^4/u$
$0 : l_2$
$u \leftarrow \bar{\epsilon}(8)$
$l_2 \leftarrow l_2 - 1$
f0
f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
f13
f14
f15
f16
f17
f18
f19
f20
f21
f22
f23
f24
f25
f26
f27
f28
f29
f30
f31
f32
g0
g1
g2
g3
g4
g5
g6
g7
g8
g9
g10
g11

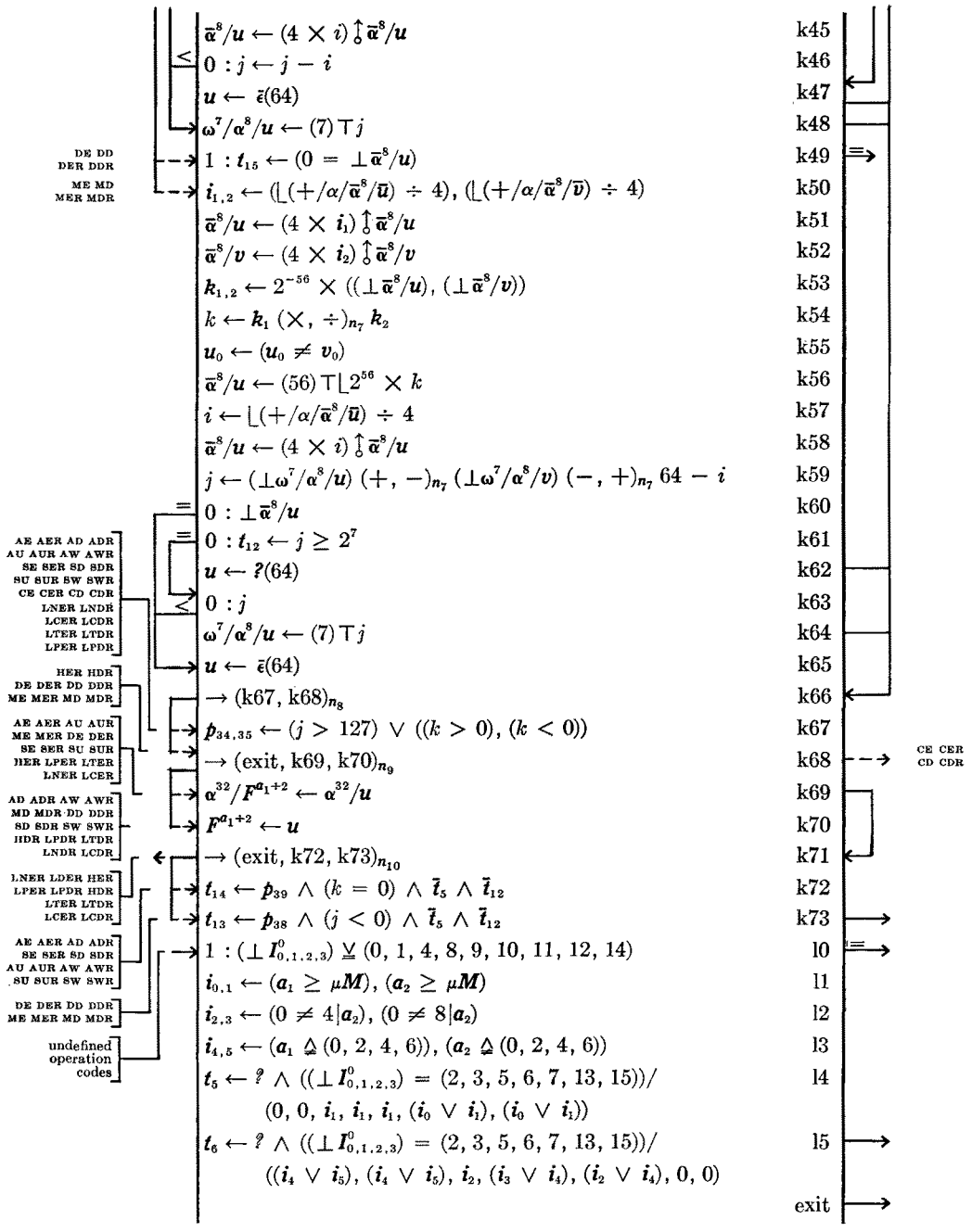

PACK
$i, j \leftarrow 0, l_1$ g12
$MAC^9((2^{24}|a_2 + l_2), 1, f, d; u)$ g13
$l_2 \leftarrow l_2 - 1$ g14
$v \leftarrow (((\omega^4/u), \alpha^4/u) \oplus ((\alpha^4/v), \omega^4/u) \oplus ((\omega^4/u), \omega^4/v) \oplus$ g15
$(\bar{\epsilon}(4), \omega^4/v) \oplus \bar{\epsilon}(8))_i$
$i \leftarrow /(4, 3, 4, 4, 4)_i; (l_2 \geq 0); (1, 2, 1, -, -)_i/$ g16
$i : 2$ g17
$i : 3$ g18
$MAC^9((2^{24}|a_1 + l_1), 1, s, d; v)$ g19
$1 : t_4 \vee t_5$ g20
$0 : l_1 \leftarrow l_1 - 1$ g21
$i : 4$ g22
UNPK
$i, j \leftarrow 0, l_1$ g23
$j \leftarrow /(1, 1, 1, 1); p_{12}; (0, 1, 0, 1)/$ g24
$MAC^9((2^{24}|a_2 + l_2), 1, f, d; v)$ g25
$l_2 \leftarrow l_2 - 1$ g26
$u \leftarrow (((\omega^4/v), \alpha^4/v) \oplus (j, \omega^4/v) \oplus (j, \alpha^4/v) \oplus (j, \bar{\epsilon}(4)))_i$ g27
$MAC^9((2^{24}|a_1 + l_1), 1, s, d; u)$ g28
$1 : t_4 \vee t_5$ g29
$0 : l_1 \leftarrow l_1 - 1$ g30
$1 : i \leftarrow /(3, 2, 3, 3)_i; (l_2 \geq 0); (1, 2, 1, 3)_i/$ g31
$l_1 \leftarrow j$ g32
$MAC^9((2^{24}|a_1 + l_1), 1, s, d; ?(8))$ g33
$0 : l_1$ g34
$l_1 \leftarrow l_1 - 1$ g35
TRT
$a_0 \leftarrow a_1$ g36
$MAC^9(a_1, 1, f, d; u)$ g37
$MAC^9((2^{24}|a_2 + \perp u), 1, f, d; v)$ g38
$1 : t_5$ g39
$1 : \vee/v$ g40
$a_1 : 2^{24}|a_0 + l_0$ g41
$a_1 \leftarrow 2^{24}|a_1 + 1$ g42
$(\omega^{24}/R^1), (\omega^8/R^2) \leftarrow ((24) \top a_1), v$ g43
$p_{34,35} \leftarrow (\vee/v) \wedge ((a_1 = 2^{24}|a_0 + l_0), (a_1 \neq 2^{24}|a_0 + l_0))$ g44
$(\omega^{24}/R^1), (\omega^8/R^2), p_{34,35} \leftarrow ?(34)$ g45
TR
$a_0 \leftarrow a_1$ g46
$MAC^9(a_1, 1, f, d; u)$ g47
$MAC^9((2^{24}|a_2 + \perp u), 1, f, d; v)$ g48
$MAC^9(a_1, 1, s, d; v)$ g49
$1 : t_4 \vee t_5$ g50
$a_1 : 2^{24}|a_0 + l_0$ g51
$a_1 \leftarrow 2^{24}|a_1 + 1$ g52
MVI
$MAC^9(a_1, 1, s, d; \omega^8/I^0)$ g53
MVC MVN MVZ
$a_0 \leftarrow a_1$ g54
$MAC^9(a_2, 1, f, d; u)$ g55


MVN MVZ
MVC
ED EDMK
ED
EDMK
EDMK
ED
TR MVC MVN MVZ
→ (g57, g60)$_{n_4}$ g56
$MAC^9(a_1, 1, f, d; v)$ g57
$u \leftarrow /(\alpha^4/v), (\omega^4/u); I_6^0; (\alpha^4/u), (\omega^4/v)/$ g58
$MAC^9(a_1, 1, s, d; u)$ g59
$1 : t_4 \vee t_5$ g60
$a_1 : 2^{24} | a_0 + l_0$ g61
$a_{1,2} \leftarrow 2^{24} | a_{1,2} + 1$ g62
→ (h1, l0)$_{t_1}$ h0
$a_0, i, j, k, s \leftarrow a_1, 0, 1, 0, 0$ h1
$j \leftarrow /(1, 1, 1, 1); p_{12}; (0, 1, 0, 1)/$ h2
$p_{34,35} \leftarrow (i \wedge \bar{s}), (i \wedge s)$ h3
$1 : \vee/t_{4,5,7}$ h4
$a_1 : 2^{24} | a_0 + l_0$ h5
$a_1 \leftarrow 2^{24} | a_1 + 1$ h6
$i \leftarrow i \wedge (class \neq 2)$ h7
$s \leftarrow j \wedge ((s \vee \vee/\omega^4/u), 1, 0, s)_{class}$ h8
$j \leftarrow 1$ h9
$u \leftarrow \bar{\epsilon}(8)$ h10
$MAC^9(a_1, 1, f, d; pattern)$ h11
$class \leftarrow +/\alpha/\sim pattern \triangleq \epsilon^2 \oplus \epsilon^{2 \cdot 7} \oplus \epsilon^{2 \cdot 6}$ h12
$fill \leftarrow /fill; (a_1 = a_0); pattern/$ h13
$1 : class \leq 0, 1$ h14
$1 : (s \wedge class = 3) \vee (a_0 = a_1)$ h15
$MAC^9(a_1, 1, s, d; fill)$ h16
$1 : k$ h17
$MAC^9(a_2, 1, f, d; source)$ h18
$a_2 \leftarrow 2^{24} | a_2 + 1$ h19
$t_7 \leftarrow 9 < \perp\alpha^4/source$ h20
$u \leftarrow j, (k = \omega^4)/source$ h21
$k \leftarrow /\bar{k}; 9 < \perp\omega^4/source; 0/$ h22
$j \leftarrow (\perp\omega^4/source) \triangleq (10, 12, 14, 15)$ h23
$1 : s \vee \vee/\omega^4/u$ h24
$MAC^9(a_1, 1, s, d; fill)$ h25
$MAC^9(a_1, 1, s, d; u)$ h26
$i \leftarrow i \vee \vee/\omega^4/u$ h27
→ (h3, h29)$_{n_4}$ h28
$1 : s \vee \sim \vee/\omega^4/u$ h29
$\omega^{24}/R^1 \leftarrow (24) \top a_1$ h30
→ (h32, h33, h34)$_{n_5}$ h31
$\omega^{24}/R^1 \leftarrow ?(24)$ h32
$p_{34,35} \leftarrow ?(2)$ h33
$a_1 \leftarrow a_0$ h34
$MAC^9(a_1, 1, s, d; ?(8))$ h35
$a_1 : 2^{24} | a_0 + l_0$ h36
$a_1 \leftarrow 2^{24} | a_1 + 1$ h37


