THE *National*® 315

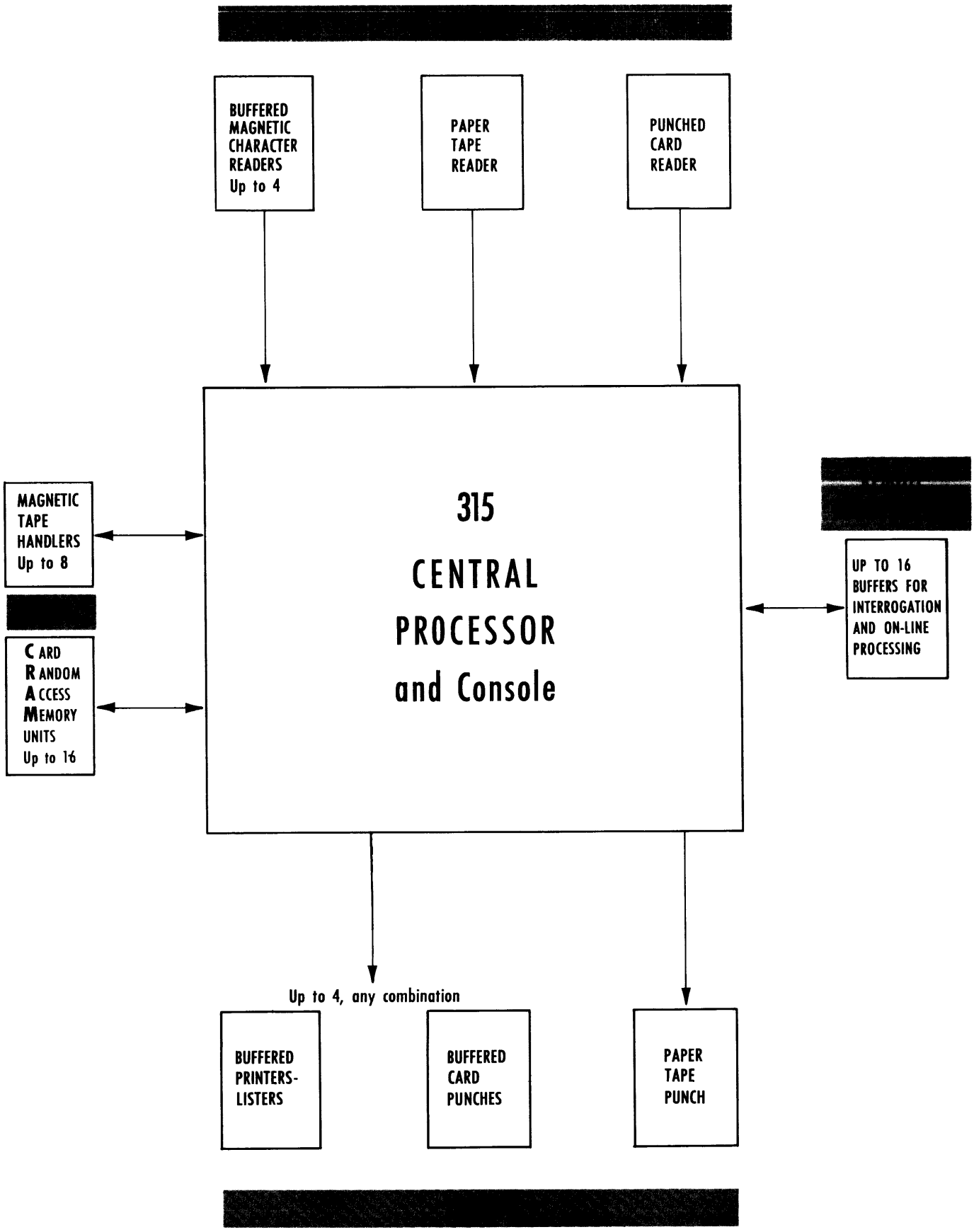
ELECTRONIC DATA PROCESSING SYSTEM

INTERNAL OPERATIONS

B

THE NATIONAL CASH REGISTER COMPANY • DAYTON 9, OHIO

PRINTED IN U.S.A. F-7403 10-61 The NCR Co.


BUFFERED MAGNETIC CHARACTER READERS Up to 4
PAPER TAPE READER
PUNCHED CARD READER
MAGNETIC TAPE HANDLERS Up to 8
CARD RANDOM ACCESS MEMORY UNITS Up to 16
315
CENTRAL PROCESSOR and Console
UP TO 16 BUFFERS FOR INTERROGATION AND ON-LINE PROCESSING
Up to 4, any combination
BUFFERED PRINTERS-LISTERS
BUFFERED CARD PUNCHES
PAPER TAPE PUNCH

CONTENTS

TABLE OF CONTENTS

*These instructions permit a "literal" to be named as "A" if desired.

* These instructions permit a "literal" to be named as "A" if desired.
** These instructions *require* a "literal" to be named as "A".

MEMORY, STORAGE OF INFORMATION:

A fundamental characteristic of any Electronic Data Processor is its internal information-storage, or *memory,* in which it is able to store both that part of the data which is being operated on at the moment, and also the program for processing that data. National's 315 Data Processor is available with memories of 2 000, 5 000, 10 000, 15 000, 20 000 or 40 000 permanently-numbered storage *locations,* which contain stored information. The number assigned to each location is its *address.* The range of addresses is:

MEMORY SIZE	ADDRESSES
2 000	00 000 thru 01 999
5 000	00 000 thru 04 999
10 000	00 000 thru 09 999
15 000	00 000 thru 14 999
20 000	00 000 thru 19 999
40 000	00 000 thru 39 999

Memory references are *cyclic modulo memory-size.* That is, if an address is used which is beyond the memory, the memory-size is automatically subtracted from this address again and again, until a new address is obtained which is within the memory. However, with 15 000-slab memory, 5 000 is subtracted and then, if necessary, 15 000.

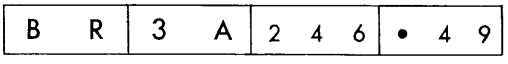
Information is stored in memory by means of magnetic cores, which are tiny rings of ferrite material, strung on a lattice of wires. Each core may be selectively magnetized in either of two states which, for convenience, are designated **0** and **1**. These symbols are not numbers; they are merely convenient marks used to distinguish the two states of a single magnetic core, and any other pair of conventional symbols would serve as well. The marks **0** and **1**, corresponding to the two possible magnetized states of a core, are called *bits* and therefore a single core may store either a **0-bit** or a **1-bit.**

Information may be either numeric or alpha-numeric. Numeric information (spoken of as "Digits") comprises the 10 decimal digits and the six symbols (the *non-decimal digits*) shown in the first row of the Language Code Table. Alphanumeric information (spoken of as "Alphas") comprises the entire set of 64 characters shown in the four rows of the table.

A Digit is represented by a combination of four bits, stored in four magnetic cores, whereas an Alpha is represented by a combination of six bits, stored in six magnetic cores. Of these six bits, the right-hand four are called *numeric bits* and the left-hand two are called *zone bits.*

It will be evident that the sixteen characters which appear in the first row of the table may be represented within the processor memory as either 4-bit Digits or 6-bit Alphas, and in practice they are stored in both forms at different times. All input-output communication with paper tape, punched cards, and printer is performed in terms of Alphas; all arithmetic operations are performed in terms of Digits; at other times, the convenience of the programmer will determine the form in which numeric information is stored. Special operations are included in the processor to condense and expand information from one form to the other.

The information stored in a single memory location is called a *slab,* and consists of 12 bits. Since these 12 bits may be divided into two groups of 6, or into three groups of 4, a slab may store either two Alphas or three Digits:



The term *slab* is a contraction of "syllable": part of a word.

LANGUAGE TABLE

ZONE BITS	NUMERIC BITS 0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0	1	2	3	4	5	6	7	8	9	@	,	SPACE	&	.	–
01	¢	A	B	C	D	E	F	G	H	I	□	△	m	n	o	p
10	+	J	K	L	M	N	O	P	Q	R	%	£	$	(	)	/
11	*	#	S	T	U	V	W	X	Y	Z	d	s	u	v	w	x

The basic unit of information for processing is the *word,* which contains a single item of information, such as Account Number, Name, Gross Pay Year-to-Date, Quantity on Hand, etc. A word may be up to 8 slabs long, and will usually contain all Digit, or all Alpha, information.

The algebraic sign of a Digit-word is determined by its extreme LH (left-hand) digit. If the LH digit is the character *hyphen,* then the word is negative; if the LH digit is anything else, then the word is positive. Thus the number +7968 would be stored in a 2-slab word as:

0 0 7	9 6 8

and in a 3-slab word as:

0 0 0	0 0 7	9 6 8

whereas the number -7968 would be stored in a 2-slab word as:

- 0 7	9 6 8

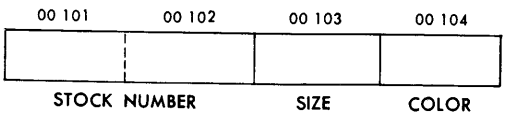
and in a 3-slab word as:

- 0 0	0 0 7	9 6 8

Therefore the longest negative number that can be stored in a given word is one digit shorter than the longest positive number that can be stored in the same word.

A word is referred to by naming the address of its LH slab, and by specifying its length (1 to 8 slabs). There are no markers within the information to designate the beginning and end of a word, and therefore the programmer may, at his convenience, regard a sequence of slabs sometimes as a single word, and at other times as several words.

Suppose the following three words are in memory, describing an item in the inventory file:



The programmer may, if he chooses, compare the 4-slab word starting at location 00 101 with a similar 4-slab word elsewhere in memory (containing the same information about an item received) to see if they are the same. Then, for some succeeding operation, he may again regard this information as comprising three different words.

ACCUMULATOR:

In addition to the numbered locations of memory, the processor contains an 8-slab storage called the Accumulator, and referred to as @. It is implicitly involved in almost every operation performed by the processor, although it is never named explicitly. The capacity of the Accumulator is 16 Alphas, or 24 Digits. Since the sign of the Accumulator is held in the Sign flag (described later) rather than in the Accumulator itself, the intermediate results of a computation may range up to 24 digits, positive or negative. However, the final result which is to be stored in memory may not exceed 24 digits positive, or 23 digits negative, unless double-precision techniques are used.

