MONO-OPERATIONS

POLY-OPERATIONS

DECLARATIVE OPERATIONS

CS-1 INPUT

MONO-OPERATIONS

MONO-OPERATIONS (Cont.)

MONO-OPERATIONS

Operations which mnemonically express a machine instruction are *mono-operations*. Each mono-operation in the source language (L_0) is translated by CS-1 to one machine instruction in the object language (L_4), i.e., the translation is one-to-one.

Mono-operations have a definite format:

W V_0 V_1 V_2

➡ [operator] • [allied operand] • [y -operand] • [j -operand] ➡

W - gives a mono-code which defines a class of machine instructions, such as *enter*, *store*, etc

V_0 - gives added information which further defines a machine instruction, thus is called the *allied operand*. The allied operand may specify a register, r, or a simple logical or arithmetic expression, e. It is absent in some operations

Some of the mono-codes are multipurpose. They form a class of operations. In such cases, the allied operand combines with and modifies the operator to generate a distinct instruction in the object language. An example is the selective operator, **SEL**. When combined with the V_0 operand, **SET**, it generates a computer function code f of 50. Similarly, **SEL • CP** generates an f of 51, **SEL • CL** generates an f of 52, and **SEL • SU** generates 53. Another example of a multipurpose operator is **ADD**:

ADD • A
ADD • Q
ADD • LP

In each case the compiler generates a separate machine code instruction.

V_1 - specifies either 1) a numeric value, 2) the address of a memory location, or 3) a register (A, Q, or B^n). The y-operand is a Read-class operand, a Store-class operand, or a Replace-class operand (see COMPUTER-ORIENTED OPERATIONS for a discussion of these). V_1 is absent in some operations

Note: Subsequent references to y include all of the above interpretations unless otherwise specified.

V_2 - specifies a j-operand which is primarily used for jump or skip determination or for repeat status interpretation. The action caused by these may be conditional or unconditional as directed by the operand used. Seven j-operands are applicable to the majority of mono-operations; these are called *normal* j-operands. Certain operations require the usage of unique j-operands, called *special* j-operands. These are explained in the discussions of those operations. The j-operand is absent on other operations

Normal j-operands are as follows:

Operand, j	Performance
(blank)	Will not skip the next operation
SKIP	Skip the next operation unconditionally
QPOS	Skip the next operation if Q is positive
QNEG	Skip the next operation if Q is negative
AZERO	Skip the next operation if A is zero
ANOT	Skip the next operation if A is non-zero
APOS	Skip the next operation if A is positive
ANEG	Skip the next operation if A is negative

Special j-operands are required for use with the following operations: Jump, Return Jump, Divide, Repeat, Add Q, Subtract Q, and all non-mask Compares.

Mono-code operators, combining with allied operands in most cases, are capable of generating all the irredundant instructions of the Unit Computer's repertoire. Additional operations such as "do nothing" operation, **NO-OP**, and "complement a register", **CP**, produce single instructions which achieve such actions which are not apparent in the names of computer function codes.

ENTer Operation:

$$\begin{array}{cccccc} & W & & V_0 & & V_1 & & V_2 \\ \Rightarrow & \textbf{ENT} & \bullet & [r \text{ or } e] & \bullet & [y] & \bullet & [j] & \Rightarrow \end{array}$$

The **ENT** operation either 1) first clears the register, r, and then transmits the numerical value expressed by y to register r, or 2) performs the function expressed by e and enters the result in A. The **Y** that appears in e refers to the numerical value which y defines.

V_0 - designates the register into which the numerical value is entered; r can be:

A, **Q**, or **B0** through **B7**

or

V_0 - states one of several simple arithmetic or logical expressions, e, to be performed, which are then entered into A. These are:

Expression, e		Performance
(1)	**LP**	LP (y) (Q)* ⇒ A
(2)	**Y+Q**	y + Q ⇒ A
(3)	**Y-Q**	y - Q ⇒ A

V_1 - gives a Read-class operand that defines y

V_2 - specifies a normal j-operand; it is optional when V_0 is **A**, **Q**, **Y+Q** or **Y-Q**

or

V_2 - specifies a j-operand when V_0 is **LP**. In this case the operation permits all normal j-operands except **QPOS** and **QNEG**. Substituted for **QPOS** and **QNEG** are two special j-operands as follow:

EVEN - Even parity (even number of "ones" in A)

ODD - Odd parity (odd number of "ones" in A)

Note: If V_0 is B^0 through B^7, V_2 must be absent.

*LP (y) (Q) means the bit-by-bit product of (y) and (Q)

Examples:

➡ ENT • Y+Q • UX(SACK+B4) ⇨

➡ ENT • Q • X77776 • AZERO ⇨

➡ ENT • LP • W(BAG9+3) • EVEN ⇨

STORe Operation:

W V_0 V_1 . V_2

➡ **STR** • [r or e] • [y] • [j] ➡

The **STR** operation stores either 1) the content of register r, or 2) the result of an expression, e, in a storage location delegated by y.

V_0 - designates the register, r, whose content is stored in a memory location. V_0 can be:

A , Q , B0 through **B7**

or

V_0 - states one of several simple arithmetic or logical expressions, e, to be performed, which are then stored in a memory location. These are:

Expression, e	Performance
(1) **LP**	LP(A)(Q)* ➡ y
(2) **A+Q**	A+Q ➡ y and A
(3) **A-Q**	A-Q ➡ y and A

V_1 - gives a Store-class operand that defines a memory location y

V_2 - specifies a normal j-operand; it is optional

Note: If r is **B0** through **B7**, V_2 must be absent.

Examples:

➡ **STR** • **B7** • **L(PEN-5)** ➡

➡ **STR** • **A-Q** • **W(INK)** • **QNEG** ➡

*LP(A)(Q) means the bit-by-bit product of A and Q

STo Re (channel) Operation

W V_0 V_1 V_2

➡ **STR** • [channel] • [y] • [sub-function code] ➡

This operation provides the interrupt word at the specified location.

V_0 - Specifies the channel of the desired interrupt word. Channels C0 - C7, C10 - C17 are permitted. V_0 may specify a name which is identified by a **MEANS** operation or a **CHAN-SET** tape

V_1 - Specifies the location at which the interrupt word is to be stored. This operand may specify only the whole contents of a memory location

V_2 - Specifies the sub-function code:

(absent)* - means the contents of the appropriate address reserved for interrupt word storage will be transferred to Y as specified by V_1. This instruction is necessary with new line equipment to reset the Interrupt Request

FORCE - provides forcing the word on the line to be stored at Y as specified by V_1. Program will hold until the word is read causing an Input Acknowledge signal (this is an abnormal mode used for testing some equipment)

Examples:

➡ **STR • C3 • W(CAT)** ➡

➡ **STR • SMPCHAN • W(DOG) • FORCE** ➡

*The Input Acknowledge is set automatically when the interrupt word is read into the special address, which occurs in both old and new line equipment

NO-OP Operation:

The **NO-OP** operation is a "do nothing" operation. It generates a 12000 00000 in the object program, causing the computer to move on to the next operation.

CLear Operation:

$$\overset{W}{} \qquad \overset{V_0}{}$$

$$\Rightarrow \quad \texttt{CL} \quad \bullet \quad \left[r \text{ or } y \right] \quad \Rightarrow$$

The **CL** operation clears the memory location specified by y or the register specified by r.

V_0 - designates the register to be cleared; r can be:

A , Q , B1 through **B7**

or

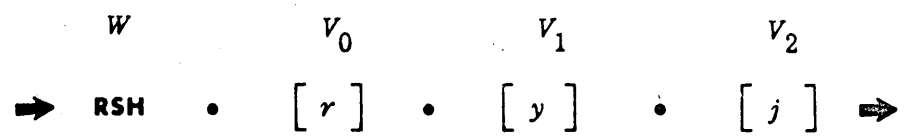
V_0 - gives a Store-class operand that defines y

Examples:

➡ **CL • Q** ➡

➡ **CL • L(GIMME)** ➡

Right **SH**ift Operation:



The **RSH** operation shifts the content of the register, r, to the right y bit positions. As the information is shifted, the original sign bit replaces the higher order bits of register r; the lower order bits are shifted off the end.

Only the lower-order 6-bits of y are recognized. The higher-order 24 bits are ignored.

V_0 - designates the register that the operation shifts; r can be:

A , **Q** , or **AQ**

AQ represents the 60-bit register consisting of A and Q

V_1 - gives a Read-class operand that defines y

V_2 - specifies a normal j-operand; it is optional

Examples:

➡ **RSH** • **AQ** • **15D** • **AZERO** ➡

➡ **RSH** • **A** • **L(FLIP+6)** ➡

Left SHift Operations:

W V_0 V_1 V_2

➡ **LSH** • [r] • [y] • [j] ➡

The **LSH** operation shifts the content of the register, r, to the left y bit positions. The shift is circular; the low-order bits of r are replaced by the upper-order bits. Only the lower-order 6 bits of y are recognized. The higher-order 24 bits are ignored.

V_0 - designates the register that the operation shifts; r can be:

A , Q , *or* AQ

AQ represents the 60-bit register consisting of A and Q

V_1 - gives a Read-class operand that defines y

V_2 - specifies a normal j-operand; it is optional

Examples:

➡ **LSH • A • L(CAT) • QNEG** ➡

➡ **LSH • Q • B4** ➡

ADD Operation:

W V_0 V_1 V_2

➡ **ADD** • [r or e] • [y] • [j] ➡

The **ADD** operation either 1) adds the numeric value expressed by y to the contents of r and replaces the result in r, or 2) performs the expression, e, and then adds its result to A.

V_0 - designates the register to which the numerical value is added

Register, r	Performance
A	A + y ➡ A
Q	Q + y ➡ Q

or

V_0 - states a logical function, e

Expression, e	Performance
LP	A + LP(y)(Q)* ➡ A

V_1 - gives a Read-class operand that defines y

V_2 - specifies a normal j-operand if V_0 is **A** or **LP** . If V_0 is **Q**, **AZERO** and **ANOT** are not permitted; **QZERO** and **QNOT** are substituted instead. V_2 is optional

Examples:

➡ **ADD** • **LP** • **W(BOOK)** ➡

➡ **ADD** • **Q** • **12D** • **QZERO** ➡

*LP(y)(Q) means the bit-by-bit product of y and Q

SUBtract Operation:

W	V_0	V_1	V_2

➡ **SUB** • [r or e] • [y] • [j] ➡

The **SUB** operation either 1) subtracts the numeric value expressed by y from the contents of r and replaces the result in r, or 2) performs the expression, e, and then subtracts its result from A.

V_0 - designates the register from which the numerical value is subtracted

Register, r	Performance
A	A - y ➡ A
Q	Q - y ➡ Q

or

V_0 - states a logical function, e

Expression, e	Performance
LP	A - LP(y)(Q)* ➡ A

V_1 - gives a Read-class operand that defines y

V_2 - specifies a normal j-operand if V_0 is **A** or **LP** . If V_0 is **Q** , **AZERO** and **ANOT** are not permitted. **QZERO** and **QNOT** are substituted instead. V_2 is optional

Examples:

➡ **SUB** • **A** • **12D** ➡

➡ **SUB** • **Q** • **B6** ➡

*LP(y)(Q) means the bit-by-bit product of y and Q

MUL tiply Operation:

W V_0 V_1 V_2

➡ **MUL** • [absent] • [y] • [j] ➡

The **MUL** operation multiplies Q by the numerical value expressed by y, leaving the double length product in AQ. All numbers involved are treated as integers.

V_0 - always absent

V_1 - gives a Read-class operand that defines y. A is not permitted

V_2 - specifies a normal j-operand

The actual multiplication is performed with positive numbers only; therefore, if the original sign bits of y and Q are not similar, an *end correction* is made by complementing the product. The branch condition j-operand is interpreted prior to the *end correction*, thus **ANEG** has no effect and **APOS** always gives an unconditional skip.

Examples:

➡ **MUL** • **L(PAPER-2)** ➡

➡ **MUL** • **4** ➡

DIVide Operation:

W V_0 V_1 V_2

➡ **DIV** • [absent] • [y] • [j] ➡

The **DIV** operation divides AQ by the numerical value expressed by y, leaving the quotient in the Q register and the remainder in the A register. The remainder bears the same sign as the quotient.

V_0 - always absent

V_1 - gives a Read-class operand that defines y. A is not permitted

V_2 - specifies a skip-the-next-operation condition

Operand, j	Condition
(blank)	Does not skip on divide
SKIP	Unconditional skip
OF	Skip if there is an *overflow*
NOOF	Skip if there is no *overflow*
AZERO	Skip if A = 0
ANOT	Skip if A $\neq$ 0
APOS	Skip if A is positive
ANEG	Skip if A is negative

Note: There is no indicator on the console to represent a *divide fault*. However, by coding each operation with a j of **OF**, a program test for a *divide fault* is automatic. With this selection for j, a skip of the next operation occurs if the *divide fault* exists. The skip would be made to a **JP** operation which provides remedial means of noting the error or of correcting it. Therefore, the operation which follows the **DIV** operation should have a j-operand of **SKIP** in order to preclude the **JP** operation whenever the divide sequence culminates in a correct answer. A divide fault can be detected also if the **DIV** operation is executed with a j of **NOOF**. In this case, a correct answer is indicated when a skip occurs. Since A is always positive at the time j is sensed, **ANEG** becomes meaningless.

Examples:

➡ **DIV • W(PAD+B2) • OF** ➡

➡ **DIV • B6** ➡

SQuare **R**oo**T** Operation:

W	V_0	V_1	V_2
➡ **SQRT**	• [absent]	• [absent]	• [j] ➡

The **SQRT** operation finds $\sqrt{|Q|}$ and places it in Q. The remainder goes to A, always destroying the previous contents. The radix point of (Q) is assumed to be at the low order end of the register.