D DR M MR
$1 : t_6 \leftarrow 0 \neq 2|a_1$
i0
$\rightarrow (\text{i2, i3})_{n_9}$
i1
M MR
$k_1 \leftarrow (\perp R^{a_1+1}) - R_0^{a_1+1} \times 2^{32}$
i2
D DR
$k_1 \leftarrow (\perp R^{a_1}, R^{a_1+1}) - R_0^{a_1} \times 2^{64}$
i3
A AH AR
C CH CR
MH
$k_1 \leftarrow (\perp R^{a_1}) - R_0^{a_1} \times 2^{32}$
i4
S SH SR
$k_1 \leftarrow \perp R^{a_1}$
i5
AL ALR
SL SLR
$\rightarrow (\text{i7, i8, i9})_{n_4}$
i6
AR CR DR
MR SR
ALR SLR
$u \leftarrow R^{a_2}$
i7
A C D M S
AL SL
$MAC^9(a_2, 4, \text{f, d}; u)$
i8
AH CH MH
SH
$MAC^9(a_2, 2, \text{f, d}; u)$
i9
$1 : t_6$
i10
$0 : t_5$
i11
$k \leftarrow \perp\ ?(32)$
i12
$\rightarrow (\text{i14, i15, i16})_{n_8}$
i13
A C D M S
AR CR DR
MR SR
$k_2 \leftarrow (\perp u) - u_0 \times 2^{32}$
i14
AH CH MH SH
$k_2 \leftarrow (\perp u) - u_0 \times 2^{16}$
i15
AL ALR
SL SLR
$k_2 \leftarrow \perp u$
i16
$k \leftarrow k_1\ (+, -, \times, \div)_{n_5}\ k_2$
i17
A AH AR
AL ALR
S SH SR
SL SLR
MH
$\rightarrow (\text{i19, i25, i26, i27})_{n_6}$
i18
$R^{a_1} \leftarrow /(32) \top k; (k < 0); \sim(32) \top |(1 + k)/$
i19
MH
$\rightarrow (\text{i21, i22, i23, exit})_{n_7}$
i20
AL ALR
$p_{34,35} \leftarrow (k \geq 2^{32}), \vee/R^{a_1}$
i21
SL SLR
$p_{34,35} \leftarrow (k \geq 0), \vee/R^{a_1}$
i22
A AH AR
S SH SR
$p_{34,35} \leftarrow (k \geq 2^{31}) \vee (k < -2^{31}) \vee ((k > 0), (k < 0))$
i23
$t_8 \leftarrow \bar{t}_5 \wedge \wedge/p_{34,35,36}$
i24
C CH CR
$p_{34,35} \leftarrow (k > 0), (k < 0)$
i25
M MR
$R^{a_1}, R^{a_1+1} \leftarrow /(64) \top k; (k < 0); \sim(64) \top |(1 + k)/$
i26
D DR
$1 : t_9 \leftarrow \bar{t}_5 \wedge ((\lfloor k) \geq 2^{31}) \vee ((\lceil k) < -2^{31}) \vee (k_2 = 0)$
i27
$R^{a_1+1} \leftarrow /(32) \top \lfloor k; (k < 0); \sim(32) \top |(1 + \lceil k)/$
i28
$k \leftarrow (|k_2)|(|k_1)$
i29
$R^{a_1} \leftarrow /(32) \top k; (k_1 < 0); \sim(32) \top |(1 + k)/$
i30
CVD
$t_6 \leftarrow (0 \neq 2^3|a_2)$
j0
$k \leftarrow (\perp R^{a_1}) - R_0^{a_1} \times 2^{32}$
j1
$a_1, l_1 \leftarrow a_2, 7$
j2
CVB
$1 : t_6 \leftarrow (0 \neq 2^3|a_2)$
j3
$i, l_2 \leftarrow 2, 7$
j4
AP CP
DP MP
ZAP SP
$\rightarrow (\text{j6, l0})_{t_1}$
j5
$\rightarrow (\text{j7, j9, j10})_{n_4}$
j6
ZAP
$i \leftarrow 2$
j7
$t_7 \leftarrow (a_2 \leq a_1 + l_1) \wedge ((a_1 + l_1) < (a_2 + l_2))$
j8
MP DP
$1 : t_6 \leftarrow (l_2 > 7) \vee (l_2 \geq l_1)$
j9
AP CP SP
$i \leftarrow 1$
j10
$v \leftarrow \epsilon(0)$
j11
$j \leftarrow 2^{24}|a_i + l_i$
j12


$MAC^9(j, 1, \mathrm{f}, \mathrm{d}; u)$
$1 : t_5$
$v \leftarrow (\perp\alpha^4/u), (\perp\omega^4/u), v$
$j : a_i$
$j \leftarrow 2^{24}|j - 1$
$1 : t_7 \leftarrow \vee/(9 < \bar{\omega}^1/v), (9 \geq \omega^1/v), t_7$
$k_i \leftarrow /10\perp\bar{\omega}^1/v; (\omega^1/v) = (13, 11)_{p_{12}}; -10\perp\bar{\omega}^1/v/$
$3 : i \leftarrow i + 1$
$\rightarrow (\mathrm{j22}, \mathrm{j23})_{n_5}$
AP CP DP MP SP
$k \leftarrow k_1 (+, -, \times, \div)_{n_6} k_2$
ZAP CVB
$k \leftarrow k_2$
$\rightarrow (\mathrm{j25}, \mathrm{j27}, \mathrm{j28}, \mathrm{j30}, \mathrm{j31})_{n_7}$
CVB
$t_9 \leftarrow (k \geq 2^{31}) \vee (k < -2^{31})$
$R^{a_1} \leftarrow /(32)\top k; (k < 0); \sim(32)\top|(1 + k)/$
CP
$p_{34,35} \leftarrow (k > 0), (k < 0)$
ZAP AP SP
$p_{34,35} \leftarrow (10^{1+2\times l_1} \leq |k) \vee ((k > 0), (k < 0))$
$t_{10} \leftarrow \wedge/p_{34,35,37}$
MP
$1 : t_7 \leftarrow 10^{1+2\times l_1} \leq |k$
DP
$k \leftarrow /\lfloor k; (k < 0); \lceil k/$
$1 : t_{11} \leftarrow (10^{(2\times(l_1-l_2))-1} \leq |k) \vee (0 = k_2)$
$j \leftarrow 2^{24}|a_1 + (l_1 - l_2) - 1$
$j \leftarrow 2^{24}|a_1 + l_1$
$v \leftarrow 10(1 + 2 \times l_1)\top|k$
$v \leftarrow v, /(12, 10)_{p_{12}}; (k < 0); (13, 11)_{p_{12}}/$
$u \leftarrow ((4)\top\alpha^1/\omega^2/v), ((4)\top\omega^1/v)$
$MAC^9(j, 1, \mathrm{s}, \mathrm{d}; u)$
$1 : \vee/t_{4,5,6}$
$v \leftarrow \bar{\omega}^2/v$
$j : a_1$
$j \leftarrow 2^{24}|j - 1$
AP MP SP ZAP CVD
$\rightarrow (\mathrm{exit}, \mathrm{j44})_{n_8}$
DP
$5 : i \leftarrow i + 1$
$j \leftarrow 2^{24}|a_1 + l_1$
$a_1 \leftarrow 2^{24}|a_1 + l_1 - l_2$
$k \leftarrow /(|k_2)|(|k_1); (k_1 < 0); -(|k_2)|(|k_1)/$
CVD
$\rightarrow (\mathrm{exit}, \mathrm{j49}, \mathrm{j50}, \mathrm{j51}, \mathrm{j52})_{n_9}$
CVB
$R^{a_1} \leftarrow ?(32)$
CP
$p_{34,35} \leftarrow ?(2)$
ZAP AP SP
$p_{34,35} \leftarrow ?(2)$
MP DP
$j \leftarrow 2^{24}|a_1 + l_1$
$MAC^9(j, 1, \mathrm{s}, \mathrm{d}; ?(8))$
$j : a_1$
$j \leftarrow 2^{24}|j - 1$
j13
j14
j15
j16
j17
j18
j19
j20
j21
j22
j23
j24
j25
j26
j27
j28
j29
j30
j31
j32
j33
j34
j35
j36
j37
j38
j39
j40
j41
j42
j43
j44
j45
j46
j47
j48
j49
j50
j51
j52
j53
j54
j55


all floating-point instructions
LE STE
LD STD
LER
LDR
AE AU CE DE ME SE SU
AER AUR CER DER MER SER SUR
HER LPER LNER LTER LCER
AD AW CD DD MD SD SW
ADR AWR CDR DDR MDR SDR SWR
HDR LPDR LNDR LTDR LCDR
HER HDR
LPER LPDR LNER LNDR LTER LTDR LCER LCDR
CD CDR AE AD SE SD AER ADR SER SDR AU AW SU SW AUR AWR SUR SWR CE CER
AE AER AU AUR CE CER SE SER SU SUR
AD ADR AW AWR CD CDR SD SDR SW SWR
AU AUR SU SUR
AE AER AD ADR SE SER SD SDR AW AWR SW SWR
AU AUR AW AWR SU SUR SW SWR
CE CER CDR CD
k0
k1
k2
k3
k4
k5
k6
k7
k8
k9
k10
k11
k12
k13
k14
k15
k16
k17
k18
k19
k20
k21
k22
k23
k24
k25
k26
k27
k28
k29
k30
k31
k32
k33
k34
k35
k36
k37
k38
k39
k40
k41
k42
k43
k44