Consider the problem of adding the contents of two Digit-words, and storing their sum in a third word. Suppose the initial contents of the three words, and of the Accumulator are:

02 344	02 345	02 346	02 347
8 4 2	4 7 7	5 2 3	9 4 6

16 999	17 000	17 001	17 002	17 003
3 2 8	0 0 5	0 0 0	0 0 1	7 3 1

12 210	12 211	12 212	12 213	12 214
7 2 6	3 6 8	3 2 0	8 3 6	4 3 8

2	4 0 8	4 6 9	2 5 0	8 6 2	4 3 6	7 9 9	@

- First LOAD the Accumulator with the contents of the 2-slab word starting at location 02 345. The Accumulator now contains:

0	0 0 0	0 0 0	0 0 0	0 0 0	4 7 7	5 2 3	@

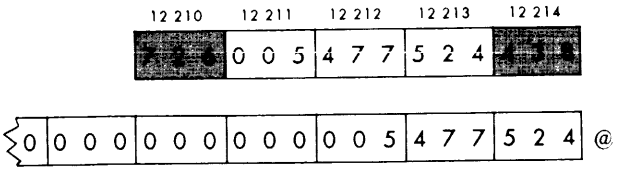
- Then ADD to the Accumulator the contents of the 3-slab word starting at location 17 000. The Accumulator now contains:

0	0 0 0	0 0 0	0 0 0	0 0 5	4 7 7	5 2 4	@

- Now STORE the contents of the Accumulator in the 3-slab word starting at location 12 211. The three words and the Accumulator now contain:

02 344	02 345	02 346	02 347
8 4 2	4 7 7	5 2 3	9 4 6

16 999	17 000	17 001	17 002	17 003
3 2 8	0 0 5	0 0 0	0 0 1	7 3 1



This illustrates several important characteristics of the processor:

- The processor selects the desired words within memory, without being concerned about the information on either side of a word.
- "Reading" or copying information from a word, or from any register, does not alter the information in that word or register.
- Placing information into a word or a register completely replaces the information previously there, and the previous information is then lost.
- All transfers of information within the processor are *right-justified*. That is, the right-hand end of the information-source is always lined up with the right-hand end of the information-destination.
- If the destination is longer than the source, the destination is always filled out to the left with zeros.

INDEX REGISTERS (R-registers):

An instruction of the type just illustrated names an **Operation**, an **Address** in memory, and a **Word Length**. The instruction also names one of 32 **Index Registers** which are always used by the processor in executing any instruction. An index register holds an address—a positive number up to 39 999.

In order to determine the actual address in memory to which the instruction refers, the processor automatically adds the address named in the instruction, plus the address stored in the index register.

Suppose that index registers 16, 17, 18 contained:

16	0 2 0 0 5
17	1 6 9 0 0
18	1 2 2 0 0

Then the three instructions just illustrated could have been written:

LOAD 2 slabs from (R16) 340. [from 02 345]
ADD 3 slabs from (R17) 100. [from 17 000]
STORE 3 slabs in (R18) 011. [in 12 211]

ADDRESSING METHODS:

An instruction names 3 digits in the address column, specifying the *position* of a word within an item of data. The index register names the *base* of the item—the address of the first slab of the item.

Suppose a series of transactions in memory which are to be posted by a Commercial Bank to its checking-account file. Each transaction comprises four words, occupying nine slabs of memory, and the first transaction starts in location 04 996:

	Address	L	Pos.	D/A	0	1	2	3	4	5	6	7	Remarks
1st TRANSACTION	04 996	3	0	D									Account number
	04 999	4	3	D									Amount of trans.
	05 003	1	7	D									Transaction code
	05 004	1	8	D									Batch number
2nd TRANSACTION	05 005	3	0	D									Account number
	05 008	4	3	D									Amount of trans.
	05 012	1	7	D									Transaction code
	05 013	1	8	D									Batch number
3rd TRANSACTION	05 014	3	0	D									Account number
	05 017	4	3	D									Amount of trans.
	05 021	1	7	D									Transaction code
	05 022	1	8	D									Batch number
4th TRANS.	05 023	3	0	D									Account number
	05 026	4	3	D									[illegible] trans.

In the program for posting these transactions, every instruction referring to Account Number will contain **000** as its address reference; every instruction referring to Amount will contain **003**; every instruction referring to Transaction Code will contain **007**; etc.

Before starting to post, an index register will be preset to contain **04 996**. Every instruction will then refer to the appropriate word in the first transaction. When that transaction is completely posted, the index register will be augmented by **9**, and will then contain **05 005** (the address of the first slab of the second transaction) whereupon each instruction will refer to the appropriate word in the second transaction. And so on.

In general, one Index Register will be used to control each data stream.

JUMP REGISTERS (J-registers):

A total of 32 jump registers are also provided in the processor. Each of these holds an address—a positive number up to 39 999.

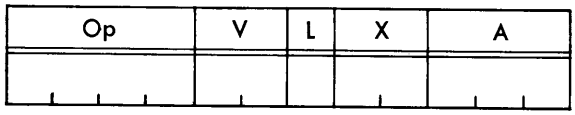
Many instructions provide for alternate *exits,* depending on conditions encountered while the instruction is being executed. If such a condition is found, then the processor will not proceed to the next instruction in sequence when the instruction is completed, but will instead *jump* to an instruction whose address is stored in one of the J-registers.

Such an instruction will name a J-register as being the first register in the *jump table* for that instruction. The jump table contains as many J-registers as there are possible exits from that particular instruction. Suppose, for example, that some instruction has three possible exits, corresponding to conditions "A", "B", "C"; and in writing the instruction the programmer specifies J23 as the beginning of the jump table. Then if the instruction finds condition "A", the processor will jump to the address stored in J23 after this instruction is complete; if condition "B", it will jump to the address stored in J24; if condition "C", it will jump to the address stored in J25. If none of these conditions exist, the processor will execute the next instruction in the normal program sequence.

Certain of the R-registers and J-registers perform special functions, and are not normally used in the fashion just described. A detailed discussion of the characteristics of those registers is given later.

SINGLE STAGE INSTRUCTIONS:

A single stage instruction is written in the following format:

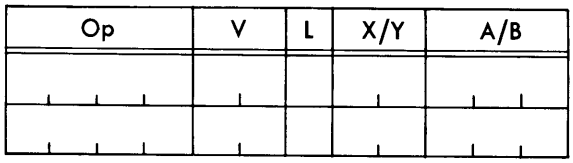


Op and **V** name the operation to be performed, and the variation if any. **L** indicates the length of the word (up to 8 slabs), **X** the index register to be used, and **A** the address reference. The processor obtains the actual address by adding **A** to the contents of the index register.

When this instruction is actually stored in the processor memory, as part of a program to be executed, it occupies two slabs of memory. The first slab contains **Op, V, L** and **X**, all condensed into 12 bits; the second slab contains **A**.

DOUBLE STAGE INSTRUCTIONS:

A double stage instruction is written in the following format:



Note that the instruction requires two lines, and that the **L** column is not used. **A** is the address reference; **B** specifies other information as required by each instruction. **X** *usually* specifies an R-register, and **Y** *usually* specifies a J-register, but not invariably. Therefore in the description of each instruction, an R-register will be called **RX** if it is specified by **X**, and **RY** if it is specified by **Y**; a J-register will be called **JX** if it is specified by **X**, and **JY** if it is specified by **Y**.

When this instruction is stored in memory, it occupies four slabs. **Op, V, X** and **Y** are condensed into the 24 bits of the first and third slabs; the second and fourth slabs contain **A** and **B** respectively.

FLAGS:

A number of *flags* are provided in the processor. These may be turned on and off by the program at will, and perform certain automatic functions, as well as storing conditions which the program may test at a later time. The terms *on/off, set/cleared* are used interchangeably to describe the two states of a flag.

SIGN FLAG:

This flag is associated with the Accumulator; it automatically indicates the algebraic sign of the Accumulator contents, and governs all arithmetic operations accordingly. However, for special purposes, it may be set *positive* or *negative* by the program, independently of the Accumulator contents.

Testing the Sign flag does not change its setting.

OVERFLOW FLAG:

Whenever an attempt is made to store more information into a word than that word can hold, putaway stops at the LH (left hand) end of the word, and the remaining information is not stored. When this occurs, or when certain other conditions arise, the processor usually sets the Overflow flag. The precise circumstances under which each operation might set Overflow are described specifically for the individual operations.

The Overflow flag may also be set for special purposes by the program, independently of other operations.

Once Overflow has been set, the flag remains set until the program tests it, at which time it is automatically cleared.

GREATER, LESS, EQUAL FLAGS:

These flags are set automatically, to indicate the result of a COMPARE or a COUNT instruction, and also to record detailed information about the execution of a SUPPRESS or a SCAN instruction.

The **G**, **L** and **E** flags are not independent; only one of them may be on at a time. However, they may all be off at the same time.

Testing these flags does not change their setting.

MEMORY FLAGS:

It is often convenient for the programmer to designate flags of his own, to record information for later use. For example, it may take many tests to determine if an individual employee should receive overtime pay, and this *yes* or *no* answer may be required at two or more widely separated points in the payroll program. Rather than repeat the entire series of tests each time this answer is needed, the program obtains the answer the first time, and records it in a memory flag. Thereafter it is only necessary to test this flag each subsequent time the answer is needed.

Any slab in memory may be designated as containing a pair of flags, corresponding to the two Alphas which the slab can store. These then become the LH (left hand) and RH (right hand) flags in that slab. If a flag contains the Alpha character *zero* it is **off**, if it contains any other Alpha character it is **on**. These flags may be set, cleared, and tested independently of each other. Since any slab of memory may contain a pair of flags, the number of such flags available to the programmer is practically unlimited.

Testing a memory flag does not change its setting.

DEMAND PERMIT FLAG:

Many of the peripheral units have the ability to interrupt the main program (*to demand* processor attention) when they have completed an operation previously assigned to them. In this fashion, the relatively slow input-output units may be kept running at maximum rate, while the processor is performing some other job; occasionally the main program will be interrupted for a brief interval to attend to one of the input-output units, and then immediately resume, while the slow unit continues to operate independently at its own speed.

The programmer will wish to permit Demand interruption during certain portions of his program, and to forbid it during other portions. The Demand Permit flag gives him this facility.

While the Demand Permit flag is on, any peripheral unit whose Unit Demand flag (see below) is on, will exercise Demand whenever it is ready to receive information from the processor, or to deliver information to the processor.

When Demand is exercised, the processor always completes the current instruction in the main program, then jumps to the demand program. The programmer may specify different demand programs for different portions of the main program, if he wishes. There is a single entry-point to any demand program, regardless of which among several possible peripheral units may have interrupted, and no indication is furnished to show which unit actually did exercise Demand. This gives the programmer complete flexibility in assigning priorities among competitively demanding units. The demand program merely attempts to SELECT each unit in turn, until it finds one which is in the *ready* state, and then gives attention to that unit. The assigning of priorities among units is performed in the simplest

possible fashion—by specifying the sequence in which the units are tested.

Entering a demand program turns off the Demand Permit flag, since normally it is not desirable to have the demand program itself interrupted. However, the programmer may permit this if he chooses, merely by having the demand program turn the Demand Permit flag back on. Just before this, he will probably have specified a different demand program to be used for the time being.

The instructions TEST:D, TEST:T, SETF:D, SETF:T are *protected* against Demand; the processor never permits interrupt after completion of one of these four instructions.

The program may set and test the Demand Permit flag. It is turned off automatically either when tested, or when Demand interrupt occurs.

UNIT DEMAND FLAGS:

The following peripheral units are capable of exercising Demand: Printers, Card Readers, Card Punches, CRAMs (Card Random Access Memories), Magnetic Character Readers, Inquiry Stations. Each of these units has within it a Unit Demand flag, which may be set and cleared by the program.