V_0 - always absent

V_1 - always absent

V_2 - specifies a skip-the-next-instruction condition

Operand, j	Condition
(blank)	Does not skip
SKIP	Always skip
REM	Skip if A ≠ 0
NO REM	Skip if A = 0

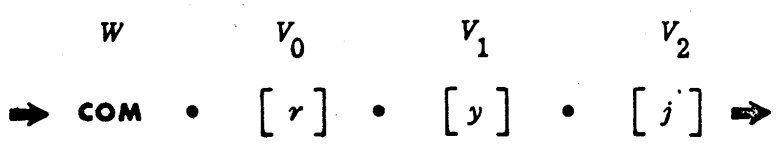
Examples:

➡ **SQRT** ➡

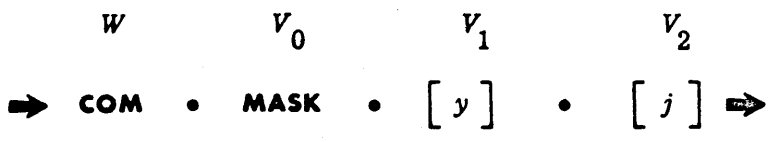
➡ **SQRT** • **NO REM** ➡

COM pare Operation:

Type A



Type B



Type A:

The **COM** operation compares the value expressed by y with r. A skip of the next operation takes place if the condition specified by j is satisfied. The content of r is not changed.

V_0 - designates the register with which the numeric value is compared

Register, r	Performance
A	A: y
Q	Q: y
AQ *	A: y and Q: y

V_1 - gives a Read-class operand that defines y

V_2 - specifies a skip condition; it *must* be present. The special meanings of j are:

Operand, j	Condition
YLESS	Skip if the value expressed by $y \leq$ Q Skip if the value expressed by $y \leq$ A
YMORE	Skip if the value expressed by $y >$ Q Skip if the value expressed by $y >$ A
YIN	Skip if Q $\geq$ value expressed by y and the value expressed by $y >$ A. $Q \geq y > A$
YOUT	Skip if Q $<$ value expressed by y or the value expressed by $y \leq$ A. $Q < y \leq A$

* Use only with j-operands **YIN** or **YOUT**. y is compared with A and Q as individual 30 bit registers

Type B:

The **COM • MASK** operation compares A with the bit-by-bit product of the values expressed by y and Q. A skip of the next operation takes place if the condition specified by j is satisfied. The contents of A and Q are not changed.

V_0 - says **MASK**

V_1 - gives a Read-class operand that defines y

V_2 - specifies a normal j-operand; it must be present. The condition of A is tested after LP(y)(Q)* is subtracted from A. The LP(y)(Q)* is then added to A

Examples:

➡ **COM • AQ • W(TAB-2) • YIN** ➡

➡ **COM • MASK • L(TAB) • AZERO** ➡

*$\overline{\text{LP}(y)}$ (Q) means the bit-by-bit product of y and Q

ComPlement Operation:

W V_0

➡ **CP** • $[r]$ ➡

The **CP** operation complements all bits of the register specified by r.

V_0 - designates the register which is complemented; r can be:

A or **Q**

Example:

➡ **CP** • **Q** ➡ (Gen: 14000 00000)

SELective Operation:

W V_0 V_1 V_2

➡ **SEL** • [e] • [y] • [j] ➡

The **SEL** operation performs logical manipulations specified by e on the content of A. A string of bits expressed by y controls these manipulations.

V_0 - states one of several logical functions. These are:

Expression, e	Performance
SET	Sets the individual bits of register A corresponding to *ones* in the numeric value expressed by y, leaving the remaining bits of A unaltered
CP	Complements the individual bits of register A corresponding to *ones* in the numeric value expressed by y, leaving the remaining bits of A unaltered
CL	Clears the individual bits of register A corresponding to *ones* in the numeric value expressed by y, leaving the remaining bits of A unaltered
SU	Replaces the bits of A with bits of the numeric value expressed by y corresponding to *ones* in Q

V_1 - gives a Read-class operand that defines y. A is not permitted

V_2 - specifies a normal j-operand; it is optional

Examples:

➡ **SEL • CP • X77774** ➡

➡ **SEL • SET • W(CLIP) • AZERO** ➡

RePLace Operation:

$$\underset{W}{} \quad \underset{V_0}{} \quad \underset{V_1}{} \quad \underset{V_2}{}$$

$$\Rightarrow \quad \text{RPL} \bullet [e] \bullet [y] \bullet [j] \quad \Rightarrow$$

The **RPL** operation performs the function expressed by e, and stores the result in A and in a memory location established by y. The **Y** that appears in e refers to the numerical value which y defines.

V_0 - states a simple arithmetic or logical expression to be performed. These are:

	Expression, e	Performance
(1)	**A+Y**	A + y ⇒ y and A
(2)	**A-Y**	A - y ⇒ y and A
(3)	**Y+Q**	y + Q ⇒ y and A
(4)	**Y-Q**	y - Q ⇒ y and A
(5)	**Y+1**	y + 1 ⇒ y and A
(6)	**Y-1**	y - 1 ⇒ y and A
(7)	**LP**	LP (y) (Q)* ⇒ y and A
(8)	**A+LP**	A + LP (y) (Q) ⇒ y and A
(9)	**A-LP**	A - LP (y) (Q) ⇒ y and A

V_1 - gives a Replace-class operand which defines address y

V_2 - specifies a normal j-operand; this is valid with all V_0 operands *except* **LP** *or*

V_2 - specifies the j-operand when V_0 is **LP**. In this case the operation permits all normal j-operands except **QPOS** and **QNEG**. Substituted for **QPOS** and **QNEG** are two special j-operands as follows:

EVEN - Even parity (even number of "ones" in A)

ODD - Odd parity (odd number of "ones" in A)

Examples:

⇒ **RPL • A+LP • W(CRUNCH) • QNEG** ⇒

⇒ **RPL • Y-Q • UX (HOPTO+B6)** ⇒

⇒ **RPL • LP • W (DOP+B4) • ODD** ⇒

*LP (y) (Q) means the bit-by-bit product of (y) and (Q)

Replace SElective Operation:

$$\begin{array}{cccccccc} & W & & V_0 & & V_1 & & V_2 \\ \Rightarrow & \textbf{RSE} & \bullet & [e] & \bullet & [y] & \bullet & [j] \Rightarrow \end{array}$$

The **RSE** operation performs logical manipulations specified by e on the content of A and then stores A in the memory location whose address is expressed by y. A string of bits in the same memory location controls these manipulations before the store takes place.

V_0 - states one of several logical functions. These are:

Expression, e	Performance
SET	Sets the individual bits of register A to *one* corresponding to *ones* in the numeric value expressed by y, leaving the remaining bits of A unaltered, then stores A at the storage address expressed by y
CP	Complements the individual bits of register A corresponding to *ones* in the numeric value expressed by y, leaving the remaining bits of A unaltered, then stores A at the storage address expressed by y
CL	Clears the individual bits of register A corresponding to *ones* in the numeric value expressed by y, leaving the remaining bits of A unaltered, then stores A at the storage address expressed by y
SU	Replaces the bits of A with bits of the numeric value expressed by y corresponding to *ones* in Q, then stores A at the storage address expressed by y

V_1 - gives a Replace-class operand that defines y

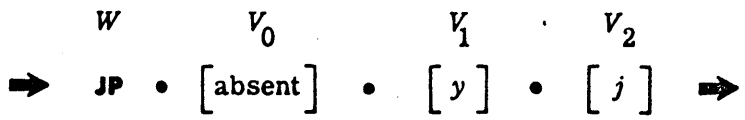
V_2 - specifies a normal *j*-operand; it is optional

Examples:

➡ **RSE • SU • W(COVER+B4)** ➡

➡ **RSE • CL • LX(POW5)** ➡

JumP Operation:



The **JP** operation clears the program address register P, and enters the address designated by y in P for certain conditions specified by j. Thus y becomes the address of the next operation and the beginning of a new program sequence. If a jump condition is not satisfied, the next sequential operation in the current sequence is executed in the normal manner.

V_0 - always absent

V_1* - gives a Read-class operand which defines address y

V_2 - specifies a jump condition

Operand j	Condition
QPOS	Jump if Q is positive
QNEG	Jump if Q is negative
AZERO	Jump if A is equal to zero
ANOT	Jump if A is not equal to zero
APOS	Jump if A is positive
ANEG	Jump if A is negative
(blank)	Unconditional jump
KEY1	Jump if Key 1 is set
KEY2	Jump if Key 2 is set
KEY3	Jump if Key 3 is set
STOP	Jump and then stop
STOP5	Jump and then stop if Key 5 is set
STOP6	Jump and then stop if Key 6 is set
STOP7	Jump and then stop if Key 7 is set
C^{n}ACTIVEIN	See next page for condition description
C^{n}ACTIVEOUT	

* If j is **C^{n}ACTIVEIN** or **C^{n}ACTIVEOUT**, an operand code of **X**, **LX**, **UX**, and **A** is not permitted

C^{n}ACTIVEIN*	Jump if the input buffer mode on channel n is active (n = 0, ——, 17)
C^{n}ACTIVEOUT*	Jump if the output buffer on channel n is active (n = 0, ——, 17)

Examples:

➡ **JP • TRACE** ➡

➡ **JP • L(TRIG + B2) • KEY1** ➡

➡ **JP • ROAR • C14ACTIVEIN** ➡

* May be a *name* which is defined by a **MEANS** operation or a **CHAN-SET** tape

JumP Operation

W V_0 V_1 V_2 V_3

➡ **JP** • [absent] • [location] • [channel] • **COMACTIVE** ➡

This operation provides a means for determining whether an external function command buffer is active.

V_0 - Always absent

V_1 - Specifies the location to which control is to be transferred if the specified external function command buffer is active. This operand may contain only a tag or a tag with a K designator of L

V_2 - Specifies the channel on which the external command buffer is to be tested. Channels C0-C7, C10-C17 are permitted. V_2 may specify a name which is identified by a **MEANS** operation or a **CHAN-SET** tape

V_3 - Specifies that this test is for an active external function command buffer

Examples:

➡ **JP • PTH • C10 • COMACTIVE** ➡

➡ **JP • PTH • TAPECHAN • COMACTIVE** ➡

Return JumP Operation:

W V_0 V_1 V_2

➡ **RJP** • [absent] • [y] • [j] ➡

The **RJP** operation performs the following steps if conditions specified by j are satisfied: 1) it stores the content of the program address counter P, which is the address of the **RJP** operation plus one, into the lower 15 bits of the memory location which has the address specified by y, and 2) then it enters P with $y + 1$. Thus, $y + 1$ becomes the address of the next operation and the beginning of a new program sequence.

If the j condition is not satisfied, the next sequential operation in the current sequence is executed in the normal manner.

V_0 - always absent

V_1 - gives a Read-class operand which defines address y

V_2 - specifies a jump condition

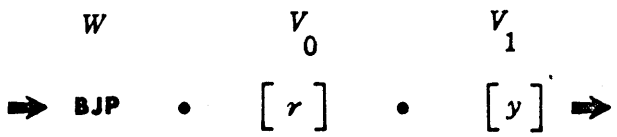
Operand, j	Condition
QPOS	Return jump if Q is positive
QNEG	Return jump if Q is negative
QZERO	Return jump if A is equal to zero
ANOT	Return jump if A is not equal to zero
APOS	Return jump if A is positive
ANEG	Return jump if A is negative
(blank)	Unconditional return jump
KEY1	Return jump if Key 1 is set
KEY2	Return jump if Key 2 is set
KEY3	Return jump if Key 3 is set
STOP	Return jump and then stop
STOP5	Return jump and then stop if Key 5 is set
STOP6	Return jump and then stop if Key 6 is set
STOP7	Return jump and then stop if Key 7 is set

Examples:

➡ **RJP • TRACE • STOP** ➡

➡ **RJP • U(FLAT+B7)** ➡

B JumP Operation:



The **BJP** operation tests the content of the B register specified by r. If (r) is zero, the normal sequence of operations continues. If (r) is non zero, (r) decreases by one, and a new sequence of operations begins at the address expressed by y.

V_0 - designates a B register: **B1** through **B7**

V_1 - gives a Read-class operand that defines y

Note: A j-operand is not permitted.

Examples:

➡ **BJP • B5 • DESK** ➡

➡ **BJP • B1 • U(EXIT+B2)** ➡

B SKip Operation:

W V_0 V_1

➡ **BSK** • $[r]$ • $[y]$ ➡ .

The **BSK** operation tests the content of the B register specified by r. If (r) is equal to the numeric value expressed by y, the control sequence skips the next operation and (r) is cleared. If (r) is not equal to the numeric value expressed by y, the normal sequence of operations continues, and (r) increases by one.

V_0 - designates a B register: **B1** through **B7**

V_1 - gives a Read-class operand that defines y

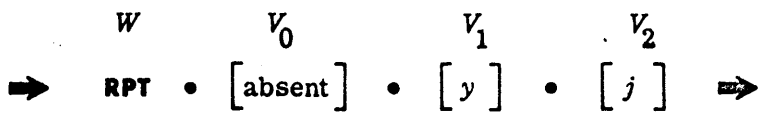
Note: A j-operand is not permitted

Examples:

➡ **BSK • B3 • 56** ➡

➡ **BSK • B4 • B2** ➡

RePeaT Operation:



The **RPT** operation initiates a repeat mode of control which causes execution of the next sequential operation the number of times expressed by y, or until the j-operand condition of the next operation is satisfied, whichever occurs first. B^7 keeps count of the number of times execution is to take place. (B^7 decreases by one after each execution.)