FORMAL DESCRIPTION OF SYSTEM/360 249

Table 3 System reference table for programs and variables

symbol	*dimension*	*significance*	*references*: **set**, *used*, [*via local variable*]
BMT		burst mode timer	
CH		channel	*HFC* **3, 7**; *RESET* **3**
CP		control panel	*EP* **3**
CPU		central processing unit	*IPL* **1**; *MCIE* **2**; *RESET* **2**
EIE		external interruption entry	
EP		emergency pull	
ES		external signals	
HFC		hardware failure in channel	
IOIE		input/output interruption entry	
IPL		initial program load	*RESET* **4**
MCIE		machine-check interruption entry	
T		timer	
TOL		time-out limiter	
TU		timer update	
DELAY (k units)		time delay	*BMT* 3; *EXC* a11, 15; *MCIE* 4; *T* 0; *TOL* 3, 7
DIAGNOSE		recovery procedure	*EXC* a21; *MCIE* 7
EXC		instruction execution	*CPU* 20
MAC^i (j; k)		memory access	*CH* 11, 18, 23, 102, 105, 136–159; *CP* 18, 23; *CPU* 3, 31, 32; *EXC*; *IOIE* 13, 18; *IPL* 6, 7; *MCIE* 8; *TU* 1, 3
MALFUNCTION RESET		channel recovery procedure	*HFC* 0
MODEL-DEPENDENT RESET		model-dependent system reset	*RESET* 6
POWER-OFF SEQUENCE		power-off sequence	*CP* 0
POWER-ON SEQUENCE		power-on sequence	*CP* 2
RESET		system reset	*CP* 3, 9
SYSTEM STOP		system shutdown	*EP* 1
B	$\leq$7, 20	channel state	
α^8/B		device status byte	*CH*2, **53, 60, 74, 81,** 136; *HFC* 8; *IOIE* 18; *IPL* 5
B_0		attention	
B_1		status modifier	*CH* 70 [*i*]
B_2		control unit end	
B_3		busy	
B_4		channel end	
B_5		device end	*CH* 69 [*j*]
B_6		unit check	
B_7		unit exception	
B_8		interruption pending in channel	*CH* **2,** 33, 36, 39, 52, **53,** 59, **60,** 73, **74, 81,** 94, **96, 98, 100, 137, 155**; *CPU* 25, 26; *EXC* c12; *IOIE* **0,** 1, 4

Table 3 Continued

symbol	*dimension*	*significance*	*references*: **set**, *used*, [*via local variable*]
B_9		channel working	*BMT* 5; *CH* **2**, **48**, **57**, **58**, 104, 112, 115, 117, 118, 122, 127, **165**; *EXC* c6, 14; *IOIE* **7**, 9
B_{10}		interplay with *CPU*	*CH* **2**, 37, 43, 56, 62, 64, 72, **79**, **134**, **165**; *EXC* c9, 10, **13**, 15; *HFC* 6; IOIE **8**, 10, **11**, **14**
B_{11}		not operational	*CH* **0**, 1, **2**; *EXC* c2
ω^8/B		device address for interruption	*CH* **53**, **60**, 73, **74**, **81**, 82, 88, 92, 97, 99; *IOIE* 0, 12, 15
C	≤ 7, 64	channel command	*CH* **2**, **6**, **27**, **48**, 58, **86**, **113**
α^8/C		command code	*CH* **2**, **6**, 19, 25 [***k***], **48**, **86**, 108 [***k***], 111 [***k***], **113**, 123
C_7		read/write	*CH* 10, 28
$\omega^{24}/\alpha^{32}/C$		data address	*CH* 11, 18, **19**
$\omega^8/\alpha^{32}/C$		device address (temporary)	*CH* **77**, 93
C_{32}		chain-data	*CH* 8, 16, 21, 66, 67, 81, 156
C_{33}		chain-command	*CH* 67, **79**, **91**, 156
C_{34}		suppress length indication	*CH* 8, 66, 81
C_{35}		skip	*CH* 17
C_{36}		program-controlled interruption	*CH* 28, 114
$C_{37,38,39}$		unused positions	*CH* 26 [***k***], 111 [***k***]
$C_{\mathfrak{i}^{40(8)}}$		ignored positions	
ω^{16}/C		count	*CH* 8, 9, 16, **20**, 21, 26 [***k***], 66, 81, 111 [***k***], 136, 154; *HFC* 8
CAW	≤ 7, 32	channel address word	*CH* **2**, **48**, 58, **86**, **102**, 136, 154; *HFC* 8; *IOIE* 13
α^4/CAW		protection key	*CH* 103; *MAC* **10**
$CAW_{4,5,6,7}$		unused positions	*CH* 103
ω^{24}/CAW		command address	*CH* 6, 23, **24**, **31**, **70**, 105, **106**, **110**
CM	≤ 7, 4	channel model characteristics	
CM_0		scan always/on request-in only	*CH* 35
$CM_{1,2,3}$		degrees of hardware sharing between *CPU* and channels	*HFC* 1, 2; *TOL* 1
E	4, 8	external lines	
E^0		direct control out	*EXC* a**10**
E^1		direct control in	*EXC* a18
E^2		timing signal out	*EXC* a**10**, **12**, **14**, **16**
E^3		timing signal in	*ES*
F	4, 64	floating-point registers	*CP* 20, **25**; *EXC* k **3–13**, **69**, **70**; Table **5**
I	3, 16	instruction register	*CPU* **3**, 12–19; *EXC* b**1**; *EXC*
α^8/I^0		instruction code	*CH* 88–93, 101, 122, 123, 143–163; *CPU* **7**, 8; *EXC* b**1**, 3, 5, d9, g58, k1, 19, l0, 4, 5; *HFC* 9
ω^8/I^0		immediate data byte	*EXC* a10, 14, 27, b**4**, 7, f4, g53
^iJ	$-$, 9	from memory in MAC^i operation	*CH* 13, 14; *MAC* **19**, 20, 22

Table 3 Continued

symbol	*dimension*	*significance*	*references*: **set**, *used*, [*via local variable*]
K	$\leq 2^{13}$, 4	memory protection keys (one per bank of 2^{11} bytes)	*EXC* a**6**, 7; *MAC* 10
M	$\leq 2^{24}$, 9	main memory	*CH* 30, 109; *EXC* a3, 19, l1; *MAC* 8, 9, 19, **23**; Table 5 Cols. A and **P**
M_0		parity column	*CH* 13, 14 [$^iJ_0$]
N	144, 11	navigation matrix	*CPU* 9
O	16, 16	decoding matrix	*CPU* 8
P	≤ 7, 9	polling lines	*CH* **2**, **34**, **39**, 40, **44**, **78**, **115**, 116, 117, 119, **121**; *TOL* **5**
R	16, 32	general registers	*CP* 19, **24**; *CPU* 14–19; *EXC*; Table **5**
S	≤ 7, 18	active subchannel status	
α^8/S		channel status byte	*CH* **2**, 9, **48**, 58, 67, **86**, **131**, 136–159; *HFC* 8; *IOIE* 13; *IPL* 5
S_0		program-controlled interruption	*CH* **28**, 59, **114**, 136; *IOIE* **14**
S_1		incorrect length	*CH* **8**, **66**, **81**
S_2		program check	*CH* 12, **26**, **30**, 66, 81, **103**, 107, **109**, **111**, 123, 128, 136, 154, 156; *MAC* **16**
S_3		protection check	*CH* 66, 81, 136, 154; *MAC* **16**
S_4		channel data check	*CH* **13**, **15**; *HFC* 1
S_5		channel control check	*HFC* 1
S_6		interface control check	*CH* **41**, 51, **52**, 62, **63**, 116, **117**, 125, **126**, **141**; *HFC* 1; *TOL* **0**
S_7		chaining check	*CH* **28**, 66, 81
S_8		interruption pending in subchannel	*CH* **2**, 9, **48**, 55, 58, **71**, **81**, **86**, 87–93, 136, **137**, **145**, **148**, 153, **155**
S_9		subchannel working	*CH* **2**, **48**, 50, 58, **71**, **81**, **86**, 87–93, 137, 145, **148**, **165**
ω^8/S		working-device address	*CH* **6**, **45**, 47, 53, 60, 73, 74, 77, 81, **82**, 84, 88, 92, 93, 115, 117; *IOIE* 12
T	≤ 256, 110	subchannel storage	*CH* **4**, 48, **58**, 85, 86, 99; *IOIE* 16, 18, **19**
U	≤ 7, 16	channel/control unit interface	*CH* **7**, **14**, **16**, **39**, **49**, **55**, **68**, **76**, **78**, **96**, **115**, **124**, **130**, **151**, **160**, **164**
U_0		command-out or status-in	*CH* **36**, 41, 52, 63, 117, 120, 126; *TOL* 1
U_1		service-out or service-in	*CH* **36**, 41, 52, 63, 65, 117, 126; *TOL* 1
U_2		address-out or address-in	*CH* **36**, 41, 42, 52, 63, 117, 126; *TOL* 1
U_3		interplay in/out	*CH* 28, 40, 41, 51, 52, 62, 63, 116, 117, 125; *TOL* 1, 5, 9
U_4		suppress-out	*CH* **2**, **29**, **36**; *HFC* **5**
U_5		operational-out	*CH* **2**; *HFC* **5**
U_6		operational-in	*CH* 61; *TOL* 5, 9
U_7		parity bit for bus	*CH* 15, 41, 52, 63, 117, 126
ω^8/U		bus for data or device status	*CH* **11**, 13–15, 18, 41, 45, 52, 53, 63, 67, 68, 74, 117, 126, 128, 129, 149–159
V	≤ 56, 21	control unit status	
V_8		service request	*CH* 37, 38
V_{12}		channel number	*CH* 37, 38