The programmer will often wish to permit some of the peripheral units, but to forbid others, to exercise Demand during a particular part of the program. He then has the program turn the Unit Demand flags in the permitted units *on*, and those in the forbidden units *off*. Any device whose Unit Demand is off may still be used by the processor at any time; but it must await the program's convenience, rather than being able to demand attention at its own convenience.

TRACER PERMIT FLAG:

To facilitate code-checking, it is customary to use supplemental *tracing* or *automonitoring* programs which permit the operator to follow the execution of the program being checked. In order to provide communication between the main program (the one being checked) and the tracing program, a Tracer interrupt facility is provided, controlled by the Tracer Permit flag.

When this flag is on, *and* the appropriate Console switch is also on, then at the conclusion of the execution of each instruction in the main program the processor automatically jumps to the tracing program.

The instructions TEST:D, TEST:T, SETF:D, SETF:T, DLR are *protected* against tracing; the processor never permits Tracer interrupt after completion of one of these five instructions.

The program may set and test the Tracer Permit flag. It is turned off automatically either when tested, or when Tracer interrupt occurs.

REGISTERS:

The 32 R-registers and 32 J-registers may be thought of as existing in the following array, which indicates the special functions assigned to some of the registers:

RELATIVE ADDRESS (INDEX) R—REGISTERS		JUMP J—REGISTERS	
00	16	00	16
01	17	01	17
02	18	02	18
03	19	03	19
04	20	04	20
05	21	05	21
06	22	06	22
07	23	07	23
08	24	08	24
09	25	09	25
10	26	10	26
11	27	11	27
12	28 Registers 28 and 29 are used by STEP, PACE, macros and subroutines. Their contents are not preserved.	12	28 Registers 28 and 29 are used by macros and subroutines. Their contents are not preserved.
13	29	13	29
14	30 Processor stores an Address here	14 Jump-Table Link	30 Demand-Program Jump Address
15 Main Link (Program Decision)	31 Sequence-Control Register	15 Demand-Program Link	31 Tracer-Program Jump Address

The numbers 00 000 through 09 000 are always stored in these Registers

The numbers 10 000 through 19 000 are always stored in these Registers in Processors with 20,000 slab memory

STEP uses Registers 00 through 05

PACE uses Registers 06 through 11

MICR Sorter-Reader uses Registers 16 through 21

Inquiry System uses Registers 22 through 24

USING THE REGISTERS:

LOADING THE REGISTERS:

An address is loaded into a register from a *memory pair*—a 2-slab word of memory. In this loading operation only the RH 18 bits (4½ digits) of the pair are placed into the register.

This is equivalent to saying that, if an attempt is made to load a negative number into a register, the negative sign is ignored, and the number is loaded as positive. If an attempt is made to load a number greater than 39 999, the processor automatically subtracts 40 000 from that number again and again until the result is less than 40 000.

STORING THE REGISTERS:

An address is stored from a register into the RH 18 bits of a memory pair. The LH 6 bits (1½ digits) of the pair are automatically set to zero.

ADDING AND SUBTRACTING IN THE REGISTERS:

Any addition and subtraction performed in the registers is modulo 40 000.

This is equivalent to saying that, if a number is added to the contents of a register and the sum is greater than 39 999, the processor automatically subtracts 40 000 again and again until the result is less than 40 000. If a number is subtracted from the contents of a register, and the result is negative, the processor automatically adds 40 000 again and again until the result is positive (or zero).

NOTE: The contents of a register is always a positive number from 00 000 thru 39 999, since each of the registers contains 18 bit-positions. In loading a register, only the RH 18 bits of the memory pair are loaded, with the LH 6 bits ignored. In storing a register, the RH 18 bits of the pair are stored, and the LH 6 bits set to zero. In adding and subtracting in a register, the augmenter is in a memory pair, and only the RH 18 bits of the augmenter are used; *except* that if the LH 4 bits of the augmenter form the Digit *hyphen* then the augmenter is treated as negative. The result of the addition or subtraction is stored in the register modulo 40 000 as a positive number.

However, once the processor is given an address to *lookup* in memory, that address is interpreted *modulo memory-size* in memories of less than 40 000 slabs, except for the special rule for 15 000-slab memory, stated on page 4.

SPECIAL FUNCTIONS OF SOME REGISTERS:

In order to obtain cross-references within the program, the first 10 index registers (**R00** thru **R09**) always contain the addresses 00 000, 01 000, ... 09 000 respectively. Thus a jump to the instruction in location 06 785 would be written:

Op	V	L	X	A
J U M P			0 6	7 8 5

R30. When the processor performs an operation whose scope is variable, it automatically stores the terminating address in R30. Therefore R30 will not normally be used by the programmer for routine address-modification. Magnetic Tape and Input instructions, as well as SCAN, store information in R30.

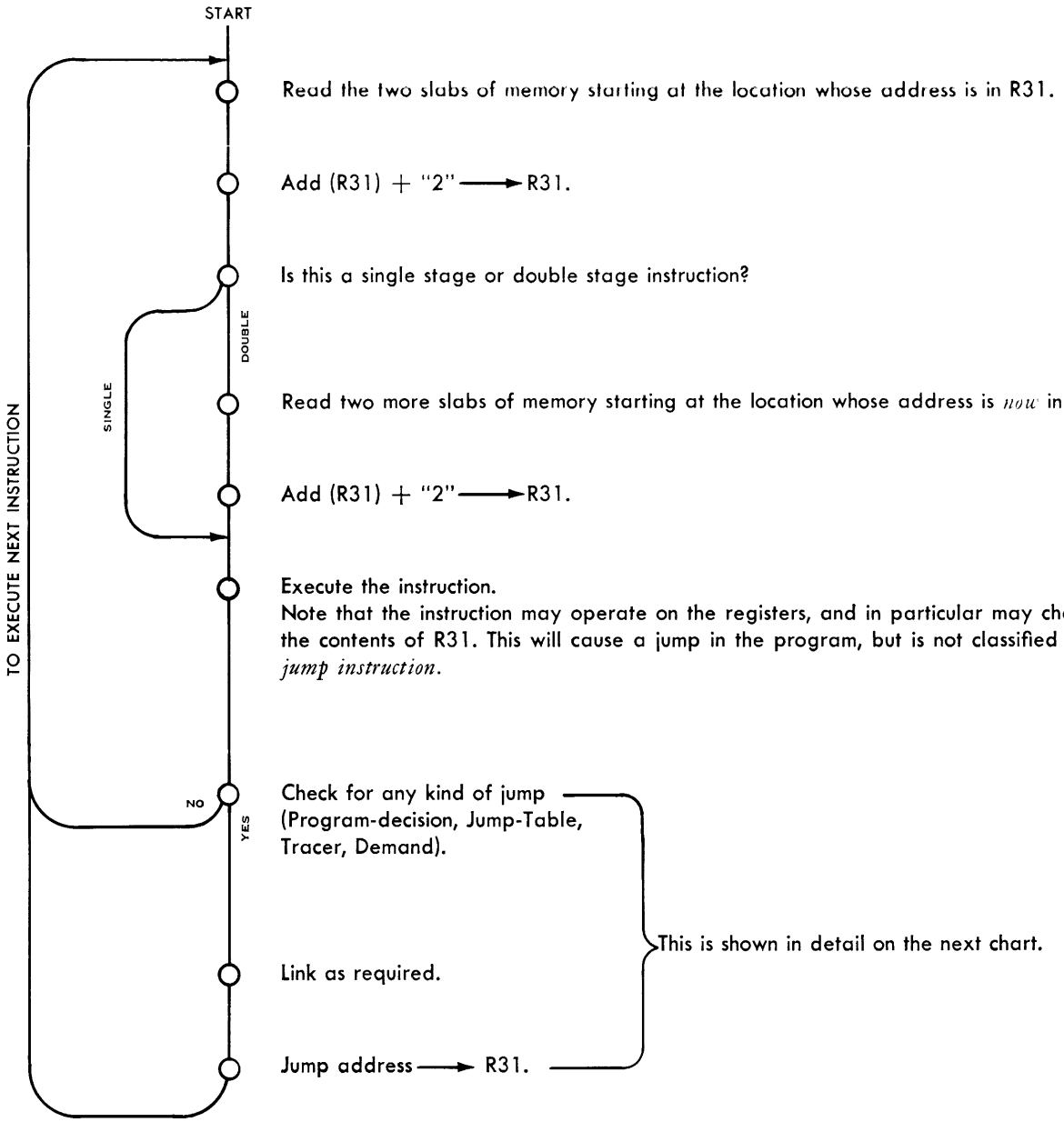
R15 and **R31** may never be used for address-modification.

R31 is used by the processor as its *Sequence-Control Register;* each time a new instruction is to be executed, the processor finds the address of that instruction in R31, and immediately replaces that address with the address of the next instruction in the normal sequence. If an instruction requires a *branch* or jump out of the normal program sequence, the processor saves the new contents of R31 as a *link* back to the normal sequence, and then plants the jump address into R31.

R15 is used as the Link Register for program-decision jumps, and is called the *main* link. When the jump is the result of a program decision (such as TEST FLAG or Unconditional Jump) the contents of R31 are saved in R15 before the jump address is placed into R31. There are only three branching instructions (JUMP:IP; TEST:D; TEST:T) which do not link.

The programmer may at any time impose a jump on the program (without the use of a branching instruction) by changing or replacing the contents of R31. Since these are *operations* on one of the registers, they are not classified as jumps, and do not link. Similarly, the programmer is free to change or replace the contents of any of the link registers.

This process can best be illustrated by a flow chart which shows the detailed steps performed by the processor in executing an instruction. The notation (*R31*) means *the contents of R31.*



J30 and **J31** hold the Demand program and Tracer program jump addresses, respectively. The programmer stores the appropriate addresses in these registers at the beginning of the program, and of course he has the privilege of changing them at any time during the program if he sees fit to do so.

When either Demand interrupt or Tracer interrupt occurs, the contents of R31 are preserved in J15; then the contents of J30 or J31, as appropriate, are planted into R31, imposing a jump to the Demand or to the Tracer program.