V_0 - always absent

V_1 - gives a Read-class operand that defines y. If y is zero, the next instruction is skipped

V_2 - specifies the mode of address modification of the repeated operation

Operand, j	Control
(blank)	Unmodified repeat of next operation
ADV	Advance the operand address of the repeated operation by one after each individual execution
BACK	Decrease the operand address of the repeated operation by one after each execution of the repeated operation
ADDB	Adds cumulatively the B register indicated in the repeated operation to its operand during each execution
R	Increase the operand address of the repeated Replace-class operation by the content of B6 for the *store* portion of the replace only
ADVR	Increase the operand address of the repeated Replace-class operation by the content of B^6 for the *store* portion of the replace only; then increment the operand address of the repeated operation by one after each execution

Operand	Control
BACKR	Increase the operand address of the repeated Replace-class operation by the content of B^6 for the *store* portion of the replace only; then decrement the operand address of the repeated operation by one after each execution
ADDBR	Adds cumulatively the B register indicated in the repeated Replace-class operation to its operand address during each execution; in addition to the above, increase the operand address of the repeated operation by the content of B^6 only for *store* portion of the replace

Note: Use *j*-operands **R**, **ADVR**, **BACKR**, and **ADDBR** only when a **RPL** operation follows the **RPT** operation.

Examples:

➡ **RPT • 39D** ➡

➡ **RPT • B7 • BACK** ➡

➡ **RPT • L(TRADE3) • ADDBR** ➡

Note: All interrupts are locked out once the repeat mode has been initiated.

INput Operation (With or Without Monitoring):

W V_0 V_1 V_2

➡ **IN** • [channel] • [y] • [absent or **MONITOR**] ➡

The **IN** operation establishes the control to transfer data from external equipment to the core memory via a specified channel. The address limits are defined by a numeric value expressed by y, which are transferred to memory address 00100+n, where n is the number of the channel. Subsequent to this operation, but not as part of it, the individual buffer operations are executed at a rate determined by the external device. The starting address, initially established by this operation, is advanced by *one* following each individual buffer operation. The next current address is maintained throughout the buffer process in the lower order 15-bit positions of memory location with storage address 00100+n. This mode continues until it is superseded by a subsequent initiation of an input buffer via the same channel, or until the higher order half and the lower order half of storage address 00100+n contain equal quantities, whichever occurs first. The first and last address of the memory area is specified in location 00100+n

V_0 - designates the Channel, C^n, through which buffering takes place:

C0, —, C17

V_1 - gives an operand that defines y. If V_1 is a number of five digits or less, or has an operand code of **L**, y replaces the lower half of address 00100+n. If V_1 is a number of more than five digits, or has an operand code of **W**, y replaces the whole word of address 00100+n. Operand codes of **X**, **U**, **LX**, **UX**, or **A**, are not permitted

V_2 - specifies whether the buffer operation is to be monitored or not. Monitoring is specified by V_2 being **MONITOR**. Otherwise V_2 is absent

A buffer operation is monitored if the main program is interrupted and control is transferred to 00040+n when the buffer operation is terminated *by* the control addresses in address 00100+n becoming equal.

Examples:

➡ **IN • C5 • 52367** ➡

➡ **IN • C14 • W(LIMIT) • MONITOR** ➡

OUTput Operation (With or Without Monitoring):

W V_0 V_1 V_2

➡ **OUT** • [channel] • [y] • [absent or **MONITOR**] ➡

The **OUT** operation establishes the control to transfer data to external equipment from the core memory via a specified channel. The address limits are defined by a numeric value expressed by y; these are transferred to memory address 00120+n, where n is the number of the channel. Subsequent to this operation, but not as part of it, the individual buffer operations are executed at a rate determined by the external device. The starting address, initially established by this operation, is advanced by *one* following each individual buffer operation. The next current address is maintained throughout the buffer process in the lower order 15-bit positions of memory location at storage address 00120+n. This mode continues until it is superseded by a subsequent initiation of an input buffer via the same channel, or until the higher order half and the lower order half of storage address 00120+n contain equal quantities, whichever occurs first. The first and last address of the memory area are specified in location 00120+n

V_0 - designates the Channel, C^n, through which buffering takes place:

C0, —, C17

V_1 - gives an operand that defines y. If V_1 is a number of five digits or less, or has an operand code of **L**, y replaces the lower half of address 00120+n. If V_1 is a number of more than five digits, or has an operand code of **W**, y replaces the whole word of address 00120+n. Operand codes of **X**, **U**, **LX**, **UX**, or **A** are not permitted

V_2 - specifies whether the buffer operation is to be monitored or not. Monitoring is specified by V_2 being **MONITOR**. Otherwise V_2 is absent

A buffer operation is monitored if the main program is interrupted and control is transferred to 00060+n when the buffer operation is terminated by the control addresses in address 00120+n becoming equal

Examples:

➡ **OUT • C7 • 41456** ➡

➡ **OUT • C12 • L(LOC) • MONITOR** ➡

EXternal-**COM**mand Operation:

$$\overset{W}{\Rightarrow}\ \textbf{EX-COM}\ \bullet\ \overset{V_0}{[\text{channel}]}\ \bullet\ \overset{V_1}{[\text{external function code}]}\ \bullet\ \overset{V_2}{[\text{sub-function code}]}\ \Rightarrow$$

The **EX-COM** operation initiates a one word external function buffer

V_0 - Specifies the channel on which the external function code is transferred. Channels C0 - C7, C10 - C17 are permitted. V_0 may specify a name which is identified by a **MEANS** operation or a **CHAN-SET** tape

V_1 - Function code, this may be a ten digit number or less, or the whole contents of a memory location (i.e., operand code of w). Other operand codes are not permitted. B-Box modification is not allowed if V_1 is a constant

V_2 - Specifies the sub-function code

(absent) - The external function command is sent without force or monitor

FORCE - Used when the communication is with external equipment which has not been designed to send an "external function request" to the computer. **MONITOR** may be used in conjunction with an **EX-COM** with a V_2 operand of **FORCE**

MONITOR - Provides a transfer of control to location 00500+j when the buffer of the external function word is completed

MONFORCE - Provides the combined capabilities of **MONITOR** and **FORCE**

Examples:

⇒ **EX-COM • C0 • 4300000016 • FORCE** ⇒

⇒ **EX-COM • C17 • W(EFUN)** ⇒

⇒ **EX-COM • TTY • CHAN • W(EFT) • MONITOR** ⇒

⇒ **EX-COM • SPILL • W(EFF) • MONFORCE** ⇒

$$\overset{W}{\Rightarrow}\ \textbf{EX-COM-MW}\ \bullet\ \overset{V_0}{[\text{channel}]}\ \bullet\ \overset{V_1}{[y]}\ \bullet\ \overset{V_2}{[\text{sub-function code}]}\ \Rightarrow$$

The **EX-COM-MW** operation sets up the appropriate external function buffer control word (at 00140+j) and initiates output buffering of the specified external function commands

V_0 - Specifies the channel on which the external function codes are sent. Channels C0 - C7, C10 - C17 are permitted. V_0 may specify a name which is identified by a **MEANS** operation or a **CHAN-SET** tape

V_1 - Gives the buffer limits of the function codes to be transmitted. This may be the contents of a whole memory location only (i.e., operand code of w). Other operand codes are not permitted. B register modification is not allowed if V_1 is a constant

V_2 - Specifies whether the buffering of the external function command words is to be monitored. When monitored, the completion of the buffer will cause transfer of control to external function buffer monitor interrupt entrance address 00500+J

Examples:

⇒ **EX-COM-MW • C3 • W(FCCW) • MONITOR** ⇒

⇒ **EX-COM-MW • TAPECHAN • W(SFCC)** ⇒

TERMinate Buffer Operation

W V_0 V_1

➡ **TERM** • [channel number or **ALL**] • [buffer mode] ➡

The **TERM** function terminates input, output, external function command, or all buffers as specified by the V_0 and V_1 operands.

V_0 - Specifies the channel on which buffering is to be terminated. Channels CO - C7, C10 - C17 are permitted. V_0 may specify a name which is identified by a **MEANS** operation or a **CHAN-SET** tape

ALL - Causes all buffering including that of external function commands, input data, and output data to be halted. No V_1 operand is allowed when the V_0 operand is **ALL**

V_1 - Specifies the mode of buffering to be terminated

(absent) - The V_1 operand must be omitted if the V_0 operand was **ALL**

COM - Terminates the buffering of external function commands on specified channel

INPUT - Terminates the buffering of input data on specified channel

OUTPUT -Terminates the buffering of output data on specified channel

Examples:

➡ **TERM • C6 • COM** ➡

➡ **TERM • ALL** ➡

➡ **TERM • C17 • OUTPUT** ➡

Example: (illegal)

➡ **TERM • ALL • INPUT** ➡

Set Interrupt Lockout Operation

W V_0

➡ **SIL • ALL** ➡

The operator **SIL** locks out both internal and external interrupts on all channels.

V_0 - The only V_0 operand allowed is **ALL**

Examples:

➡ **SIL • ALL** ➡

Example: (illegal)

➡ **SIL • C6** ➡

Set Interrupt Lockout - EXternal Operation

$$\overset{W}{\quad} \qquad \overset{V_0}{\quad}$$
$$\Rightarrow \textbf{SIL-EX} \bullet [\text{channel}] \Rightarrow$$

The operator **SIL-EX*** sets external interrupt lockout for the specified channel.

V_0 - Specifies the channel on which external interrupts are to be locked out. Channels C0 - C7, C10 - C17 are permitted. V_0 may specify a name which is identified by a **MEANS** operator or a **CHAN-SET** tape

ALL - Locks out external interrupts on all channels

Examples:

⇒ **SIL-EX • C10** ⇒

⇒ **SIL-EX • ALL** ⇒

⇒ **SIL-EX • FLEXCHAN** ⇒

* The interrupts locked out by **SIL-EX**, can be released only by the **RIL-EX** operation

Remove Interrupt Lockout Operation

$$\overset{W}{\rightarrow}\ \textbf{RIL}\ \bullet\ \overset{V_0}{[\text{absent or } \textbf{ALL}]}\ \rightarrow$$

The operator **RIL** removes interrupt lockouts on all internal channels, and all external channels not previously locked out by **SIL-EX** operations.

V_0 - The effect on the computer is the same whether V_0 is **ALL** or absent

(absent) - If V_0 is absent an instruction of the type 600XX XXXXX will be generated

ALL - If V_0 is **ALL** an instruction of the type 66X10 XXXXX will be generated

Examples:

➡ **RIL** ➡

➡ **RIL • ALL** ➡

Remove Interrupt Lockout - EXternal Operation

$$\overset{W}{\blacktriangleright}\ \textbf{RIL-EX}\ \bullet\ \overset{V_0}{[\text{channel}]}\ \blacktriangleright$$

The operator **RIL-EX*** releases the interrupt lockout for external interrupts.

V_0 - Specifies the channel on which the external interrupt lockout is to be released. Channels C0 - C7, C10 - C17 are permitted. V_0 may specify a name which is identified by a **MEANS** operator or a **CHAN-SET** tape

ALL - Removes the external interrupt lockout on all channels

Examples:

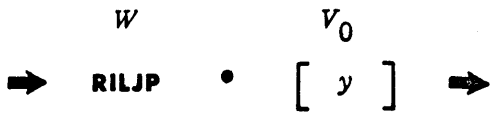
➡ **RIL-EX • C10** ➡

➡ **RIL-EX • ALL** ➡

➡ **RIL-EX • TTYCHAN** ➡

* This instruction must be used to remove interrupt lockouts on channels previously locked out by **SIL-EX** operations. **RIL** operations only release lockouts for external interrupts not locked out by **SIL-EX** , and of course internal interrupts

Remove Interrupt Lockout and JumP Operation



The **RILJP** operation removes the interrupt lockout, thus allowing a subsequent interrupt, and jumps to address y unconditionally. The operation generates a 601nn nnnnn instruction.

V_0 - gives a Read-class operand which defines address y

NORMalize Operation:

$$\overset{W}{\blacktriangleright\ \textbf{NORM}\ \bullet\ [\ r\ ]}\ \overset{V_0}{}$$

The **NORM** operation shifts AQ left circularly until the upper two bits of A are unequal or until AQ has been shifted 72_8 times.

V_0 - designates AQ

Enable **C**ontinuous **D**ata **M**ode Operation:

W	V_0	V_1
➡ **ECDM** •	[channel] •	[sub-function code]

The **ECDM** operation will automatically reinitiate a buffer when termination occurs. Upon termination, a new pair of control words are transferred to the buffer control address for this channel and the buffer re-activated. If a monitor interrupt has been selected, it will occur at this time.

V_0 - Specifies the CDM channel. Channels CØ-C7, C1Ø-C17 are permitted if they have the necessary hardware modification. V_0 may specify a name which is identified by a **MEANS** operation or a **CHAN-SET** tape

V_1 - Establishes direction of data transfer

INPUT — Data transferred into the computer. Buffer Control Word is located at address $(00200 + C^n)$

OUTPUT — Data transferred to external equipment. Buffer Control Word is located at address $(00220 + C^n)$

Disable **C** ontinuous **D**ata **M**ode Operation:

$$\overset{W}{\blacktriangleright\ \textbf{DCDM}} \bullet \overset{V_0}{[\text{channel}]} \bullet [\text{sub-function code}]$$

The **DCDM** operation disables the CDM for a given channel

V_0 - Specifies the CDM channel. Channels CØ-C7, C1Ø-C17 are permitted. V_0 may specify a name which is identified by a **MEANS** operation or a **CHAN-SET** tape

V_1 - INPUT Disables input CDM control

OUTPUT Disables output CDM control

POLY-OPERATIONS

POLY-OPERATIONS

Quite frequently, a sequence of instructions appears iteratively in a program. This sequence performs a specific job or function. It is possible in cases such as this to generate the sequence of instructions with a single CS-1 operation. This is the familiar one-to-many relationship between instructions which shall be herein termed poly-coding, with the parent instruction being called a poly-operation. A poly-operation is capable of generating within the compiler system a unique sequence of computer instructions (in some cases a single instruction) designed to perform the specific task required.

It is permissible during the coding of a routine to intermix mono- and poly-operations in any order desired. However, the programer must not attempt to skip a poly-operation with the j-operand of a mono-operation. The poly-operation usually results in the generation of more than one instruction in the assembled object program; the computer skips the first of these instead of the intended next mnemonic operation in the source program. The compiler-generated computer instructions appear in the object program in the order specified by the CS-1 coding.