Table 3 Continued

symbol	*dimension*	*significance*	*references*: **set**, *used*, [*via local variable*]
a	4	effective addresses	
a_0		temporary address	
a_1		first address	*CH* 82; *CPU* **16, 19**; *EXC* a, b11, b, c1, d, e, f, g, h, i0–5, 19–30, j, k1–13, 69, 70, l1–3
a_2		second address	*CPU* **12, 15, 18**; *EXC* a3–7, b1, 2, 13, 14, 23, 24, d6, d, e3, f0, 1, 16, 18, g, h18, 19, i7–9, j0–16, k1–12, l1–3
a_3		third address	*CPU* **13**; *EXC* b18, 19, d17, 23
address sw	24	address switch	*CP* 16–25; *MAC* 6
alternate pfx	12	alternate prefix	*MAC* 4
b	8	console buttons	*CP* 5–7
b_0		load key	
b_1		reset key	
b_2		interrupt key	
b_3		stop key	
b_4		start key	
b_5		set-instruction-counter key	
b_6		display key	
b_7		store key	
cpu status	$8 \times m_6$	*CPU* registers stored in diagnosis	*MCIE* 8
data sw	32	data switch	*CP* 16, 23–25
display lights	32	display lights	*CP* **18–21**
e	6	external signal lines	
e_0		write out	*EXC* a**10, 12**
e_1		read out	*EXC* a**14, 16**
e_2		hold in	*EXC* a17; *TU* 0
e_3		machine-check out	*MCIE* **3, 5**
$e_{4,5}$		IPL in-lines	*CP* 5–7
external signals	6	set by E^3	*EIE* 0, 2, **3**; *ES* **0**
f		failures (parity, etc.)	*HFC* 4; *IPL* 9; *MAC* **22**; *MCIE* 0, **9**; *RESET* **0**
fill	8	fill character	*EXC* h**13**, 16, 25
g	3	general interlock bits	*RESET* **0**
g_0		interruption interlock	*CH* 62, 64, 72, 82, 87–92, 117, 122, 123, 132–157; *CPU* **27**, 28; *EIE* 1, **4**; *HFC* 9; *IOIE* **0**, 2; *MCIE* **9**
g_1		I/O interruption retraction	*CPU* 30; *IOIE* **3, 17**
g_2		burst-timer control	*BMT* **0, 1**; *CH* **48, 58, 165**; *EXC* c5; *IOIE* 6
h	5	interruption holder	*CPU* 25, 26, **29**; *RESET* **0**
h_0		machine check	*MCIE* **1**
h_1		program check	*CPU* **24**; *MCIE* **9**
h_2		supervisor call	*EXC* a**27**; *MCIE* **9**
h_3		external	*EIE* **0**
h_4		I/O	*IOIE* **1**

Table 3 Continued

symbol	*dimension*	*significance*	*references*: ***set***, *used*, [*via local variable*]
i, j, k		local variables	
l	3	field lengths	*CPU* **17**; *EXC* f22, 26, 31, g, h5, 36, j
load unit sw	11	load unit switch	*CH* 5, 6; *IPL* 5, 6
m	11	machine (model) characteristics	
m_0		protection feature	*CH* 103; *CPU* 2; *MAC* 11; N_0
m_1		decimal feature	N_0
m_2		floating-point feature	N_0
m_3		direct control feature	N_0
m_4		interlock feature	N_0
m_5		memory width in bytes	*CP* 18, 23; *MAC* 6
m_6		number of bytes in machine check	*MCIE* 8
m_7		number of final 0's in diagnose	*EXC* a19
m_8		diagnose completion option	*EXC* a22
m_9		set IC key option	*CP* 16
m_{10}		burst mode interval	*BMT* 2
main pfx	12	main prefix	*MAC* 4
n	11	navigation vector	*CPU* **9**
n_0		instruction set options (m_0, m_1, m_2, m_3, m_4)	*CPU* 10
n_1		privileged operations	*CPU* 10
n_2		format (RR, RX, RS, SS, SI)	*CPU* 11
n_3		starting line in *EXC*	*CPU* 21; *EXC* 0; *TU* 0
$\bar{\alpha}^4/n$		branch control in *EXC*	*EXC*
p	64	program status word	*CPU* 31, **32**; *EXC* a**26**; *IPL* **7**
α^8/p		system mask	*CPU* 25, 26; *EXC* a**2**; *IOIE* 4
$p_{8,9,10,11}$		protection key	*CPU* 2; *MAC* 10
p_{12}		(extended BCD/American standard) code	*EXC* g24, h2, j19, 36
p_{13}		machine check mask	*MCIE* 0
p_{14}		(running/wait) state	*CPU* 36
p_{15}		(supervisor/program) state	*CPU* 10
$p_1{}^{16}{}_{(16)}$		interruption code	*CPU* **24, 33**; *EIE* **2**, 3; *EXC* a**27**; *IOIE* **0**; *IPL* **8**; *MCIE* **6**
$p_{32,33}$		instruction length code	*CPU* **1–7, 23, 33**; *EXC* b10; *IPL* **8**
$p_{34,35}$		condition code	*CH* **133, 138, 146, 161**; *EXC* b10, b11; Table **5**
p_{36}		fixed-point overflow mask	*EXC* a**0**, b10, d12, e9, i24
p_{37}		decimal overflow mask	*EXC* a**0**, b10, j29
p_{38}		exponent underflow mask	*EXC* a**0**, b10, k73
p_{39}		lost significance mask	*EXC* a**0**, b10, k41, 42, 72
ω^{24}/p		instruction address	*CP* **16**, 21; *CPU* 3, **5**; *EXC* b10, **13, 14, 23, 24**
pattern	8	character from pattern field	*EXC* h**11**, 12, 13

Table 3 Continued

symbol	*dimension*	*significance*	*references*: **set**, *used*, [*via local variable*]
q	10	memory-access queue	*MAC* **0**, 1, **24**
r	10	memory-access request	*MAC* **0**, **2**, 24
rank	10	*MAC* priority (7 channels with 0 in arbitrary position, timer update, *CPU*)	*IOIE* 4; *MAC* 1
s		local variable	
source	8	character from source field	*EXC* h**18**, 20–23
t	16	program exceptions	*CPU* **1–10**, 21–24; *EXC*; *MAC* **15**; Table **5**
t_i		exception with code i	
time-out limit	2	time limits	*TOL* 2, 6
u, v		local variables	
w	≤ 7	number of control units on interface	*CH* 40, 116, 117, 119; *TOL* 5
address compare sw		address compare switch	*MAC* 5
b		index of console button being serviced	*CH* 5; *CP* **6**, 7–14; *IPL* 3, 4
c		channel index	*CH* 5, 37, 38; *CH*; *HFC*; *TOL*
		distinction between selector and multiplexor channels	*CH* 3, 32, 36, 39, 44, 46, 50, 52, 54, 61, 73, 75, 80, 83, 87–95, 121, 145, 148, 155, 165; *EXC* [i] c4, 14; *IOIE* [i] 5, 9, 12
class		(digit select/significance start/field separator/ other)	*EXC* h7, 8, **12**, 14, 15
console interrupt		set by interrupt key	*CP* **11**; *EIE* 0, 2, **3**
emergency pull sw		emergency pull switch	*EP* 0, 2
h		index of interruption being serviced	*CPU* **26**, 27–32; *EIE* 1; *IOIE* 2
i		local variable	
ipl		initial program load	*CH* 5; *CP* **10**; *CPU* 0; *IPL* 0, **10**; *RESET* **1**
j, k		local variables	
load light		on during initial program load	*IPL* **2**, **10**; *RESET* **5**
manual light		on when *CPU* stopped	*CPU* **35**, *TU* 0
operating state		*CPU* operate or stop	*CP* **12**, 13, **15**; *CPU* **34**, 35; *IPL* **10**; *MAC* **5**, **6**; *RESET* **1**
pfx sw		prefix select switch	*IPL* 3
pfx tgr		prefix trigger	*IPL* **3**; *MAC* 4
power-off key		power-off key	*CP* 4
power-on key		power-on key	*CP* 1
rate sw		rate switch	*CPU* 34; *TU* 0
s		local variable	
storage select sw		storage select switch	*CP* 17, 22
t		subchannel index	*CH* **47**, 48, 58, **84**, 85, 86, 97
tick		timer pulse	*T* **1**; *TU* 0, **5**
timer alarm		interval timer alarm	*EIE* 0, 2, **3**; *TU* **4**
timer frequency		50, 60, 300×2^i c.p.s. $(0 \leq i \leq 8)$	*T* 0; *TU* 2
w		wait for timer update	*MAC* **2**, 24
wait light		on during wait state	*CPU* **36**