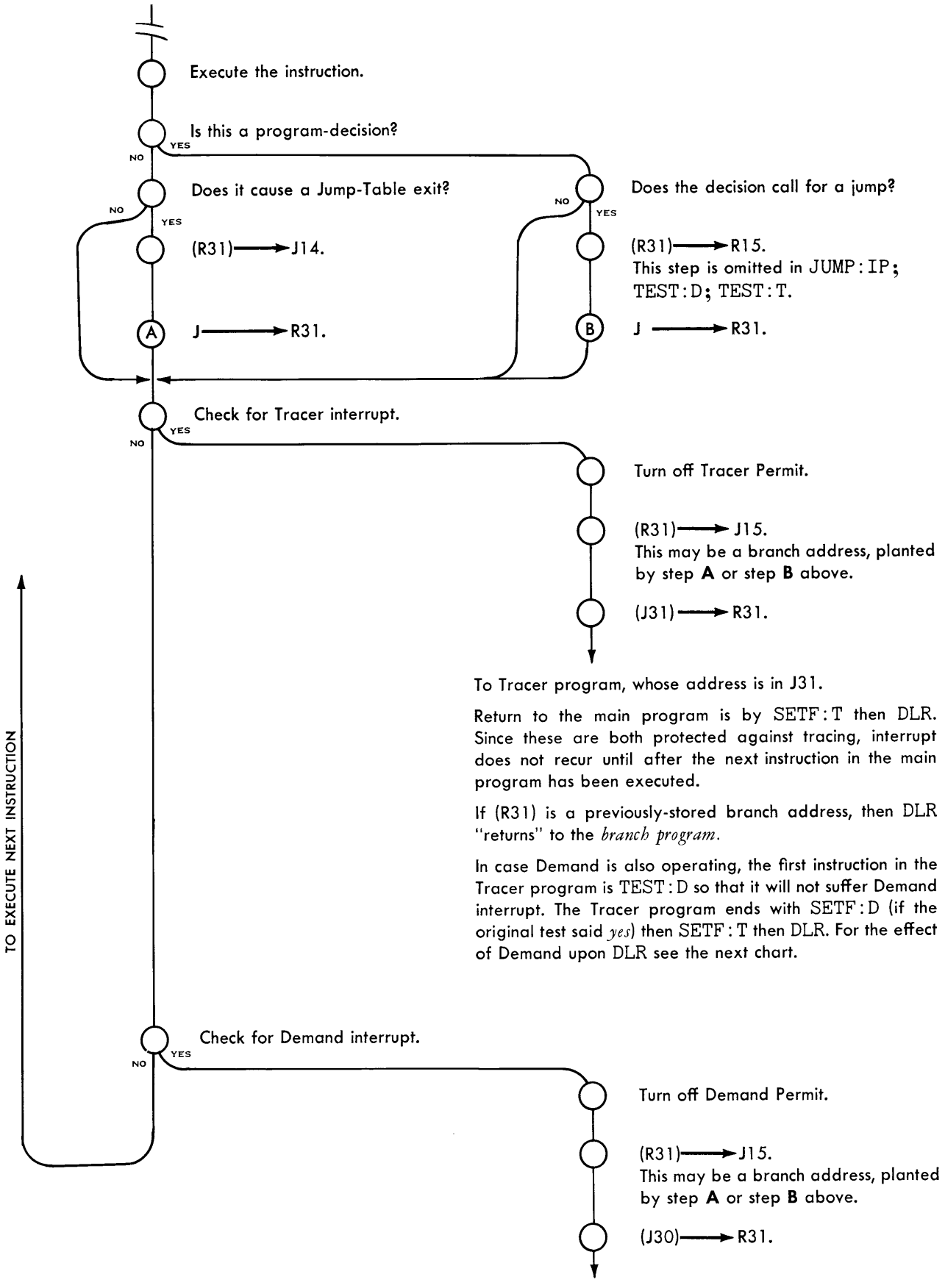
J15 holds the link—the address of the next instruction in the interrupted program—for either Demand or Tracer interrupt.

J14 holds the link if any instruction takes a jump table exit. The contents of R31 are saved in J14; then the contents of the designated one of the J-registers are planted into R31.

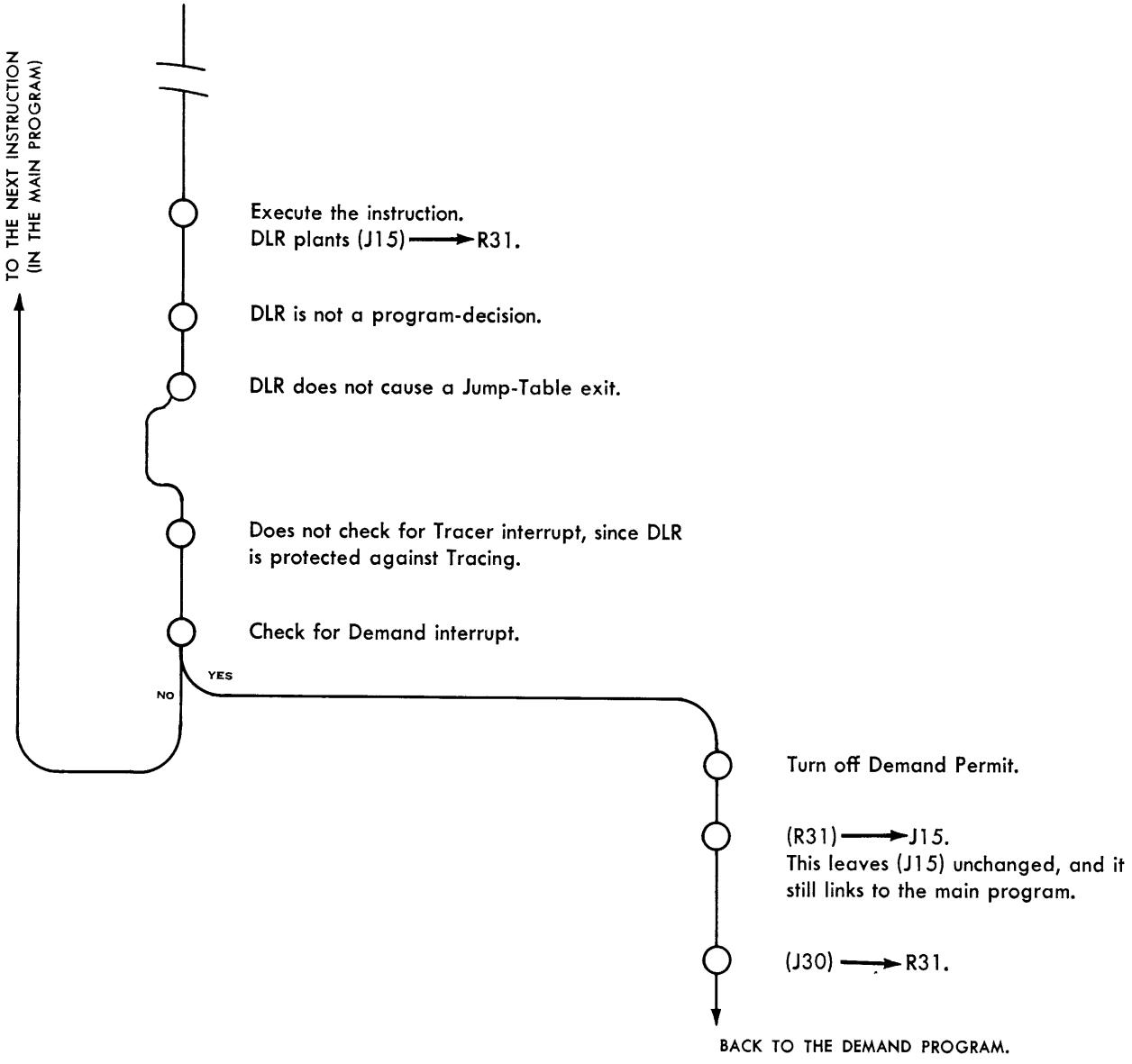
It occasionally happens that the programmer does not care whether an exit condition occurs or not during execution of an instruction. He can then suppress the exit by naming J14 as the jump register for that instruction. The sequence of events within the processor after detection of the exit condition then is:

- Plant the contents of R31 (the address of the next instruction in sequence) into J14.
- Plant the contents of the designated J-register (which is J14, and which now has the same contents as R31) into R31.
- The contents of R31 thus remain unchanged, and the processor takes the next instruction in sequence.

All these processes are illustrated on the following flow chart (an expansion of one section of the previous chart) which also shows the result if two, or all three, of these jumps are required simultaneously.


Execute the instruction.
Is this a program-decision?
YES
NO
Does it cause a Jump-Table exit?
NO
YES
(R31)→J14.
A
J→R31.
Does the decision call for a jump?
NO
YES
(R31)→R15.
This step is omitted in JUMP:IP; TEST:D; TEST:T.
B
J→R31.
Check for Tracer interrupt.
YES
NO
Turn off Tracer Permit.
(R31)→J15.
This may be a branch address, planted by step A or step B above.
(J31)→R31.
To Tracer program, whose address is in J31.
Return to the main program is by SETF:T then DLR. Since these are both protected against tracing, interrupt does not recur until after the next instruction in the main program has been executed.
If (R31) is a previously-stored branch address, then DLR "returns" to the branch program.
In case Demand is also operating, the first instruction in the Tracer program is TEST:D so that it will not suffer Demand interrupt. The Tracer program ends with SETF:D (if the original test said yes) then SETF:T then DLR. For the effect of Demand upon DLR see the next chart.
Check for Demand interrupt.
YES
NO
Turn off Demand Permit.
(R31)→J15.
This may be a branch address, planted by step A or step B above.
(J30)→R31.
To Demand program, whose address is in J30.
Return to the main program (or branch program) is by SETF:D then DLR. For the effect of a new Demand upon DLR see the next chart.
TO EXECUTE NEXT INSTRUCTION

It is not immediately obvious that a Demand interrupt occurring at the conclusion of a DLR instruction will work correctly. It will, however, because DLR is an *operation* on the contents of R31 (thus imposing a jump) but is not a *jump instruction.* The following flow chart of part of the DLR instruction shows how it works:



NAMING OF LITERALS:

Certain of the instructions permit naming index registers R15 and R31 *as though* they were to be used for address-modification, but when this is done, the processor accepts it as an indication to perform a special function.

Whenever R15 or R31 is named in the **X** column (in those instructions which permit it) then the processor interprets the contents of the **A** column not as the address of the data, but as the data itself. The **L** column is then not used.

Thus suppose it is desired to multiply the Accumulator contents by 17, and then to add 125 to the result. The instructions would be:

Op	V	L	X	A
M U L T			1 5	1 7
A D D			1 5	1 2 5

Whenever it is desired to name a literal, rather than an address, in one of the instructions which permit it, the programmer puts an A or a D into the **X** column, and this will cause the instruction to be stored in the program with a reference to R15. D specifies a Digit literal, and A specifies an Alpha literal. Thus:

Op	V	L	X	A
L D			D	7 5

will load Digits "075" (binary 000001110101)

Op	V	L	X	A
L D			A	7 5

will load Alphas "75" (binary 000111000101)

Those instructions, and variations, which permit naming a literal in this fashion are marked throughout this manual with an *.

THE "EFFECTIVE LENGTH" OF THE ACCUMULATOR:

Although the actual length of the Accumulator is, and always remains, 8 slabs, the concept of its "effective length" is a useful one when discussing overflow conditions, and in the formulas for the execution times of the instructions.

The effective length of the Accumulator is simply the number of slabs, counting from the RH end, which contain all the non-zero information in the Accumulator. However, the effective length is never zero, so a cleared Accumulator has an effective length of one slab.