Poly-operations are capable of producing compiler-generated, unique labels and tags for internal use during compiling. The Appendix gives a complete discussion of compiler-generated tags.

The computer frequently employs registers A, Q, and B^7 in object program instructions resulting from poly-operations. In so doing, it destroys any previous information contained in these registers. The programer should therefore exercise caution in the use of these registers in statements preceding poly-operations. In cases where their use is necessary and the content of any of these registers is required later, the programer must save their content by transfer to a temporary storage location for later reference.

ENTRY Operation:

L	*W*	V_0
[subroutine name] ➡	**ENTRY** •	[stop condition] ➡

The **ENTRY** operation establishes a standard means of starting all subroutines. It produces either a normal entry with no jump conditions or a jump capability with Key Stop options. This operation is the first one in each subroutine; because of this it must have a label which gives the subroutine name. Although each **ENTRY** operation generates only one instruction, the variations which the instruction can assume make it a poly-operation.

V_0 - names the key which must be set on the computer console if the programmer wishes the computer to stop on exiting from the subroutine. If operand V_0 is absent (no key stop specified), the compiler generates a word of 0's for the first subroutine word in the object program. If V_0 is present, the compiler generates a 61j00 00000, where j is determined by the V_0 operand. The allowable entries for V_0 are:

STOP5 ; j = 5
STOP6 ; j = 6
STOP7 ; j = 7

Example a:

L	*W*	V_0
TYPEC ➡	**ENTRY** •	**STOP6** ➡

Generated:	*Mnemonic Equivalent*
61 600 00000	**JP • 0 • STOP6**

Example b:

L	*W*
TYPET ➡	**ENTRY**

Generated:	*Mnemonic Equivalent*
00000 00000	None

EXIT Operation:

L	W	V_0
[optional label] ➡	**EXIT** •	[jump condition] ➡

The **EXIT** operation provides a means of exiting normally from a subroutine, i.e., it generates a jump back to the **ENTRY** operation of the subroutine and thence to the main routine. The **EXIT** operation is used at every place in the subroutine where an exit from it is desired; hence, any number of exits is permitted. The label is optional.

Although the compiler generates only one instruction per **EXIT** operation, the generated instruction can assume a variety of formats. The format depends on 1) whether the V_0 operand of the foregoing **ENTRY** of the subroutine is present or absent, and 2) the **EXIT** V_0 operand itself. Because of these variations this operation is classed as a poly-operation. If the V_0 operand of the preceding **ENTRY** operation is absent, the compiler generates a $61j\underline{1}0$ nnnnn or a $60j\underline{1}0$ nnnnn. If the V_0 operand of the preceding **ENTRY** operation is present, the compiler generates either $61j\underline{0}0$ nnnnn or $60j\underline{0}0$ nnnnn. The address assigned to the preceding **ENTRY** position is nnnnn. The compiler looks for this address, then inserts it in the tag position, nnnnn, of the **EXIT** operation.

V_0 - determines j in the instruction generated by the **EXIT** operations as follows:

If the **EXIT** V_0 is:	The generated instruction is:				
		j	k	b	y
Absent	61	0	(1 or 0)*	0	nnnnn
QPOS	60	2	↓	↓	↓
QNEG	60	3			
AZERO	60	4			
ANOT	60	5			
APOS	60	6			
ANEG	60	7			

*If V_0 of previous **ENTRY** is absent, $k = 1$
If V_0 of previous **ENTRY** is present, $k = 0$

If the **EXIT** V_0 is:	The generated instruction is:				
		j	*k*	*b*	*y*
KEY1	61	1	(1 or 0)*	0	nnnnn
KEY2	61	2			
KEY3	61	3			
STOP	61	4			
STOP5	61	5			
STOP6	61	6			
STOP7	61	7			

*If V_0 of previous **ENTRY** is absent, $k = 1$
If V_0 of previous **ENTRY** is present, $k = 0$

Examples:

a. Previous **ENTRY** V_0 absent

L *W* V_0

1) **MAP2** ➡ **EXIT** • **KEY3** ➡

Generated: *Mnemonic Equivalent*

61310 nnnnn	**JP** • **L(nnnnn)** • **KEY3**

2) ➡ **EXIT** ➡

Generated: *Mnemonic Equivalent*

61010 nnnnn	**JP** • **L(nnnnn)**

b. Previous **ENTRY** V_0 present

W V_0

1) ➡ **EXIT** • **ANOT** ➡

Generated: *Mnemonic Equivalent*

60500 nnnnn	**JP** • nnnnn • **ANOT**

L *W* V_0

2) **MAP3** ➡ **EXIT** • **QNEG** ➡

Generated: *Mnemonic Equivalent*

60300 nnnnn	**JP** • nnnnn • **QNEG**

CLEAR Operation:

W V_0 V_1

➡ **CLEAR** • [number of words] • [starting address] ➡

The **CLEAR** operation clears (fills with 0's) a number of words of an area of core memory.

V_0 - specifies the number of words to be cleared. This is a Read-class operand; however, the operand code **A** is not permitted. If a value of 0 is used, the operation is a "do nothing" instruction which causes a delay of the computer. If a 1 is specified, the end result is the same as that of a mono-code CL operation

V_1 - gives the starting address of the area to be cleared. This may be a constant of maximum five digits, a tag, a tag with an increment, or a tag with an increment and a B-register designation

Example:

➡ **CLEAR • 6 • CAT+B6-2** ➡

Generated:

		Mnemonic Equivalent
70100	00006	**RPT • 6 • ADV**
16036	nnnnn*	**STR • B0 • W(CAT+B6-2)**

*nnnnn = constant specified by **CAT -2**

PUT Operation:

W V_0 V_1

➡ **PUT** • [the word] • [destination address] ➡

The **PUT** operation places a single word or half word in a designated storage address.

V_0 - expresses a Read-class operand; it may be a tag, a constant, or the content of an address. This represents the source information

V_1 - specifies the address in memory at which the word or half word is to be stored. This is a Store-class operand; it gives a constant, a B register, a tag, a tag with increment, or a tag with increment and B-register designation preceded by an appropriate operand code. A and Q are not permitted

Since register Q is used for the movement of the word, its original content is destroyed by this operation. The programer must provide for preservation of the initial contents if desired

Example a:

➡ **PUT • L(CAT+B6) • U(DOG+B3-2)** ➡

Generated:

10016	nnnnn*
14023	nnnnn

Mnemonic Equivalent

ENT • Q • L(CAT+B6)
STR • Q • U(DOG+B3-2)

Example b:

➡ **PUT • -0 • W(B6)** ➡

Generated:

10040	77777
14036	00000

Mnemonic Equivalent

ENT • Q • -0
STR • Q • W(B6)

*nnnnn is an allocated address corresponding to a tag

Example c:

➡ **PUT • 77342106 • W(DOG)** ➡

Generated:

10030	nnnnn*
14030	nnnnn

ENT • Q • W(A|||||nnnn)*
STR • Q • W(DOG)

*A ||||| nnnn is a compiler-generated tag (see Appendix); nnnnn is an allocated address corresponding to a tag

MOVE Operation:

W V_0 V_1 V_2

➡ **MOVE** • [number of words] • [from address] • [to address] ➡

The **MOVE** operation moves masses of data from one area to another. The computer moves the words of information sequentially through the Q register and may use B^7 for indexing. It does not reinstate the original content to either the B^7 or the Q register; the programer must save and restore such information if he wishes to retain it.

V_0 - specifies the number of words to be moved; the programer inserts a Read-class operand to indicate the number of words to be transferred; however, the operand codes **X**, **LX**, **UX**, or **A** are not permitted

V_1 - indicates the initial address of the area from which data will be moved; it can be an absolute address, a B register, a tag, or a tag with an increment and/or a B register designation

V_2 - states the initial address of the area to which data will be transferred; it can be an absolute address, a B register, a tag, or a tag with an increment and/or a B register designation

The compiler generates instructions in numbers varying from 2 to 10, depending upon 1) the number of words to be moved, 2) whether the V_0 operand is mnemonic or not, and 3) whether B^n designations appear in V_1 and/or V_2. If only one word is moved, the minimum number of instructions generated is two; if V_0 is mnemonic, the minimum is five instructions. Since the use of B-register designations in operands V_1 and V_2 changes the number of instructions generated by the compiler, two examples are given below. The first shows an operation with no B^n in either operand V_1 or V_2; the second contains a B^n in both operands.

Example a:

➡ **MOVE • 4 • CAT • DOG** ➡

Generated:

Mnemonic Equivalent

	12700	00003	**ENT • B7 • 3**
	10037	nnnnn	**ENT • Q • W(CAT+B7)**
	14037	nnnnn	**STR • Q • W(DOG+B7)**
[α]	72700	[α -2]	**α BJP • B7 • α -2**

Example b:

➡ **MOVE • B5 • CAT+B4 • DOG+B7-3** ➡

Generated:		*Mnemonic Equivalent*
10004	nnnnn	**ENT • Q • CAT+B4**
14010	[α -2]	**STR • Q • L(α-2)**
10007	nnnnn	**ENT • Q • DOG+B7-3**
14010	[α -1]	**STR • Q • L(α-1)**
12705	00000	**ENT • B7 • B5**
72700	[α -2]	**BJP • B7 • α -2**
61000	[α +1]	**JP • α+1**
10037	[000000]	**ENT • Q • W(0+B7)**
14037	[000000]	**STR • Q • W(0+B7)**
[α] 72700	[α -2]	**α BJP • B7 • α -2**

INCREMENT Operation:

W V_0 V_1

➡ **INCREMENT** • [B register] • [increment] ➡

The **INCREMENT** operation provides a means to either increase the number contained in a B register (B^n) by a fixed increment or decrease the number in B^n by a fixed decrement.

V_0 - specifies the B register to be incremented

V_1 - states the value of the increment by which the content of the B register is to be altered. The increment is defined by a Read-class operand

Example a:

➡ **INCREMENT • B2 • -1** ➡

Generated: *Mnemonic Equivalent*

[α] 72 200 [α+1]	α	**BJP • B2 • α+1**
[α+1] Next Instruction	α+1	Next Instruction

Example b:

➡ **INCREMENT • B5 • 32D** ➡

Generated: *Mnemonic Equivalent*

12 505 00040	**ENT • B5 • B5+32D**

Example c:

➡ **INCREMENT • B3 • -12** ➡

Generated: *Mnemonic Equivalent*

11 003 00000	**ENT • A • B3**
20 040 77765	**ADD • A • X(77765)**
	ENT • B3 • A

Example d:

➡ **INCREMENT • B4 • L(CAT+6+B2)** ➡

Generated :	*Mnemonic Equivalent*
11 004 00000	**ENT • A • B4**
20 052 nnnnn*	**ADD • A • LX(CAT+6+B2)**
12 470 00000	**ENT • B4 • A**

This poly-operation generates a variable number of object language instructions depending on the nature of the V_1 operand. A positive constant in V_1 causes a single instruction to be generated, a negative constant causes two instructions, and a symbolic name results in three instructions.

A special case occurs when the V_1 value is: -1. A B register can be decremented by one to reach zero, but not through zero; i.e., a B register containing zero, if decremented by one, remains zero.

The programer should note that the A register is used in some cases and is not restored. If he wishes to preserve the previous content of register A for later use, he must provide for its storage in another location.

---

*nnnnn: The value allocated to the tag **CAT+6** by the compiler

Upper- TAG Operation:

W V_0 V_1

➡ **U-TAG** • [upper tag name] • [lower tag name, constant, or zero] ➡

The **U-TAG** operation provides the programer with a means of expressing the upper half of a storage address by means of a symbolic tag. This is the only method by which this may be done. The programer has the option of specifying a tag in the lower half of the word also. This operation is useful for such purposes as the preparation of jump tables and the specification of upper and lower buffering limits.

V_0 - gives the name of the upper tag. A constant is not permitted

V_1 - gives the name of a lower tag if desired. If no tag is desired, this must be **0** (see example b, below)

Example a:

DOG16 ➡ U-TAG • CAT4 • MOUSE7 ➡

Tags **CAT4** and **MOUSE7** represent the upper and lower 15 bits respectively of the storage location represented by the label **DOG16.** Assume that the following allocation values are given on an allocation tape:

MOUSE7 ➡ 563

CAT4 ➡ 53210

DOG16 ➡ 3000

The computer word produced as a result of the **U-TAG** poly-operation is: 03000 53210 00563.

Example b:

RAT 13 ➡ U-TAG • DCON • 0 ➡

The tag **DCON** represents the upper 15 bits of the storage location represented by the label **RAT13.** The V_1 operand of 0 causes the lower half of the word produced to be filled with 00000.

PRINT Operation:

W V_0 V_1

➡ **PRINT** • [base address of print buffer (β)] • [jump condition] ➡

The **PRINT** operation provides the programer with a method of activating the print out of information on the High-Speed Printer. The operation initiates a subroutine, PRINTB, which causes the content of a 24D-word core buffer area (in printer code) with a base address, β, to be transferred to the High-Speed Printer as a 120D-character line of print. The PRINTB subroutine is capable of transferring the content of the buffer area either directly to the High-Speed Printer (on-line), or to a tape unit for subsequent off-line printing. The programer must, however, enable the printer once in his routine before using the **PRINT** operation.