Table 5 Navigation matrix and reference table

*Navigation matrix N*①													*Machine instruction*②			*Results*③			*Program exceptions*④																
0	1	2	3	4	5	6	7	8	9	10	*Index*	*Mnemonic*	*Name*	*Code*	*R*	*F*	$p_{24,35}$	OP 1	M 2	EX 3	P 4	A 5	S 6	D 7	IF 8	IK 9	DF 10	DK 11	E 12	U 13	LS 14	FK 15
0	0	$\perp I^0_{0,1}$	10	–	–	–	–	–	–	–	0		undefined operation codes					*S*				*S*	*S*												
1	0	1	i4	1	0	0	2	0	–	–	1	A	Add	5A	1		4					*T*	*S*		*C*									
m_2	0	1	k0	7	2	2	0	0	2	1	2	AD	Add Normalized (Long)	6A		1	4	*S*				*T*	*S*						*T*	*C*	*C*			
m_2	0	0	k0	8	2	2	0	0	2	1	3	ADR	Add Normalized (Long)	2A		1	4	*S*					*S*						*T*	*C*	*C*			
m_2	0	1	k0	4	2	0	0	0	1	1	4	AE	Add Normalized (Short)	7A		½	4	*S*				*T*	*S*						*T*	*C*	*C*			
m_2	0	0	k0	5	2	0	0	0	1	1	5	AER	Add Normalized (Short)	3A		½	4	*S*					*S*						*T*	*C*	*C*			
1	0	1	i4	2	0	0	2	1	–	–	6	AH	Add Halfword	4A	1		4					*T*	*S*		*C*									
1	0	1	i5	1	0	0	0	2	–	–	7	AL	Add Logical	5E	1		4					*T*	*S*											
1	0	0	i5	0	0	0	0	2	–	–	8	ALR	Add Logical	1E	1		4																	
m_1	0	3	j5	2	0	0	2	0	3	–	9	AP	Add Decimal	FA			4	*S*			*T*	*T*		*T*			*C*								
1	0	0	i4	0	0	0	2	0	–	–	10	AR	Add	1A	1		4								*C*									
m_2	0	1	k0	4	2	1	0	0	1	1	11	AU	Add Unnormalized (Short)	7E		½	4	*S*				*T*	*S*						*T*		*C*			
m_2	0	0	k0	5	2	1	0	0	1	1	12	AUR	Add Unnormalized (Short)	3E		½	4	*S*					*S*						*T*		*C*			
m_2	0	1	k0	7	2	3	0	0	2	1	13	AW	Add Unnormalized (Long)	6E		1	4	*S*				*T*	*S*						*T*		*C*			
m_2	0	0	k0	8	2	3	0	0	2	1	14	AWR	Add Unnormalized (Long)	2E		1	4	*S*					*S*						*T*		*C*			
1	0	1	b10	1	1	–	–	–	–	–	15	BAL	Branch and Link	45	1																			
1	0	0	b10	0	1	–	–	–	–	–	16	BALR	Branch and Link	05	1																			
1	0	1	b11	1	0	–	–	–	–	–	17	BC	Branch on Condition	47																				
1	0	0	b11	0	0	–	–	–	–	–	18	BCR	Branch on Condition	07																				
1	0	1	b8	1	1	–	–	–	–	–	19	BCT	Branch on Count	46	1																			
1	0	0	b8	0	1	–	–	–	–	–	20	BCTR	Branch on Count	06	1																			
1	0	2	b17	0	–	–	–	–	–	–	21	BXH	Branch on Index High	86	1																			
1	0	2	b17	1	–	–	–	–	–	–	22	BXLE	Branch on Index Low or Equal	87	1																			
1	0	1	i4	1	1	1	–	0	–	–	23	C	Compare	59			3					*T*	*S*											
m_2	0	1	k0	7	2	2	2	0	0	–	24	CD	Compare (Long)	69			3	*S*				*T*	*S*											
m_2	0	0	k0	8	2	2	2	0	0	–	25	CDR	Compare (Long)	29			3	*S*					*S*											
m_2	0	1	k0	4	2	0	2	0	0	–	26	CE	Compare (Short)	79			3	*S*				*T*	*S*											
m_2	0	0	k0	5	2	0	2	0	0	–	27	CER	Compare (Short)	39			3	*S*					*S*											
1	0	1	i4	2	1	1	–	1	–	–	28	CH	Compare Halfword	49			3					*T*	*S*											
1	0	1	f0	0	–	–	–	–	–	–	29	CL	Compare Logical	55			3					*T*	*S*											
1	0	3	f14	0	–	–	–	–	–	–	30	CLC	Compare Logical	D5			3					*T*												
1	0	4	f3	0	–	–	–	–	–	–	31	CLI	Compare Logical	95			3					*T*												
1	0	0	f1	0	–	–	–	–	–	–	32	CLR	Compare Logical	15			3																	
m_1	0	3	j5	2	0	1	1	0	2	–	33	CP	Compare Decimal	F9			3	*S*				*T*		*T*										
1	0	0	i4	0	1	1	–	0	–	–	34	CR	Compare	19			3																	
1	0	1	j3	–	1	–	0	–	1	–	35	CVB	Convert to Binary	4F	1							*T*	*S*	*T*		*C*								
1	0	1	j0	–	–	–	–	0	0	–	36	CVD	Convert to Decimal	4E							*S*	*S*	*S*											
1	0	1	i0	1	3	3	–	0	1	–	37	D	Divide	5D	2							*T*	*S*			*S*								
m_2	0	1	k0	7	3	–	1	1	2	2	38	DD	Divide (Long)	6D		1		*S*				*T*	*S*						*T*	*C*		*S*		
m_2	0	0	k0	8	3	–	1	1	2	2	39	DDR	Divide (Long)	2D		1		*S*					*S*						*T*	*C*		*S*		
m_2	0	1	k0	4	3	–	1	1	1	2	40	DE	Divide (Short)	7D		½		*S*				*T*	*S*						*T*	*C*		*S*		
m_2	0	0	k0	5	3	–	1	1	1	2	41	DER	Divide (Short)	3D		½		*S*					*S*						*T*	*C*		*S*		
1	1	4	a19	–	–	–	–	–	–	–	42	diagnose	Diagnose	83			4		*S*			*S*	*S*											
m_1	0	3	j5	1	0	3	4	1	4	–	43	DP	Divide Decimal	FD				*S*			*T*	*T*	*S*	*T*				*S*						
1	0	0	i0	0	3	3	–	0	1	–	44	DR	Divide	1D	2								*S*			*S*								
m_1	0	3	h0	0	1	–	–	–	–	–	45	ED	Edit	DE			3	*S*			*T*	*T*		*T*										
m_1	0	3	h0	1	0	–	–	–	–	–	46	EDMK	Edit and Mark	DF	1†		3	*S*			*T*	*T*		*T*										
1	0	1	b0	–	–	–	–	–	–	–	47	EX	Execute	44						*S*		*S*	*S*											
m_2	0	0	k0	9	0	–	–	1	2	0	48	HDR	Halve (Long)	24		1		*S*					*S*											
m_2	0	0	k0	6	0	–	–	1	1	0	49	HER	Halve (Short)	34		½		*S*					*S*											
1	1	4	c0	1	1	–	–	–	–	–	50	HIO	Halt I/O	9E			4		*S*															
1	0	1	d7	f	–	–	–	–	–	–	51	IC	Insert Character	43	1							*T*												
m_0	1	0	a3	1	0	0	0	–	–	–	52	ISK	Insert Storage Key	09	1			*S*	*S*			*S*	*S*											
1	0	1	d1	f	–	–	–	–	–	–	53	L	Load	58	1							*T*	*S*											
1	0	1	d6	–	–	–	–	–	–	–	54	LA	Load Address	41	1																			
m_2	0	0	k0	9	1	–	–	0	2	0	55	LCDR	Load Complement (Long)	23		1	3	*S*					*S*											
m_2	0	0	k0	6	1	–	–	0	1	0	56	LCER	Load Complement (Short)	33		½	3	*S*					*S*											
1	0	0	d8	–	–	–	–	–	–	–	57	LCR	Load Complement	13	1		4								*C*								
m_2	0	1	k0	1	f	–	–	–	–	–	58	LD	Load (Long)	68		1		*S*				*T*	*S*											
m_2	0	0	k0	3	–	–	–	–	–	–	59	LDR	Load (Long)	28		1		*S*					*S*											
m_2	0	1	k0	0	f	–	–	–	–	–	60	LE	Load (Short)	78		½		*S*				*T*	*S*											
m_2	0	0	k0	2	–	–	–	–	–	–	61	LER	Load (Short)	38		½		*S*					*S*											
1	0	1	d3	–	–	–	–	–	–	–	62	LH	Load Halfword	48	1							*T*	*S*											
1	0	2	d13	f	0	–	–	–	–	–	63	LM	Load Multiple	98	*							*T*	*S*											
m_2	0	0	k0	9	1	–	–	0	2	0	64	LNDR	Load Negative (Long)	21		1	2	*S*					*S*											
m_2	0	0	k0	6	1	–	–	0	1	0	65	LNER	Load Negative (Short)	31		½	2	*S*					*S*											
1	0	0	d8	–	–	–	–	–	–	–	66	LNR	Load Negative	11	1		2																	
m_2	0	0	k0	9	1	–	–	0	2	0	67	LPDR	Load Positive (Long)	20		1	2	*S*					*S*											
m_2	0	0	k0	6	1	–	–	0	1	0	68	LPER	Load Positive (Short)	30		½	2	*S*					*S*											
1	0	0	d8	–	–	–	–	–	–	–	69	LPR	Load Positive	10	1		3								*C*								
1	1	4	a24	–	–	–	–	–	–	–	70	LPSW	Load PSW	82			4		*S*			*T*	*S*											
1	0	0	d0	–	–	–	–	–	–	–	71	LR	Load	18	1																			
m_2	0	0	k0	9	1	–	–	0	2	0	72	LTDR	Load and Test (Long)	22		1	3	*S*					*S*											
m_2	0	0	k0	6	1	–	–	0	1	0	73	LTER	Load and Test (Short)	32		½	3	*S*					*S*											
1	0	0	d8	–	–	–	–	–	–	–	74	LTR	Load and Test	12	1		3																	
1	0	1	i0	1	2	2	–	0	0	–	75	M	Multiply	5C	2							*T*	*S*											
m_2	0	1	k0	7	4	–	0	1	2	2	76	MD	Multiply (Long)	6C		1		*S*				*T*	*S*						*T*	*C*				
m_2	0	0	k0	8	4	–	0	1	2	2	77	MDR	Multiply (Long)	2C		1		*S*					*S*						*T*	*C*				
m_2	0	1	k0	4	4	–	0	1	2	2	78	ME	Multiply (Short)	7C		½		*S*				*T'*	*S*						*T*	*C*				
m_2	0	0	k0	5	4	–	0	1	2	2	79	MER	Multiply (Short)	3C		½		*S*					*S*						*T*	*C*				