ACCUMULATOR CONTENTS	EFFECTIVE LENGTH
0 0 0 | 0 0 0 | 0 0 1 | 0 0 0 | 0 0 0 | 0 0 0 | 3 1 5 | 0 0 0	6
A B | C D | E F | G H | I J | K M | N ∅ | P Q	8
0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 SP | N C | R SP	3
0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0	1

STUDENT'S NOTES

DEFINITIONS

LH	Left-hand.
RH	Right-hand.
@	The Accumulator.
(@)	The contents of the Accumulator.
Op:V	Operation and variation.
A	The A-address named in an instruction.
L	The length of the word referred to in an instruction.
A-word	The word referred to by the address A. The address of the LH slab of the word is obtained by adding A + contents of an index register. The length of the word is L slabs.
RX	The R-register specified by X.
RY	The R-register specified by Y.
JX	The J-register specified by X.
JY	The J-register specified by Y.
G	An augmenter named in an instruction. May range from —99 thru 999.
Alpha	A 6-bit character.
Digit	A 4-bit character.
slab	The information stored in a single memory location. A slab comprises 2 Alphas or 3 Digits. The term "slab" is a contraction of "syllable": part of a word.
word	A single unit of information, consisting of 1 to 8 slabs. A few operations permit longer words than this.
pair	A 2-slab word containing an address or an augmenter in its RH 18 bit-positions. If the pair contains an augmenter, then it may be positive or negative.
N	The number of times a sub-unit of an operation is to be performed. (ie- Load N registers; Move N slabs; etc.)
J	Jump address named in an instruction.
*	Indicates an instruction which permits naming of a "literal" if desired.
**	Indicates an instruction which requires naming of a "literal".

* LOAD Accumulator

Op	V	L	X	A
L D		L	X	A

(**A**) replaces (@)

or "**A**" replaces (@)

This operation may be performed on either Digit or Alpha information.

The A-word is transcribed into the Accumulator, right-justified, and the Accumulator is filled out with zeros to the left.

SIGN FLAG: The processor does not "know" whether the programmer regards the A-word as made up of Digits or of Alphas. Therefore, if the left-hand four bits of the A-word are all 1-bits, the processor assumes a negative digit-word, sets the Sign flag negative, and replaces those four 1-bits in the Accumulator with four 0-bits.

Otherwise the Sign flag will be set positive.

OVERFLOW: Cannot occur.

* STORE Accumulator

Op	V	L	X	A
S T		L	X	A

(@) replaces (**A**)

or (@) replaces "**A**"

This operation may be performed on either Digit or Alpha information.

The Accumulator is transcribed into the A-word right-justified. If the A-word is shorter than the effective length of the Accumulator, overflow occurs.

SIGN FLAG: If the Sign flag was previously set negative, and if the LH digit of the stored A-word is *zero,* then that digit is replaced by *hyphen.*

If the LH digit of the stored A-word is not *zero,* then there is no room for the negative sign, and overflow will occur.

In case of data overflow, the LH digit of the stored A-word might accidentally be *zero;* in this case a negative sign is not stored.

Setting of the Sign flag remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW:

1. The A-word is shorter than the effective length of the Accumulator.

2. Sign flag is negative, and there is not room to store the sign in the A-word, even though there is room for all the digits.

* ADD to Accumulator

Op	V	L	X	A
A D D		L	X	A

(@) + (**A**) replaces (@)
or (@) + "**A**" replaces (@)

This operation may be performed only on Digit information.

If the LH digit of the A-word is *hyphen* then the word is considered negative and that digit is added as though it were a zero; otherwise the word is considered positive. The setting of the Sign flag designates the algebraic sign of the Accumulator.

The addition is performed according to the algebraic law of signs, and the sign of the result causes a new setting of the Sign flag.

NOTE: If any ADD operation yields a result of "zero", then the Sign flag remains unchanged.

SIGN FLAG: Designates the sign of the Accumulator;

Then is set by the sign of the result.

ACCUMULATOR: Holds the result of the operation.

OVERFLOW: If the result contains more than 24 digits.

The Accumulator will then hold the RH 24 digits of the result, and the Sign flag will be set correctly.

* SUBTRACT from Accumulator

Op	V	L	X	A
S U B		L	X	A

(@) − (**A**) replaces (@)
or (@) − "**A**" replaces (@)

This operation may be performed only on Digit information.

If the LH digit of the A-word is *hyphen* then the word is considered negative and that digit is subtracted as though it were a zero; otherwise the word is considered positive. The setting of the Sign flag designates the algebraic sign of the Accumulator.

The Subtraction is performed according to the algebraic law of signs, and the sign of the result causes a new setting of the Sign flag.

NOTE: If any SUB operation yields a result of "zero", then the Sign flag remains unchanged.

SIGN FLAG: Designates the sign of the Accumulator;

Then is set by the sign of the result.

ACCUMULATOR: Holds the result of the operation.

OVERFLOW: If the result contains more than 24 digits.

The Accumulator will then hold the RH 24 digits of the result, and the Sign flag will be set correctly.

* ADD to Memory

Op	V	L	X	A
A D D	M	L	X	A

(@) + (**A**) replaces (**A**)
or (@) + "**A**" replaces "**A**"

This operation may be performed only on Digit information.

If the LH digit of the A-word is *hyphen* then the word is considered negative and that digit is added as though it were zero; otherwise the word is considered positive. The setting of the Sign flag designates the algebraic sign of the Accumulator.

The addition is performed according to the algebraic law of signs, and the sign of the result is stored in the A-word with the result itself. However, the Sign flag is not changed by the result.

NOTE: If this operation yields a result of *zero* the original sign of the A-word remains unchanged.

SIGN FLAG: Designates the sign of the Accumulator;

Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW:
1. If the result contains more significant digits than the A-word can hold.
2. If the sign of the result is negative, and there is no room (see STORE) for the sign in the A-word. The sign is not stored.

* COMPARE

Op	V	L	X	A
C O M P		L	X	A

(@) is compared with (**A**) and G, L, E flag is
or (@) is compared with "**A**" set accordingly.

This operation may be performed on either Digit or Alpha information.

The G-flag is set if (@) is Greater;
The L-flag is set if (@) is Less;
The E-flag is set if (@) is Equal.

With Digit information:

If the LH digit of the A-word is *hyphen* then the word is considered negative and that digit is compared as though it were a zero; otherwise the word is considered positive. The setting of the Sign flag designates the algebraic sign of the Accumulator.

The comparison is performed according to the algebraic law of signs, whereby any negative number is smaller than any positive number; and of two negative numbers, the larger magnitude is the smaller number.

If any non-decimal digits are present, the result can be predicted by giving the digits their binary values.

With Alpha information:

Since the operation does not distinguish between Digit and Alpha information (the actual comparison is bit-by-bit) it is essential that the Sign flag be set positive before this operation is performed. In dealing with Alpha information, this will usually be the case anyway. However, note that if the LH character of the A-word is u, v, w, x then the A-word will be considered negative.

NOTE: If the two numbers being compared are "positive zero" and "negative zero" then this operation sets the E-flag.

SIGN FLAG: Designates the sign of the Accumulator.

Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

* DIVIDE Accumulator

Op	V	L	X	A
D I V		L	X	A

$\frac{(@)}{(\mathbf{A})}$ replaces LH (@)
or $\frac{(@)}{\text{"}\mathbf{A}\text{"}}$ replaces LH (@)
} Remainder replaces RH (@)

This operation may be performed only on Digit information.

The quotient appears, right-justified, in the LH 4 slabs of the Accumulator. The remainder appears, right-justified, in the RH 4 slabs of the Accumulator.

If the LH digit of the A-word is *hyphen* then the word is considered negative and the digit is treated as though it were a zero; otherwise the word is considered positive. The setting of the Sign flag designates the algebraic sign of the Accumulator.

The division is performed according to the algebraic law of signs, and the sign of the result causes a new setting of the Sign flag.

No sign is explicitly associated with the remainder; it is understood that the remainder has the same sign as the dividend (*initial* setting of the Sign flag).

After storing the result of the division, the quotient and remainder should thereafter be treated as separate words. The remainder word will appear to be positive, and the programmer must keep track of what its sign ought to be.

If for any reason the remainder alone is to be stored, observe the comments under STORE, with regard to overflow and storage of the Sign flag. Particular care must be taken at this point if there is a possibility that the quotient may be either zero or minus-zero.

SIGN FLAG: Designates the sign of the Accumulator;

Then is set by the sign of the Quotient.

ACCUMULATOR: Holds the dividend (numerator); Then receives the result of the operation.

OVERFLOW: No division is performed, and overflow occurs, in the following cases:

1. The A-word contains zero,
2. The A-word is more than 4 slabs long.
3. The effective length of the Accumulator is more than 4 slabs, *unless* the A-word contains a number of greater magnitude than the number in the LH 4 slabs of the actual Accumulator. This is the criterion for obtaining a quotient more than 4 slabs long, which is forbidden.

* MULTIPLY ACCUMULATOR

Op	V	L	X	A
M U L T		L	X	A

(@) × (**A**) replaces (@)
or (@) × "**A**" replaces (@)

This operation may be performed only on Digit information.

If the LH digit of the A-word is *hyphen* then the word is considered negative and that digit is multiplied as though it were a zero; otherwise the word is considered positive. The setting of the Sign flag designates the algebraic sign of the Accumulator.

The multiplication is performed according to the algebraic law of signs, and the sign of the result causes a new setting of the Sign flag.

SIGN FLAG: Designates the sign of the Accumulator;

Then is set by the sign of the result.

ACCUMULATOR: Holds the result of the operation.

OVERFLOW: If the sum of:
Length of the A-word and
Effective length of the Accumulator.
is more than 8 slabs.

No multiplication will be performed, and the contents of the Accumulator will be unchanged.

* BINARY ADD to Accumulator

Op	V	L	X	A
B A D D		L	X	A

(@) + (**A**) replaces (@) Addition is Mod-64 with no carry between Alpha positions.

or (@) + "**A**" replaces (@) Addition is normal, with full carries.

SIGN FLAG: Ignored. Both operands considered positive.

Remains unchanged.

ACCUMULATOR: Holds result of the operation.

OVERFLOW: When adding (**A**), overflow cannot occur.

When adding "**A**", overflow will occur if the result exceeds the 96-bit capacity of the Accumulator, but this is unlikely.

ADDITION TABLE
FOR BINARY ADD

	0	1
0	0	1
1	1	c0

"c" means carry

EDIT

Op	V	L	X	A
E D I T		L	X	A

Edit (**A**) into @ according to format-control previously in @.

This operation may be performed only on Alpha information.

The characters of the A-word are transcribed, one by one, from right to left, into the Accumulator, replacing certain of the characters previously in the Accumulator.

In order to do this, the characters in the Accumulator are scanned from right to left, until either *m* or *comma* is found.

If *m* Replace the *m* by the next (initially the rightmost) character of the A-word, and continue to scan the Accumulator.

If *comma* If the next character of the A-word is anything other than *asterisk* or *space* then leave the *comma* unchanged and continue to scan the Accumulator.

If the next character of the A-word is either *asterisk* or *space* then replace the *comma* with that character, but do not advance to the next character of the A-word.