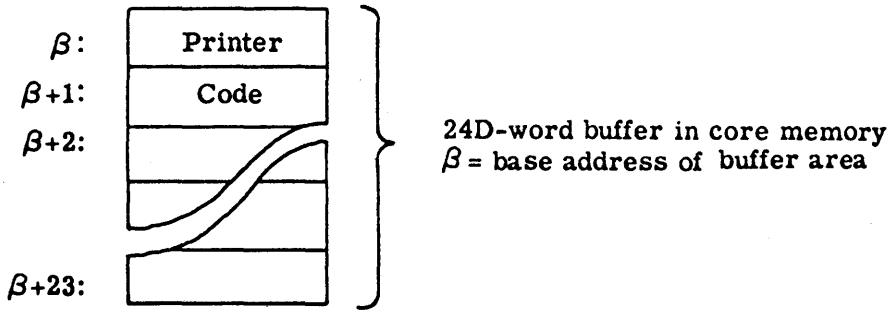


Figure 1. Buffer Area Format

V_0 - specifies the base address of the core memory buffer area. It permits a Read-class operand without an operand code or with operand codes **L** or **U**

V_1 - refers to the j-operands; these have the special j-operand meanings of the **RJP** operation. The use of this operand is optional

Example a:

GOGO ➡ PRINT • B6 • ANOT ➡

Generated:

Mnemonic Equivalent

12 706 00000	**ENT • B7 • B6**
64 500 nnnnn	**RJP • PRINTB • ANOT**

nnnnn: The address allocated to the PRINT B subroutine entry

Example b:

➡ **PRINT • HAW2** ➡

Generated:

Mnemonic Equivalent

12 700 [α]	**ENT • B7 • HAW2**
65 000 nnnnn	**RJP • PRINTB**

α : the address expressed by **HAW2**

nnnnn: the address allocated to the PRINTB subroutine entry

Example c:

➡ **PRINT • L(NUT+B4-6) • KEY2** ➡

Generated:

Mnemonic Equivalent

12 714 [α]	**ENT • B7 • L(NUT+B4-6)**
65 200 nnnnn	**RJP • PRINTB • KEY2**

α : the address expressed by **NUT-6**

nnnnn: The address allocated to the PRINT B subroutine entry

TYPEC Operation:

W $-V_0-$

➡ **TYPEC** • [information to be typed] ➡

The **TYPEC** operation causes the content (in octal) of A, Q, any B register, or any storage location to be typed by the on-line-typewriter. In addition to specifying that the numerical information in any of the above registers be typed, the programer may issue special commands to the typewriter. These commands, used as operands in the special format described below, may cause the typewriter to do the following three things:

Operand	Performance
• |**CR**| •	Causes the typewriter to do a carriage return
• |**SP**| •	Causes the typewriter to skip a space
• |**TAB**| •	Causes the typewriter to move to the next tabulation stop

By properly inserting these commands as operands between the operands denoting the information to be typed, the programer can control the format (spacing and lines) of the information typed. The vertical bars are the special control symbols for indicating that the operand is an order directing the typewriter. Each of these three special operands *must* begin with and end with a vertical bar, and *must* each be separated by point separators from other operands.

$-V_0-$ - specifies the operands in the operand position in the order in which they are to be read and/or executed. These operands are of four types: $p1$, $p2$, $p3$, and $p4$. They may appear in any order, depending on the programer's desires or needs. Point separators must separate each operand.

$p1$ - gives the locator of a value to be typed; it consists of an operand code of **L**, **U**, or **W** together with a normal Read operand in parentheses. The parentheses may contain a tag, B-register designation, or increment, or any combination of these

$p2$ - gives a tag or label allocation value, without operand code, which the typewriter will type

$p3$ - specifies a constant, of five digits or less, to be typed

Exception: The value, zero, will not be typed if expressed as an operand. Zeros may be obtained by using the **TYPET** operation.

p 4 - states a special typewriter command symbol. Valid symbols are |**CR**| , |**SP**|, and |**TAB**|; these command symbols cause the typewriter to perform a carriage return, to skip a space, and to move to a tabulator stop respectively

Example:

L *W* - V_0 -

FIRST ➡ TYPEC • U(BETA + B3-6) • 2576 • |CR| • A • |SP| • Q • BETA ➡
*p*1 *p*3 *p*4 *p*1 *p*4 *p*1 *p*2

LAST ➡ STR • Q • W (GAMMA) ➡

The **FIRST** operation above causes the following equivalent instructions and codes to be generated (except for the **LAST** operation):

FIRST ➡ RJP • TYPEC ↴
➡ 00023 • BETA - 6 ↴
➡ 00000 • 02576 ↴
➡ 77450 • 00000 ↴
➡ 00070 • 00000 ↴
➡ 77040 • 00000 ↴
➡ 00000 • 00000 ↴
➡ 00000 • BETA ↴
LAST ➡ STR • Q • W(GAMMA) ➡

The TYPEC subroutine checks the first two characters of each of the operations following the Return Jump to TYPEC subroutine. If these are 00, it replaces them with 10 or 20; if they are 77, it interprets the characters following as commands to the typewriter (type *p* 4 operands).

The operation labeled **LAST** is not a part of the **TYPEC** performance. It illustrates that the programer must follow the **TYPEC** operation with an

operation which will not cause the generation of a word of 0's in the object program. In other words, the next instruction in the object program must have a legitimate computer instruction code.

The compiled object program uses the TYPEC subroutine to produce the typeout. In general this poly-operation generates a Return Jump to the TYPEC subroutine, followed by an operation statement for each operand, directing the computer either to type the information as specified or to perform the command given. The TYPEC subroutine stores the contents of the registers it uses and restores them upon completion of the typeout.

NOTE: Because a 77 in the function code position has special meaning to the TYPEC routine, do not follow a TYPEC statement with a function code of 77.

TYPET Operation:

$$\overset{W}{} \qquad\qquad \overset{-V_0-}{}$$

$$\Rightarrow \quad \textbf{TYPET} \quad \bullet \quad \left[\text{text and typewriter commands}\right] \Rightarrow$$

The **TYPET** operation generates a section of object language program which, when run on the computer, causes the typewriter to type the message given by the $-V_0-$ operand of the **TYPET** operation. No point separators appear between the parts of the V_0 operand. The commands to the typewriter, viz., carriage return, space, and tab, intersperse with the text according to the needs and desires of the programer. These typewriter commands separate from the text by means of a vertical bar, | , before and after each command:

• text |**SP**| text |**CR**| text |**TAB**| text ➡

Exception: Where a space is desired between characters of the typed text, a space code symbol, Δ, may substitute for the command, |**SP**|.* The above example would then be:

• text Δ text |**CR**| text |**TAB**| text ➡

The programer can use either symbol for a space. Where a tab follows a carriage return, the format should be |**CR**||**TAB**| :

• text Δ text |**CR**||**TAB**| text ➡

$-V_0-$ specifies the text to be typed, interspersed with the typewriter commands needed to produce the text format desired by the programer. The typewriter commands and their symbols are:

Carriage Return:	|**CR**|
Tab:	|**TAB**|
Space:		**SP**	or Δ

*Only four consecutive space codes permitted (ΔΔΔΔ)

Example:

```
FIRST ➡ TYPET • ABC | CR | DE | TAB | ➡

      ➡ TYPET • FGH | CR ||TAB | |Δ J ➡
```

produces the object language program:

```
FIRST ➡ RJP • TYPET
        65000  TYPET

        ↑ A   B   C )
        47302     31645

        D E → STOP
        22205   17700

        RJP • TYPET
        65000  TYPET

        ↑ F   G   H )
        47261     30545

        → I   Δ  J STOP
        51140     43277
```

During the running of the object program, the TYPET subroutine then uses the above object language program to produce the typewriter printout.

Any number of space commands can precede or follow the | **CR** | and | **TAB** | commands without affecting the text. Putting more than one space command between parts of the text has the effect of spreading these parts of the text farther apart on the typewritten page.

There is no provision for controlling the case of the characters in the output message. Alphabetical information is typed in upper case, numerical information in lower case. The TYPET subroutine, which unpacks the codes taken from the object language program, recognizes the end of the message by detecting the code, 77.

W $-V_0-$

➡ **PUNCHC** • [parameters for information and/or typewriter commands] ➡

The **PUNCHC** operation causes the content (in octal) of A, Q, any B register, or any storage location to be punched by the High-Speed Punch. In addition to directing that the numeric information in any of the above registers be punched, the programer may write three special command symbols. These three symbols are typewriter commands which, when the punched paper tape is on a typewriter, will direct the typewriter to perform certain carriage operations. These operations control the format of the typewriter typeout; they include:

Operand	Performance
• |**CR**| •	Causes the typewriter to do a carriage return
• |**SP**| •	Causes the typewriter to skip a space
• |**TAB**| •	Causes the typewriter to move to the next tabulation stop

By properly inserting these commands as operands between the operands denoting the information be typed, the programer can control the format (spacing and lines) of the information typed. The vertical bars are the special control symbols for indicating that the operand is an order directing the typewriter. Each of these three special operands *must* begin with and end with a vertical bar, and each *must* be separated by point separators from other operands.

$-V_0-$ - specifies the operands in the operand position in the order in which they are to be read and/or executed. These operands are of four types: $p1$, $p2$, $p3$, and $p4$. They may appear in any order, depending on the programer's desires or needs. Point separators must separate each operand.

$p1$ - gives the locator address of a value to be typed; it consists of a normal Read-class operand

$p2$ - gives a tag or label allocation value, without operand code, which the typewriter will type

$p3$ - specifies a constant, of five digits or less, to be typed

Exception: The value, zero, will not be typed if expressed as an operand. Zeros may be obtained by using the **PUNCHT** operation.

$p4$ - states a special typewriter command symbol. Valid symbols are |**CR**| , |**SP**| , and |**TAB**| ; these command symbols cause the typewriter to perform a carriage return, to skip a space, and to move to a tabulator stop respectively

Example:

```
  L        W                        - V0 -
FIRST ➡ PUNCHC • Q  •  |CR| • L (ALPHA + B3)  • |TAB| • 50 • |SP| • INST 4  ➡
                p1     p4        p1            p4     p3    p4      p2
NEXT  ➡ ENT • A • U (GAMMA)  ➡
```

The **FIRST** operation above causes the following equivalent (in some cases, incomplete) operations to be generated:

```
FIRST ➡ RJP    •  PUNCHC )
      ➡ 00000  •  00000 )
      ➡ 77450  •  00000 )
      ➡ 00013  •  ALPHA )
      ➡ 77510  •  00000 )
      ➡ 00000  •  00050 )
      ➡ 77040  •  00000 )
      ➡ 00000  •  INST4 )
```

The PUNCHC subroutine checks the first two characters of each of the operations following the Return Jump to PUNCHC subroutine. If these are 00, it replaces them with 10; if they are 77, it interprets the characters following as commands to the typewriter (type $p4$ operands).

The operation labeled **NEXT** is not a part of the **PUNCHC** performance. It illustrates that the programer must follow the **PUNCHC** operation with an operation which will not cause

the generation of a word of 0's in the object program. In other words, the next instruction in the object program must have a legitimate computer instruction code.

The compiler object program uses the PUNCHC subroutine to produce the typeout. In general this poly-operation generates a Return Jump to the PUNCHC subroutine, followed by an operation statement for each operand, directing the computer either to type the information as specified or to perform the command given.

NOTE: Because a 77 in the function code position has special meaning to the PUNCHC routine, do not follow a PUNCHC statement with a function code of 77.

PUNCHT Operation:

W $-V_0-$

➡ **PUNCHT** • [text and/or typewriter commands] ➡

The **PUNCHT** operation causes the High-Speed Punch to punch the text(s) which the programer has written in the $-V_0-$ operand position of the **PUNCHT** operation. It also punches the codes for SP, CR, and TAB, which control the typewriter carriage movements during a listing of the punched tape. The programer controls the format of the typewriter listing by interspersing the carriage control symbols between his texts as he desires. No point separators appear between the parts of the $-V_0-$ operand. Each carriage control symbol must have a vertical bar, | , before and after it.

Exception: Where a space is desired between characters of the typed text, a space code symbol, Δ , may substitute for the command, |**SP**|*.

Example:

a. ➡ **PUNCHT** • text |**SP**| text |**CR**| text |**TAB**| text ➡

b. ➡ **PUNCHT** • text Δ text |**CR**| text |**TAB**| text ➡

Where 2 carriage control symbols appear consecutively, each one must have vertical bars before and after it.

Example:

➡ **PUNCHT** • text |**CR**| |**TAB**| text ➡

$-V_0-$ - specifies the text to be typed, interspersed with the typewriter commands needed to produce the format desired by the programer.

If the text is too long to put into one L_0 **PUNCHT** operation, successive operations can be written. Labels on these operations are optional.

Example:

FIRST ➡ **PUNCHT** • **PAY** Δ **TAX** |**CR**| **ON** |**TAB**| ➡

➡ **PUNCHT** • **OCT** |**SP**| **15** |**CR**| ➡

---

*Only four consecutive space codes permitted (ΔΔΔΔ)

The operations and codes generated in the running program by the above poly-operations are:

```
FIRST ➡ RJP • PUNCHT
        65000   PUNCHT

        ↑ P A Y Δ
        47153   02504

        T A X ⤸ 0
        01302   74503

        N →  STOP
        06517   70000

        RJP • PUNCHT
        65000   PUNCHT

        O C T  Δ ↓
        03160   10457

        1 5 ⤸ STOP
        52524   57700
```

When the running program is subsequently performed, the PUNCHT subroutine then causes the High-Speed Punch to punch out octal codes above.

Any number of space commands can appear consecutively anywhere in the text. The effect is to vary the spacing between parts of the texts on the hard copy.

There is no provision for controlling the case of the characters in the output message. Alphabetic information appears in upper case, numeric in lower case. The PUNCHT subroutine, which translates sequentially the codes taken from the object language program, recognizes the end of the message by detecting the code 77.

TYPE-DECimal Operation:

W $-V_0-$

➡ **TYPE-DEC** • [information to be typed] ➡

The **TYPE-DEC** operation causes the content (in decimal) of A, Q, any B register, or any storage location to be typed by the on-line typewriter. In addition to specifying that the numerical information in any of the above registers be typed, the programer may issue special commands to the typewriter. These commands, used as operands in the special format described below, may cause the typewriter to do the following three things:

Operand	Performance
• | **CR** | •	Causes the typewriter to do a carriage return
• | **SP** | •	Causes the typewriter to skip a space
• |**TAB**| •	Causes the typewriter to move to the next tabulation stop

By properly inserting these commands as operands between the operands denoting the information to be typed, the programer can control the format (spacing and lines) of the information typed. The vertical bars are the special control symbols for indicating that the operand is an order directing the typewriter. Each of these three special operands must begin with and end with a vertical bar, and must each be separated by point separators from other operands.