Table 5 Continued

| *Navigation matrix N*① | | |

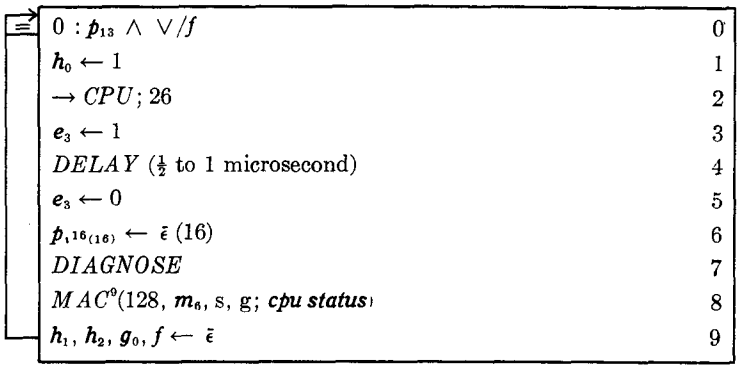
Table 6 Operation decoding matrix O

First Hexadecimal Digit	Second Hexadecimal Digit 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	0	0	0	0	SPM 112	BALR 16	BCTR 20	BCR 18	SSK 118	ISK 52	SVC 128	0	0	0	0	0	RR
1	LPR 69	LNR 66	LTR 74	LCR 57	NR 91	CLR 32	OR 95	XR 142	LR 71	CR 34	AR 10	SR 113	MR 82	DR 44	ALR 8	SLR 110	RR
2	LPDR 67	LNDR 64	LTDR 72	LCDR 55	HDR 48	0	0	0	LDR 59	CDR 25	ADR 3	SDR 100	MDR 77	DDR 39	AWR 14	SWR 130	RR
3	LPER 68	LNER 65	LTER 73	LCER 56	HER 49	0	0	0	LER 61	CER 27	AER 5	SER 102	MER 79	DER 41	AUR 12	SUR 127	RR
4	STH 124	LA 54	STC 121	IC 51	EX 47	BAL 15	BCT 19	BC 17	LH 62	CH 28	AH 6	SH 103	MH 80	0	CVD 36	CVB 35	RX
5	ST 120	0	0	0	N 88	CL 29	O 92	X 139	L 53	C 23	A 1	S 98	M 75	D 37	AL 7	SL 105	RX
6	STD 122	0	0	0	0	0	0	0	LD 58	CD 24	AD 2	SD 99	MD 76	DD 38	AW 13	SW 129	RX
7	STE 123	0	0	0	0	0	0	0	LE 60	CE 26	AE 4	SE 101	ME 78	DE 40	AU 11	SU 126	RX
8	SSM 119	0	LPSW 70	diagnose 42	WRD 138	RDD 97	BXH 21	BXLE 22	SRL 117	SLL 109	SRA 114	SLA 106	SRDL 116	SLDL 108	SRDA 115	SLDA 107	RS or SI
9	STM 125	TM 133	MVI 84	TS 136	NI 90	CLI 31	OI 94	XI 141	LM 63	0	0	0	SIO 104	TIO 132	HIO 50	TCH 131	RS or SI
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	RS or SI
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	RS or SI
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SS
D	0	MVN 85	MVC 83	MVZ 87	NC 89	CLC 30	OC 93	XC 140	0	0	0	0	TR 134	TRT 135	ED 45	EDMK 46	SS
E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SS
F	0	MVO 86	PACK 96	UNPK 137	0	0	0	0	ZAP 143	CP 33	AP 9	SP 111	MP 81	DP 43	0	0	SS

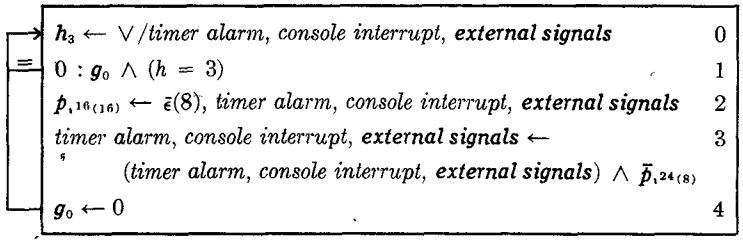
Table 7 Normal uses of effective addresses

Format	*First operand*	*Second operand*
RR (register, register)	R^{a_1} or F^{a_1}	R^{a_2} or F^{a_2}
RX (register, storage indexed)	R^{a_1} or F^{a_1}	M^{a_2}
RS (register, storage)	R^{a_1}	M^{a_2}
SI (storage, immediate)	M^{a_1}	ω^8/I^0
SS (storage, storage)	M^{a_1}	M^{a_2}

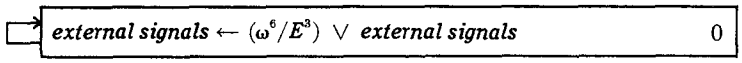
MCIE, machine check interruption entry *system program*



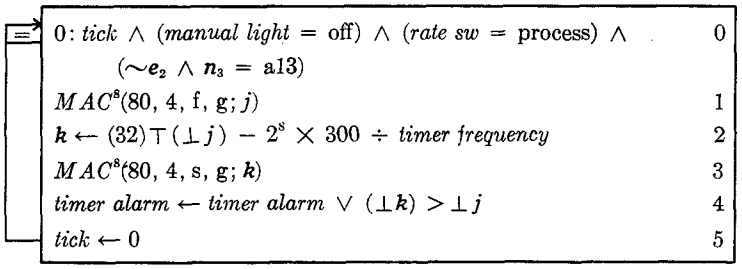
EIE, external interruption entry *system program*