The operation terminates when the A-word is exhausted.

SIGN FLAG: Remains unchanged.

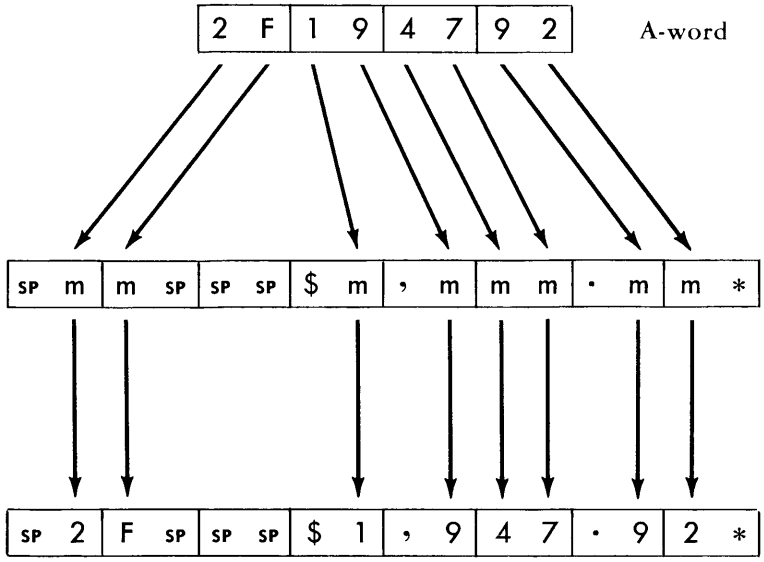
ACCUMULATOR: Holds the format-control pattern;

Then holds the edited A-word.

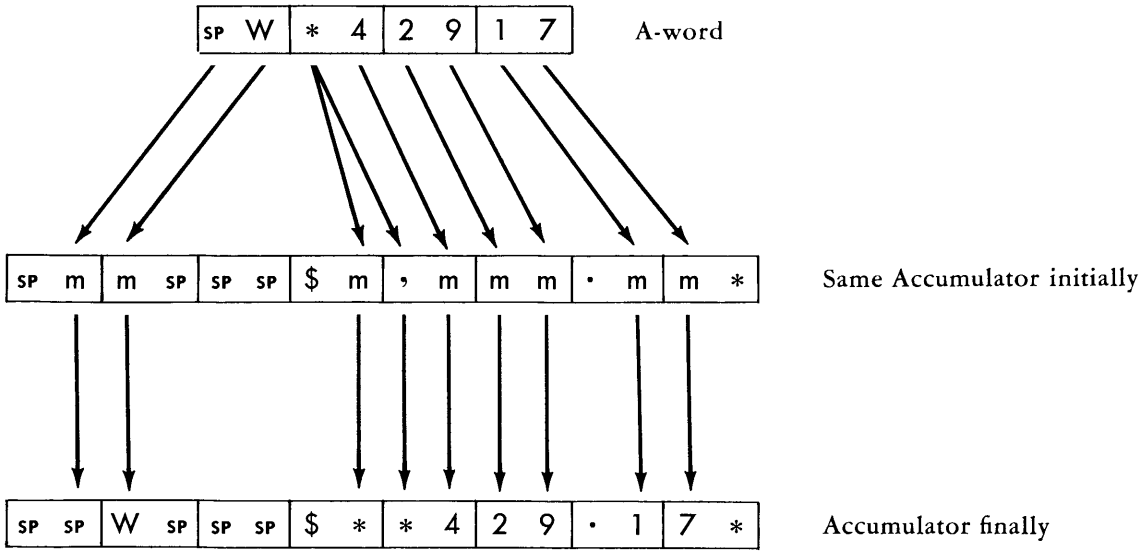
OVERFLOW: Cannot occur.

ERROR HALT: If the Accumulator is completely scanned, and some characters of the A-word have not been transcribed.

Example 1:



Example 2:



SUPPRESS

Op	V	L	X	A
SUPP		L	X	A

Leading zeros in the A-word are replaced with spaces.

This operation may be performed only on Alpha information.

The A-word is scanned from left to right. If the first character is a *zero* it is replaced by a *space* and the scanning proceeds to the next character; and so on.

The operation terminates when either:

1. The scanning process encounters any character other than *zero.*
2. The A-word has been exhausted, and now contains all spaces.

After the operation terminates:

The Accumulator contains the number of slabs *in which* suppression has occurred.

If an odd number of zeros were suppressed, the G-flag is turned on.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Holds tally of number of slabs now containing spaces.

G-FLAG: ON if odd number of spaces. OFF if even number of spaces.

L-FLAG; E-FLAG: Always turned OFF.

OVERFLOW: Cannot occur.

SET and CLEAR Processor Flags

FORMAT A: All but Memory Flags

Op	V	L	X	A
SETF	(V)			

FORMAT B: Memory Flags

Op	V	L	X	A
SETF	(V)		X	A

Op	V	L	X	A
CLRF	(V)		X	A

SETF:+ Set Sign flag plus.

SETF:− Set Sign flag minus.

SETF:Ø Set Overflow flag on.

Protected

SETF:D Set Demand Permit flag on.

SETF:T Set Tracer Permit flag on.

*SETF:LH Set LH Memory flag on, in 1-slab A-word.

*SETF:RH Set RH Memory flag on, in 1-slab A-word.

The character *space* is inserted

*CLRF:LH Set LH Memory flag off, in 1-slab A-word.

*CLRF:RH Set RH Memory flag off, in 1-slab A-word.

The character *zero* is inserted

SIGN FLAG: Remains unchanged, except by SETF:+ and SETF:−.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur, except as result of SETF:Ø

TEST (single-stage)

Op	V	L	X	A
T E S T	(V)		X	J

Each of these may cause a jump to the instruction whose address is: "J" + (contents of RX)

TEST:G Jump if G-flag is on. Set Link in R15

TEST:L Jump if L-flag is on. Set Link in R15

TEST:E Jump if E-flag is on. Set Link in R15

TEST:– Jump if Sign flag negative. Set Link in R15

(Flag not disturbed)

TEST:O Jump if Overflow flag is on. Set Link in R15

Protected

TEST:D Jump if Demand Permit flag is on. Does not link.

TEST:T Jump if Tracer Permit flag is on. Does not link.

(Flag turned off)

The programmer will use TEST:D and TEST:T, with jumps suppressed, to turn off the Demand and Tracer flags. Aside from that, these instructions are used only for housekeeping purposes in the "canned" portions of the Tracer and Demand programs.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

NOTE: When testing the G, L, E flags after SCAN, it is more convenient to use the following alternative mnemonics.

TEST:SL Did SCAN stop on left digit or alpha? Same command as TEST:E

TEST:SM Did SCAN stop on middle digit? Same command as TEST:L

TEST:SR Did SCAN stop on right digit or alpha? Same command as TEST:G

* *TEST (double-stage)*

FORMAT A:
Test Memory flags.
Jump to address named in instruction.
Link in R15.

Op	V	L	X/Y	A/B
T E S T	(V)		X	A
			JUMP ADDRESS (ALL 5 DIGITS)	

FORMAT B:
Test Console Switches.
Jump to address "J" + (contents of RY).
Link in R15.

Op	V	L	X/Y	A/B
T E S T	S W			A
			Y	J

*TEST:LH Jump if LH flag in 1-slab A-word is on (not equal to zero).

*TEST:RH Jump if RH flag in 1-slab A-word is on (not equal to zero).

*TEST:SW Jump if Console Option Switch number (**A**) or "**A**" is on. Switch number may be 000-007.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

LH, RH FLAGS: Remain unchanged.

SWITCHES: Remain unchanged. They are set by the console operator.

JUMP
JUMP INDIRECT
JUMP INDIRECT, keep PREVIOUS Link

Op	V	L	X	A
J U M P			X	J

Op	V	L	X	A
J U M P	I		X	A

Op	V	L	X	A
J U M P	I P		X	A

JUMP: Unconditional jump to address "J" + (contents of RX). Link in R15. (R31) replaces (R15). Then Jump address replaces (R31).

JUMP:I Unconditional jump to address stored in 2-slab A-word. Link in R15. (R31) replaces (R15). Then A-word replaces (R31).

JUMP:IP Unconditional jump to address stored in 2-slab A-word. **Does not link.** A-word replaces (R31).

SIGN FLAG Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW Cannot occur.

SKIP within the program

Op	V	L	X	A
S K I P				G

This instruction causes an unconditional jump to a point up to 999 slabs after, or up to 99 slabs before, the next instruction in the normal sequence.

(R31) + "**G**" replaces (R31).
G may range to 999 or to −99.

This is an *operation* upon (R31), and is not a *jump instruction.* Therefore it does not set any link.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

DEMAND LINK RETURN

Op	V	L	X	A
D L R				

This instruction causes a return to the Demand Link.

(J15) replaces (R31).

This is an *operation* upon (R31), and is not a *jump instruction.* Therefore it does not set any link.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

NOTE: This instruction is protected against Tracing. It is not protected against Demand.

MAIN LINK RETURN, and AUGMENT Control Register

Op	V	L	X	A
M L R A				G

This instruction causes an unconditional jump back to the Main Link, with the option of skipping instead to a point up to 999 slabs after the link, or up to 99 slabs before the link.

(R15) + "**G**" replaces (R31).
G may range to 999 or −99.

This is an *operation* upon (R31), and is not a *jump instruction.* Therefore it does not set any link.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

* SHIFT Accumulator

Op	V	L	X	A
S H F T	(V)		X	A

This operation may be performed on either Digit or Alpha information, as specified by the Variation.

The contents of the Accumulator are shifted N places. N is equal to: The contents of the A-word (always one slab long); or "**A**" itself.

Except for SHFT:LC and SHFT:RC all shifts are linear, ie- "off the end".

*SHFT:DL Shift digits left,
enter zeros at the right.

*SHFT:DR Shift digits right,
enter zeros at the left.

*SHFT:RR Shift digits right and roundoff,
enter zeros at the left.

Before the shift, 5 is lined up with the Nth character from the right, and added to the Accumulator contents.

*SHFT:LC Shift digits left circular.

*SHFT:RC Shift digits right circular.

The two circular shifts operate within the *effective length* of the Accumulator. They leave the effective length unchanged even though the circulation may bring zeros into the leftmost positions.

*SHFT:AL Shift alphas left,
enter spaces at the right.

If, before the shift, there were zeros at the RH end of the Accumulator, these are shifted unchanged just like any other characters; they *are not replaced* by spaces.

*SHFT:AR Shift alphas right,
enter zeros at the left.

NOTE: During an Alpha shift, all zeros within the previous effective length of the Accumulator remain significant to the new effective length.

Before a left or right Alpha shift of an odd number of positions, a slab containing **0 SP** is inserted in the Accumulator, just to the left of the previous effective length.

After an Alpha Right shift of more than the previous effective length, the Accumulator will contain the single slab **SP SP**.

SIGN FLAG: Remains unchanged.

OVERFLOW: Cannot occur.