$-V_0-$ specifies the operands in the operand position in the order in which they are to be read and/or executed. These operands are of four types: $p1$, $p2$, $p3$, and $p4$. They may appear in any order, depending on the programer's desires or needs. Point separators must separate each operand

$p1$ - gives the locator of a value to be typed; it consists of an operand code together with a normal Read operand in parentheses. The parentheses may contain a tag, B-register designation, or increment, or any combination of these

$p2$ - gives a tag or label allocation value, without operand code, which the typewriter will type

$p3$ - specifies a constant of five digits or less to be typed

Exception: The value, zero, will not be typed if expressed as an operand. Zeros may be obtained by using the **TYPET** operation.

*p*4 - states a special typewriter command symbol. Valid symbols are |**CR**|, |**SP**|, and |**TAB**|; these command symbols cause the typewriter to perform a carriage return, to skip a space, and to move to a tabulator stop respectively

Example:

L	*W*									-V_0 -							
FIRST	➡	**TYPE-DEC**	•	**U(BETA+B3-6)**	•	**2576**	•	|**CR**|	•	**A**	•	|**SP**|	•	**Q**	•	**BETA**	➡
				*p*1		*p*3		*p*4		*p*1		*p*4		*p*1		*p*2	

NOTE: Because a 77 in the function code position has special meaning to the TYPE-DEC routine, do not follow a TYPE-DEC statement with a function code of 77.

PUNCH-DECimal Operation:

W $-V_0-$

➡ **PUNCH-DEC** • [parameters for information and/or typewriter commands] ➡

The **PUNCH-DEC** operation causes the content (in decimal) of A, Q, any B register, or any storage location to be punched by the High-Speed Punch. In addition to directing that the numeric information in any of the above registers be punched, the programer may write three special command symbols. These three symbols are typewriter commands which, when the punched paper tape is on a typewriter, will direct the typewriter to perform certain carriage operations. These operations control the format of the typewriter typeout; They include:

Operand	
•| **CR** |•	Causes the typewriter to do a carriage return
•| **SP** |•	Causes the typewriter to skip a space
•| **TAB** |•	Causes the typewriter to move to the next tabulation stop

By properly inserting these commands as operands between the operands denoting the information be typed, the programer can control the format (spacing and lines) of the information typed. The vertical bars are the special control symbols for indicating that the operand is an order directing the typewriter. Each of these three special operands must begin with and end with a vertical bar, and each must be separated by point separators from other operands.

$-V_0-$ specifies the operands in the operand position in the order in which they are to to be read and/or executed. These operands are of four types: $p1$, $p2$, $p3$, and $p4$. They may appear in any order, depending on the programer's desires or needs. Point separators must separate each operand.

$p1$ - gives the locator address of a value to be typed; it consists of a normal Read-class operand

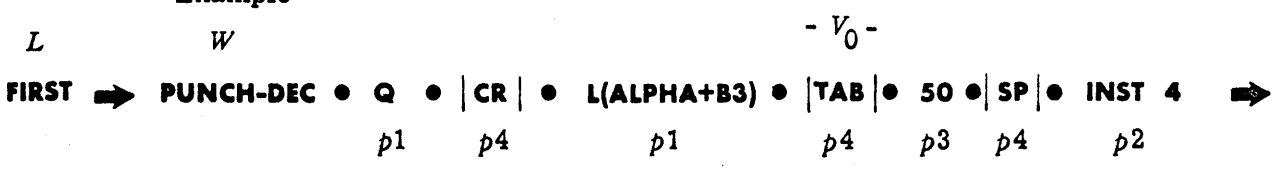
$p2$ - gives a tag or label allocation value, without operand code, which the typewriter will type

$p3$ - specifies a constant, of five digits or less, to be typed

Exception: The value, zero, will not be typed if expressed as in operand. Zeros may be obtained by using the **PUNCHT** operation.

$p4$ - states a special typewriter command symbol. Valid symbols are |**CR**|, |**SP**|, and |**TAB**| ; these command symbols cause the typewriter to perform a carriage return, to skip a space, and to move to a tabulator stop respectively

Example



NOTE: Because a 77 in the function code position has special meaning to the PUNCH-DEC routine, do not follow a PUNCH-DEC statement with a function code of 77.

DECLARATIVE OPERATIONS

DECLARATIVE OPERATIONS

The programer frequently wishes to supply to the compiler certain information for use in the compiling process which does not generate an instruction. The information may be involved in subsequent operations in constructing a machine-code instruction; or it may be substituted for already existing data or information, thereby extending the scope and power of the operation. This is especially true where, by changing one operand in an operation, the operation may perform a variety of similar tasks.

Declarative operations, therefore, are operations which do not result in the generation of instructions in the object program; they rather 1) give information about relationships, such as equality between data and/or symbolic names, 2) make assertions, and 3) define a procedure. Declarative operations state facts and provide information which the compiler either utilizes, or stores and later incorporates into the object program instructions it generates.

In all cases, the programer must state the declarative operation at some place *ahead* of the action operation which is to use it. These operations can intermingle with action operations anywhere in the program, provided they comply with the above priority restriction. It is often worthwhile for the programer to place the declarative statements on a separate **PROGRAM** tape or punched cards, to be read into the compiler before the main program.

EQUALS Operation:

L W $-V_0-$

[*unknown* tag] ➡ **EQUALS** • [*known* value: lab/tag ± i, or a constant] ➡

The **EQUALS** operation establishes an equivalence between one expression, L, whose allocation value is *unknown* and another expression, V_0, for which the allocation value is *known*. This provides the programer with a versatile allocation aid whereby he can transfer an allocation value from one label or tag to another tag. Since this operator is concerned solely with allocation, a compiler function, it generates no instructions in the internal program.

This operation permits addition, +; subtraction, -; multiplication, ( ) ( ); or division, /, with known values. A term in the arithmetic process may be a constant or a tag (± increment is permitted); a factor is an expression made up of terms connected by + or - signs; it corresponds to an address. Computations progress sequentially upon factors, with the terms in each factor accumulated separately before multiplication and/or division. Thus the computations are essentially multiplications and/or divisions of addresses

L - gives the name of the *unknown* tag to which a numeric value is to be assigned

$-V_0-$ - gives: a) the constant which the programer wishes to assign, or b) the label or tag whose value is known, with or without an increment, or c) a combination of labels, tags, and/or constants in an arithmetic relationship. Each value may consist of one or more of the following: 1) a number; 2) a label or tag; 3) a numeric increment; 4) a numeric decrement; 5) a tag with increment; or 6) a tag with decrement. Two or more of these may be joined together by either successive multiplications or by successive divisions, but not a combination of the two processes. The expression may also be an accumulation of two or more additions and/or subtractions of known values. In expressions combining addition or subtraction with either multiplication or division, the addends or subtrahends are treated as increments or decrements to the factor with which they are immediately associated

Example:

NUB-4/CHOP-5 + COB

(NUB-4) serves as the dividend, and **CHOP-5 + COB** are combined into a single divisor value. Regardless of how - V_0 - is expressed, a single absolute allocation value for the entire expression must be known by the compiler. The compiler stores this value for later use when the *unknown* tag, L, is referenced. B-register designations are not permitted in the - V_0 - operand position.

Example a:

CAT ➡ EQUALS • DOG + 2 - HORSE ➡

(e.g., if **DOG** = 300 and **HORSE** = 100; **CAT** = 202.)

Example b:

SR3 ➡ EQUALS • SR4 - 33D + 44 + RATS ➡

Example c:

TMAX ➡ EQUALS • 45600 ➡

Example d:

PECE ➡ EQUALS • (DOVE + 2) (MANY) ➡

Example e:

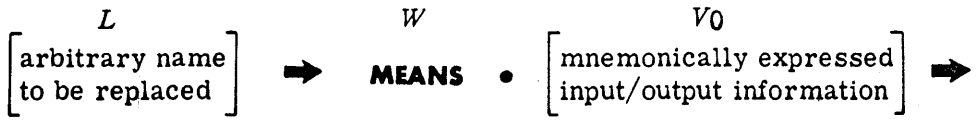
CRUMP ➡ EQUALS • RACER-4/BOOL /5 ➡

In this example, the computer subtracts 4 from the value represented by **RACER**, divides this result by the value allocated to **BOOL**, then divides this result by 5; it then assigns this value to **CRUMP**. The **EQUALS** operation thus has the power to perform arithmetic computations within the compiler.

Note: In cases of multiplication, if the product exceeds five characters, an error printout occurs.

In cases of division, if the quotient is not an integer, an error printout occurs.

MEANS Operation:



The **MEANS** operation replaces an arbitrary name in the label, L, position with input/output information expressed in mnemonics. It permits programs to be written with complete flexibility concerning the assignment of channels to external equipment. By holding the assignment of external equipment open until needed, the programer can at that time determine which of the specific channels are available for use. He then replaces the general assignment with the one he desires by entering a **MEANS** statement in the L_0 program prior to the operation performing the input/output function. The computer then takes what is in $-V_0-$ and replaces in the subsequent I/O operation the value assigned to L in the **MEANS** operation. Thus the programer can assign an external equipment to any Input/Output or Function channel. L_0 operations which may contain replaceable general operands include: **STR, JP, TERM, IN, OUT, EX-COM, EX-COM-MW, SIL-EX,RIL-EX, ECDM, AND DCDM.**

This operation does not generate any internal compiler instruction; it makes the indicated substitution, then drops out. The Input/Output operation(s) subsequently involved in the transfer then function as usual, using the substituted operand.

The **MEANS** operation is applicable only to the assignment of input/output parameters. It cannot be used interchangeably with **EQUALS**, nor can **EQUALS** be used to perform the task assigned to **MEANS**.

L - gives an arbitrary name to be replaced. This has normal label format; i.e., there are no special restrictions regarding the symbols entered in L

V_0 - states the specific input/output assignment to be substituted for L. Entries in this operand position are presently restricted to information regarding Input/Output specifications. They may consist, therefore, of any unique external equipment assignment, e.g., **C12ACTIVEIN**; **C5**; or a function constant

Example **MEANS** operations:

a) **LOB ➡ MEANS • C12**

b) **PITCH ➡ MEANS • C15ACTIVEOUT**

c) **CUT ➡ MEANS • C0**

Examples of Input/Output operations with which the foregoing examples may be used:

a) **➡ EX-FCT • LOB • 426**

b) **➡ JP • POST • PITCH**

c) **➡ OUT • CUT • W(SNAP) • MONITOR**

d) **➡ TERM • LOB • INPUT**

Note: **MEANS** operations appear either within an L_0 program or as separate input under a **PROGRAM** header. In both cases the **MEANS** operations become a part of the compiler's L_1 table storage.

RESERVE Operation:

W V_0

➡ **RESERVE** • [number of words] ➡

The **RESERVE** operation sets aside a block of memory locations in the running (object) program. It does so by adding the number expressed by the V_0 operand to the current allocation address and storing the next generated instruction at the incremented address. Thus the reservation of space begins at the location following that of the previously generated instruction and includes the V_0 number of continuous locations. The compiler does not clear these locations; it merely by-passes them during allocation. Some of the special reasons for reserving such an area include:

1. Setting aside a specific area for the storage of parameters
2. Leaving an area open for working storage
3. Reserving space, e.g., at the end of the program, for expansion purposes
4. Subsequent insertion of other program instructions

V_0 - specifies the number of words to be reserved. The programer may enter only a constant in this operand location

Example:

CAT ➡ **RESERVE** • 4 ➡

Result:

[α] ➡ [previously generated instruction] ➡

CAT ➡ - - - - - -
- - - - - -
- - - - - -
- - - - - -

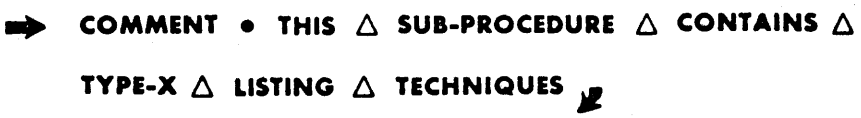
[α+5] ➡ [next generated instruction] ➡

The use of a label to identify the first word of the reserved area permits the referencing of the entire area or of any word location in it. The programer may gain access to any word of information in the area by referencing the label, or the label plus the increment required to designate the desired word. The operation **ENT • A • W(CAT+3)**; for example, reads in the A register the content of the fourth word in the reserved area in the example above.

COMMENT Operation:

The **COMMENT** operation permits the programer to place a message(s) within the input program to provide added information for edited records of the problem definition. This operation is declarative; it has no dynamic meaning to the input language.*

Example:



* **COMMENT** is not a *true* poly-operation since it does not generate machine instructions

CS-1 INPUT

Input to the CS-1 Compiler (programs, allocations or corrections) is in two media; paper tape and 80-column cards. The user selects the input medium available.

Paper tape input is in standard paper tape codes (FD, ASCII, FLEX, etc.). Card input is on standard 80-column cards. An Off-Line Card-to-Tape process transfers the data from cards to magnetic tape; therefore, the actual input to the computer is via magnetic tape. In addition to the input media types heretofore mentioned, certain data may be manually entered via console registers during compilation runs. This consists only of minor entries of data such as selecting outputs.