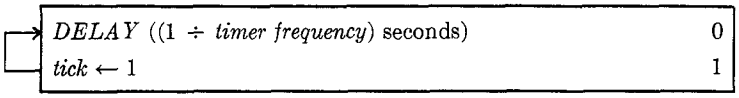
ES, external signals *system program*



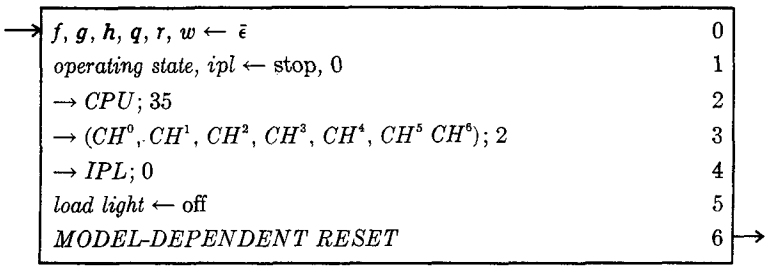
TU, timer update *system program*



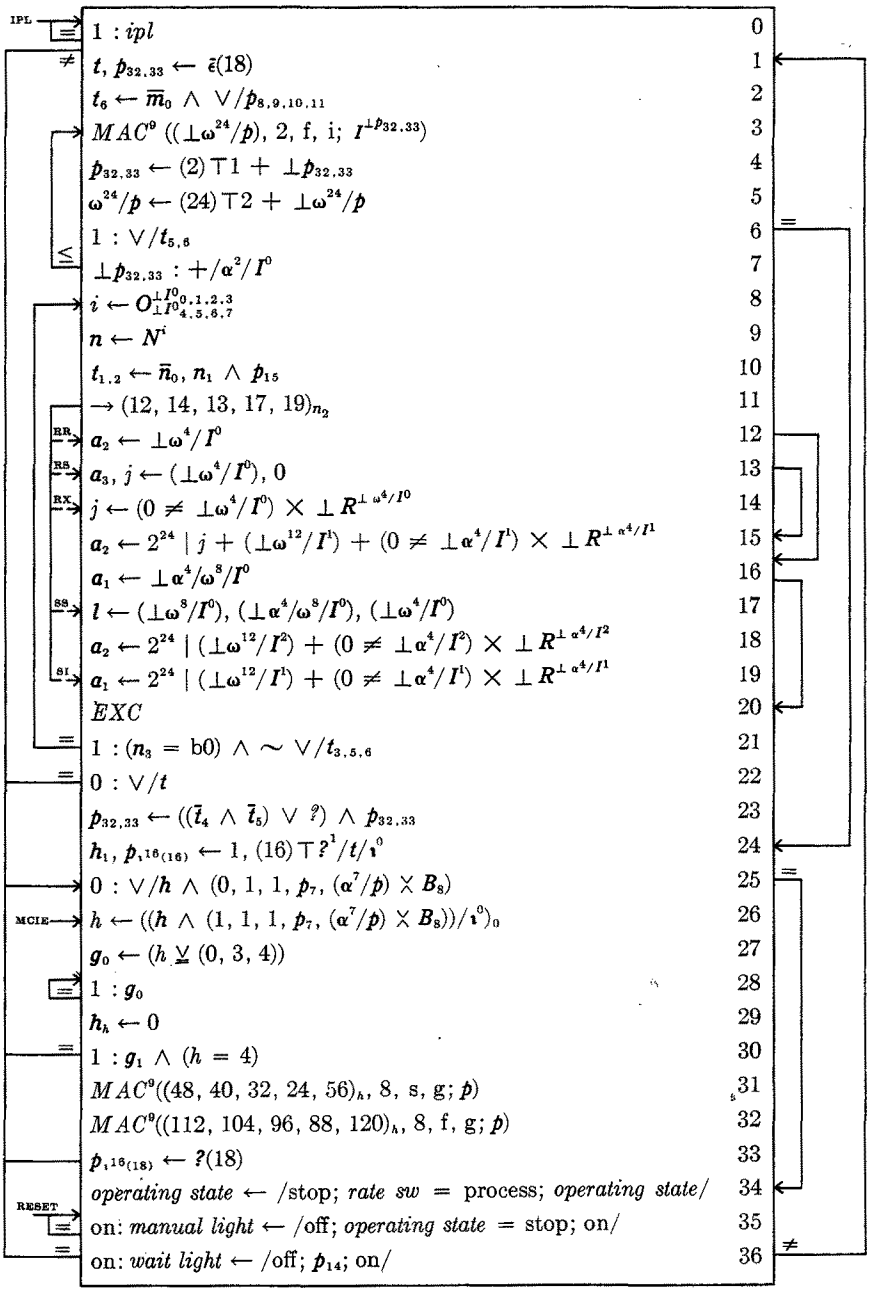
T, timer *system program*



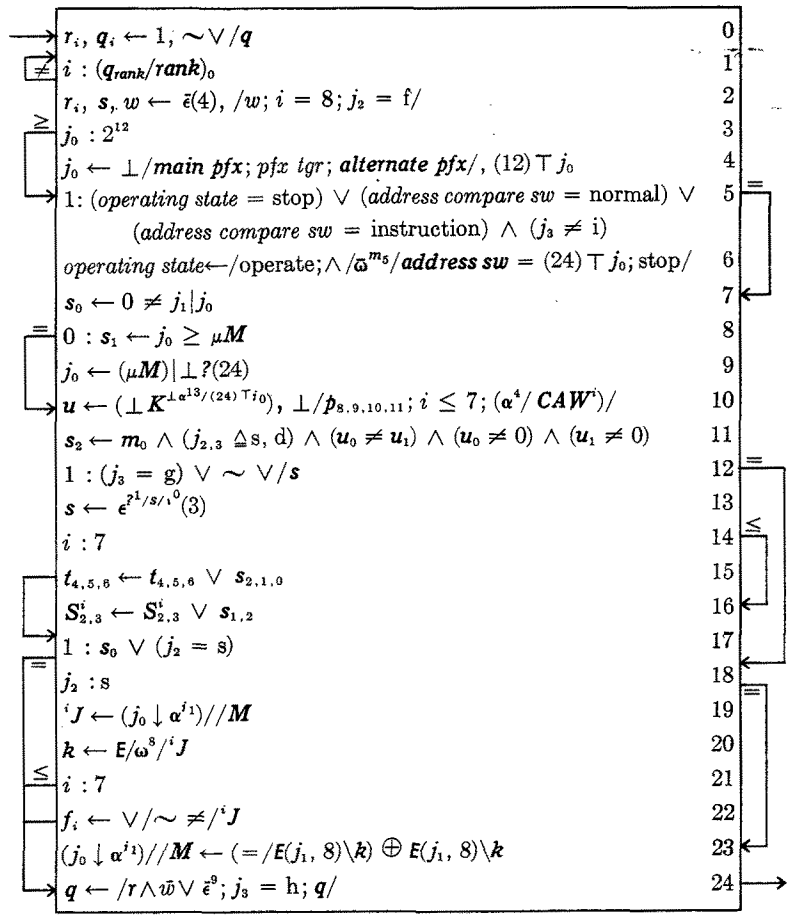
RESET, reset *defined operation*



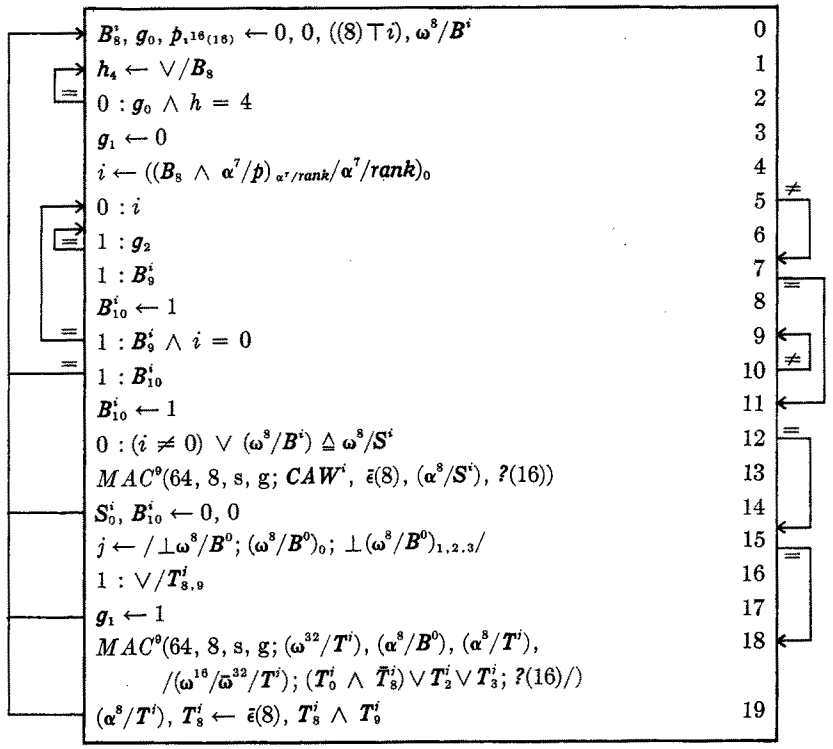
CPU, central processing unit *system program*



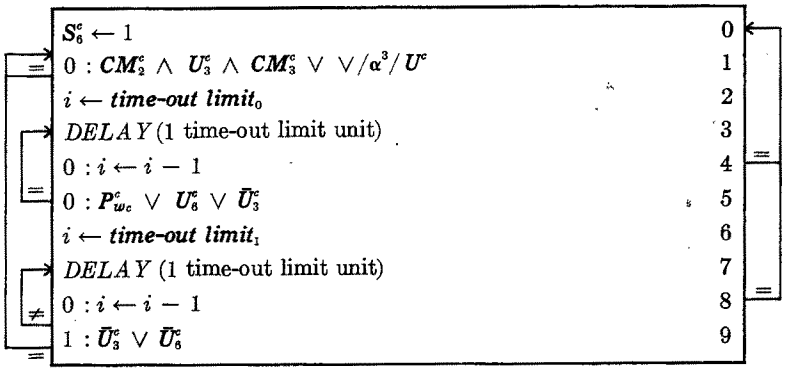
$MAC^{i}(j; k)$, memory access *defined operation*