If N = 0, these operations do nothing.

* COUNT

Op	V	L	X/Y	A/B
C N T			X	A
			Y	G

This instruction performs two distinct operations.

1. First ADD: (RY) + "**G**"
   replaces (RY)
   G ranges to 999 or −99.
2. Then COMPARE: (RY) vs (**A**) completely
   or (RY) vs "**A**" mod 1000
   and set G, L, E flag.

If the LH digit of "**G**" is the character *hyphen* then "**G**" is considered negative; otherwise "**G**" is considered positive. The contents of an Index Register are always positive.

If an A-word in Memory is named, it is always a "pair": a 2-slab word containing a number no greater than 39 999.

If "**A**" itself is used for the comparison, then only the RH 3 digits of the Index Register are used in the comparison.

SIGN FLAG: Remains unchanged.

G-FLAG: ON if (RY) is greater.
L-FLAG: ON if (RY) is less.
E-FLAG: ON if (RY) is equal.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

NOTE: Addition is always performed modulo 40 000 regardless of the memory size of the Processor.

LOAD Registers
SPREAD-LOAD Registers

Op	V	L	X/Y	A/B
L D	(V)		X	A
			Y	N

Op	V	L	X/Y	A/B
S L D	(V)		X	A
			Y	N

LD :R Transcribe N successive Memory pairs (A), (A+2), etc. into N successive R-registers starting with RY.

LD :J Transcribe N successive Memory pairs (A), (A+2), etc. into N successive J-registers starting with JY.

SLD :R Transcribe one Memory pair (A) into each of N successive R-registers starting with RY.

SLD :J Transcribe one Memory pair (A) into each of N successive J-registers starting with JY.

If any of these operations remains incomplete after referring to R31 or J31, there will be an error halt.

If any of these operations carries past the end of memory, it will cycle back to location 00 000 and continue.

NOTE: Only the RH 18 bits (4½ digits) of each pair are loaded. The LH 6 bits (1½ digits) are irrelevant.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

If N = 0, these operations do nothing.

STORE Registers

Op	V	L	X/Y	A/B
S T	(V)		X	A
			Y	N

ST:R Transcribe N successive R-registers starting with RY into N successive Memory pairs (A), (A+2,) etc.

ST:J Transcribe N successive J-registers starting with JY into N successive Memory pairs (A), (A+2), etc.

If either of these operations remains incomplete after referring to R31 or J31, there will be an error halt.

If either of these operations carries past the end of memory, it will cycle back to location 00 000 and continue.

NOTE: The contents of a register are stored as the RH 18 bits (4½ digits) of the Memory pair. The LH 6 bits of the pair are set to zero.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

If N = 0, these operations do nothing.

AUGMENT Registers
SPREAD-AUGMENT Registers

Op	V	L	X/Y	A/B
A U G	(V)		X	A
			Y	N

Op	V	L	X/Y	A/B
S A U G	(V)		X	A
			Y	N

AUG :R The contents of each of N successive R-registers starting with RY is augmented by the contents of the corresponding one of N successive Memory pairs (A), (A+2), etc.

AUG :J The contents of each of N successive J-registers starting with JY is augmented by the contents of the corresponding one of N successive Memory pairs (A), (A+2), etc.

SAUG:R The contents of each of N successive R-registers starting with RY is augmented by the contents of Memory pair (A).

SAUG:J The contents of each of N successive J-registers starting with JY is augmented by the contents of Memory pair (A).

Only the RH bits (4½ digits) of a Memory pair are used in augmenting a register.

If the LH digit of any Memory pair is *hyphen*, then the contents of that pair is a negative number and the contents of the register are diminished. The contents of the register are always positive.

Addition is performed modulo 40 000 regardless of memory size.

If any of these operations remains incomplete after referring to R31 or J31, there will be an error halt.

If any of these operations carries past the end of memory, it will cycle back to location 00 000 and continue.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

If N = 0, these operations do nothing.

Examples

REGISTERS INITIALLY

R13	0 0 0 4 7
R14	2 7 6 3 5
R15	3 9 9 9 9
R16	1 5 0 0 0
R17	0 4 2 9 6
R18	0 2 5 0 0

MEMORY PAIRS

01711	0 0 0 1 0 0
01713	0 0 0 2 0 0
01715	− 0 0 0 0 2
01717	0 3 0 0 0 0
01719	0 0 0 0 0 1
01721	− 7 0 0 0 0

RESULT OF

AUG	R		01	711
			13	006

R13	0 0 1 4 7
R14	2 7 8 3 5
R15	3 9 9 9 7
R16	0 5 0 0 0
R17	0 4 2 9 7
R18	1 2 5 0 0

RESULT OF

SAUG	R		01	711
			13	006

R13	0 0 1 4 7
R14	2 7 7 3 5
R15	0 0 0 9 9
R16	1 5 1 0 0
R17	0 4 3 9 6
R18	0 2 6 0 0

MOVE information between Registers

Op	V	L	X/Y	A/B
M Θ V E	(V)		X	
			Y	N

MΘVE:RR Transcribe N successive R-registers starting with RY into N successive R-registers starting with RX.

MΘVE:JR Transcribe N successive J-registers starting with JY into N successive R-registers starting with RX.

MΘVE:RJ Transcribe N successive R-registers starting with RY into N successive J-registers starting with JX.

MΘVE:JJ Transcribe N successive J-registers starting with JR into N successive J-registers starting with JX.

If any of these operations remains incomplete after referring to R31 or J31, there will be an error halt.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: Remains unchanged.

OVERFLOW: Cannot occur.

If N = 0, these operations do nothing.

MOVE memory } { *Start at Beginning* } up to 999 slabs
** *SPREAD memory* } { *Start at End* }

Op	V	L	X/Y	A/B
M Θ V E	(V)		X	A
			Y	B

Y specifies Index Register RY, modifying address B.

Op	V	L	X/Y	A/B
S P R D	(V)			A
			Y	B

Y specifies Index Register RY, modifying address B.

These operations may by performed on either Digit or Alpha information.

MOVE transcribes N (up to 999) consecutive slabs from an A-area to a B-area in Memory.

N is in the RH slab of the Accumulator.

MOVE:B Start at the beginning of each area:
(A) replaces (B)
then (A+1) replaces (B+1) etc.

MOVE:E Start at the end of each area:
(A) replaces (B)
then (A−1) replaces (B−1) etc.

SPREAD transcribes "A" itself into each slab of an N-slab B-area in Memory.

**SPRD:B "A" replaces (B)
then "A" replaces (B+1) etc.

**SPRD:E "A" replaces (B)
then "A" replaces (B−1) etc.

If any of these operations carries past the end of Memory, it will cycle back to location 00 000 and continue.

SIGN FLAG: Remains unchanged.

ACCUMULATOR: RH slab holds N: the number of slabs to be transcribed (up to 999).

Remains unchanged.

OVERFLOW: Cannot occur.

If N = 0, these operations do nothing.

SCAN

Op	V	L	X/Y	A/B
S C N D	(V)		X	A
			Y	L

L is length of A-word: up to 999
Y specifies Jump Register JY

Op	V	L	X/Y	A/B
S C N A	(V)		X	A
			Y	L

L is length of A-word: up to 999
Y specifies Jump Register JY

SCND: Scan Digits.

SCNA: Scan Alphas.

This operation may be performed on either Digit or Alpha information, as specified by the Variation.

Consider the Accumulator as being of the same length as the A-word, which in this operation may be up to 999 slabs long. Consider that every slab in the Accumulator exactly duplicates the RH slab.

The Accumulator and the A-word are then simultaneously scanned, from left to right, and compared Digit by Digit, or Alpha by Alpha, until one character in the A-word is found which bears the specified relationship to the corresponding character in the Accumulator. The relationship may be specified as Greater than, Less than, or Equal to, the corresponding character in the Accumulator.

The LH character of the Variation specifies whether the SCAN shall seek a character in the A-word which is:

G: greater than
L: less than
E: equal to

the corresponding character in the Accumulator.

The operation terminates when either:

1. A character in the A-word meets the test. R30 then contains the address of the slab in which this character appears.

   Flags are set to indicate the position of this character in that slab.

RH Digit or RH Alpha	G-flag set
Middle Digit	L-flag set
LH Digit or LH Alpha	E-flag set

   The Processor proceeds to the next instruction in normal sequence.

2. No character in the A-word meets the test.

   R30 remains unchanged.

   G, L, E-flags are all turned off.

   The Processor takes its next instruction from the address in JY.

   Link is set in J14.

If L = 0 the Processor immediately takes termination 2.

NOTE: It may be inconvenient to remember the correspondence of G, L, E flags to left, middle, right positions within the slab. Therefore three alternative mnemonics are provided.

TEST:SL Did SCAN stop on left digit or alpha?
Same command as TEST:E

TEST:SM Did SCAN stop on middle digit?
Same command as TEST:L

TEST:SR Did SCAN stop on right digit or alpha?
Same command as TEST:G

SIGN FLAG: Remains unchanged.

ACCUMULATOR: RH slab contains the Scan Key, which is considered to be duplicated in every slab of the Accumulator.

Note that the word "considered" indicates merely a convenient way of describing the operation. In fact, only the RH slab of the Accumulator is significant.

The entire Accumulator remains unchanged by the SCAN operation.

R30: After termination **1**, R30 holds address of slab containing the successful character of the A-word.

After termination **2**, R30 remains unchanged.

G, L, E FLAGS: After termination **1**, these indicate the position of the successful character within its own slab.

After termination **2**, all these flags are off.

JY: After termination **2**, Processor jumps to address stored in JY. If desired, this jump can be suppressed by naming J14 as JY.

OVERFLOW: Cannot occur.

SELECTIVE SCANNING:

As the characters in memory are compared with the characters in the RH slab of the Accumulator, it is not necessary that *all* the Digits or Alphas of each slab be examined, In order to specify which positions are actually to be scanned, the positions in a slab are given "scan-values":

Digits	4 2 1
Alphas	2 1

In the RH character of the variation, the programmer writes the sum of the scan-values of those positions which he wishes the SCAN to examine.

`SCND:G3` means Scan Digits for Greater, examining only the RH and middle digits of each slab (ignoring the LH digit).

`SCND:E6` means Scan Digits for Equal, examining only the LH and middle digits of each slab (ignoring the RH digit).

`SCNA:L2` means Scan Alphas for Less, examining only the LH alpha of each slab (ignoring the RH alpha).

A "blank" as the RH character of the variation means that all characters are significant. Thus:

`SCND:G` means the same as `SCND:G7`

`SCNA:E` means the same as `SCNA:E3`

PARTIAL ALPHA STORE

Op	V	L	X/Y	A/B
P A S T	(V)		X	A
				L

L is length of A-word: up to 8 (or 9)

This operation may be performed only on Alpha information.

The Accumulator is stored, right-justified, in a part of the A-word.

If L = 0, this operation does nothing.

PAST:XL Except the LH character of the A-word.

PAST:XR Except the RH character of the A-word.