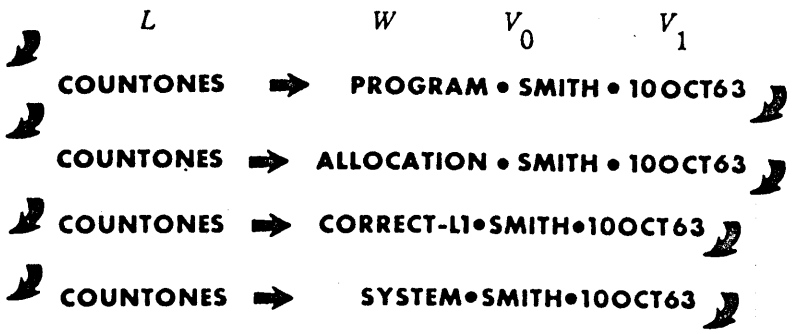
The programer uses a uniform set of symbols as separators in all coding. See Table 1 Separator interpretation differs between card and paper tape input media as described in their respective subsections.

TABLE 1. CS-1 CODING SEPARATORS

Symbol	Coding Significance
➡	Delimits the *statement.* Must always precede the statement operator. Must precede *notes;* omit if notes not given.
⮯	Signifies *end* of operation. Must precede header operations.
•	Separates statement components
( )	1) Indicates contents of a storage location
	2) Specifies data unit subname or subscript
,	Separates item from word or field in a data name.
( )( )	Specifies multiplication
/	Specifies division
+	Specifies addition
-	Specifies subtraction
Δ	Specifies space
|	Special control character

Input to CS-1 (program, allocation, or correction) requires an initial *header operation* for identification purposes. A header consists of the program name in the *L* coding position, a header-type operator in the *W* position and two identifying operands, V_0 and V_1, in which the

programer specifies his name and the date. The examples below illustrate four typical headers:



CS-1 requires a limited amount of input arrangement. Information to the compiler always precedes the routine being compiled. The programer usually places information to the compiler under a **C-CONTROL** header. Subordinate to the **C-CONTROL** header are minor headers followed by their respective operations. Independent control operations also follow. The **C-CONTROL** header merely categorizes control information to the compiler under a single header; its use is optional since the minor headers and independent control operations can be given independently. (See COMPILER CONTROL OPERATIONS.)

The extent of compiler control as CS-1 input is optional. The programer may control all compiling activity by paper tape or cards, or he may instruct the compiler operator to control much of the compiling at the console.

The routine or routines, being compiled require one of four headers, **PROGRAM**, **SYSTEM**, **SYS-DD**, or **SYS-PROC**. The **PROGRAM** header identifies a standard *Computer-Oriented* routine for compiling. The **SYSTEM** header identifies a *Problem-Oriented* routine for compiling. CS-1 requires the **SYSTEM** header whenever it is to select data designs and/or procedures from a User Library. It then compiles these with the L_0 program. CS-1 permits *Problem-Oriented* input routines with initial headers of **SYS-DD** or **SYS-PROC** *only* when no User Library data is required.

The correcting process requires a separate correction run. The correction tape or cards used contains pairs of input operations for the purpose of making corrections to the L_0 input routine(s). The header operation for this tape, **CORRECT-L1**, requires a label and two operands (programer's name and current date). The programer can receive a corrected L_0 on either paper tape or magnetic tape. The corrected L_0 tape is then used as input in subsequent compiling runs. See CS-1 OUTPUT.

A **LIBRARY** header informs CS-1 of an oncoming library manipulation or library listing requests. Nothing must precede this header. Data following this header inserts, replaces, or deletes library information. In addition, the data may also call for listings of data designs, procedures, or the library directory. (See CS-1 LIBRARIAN.)

The following skeleton program samples illustrate typical CS-1 read-in arrangement.

Sample 1. (Compile a simple routine)

Read-in-order

1.

[label] ➡ **ALLOCATION** • [programer's name] • [date]

➡ (Ref. ALLOCATION)

➡

2.

[label] ➡ **PROGRAM** • [programer's name] • [date]

➡
➡ } Mono-operations and Poly-operations
➡ (Ref. COMPUTER-ORIENTED OPERATIONS)
➡

➡ **END-DATA** (for card input only)

Instructions to compiler operator

1) Read-in L_0

2) Output types desired

Sample 2. (Correct a simple routine)

Read-in order

1.

[label] ➡ **CORRECT-L1** • [programer's name] • [current date]

➡ (Ref. PROGRAM CORRECTIONS)

2.

[label] ➡ **PROGRAM** • [programer's name] • [date]

➡ Mono-operations and Poly-operations (Ref. COMPUTER-ORIENTED OPERATIONS)

➡ **END-DATA** (for card input only)

Instructions to compiler operator

1) Read-in L_0

2) Outputs desired

Sample 3. (Compile single routine under compiler control)

Read-in order

1.

[label] ➡ **C-CONTROL** • [programer's name] • [date]

➡ **OUTPUTS** • [n] • [nn] • [nnn]

2.

➡ **ALLOCATION**

➡ (Ref. ALLOCATION)

➡

3.

[label] ➡ **PROGRAM** • [programer's name] • [date]

➡

➡ Mono-operations and Poly-operations
(Ref. COMPUTER-ORIENTED
OPERATIONS)

➡

➡

➡ **END-DATA** (for card input only)

Sample 4. (Compile a System Procedure)

Read-in order

1.

```
[label]  ➡ C-CONTROL • [ programer's name ] • [date]
         ➡ OUTPUTS • [n] • [n]
         ➡                  (Ref. COMPILER CONTROL OPERATIONS)
         ➡ ALLOCATION
BASE     ➡ 30500
```

2.

```
[label γ] ➡ SYS-PROC
          ➡ LOC-DD
          ➡
          ➡ END-LOC-DD
          ➡ PROCEDURE • [α]
          ➡                  (Ref. DATA DESIGN OPERATIONS and
                                   CS-1 LIBRARIAN)
          ➡ END-PROC • [α]
          ➡ PROCEDURE • [γ]
          ➡
          ➡ END-PROC • [γ]
```

Sample 5. (Compile a System Routine)

Read-in order

1.

```
[label] ➡ C-CONTROL • [ programer's name ] • [date]
        ➡ OUTPUTS • [n] • [nn] • [nnn]
        ➡ ALLOCATION        (Ref. COMPILER CONTROL
   BASE ➡ 2000                    OPERATIONS)
ENTRANCE ➡ [δ]
```

2.

```
[label] ➡ SYSTEM • [ programer's name ] • [date]
        ➡ SEL-DD • [label]
        ➡                  (Ref. CS-1 LIBRARIAN)
        ➡ SEL-SYS • [key] [non-unique labels only]
        ➡ SEL-PROC • [label, key] • [non-unique labels only]
```

3.

```
[label]     ➡ SYS-DD
            ➡        (Ref. PROBLEM-ORIENTED OPERATIONS and
            ➡             CS-1 LIBRARIAN)
            ➡
            ➡ END-SYS-DD
[label δ]   ➡ SYS-PROC
            ➡ LOC-DD
            ➡
            ➡ END-LOC-DD
            ➡ PROCEDURE • [α]
            ➡                  (Ref. DATA DESIGN OPERATIONS and
            ➡ END-PROC • [α]          CS-1 LIBRARIAN)
            ➡ PROCEDURE • [δ]
            ➡
            ➡ END-PROC • [δ]
```

Sample 6. (Replace a data design and insert a procedure in a User Library)

Read-in order

1.

```
[label] ➡ LIBRARY • [ programer's name ] • [date]
                                    (Ref. CS-1 LIBRARIAN)
        ➡ RPL-DD • [α] • [ programer's name ] • [date]
        ➡ INS-PROC • [δ,key] • [ programer's name ] • [date]
```

2.

```
[α] ➡ SYS-DD
    ➡
    ➡                  (Ref. PROBLEM-ORIENTED OPERATIONS and
    ➡                       CS-1 LIBRARIAN)
    ➡ END-SYS-DD
```

3.

```
[δ] ➡ SYS-PROC
    ➡ LOC-DD
    ➡
    ➡ END-LOC-DD       (Ref. DATA DESIGN OPERATIONS and
    ➡ PROCEDURE • [δi]      CS-1 LIBRARIAN)
    ➡
    ➡
    ➡ END-PROC • [δ]
```

Note: No **C-CONTROL** operations are required since librarian operations direct compiler action.

PAPER TAPE INPUT

Paper tape is a medium of input for program, allocation, or correction data to the CS-1 compiler. Standard code characters provide format control. Input requirements permit either upper- or lower-case characters for all input with the exception of separators. Upper-case characters, however, are recommended for all input except those that specifically require lower-case characters (refer to TABLE II.)

TABLE II. EQUIVALENT INPUT FORMAT CODING SYMBOLS-PAPER TAPE INPUT

Software name	Software Symbol	Flexowriter Codes	Field Data Symbol Substitution	Field Data Codes	ASCII Codes
Carriage Return	↵	45		04 03	15 12
Shift Up	↑	47		01	-
Shift Down	↓	57		02	-
Tab	→	51	Special ▭	76	137
Point Separator	•	44	Apostrophe '	72	52
Double Period	••	57 42 42		75 75	56 56
Space	△	04		05	40
Comma	,	57 46 47		56	54
Vertical Bar	|	57 50 47	Exclamation !	55	41
Plus	+	57 54 47		42	53
Minus	-	56		41	55

A double lower case period in the *L* coding position indicates the end of tape read-in. This is a special control symbol used only with paper tape to terminate input. Therefore, each paper tape begins with a header and ends with a double-period end symbol.

The following examples illustrate the basic format for program operations and the common usage of separators therein.

L	*W*	V_0	V_1	V_2	*N*
CAT4 ➡	ENT •	Q •	W(RAT3-2+B6) •	QNEG ➡	RATCHECK ↷
➡	RPT •	36 •	BACK ↷		

Notice that it is essential to use a straight arrow before each operator even when a label is not given. The second straight arrow is used only when notes are given. The point symbol separates the components of the statement. Parenthesis symbols indicate contents of a storage location modified by an operand code. Also within the parenthesis symbols are data unit subnames and subscripts or multiplication factors. Spaces are permitted throughout the operation. The curved arrow indicates the end of the operation, or of the notes if present.

Coding forms are used to prepare CS-1 L_0 programs for paper tape input. Figure 1 is a typical coded program for paper tape input preparation.

TITLE COUNT ONES
PAGE _____ of _____

UNIVAC CS-I CODING FORM

PROGRAMER WALEEN SMITH
PLT I EXT 786 MS 120
DATE 15 OCT 63

LABEL	OPERATOR	OPERANDS AND NOTES
	HEADER TYPE	
COUNT ONES	PROGRAM	SMITH • 10 OCT 63
CTONES Ø	CL	B2 • → SET WORD INDEX
CTONES 1	ENT	B1 • 35 → SET SHIFT INDEX
	CL	A • → SET SUM Ø TO ZERO
	ENT	Q • W(WORD Ø + B2)
CTONES 2	LSH	Q • 1 • Q POS → TEST EACH BIT FOR 0 OR 1
	ADD	A • 1 → INCREASE SUM IF 1 FOUND
	BJP	B1 • CTONES 2
	STR	A • W(SUM Ø + B2) → SUM STORAGE
	BSK	B2 • NWORDS
	JP	CTONES 1 • STOP 5 → CONTINUE COMPUTING. SUMS
CTONES 3	JP	CTONES 3 • STOP → END
••		

Figure 1. Typical Coded Program for Paper Tape Input Preparation

CARD INPUT

Punched cards are a medium of input to the CS-1 Compiling System. Data are first key-punched on standard 80-column cards. Computer input is via magnetic tape (card-to-tape conversion) or direct card read-in.

Basically the coding format is similar for either card or paper tape input. A four-digit card number is sequentially assigned to each operation and insert corrections are given two additional digit assignments. The card coding form provides space for the card and insert numbers; it also provides space for a four-character deck identifier on each coding sheet. This alphanumeric deck identifier is unique for each program.

Interpretation of coding separator symbols for card input is given in TABLE III.

TABLE III. CODING SYMBOLS - CARD INPUT

Symbol	*Key*	*Rows Punched*
➡ (start statement)	SKIP	(none)
➡ (end statement)	= -	4, 8
↵	REL	(none)
|	$ *	11, 3, 8
•	$ *	11, 4, 8
(	. (	0, 4, 8
¢	. (	12. 3, 8
,	,)	0, 3, 8
)	,)	12, 4, 8
/	0 /	0, 1
+	+ P	12
-	— SKIP	11

The straight coding arrow is interpreted according to its format position; it represents a SKIP Key at the beginning of a statement and three dashes at the end of a statement. The point coding separator is represented by the "*" key in all card input. (See Figure 2).

CARD CONTROL: The **END-DATA** operation indicates the end of a data-read-in segment. This stops the read-in process and allows the computer operator to initiate compiling action on the segment of input. (See Figure 3).