IOIE, input/output interruption entry *system program*



TOL^c, time-out limiter *system program*



HFC^c, hardware failure in channel *c system program*

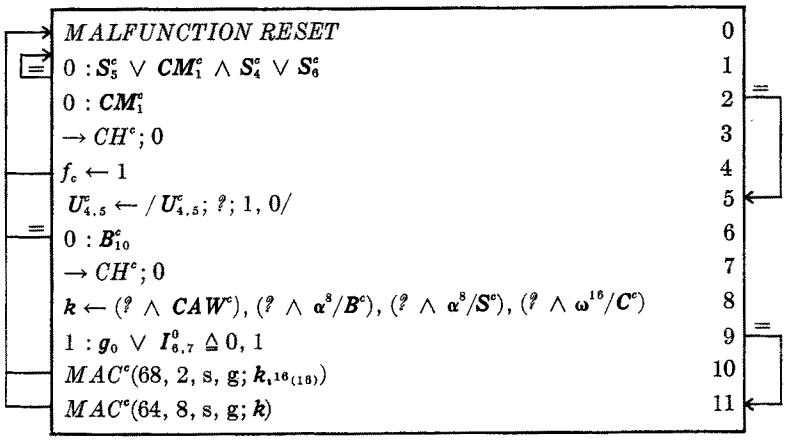
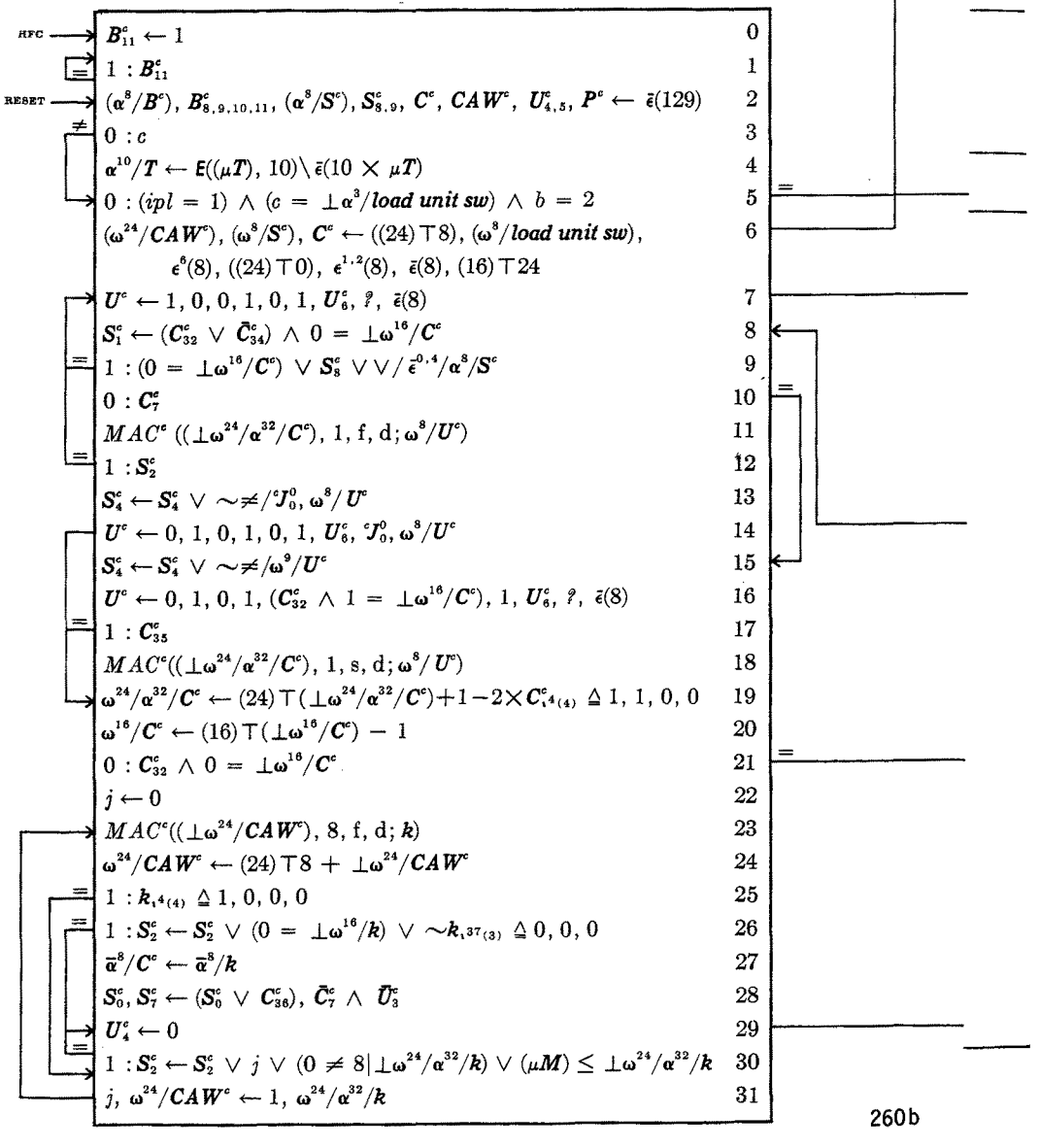
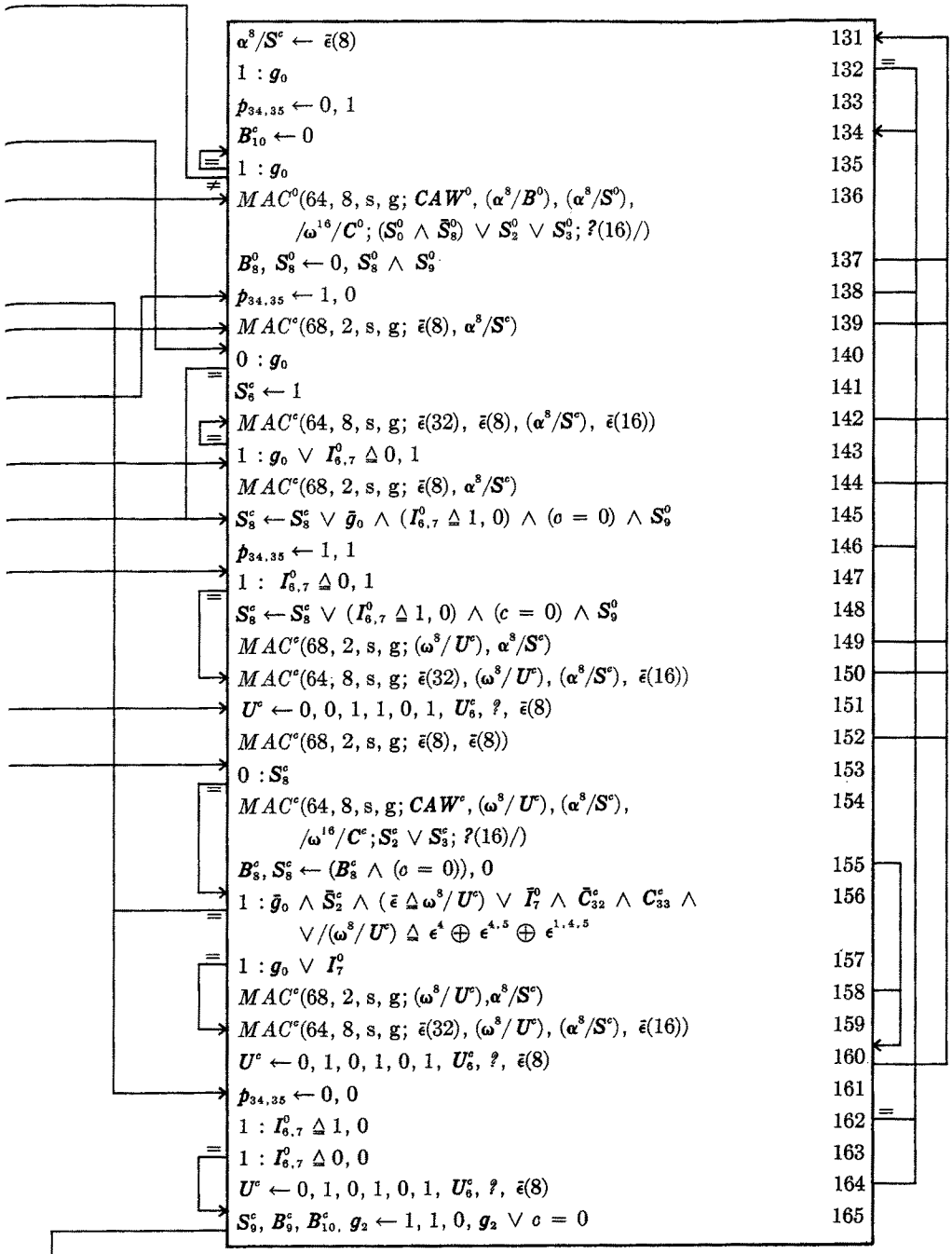


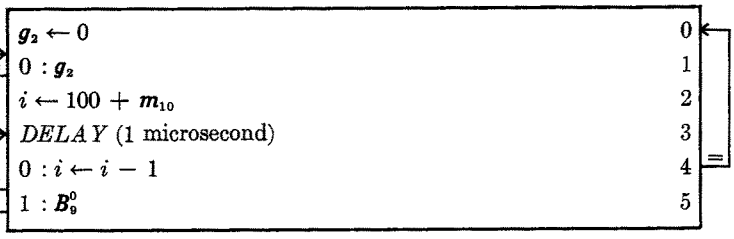
Table 10 Channel program segments

Segment	Channel type	State B_8^c	B_5^c	S_8^c	S_5^c	Extent	Enter	Exit
data transfer	mpx	?	1	?	1	7-31, 59-65	59	61, 63, 64, 65
	sel	?	1	0	1			
data chaining	both	?	1	0	1	22-31	22	29
termination	both	1	0	1	0	66-81, 56-58	56, 66, 67, 68, 78	58, 69, 70, 77, 80
command chaining	both	?	1	0	1	68-70	68	69, (70)
channel idle	mpx	?	0	?	?	35-55	35	38, 43, 50, 54, 55
	sel	0	0	0	0			
CPU service	both	?	?	?	?	82-165, 32-34	82, 104	128, 129, 130, 32, 33, 34
state analysis	mpx	?	0	?	?	82-101	82	85, 87, 88, 90, 91, 92, 93, 101
	sel	?	0	?	0			
sio	mpx	?	0	0	0	102-114	102, 104	112, 114
	sel	0	0	0	0			
command chaining	both	?	1	0	1	104-114	(104)	112, 114
channel-initiated selection	mpx	?	?	?	?	115-130	115	118, 119, 120, 122, 127, 128, 129, 130
	sel	?	?	0	?			
CSW and condition code setting	mpx	?	0	?	?	136-165	136, 138, 139, 140, 143, 145, 147, 151, 153, 161	137, 138, 139, 142, 144, 146, 149, 150, 152, 160, 162, 164, 165
	sel	?	0	?	0			
end of *CPU* service	mpx	?	0	?	?	131-135, 32-34	131, 134	32, 33, 34
	sel	?	0	?	0			
system reset and *IPL*	both	0	0	0	0	0-6	0, 2	5, 6

CH^c, channel c *system program*





$I^0_{6,7}$	OP
0 0	SIO
0 1	TIO
1 0	HIO
1 1	TCH

BMT, burst mode timer *system program*