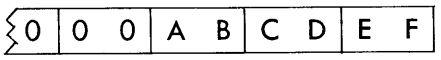
PAST:XB Except both.

SIGN FLAG: Remains unchanged. The Sign flag is not stored in the A-word.

ACCUMULATOR: Remains unchanged.

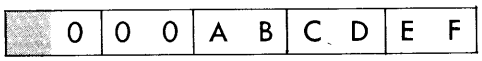
OVERFLOW: XL: L greater than 8; 16 characters stored in A-word.
XR: L greater than 8; 16 characters, plus a zero, stored in A-word.
XB: L greater than 9; 16 characters, plus a zero, stored in A-word.

Examples: Shaded areas are unchanged.



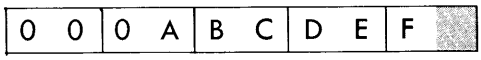
Contents of Accumulator

3-slab A-word after PAST:XL



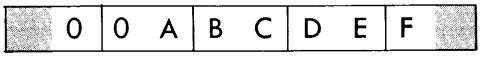
5-slab A-word after PAST:XL

3-slab A-word after PAST:XR



5-slab A-word after PAST:XR

3-slab A-word after PAST:XB



5-slab A-word after PAST:XB

LOAD ALPHA-TO-DIGIT

Op	V	L	X/Y	A/B
L D A D	(V)		X	A
			Y	L

L is length of A-word: up to 12 (or 13)
Y specifies Jump Register JY

This operation may be performed only on Alpha information in Memory, which then becomes Digits in the Accumulator.

The A-word is Loaded into the Accumulator, right-justified, with its Alphas transformed into Digits.

The Alphas of the A-word are transcribed, from right to left, into the Accumulator. As each character is transcribed, it is stripped of its zone bits, and stored in the Accumulator as a 4-bit Digit.

The operation terminates when either:

1. The A-word is exhausted; the Accumulator is filled out to the left with zeros.
2. The Accumulator is filled, and the A-word is not exhausted. Overflow then occurs.

If any of the discarded zone bits of the A-word are 1-bits, the operation still proceeds to completion; then the Processor takes its next instruction from the address in JY, and sets link in J14. Otherwise the Processor proceeds in sequence.

If L = 0, this operation clears the Accumulator.

LDAD: Load and condense the entire A-word.

LDAD:XL Except the LH character of the A-word.

LDAD:XR Except the RH character of the A-word.

LDAD:XB Except both. L may be equal to 13.

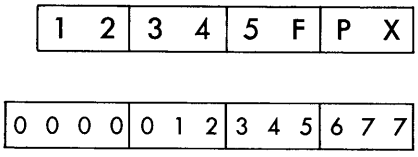
SIGN FLAG: Set positive by this operation.

ACCUMULATOR: Contains the condensed A-word.

JY: Processor jumps to address stored in JY if any 1-bits in zones. If desired, this jump can be suppressed by naming J14 as JY.

OVERFLOW: If the A-word is too long.

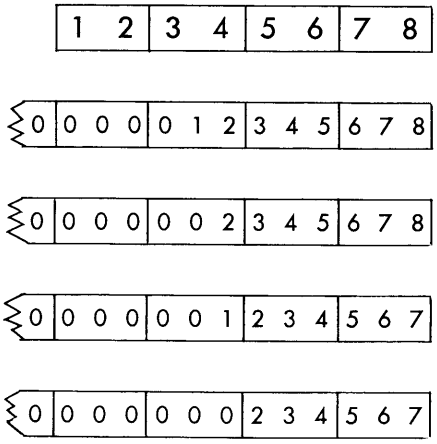
Example 1:



A-word

Accumulator after LDAD
Processor jumps to address in JY.

Example 2:



A-word

Accumulator after LDAD

Accumulator after LDAD:XL

Accumulator after LDAD:XR

Accumulator after LDAD:XB

STORE DIGIT-TO-ALPHA

Op	V	L	X/Y	A/B
S T D A			X	A
				L

L is length of A-word: up to 12

This operation may be performed only on Digit information in the Accumulator, which then becomes Alphas in Memory.

The Accumulator is stored in the A-word, right-justified, with its Digits transformed into Alphas.

The digits of the Accumulator are transcribed, from right to left, into the A-word. As each digit is transcribed, a pair of 0-bit zone bits are attached to it, and it is stored in the A-word as a 6-bit Alpha.

The operation terminates when either:

1. The A-word is filled. If any non-zero digits of the Accumulator cannot fit into the A-word, overflow occurs.
2. The A-word has received 24 characters. If the A-word is more than 12 slabs long, overflow occurs.

If L = 0, this operation does nothing.

SIGN FLAG: Remains unchanged.

The Sign flag is not stored in the A-word.

ACCUMULATOR: Remains unchanged.

OVERFLOW:
1. If the A-word is too small for all the non-zero characters in the Accumulator.
2. If the A-word is more than 12 slabs long.

ADD-SUBTRACT TABLES

ADDITION. Use this table for:

Add like signs
Subtract unlike signs
Add to Memory, like signs
Augment, add positive number
Count, add positive number

SUBTRACTION. Use this table for:

Add unlike signs
Subtract like signs
Add to Memory, unlike signs
but interchange Accumulator and Memory for entry to the table
Augment, add negative number
Count, add negative number

If the Subtraction Table indicates that the result of an operation ends with a borrow (for example **b823**) the result is automatically complemented and appears in the processor as negative (**-177**).

ADDITION

NO CARRY FROM PREVIOUS DIGIT-POSITION

Digits from Memory

Digits from Accumulator or Register	0	1	2	3	4	5	6	7	8	9	@	,	SPACE	&	.	-	F
0	0	1	2	3	4	5	6	7	8	9	@	,	SPACE	&	.	-	
1	1	2	3	4	5	6	7	8	9	c0	,	SPACE	&	.	-	0	
2	2	3	4	5	6	7	8	9	c0	c1	SPACE	&	.	-	0	1	
3	3	4	5	6	7	8	9	c0	c1	c2	&	.	-	0	1	2	
4	4	5	6	7	8	9	c0	c1	c2	c3	.	-	0	1	2	3	
5	5	6	7	8	9	c0	c1	c2	c3	c4	-	0	1	2	3	4	
6	6	7	8	9	c0	c1	c2	c3	c4	c5	0	1	2	3	4	5	
7	7	8	9	c0	c1	c2	c3	c4	c5	c6	1	2	3	4	5	6	
8	8	9	c0	c1	c2	c3	c4	c5	c6	c7	2	3	4	5	6	7	
9	9	c0	c1	c2	c3	c4	c5	c6	c7	c8	3	4	5	6	7	8	
@	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	4	5	6	7	8	9	
,	c1	c2	c3	c4	c5	c6	c7	c8	c9	c@	5	6	7	8	9	c0	
SPACE	c2	c3	c4	c5	c6	c7	c8	c9	c@	c,	6	7	8	9	c0	c1	
&	c3	c4	c5	c6	c7	c8	c9	c@	c,	c SPACE	7	8	9	c0	c1	c2	
.	c4	c5	c6	c7	c8	c9	c@	c,	c SPACE	c&	8	9	c0	c1	c2	c3	
-	c5	c6	c7	c8	c9	c@	c,	c SPACE	c&	c.	9	c0	c1	c2	c3	c4	

G

CARRY FROM PREVIOUS DIGIT-POSITION

Digits from Memory

Digits from Accumulator or Register	0	1	2	3	4	5	6	7	8	9	@	,	SPACE	&	.	-	F
0	1	2	3	4	5	6	7	8	9	c0	,	SPACE	&	.	-	0	
1	2	3	4	5	6	7	8	9	c0	c1	SPACE	&	.	-	0	1	
2	3	4	5	6	7	8	9	c0	c1	c2	&	.	-	0	1	2	
3	4	5	6	7	8	9	c0	c1	c2	c3	.	-	0	1	2	3	
4	5	6	7	8	9	c0	c1	c2	c3	c4	-	0	1	2	3	4	
5	6	7	8	9	c0	c1	c2	c3	c4	c5	0	1	2	3	4	5	
6	7	8	9	c0	c1	c2	c3	c4	c5	c6	1	2	3	4	5	6	
7	8	9	c0	c1	c2	c3	c4	c5	c6	c7	2	3	4	5	6	7	
8	9	c0	c1	c2	c3	c4	c5	c6	c7	c8	3	4	5	6	7	8	
9	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	4	5	6	7	8	9	
@	c1	c2	c3	c4	c5	c6	c7	c8	c9	c@	5	6	7	8	9	c0	
,	c2	c3	c4	c5	c6	c7	c8	c9	c@	c,	6	7	8	9	c0	c1	
SPACE	c3	c4	c5	c6	c7	c8	c9	c@	c,	c SPACE	7	8	9	c0	c1	c2	
&	c4	c5	c6	c7	c8	c9	c@	c,	c SPACE	c&	8	9	c0	c1	c2	c3	
.	c5	c6	c7	c8	c9	c@	c,	c SPACE	c&	c.	9	c0	c1	c2	c3	c4	
-	c6	c7	c8	c9	c@	c,	c SPACE	c&	c.	c-	c0	c1	c2	c3	c4	c5	

G

SUBTRACTION

NO BORROW BY PREVIOUS DIGIT-POSITION

Digits from Memory

Digits from Accumulator or Register	0	1	2	3	4	5	6	7	8	9	@	,	SPACE	&	.	-
0	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.	*b*&	*b* SPACE	*b*,
1	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.	*b*&	*b* SPACE
2	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.	*b*&
3	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.
4	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-
5	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0
6	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1
7	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2
8	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3
9	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4
@	@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5
,	,	@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6
SPACE	SPACE	,	@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7
&	&	SPACE	,	@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8
.	.	&	SPACE	,	@	9	8	7	6	5	4	3	2	1	0	*b*9
-	-	.	&	SPACE	,	@	9	8	7	6	5	4	3	2	1	0

F

G

BORROW BY PREVIOUS DIGIT-POSITION

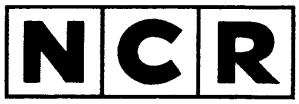
Digits from Memory

Digits from Accumulator or Register	0	1	2	3	4	5	6	7	8	9	@	,	SPACE	&	.	-
0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.	*b*&	*b* SPACE	*b*,	*b*@
1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.	*b*&	*b* SPACE	*b*,
2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.	*b*&	*b* SPACE
3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.	*b*&
4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-	*b*.
5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0	*b*-
6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1	*b*0
7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2	*b*1
8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3	*b*2
9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4	*b*3
@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5	*b*4
,	@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6	*b*5
SPACE	,	@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7	*b*6
&	SPACE	,	@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8	*b*7
.	&	SPACE	,	@	9	8	7	6	5	4	3	2	1	0	*b*9	*b*8
-	.	&	SPACE	,	@	9	8	7	6	5	4	3	2	1	0	*b*9

F

G

An Educational Publication
Marketing Services Department
THE NATIONAL CASH REGISTER COMPANY — DAYTON 9, OHIO



PRINTED IN U.S.A. SP—1128 E5TTT