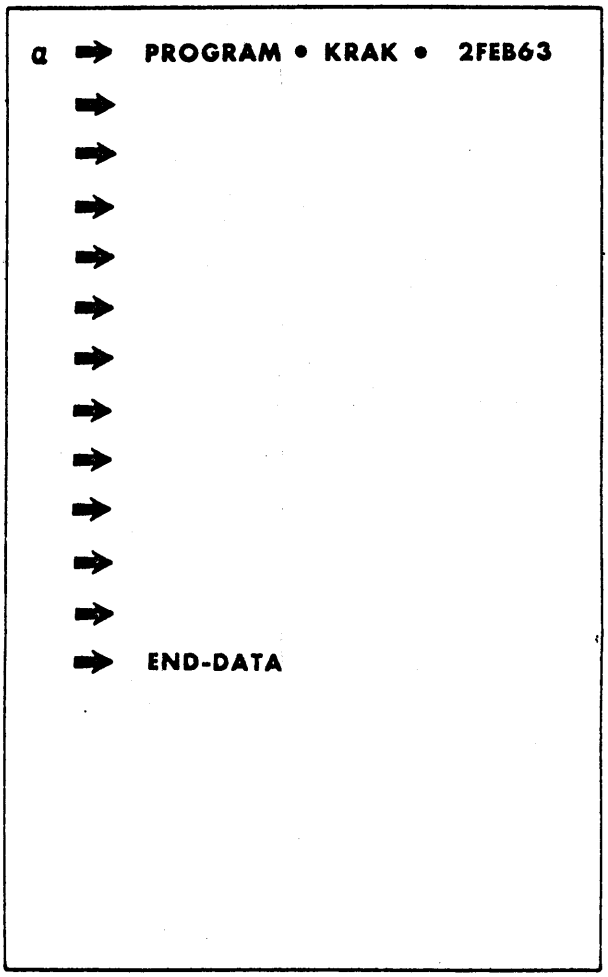


Figure 2. Single Program Input

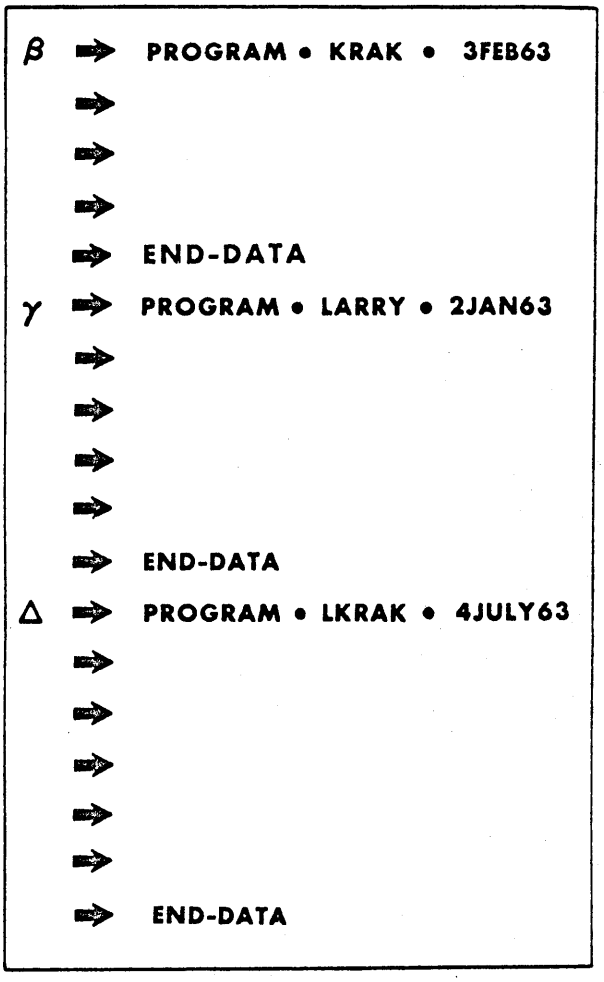
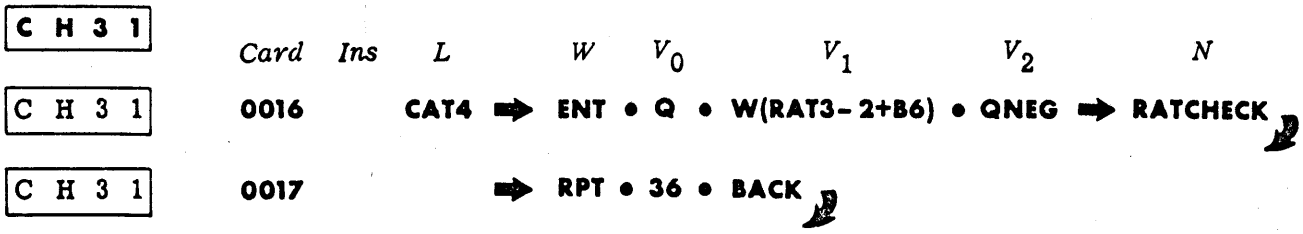


Figure 3. Multiple Program Input

CODING FORMAT: The following examples illustrate the basic coding format for card input with typical operations:

DECK ID.



The programer should notice that identical deck identifiers are punched with each operation of a given program, as indicated in the above rectangles. This identifier is therefore specified only once per coding sheet. It is essential to use a straight arrow before each operator even when a label is not used. The second straight arrow is used only when notes are given. The point separates components of the statement. Parenthesis symbols indicate contents of a storage location modified by an operand code. Also within parenthesis symbols are data unit subnames and subscripts or multiplication factors. Spaces are permitted throughout the operation. The curved arrow indicates the end of the operation, or of notes if present.

Figure 4 is a typical coded program for card input preparation.

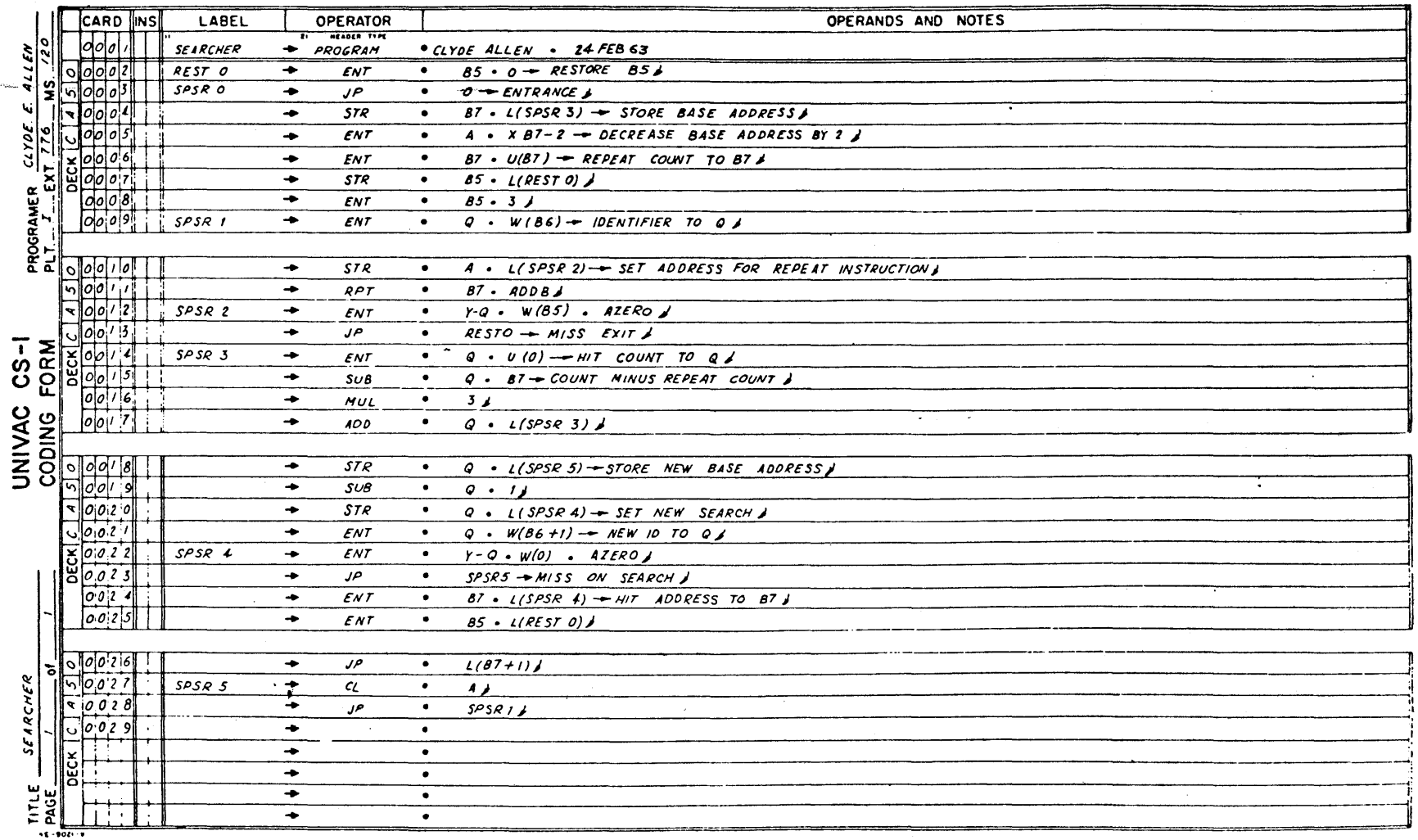


Figure 4. Typical Coded Program for Card Input Preparation

CARD FORMAT: The card format utilizes the punch columns as follows: 1-4, deck identifier; 5-8, card number; 9-10, insert card no.; 11-20, label; 21-79, statement and notes; and 80, overflow card character (0V). The statement and notes start at column 11 for the overflow cards (see Figure 5).

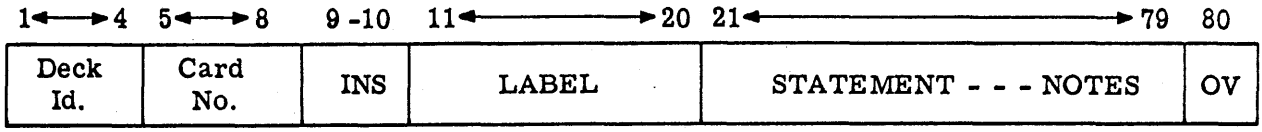


Figure 5. Card Format

The DECK ID. and CARD NO. each require four alphanumeric characters. The keypunch operator duplicates a unique DECK ID on each card. CARD NO.'s are sequential throughout the deck with the exception of insert and overflow cards; numbering starts with 0001. The INS (insert) number provides for insertions to the deck. An insertion card has the same CARD NO. as its preceding card and contains a non-zero two-digit insert number. In all other cases the INS number is blank. A triple dash (- - -) always follows the statement, with or without notes (see Figure 6).

Example:

DECK ID.	CARD NO.	INS
CH34	0092	. . . . .
CH34	0093	. . . . .
CH34	0093	01 . (first insert card) . . . . .
CH34	0093	02 . (second insert card). . . .
CH34	0094	. . . . .

Operations containing more than 59 characters in their combined Statement and Notes area require the use of an overflow card(s); two overflow cards are permitted. A "1" in column 80 indicates an overflow condition; a blank eightieth column indicates no overflow. An overflow card contains the same card number as its preceding card and a 2 or 3 placed in the eightieth column. The keypunch operator handles overflow conditions. The programer need not determine whether or not his statement will fit on one card.

Example:

DECK ID	CARD NO.		OV
CH44	0119	. . . . .	(blank)
CH44	0120	. . . (data overflows this card)	1
CH44	0120	. . . (first card receiving overflow). .	2
CH44	0120	. . . (second card receiving overflow.	3
CH44	0121	. . . . .	(blank)

The column-skip feature on the keypunch provides a convenient means to bypass unused columns reserved for the Label. The keypunch operator begins a label with column 11 and skips any unused columns between the end of the label and column 21. If no label is present, the operator hits the SKIP key and the card is automatically positioned at column 21.

The statement begins at column 21 on the first card only and at column 11 on all overflow cards. A triple dash separates the Statement from the Notes. The REL (release) key ends each operation.

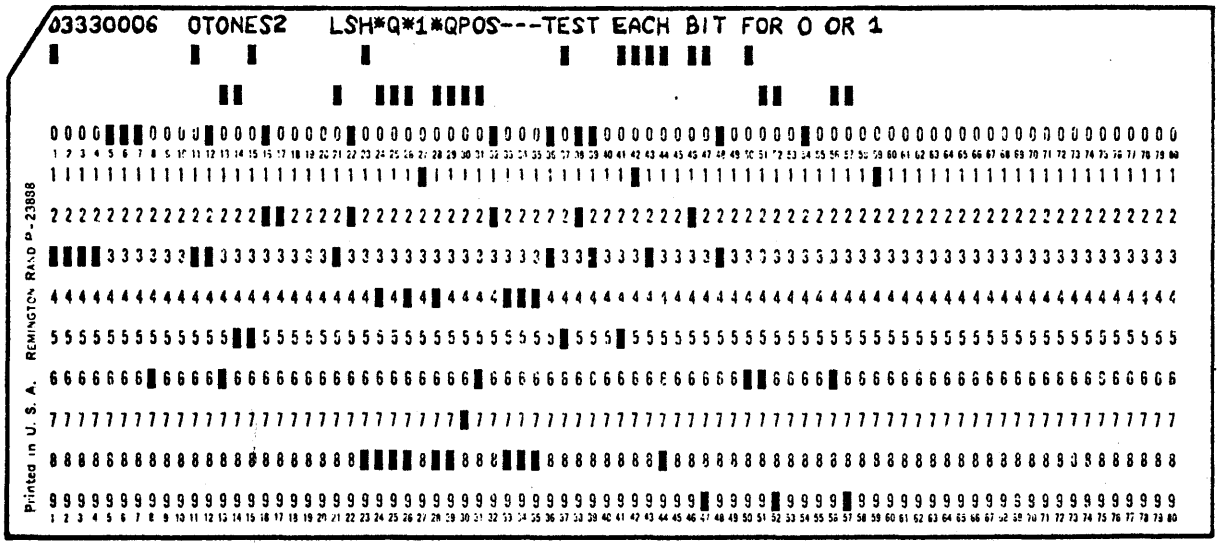


Figure 6. Typical CS-1 Operation on 80 Column Card

The following illustrates a High-Speed Printer listing of a typical CS-1 card deck:

```
C3330001    COUNTONES PROGRAM*SMITH*100CT63
C3330002    CTONESO   CL*B2     SET WORD INDEX
C3330003    CTONES1   ENT*B1*35     SET SHIFT INDEX
C3330004              CL*A   SET SUM TO ZERO
C3330005              ENT*Q*W%WORDO&B2:
C3330006    CTONES2   LSH*Q*1*QPOS      TEST EACH BIT FOR 0 OR 1
C3330007              ADD*A*1     INCREASE SUM IF 1 FOUND
C3330008              BJP*B1*CTONES2
C3330009              STR*A*W%SUM0&B2:     SUM STORAGE
C3330010              BSK*B2*NWORDS
C3330011              JP*CTONES1*STOP5    CONTINUE COMPUTING SUMS
C3330012    CTONES3   JP*CTONES3*STOP    END
C3330013              END-PROG
C3330014              END-DATA
```