would be the one in which the search continues for a whole drum cycle. The mark would be found during word-time 12. During word-time 12 there will also be a signal telling the computer to read a command from the next word-time, this signal having been generated by the N specified in the return command itself. Thus, in the worst possible case both the mark signal and the N signal will be present during the same word-time, 12. The N signal is interrogated at T21, whereas the mark signal is interrogated at T13. Therefore, even in the worst case, it will be the mark signal that picks the next command, and not the N signal.

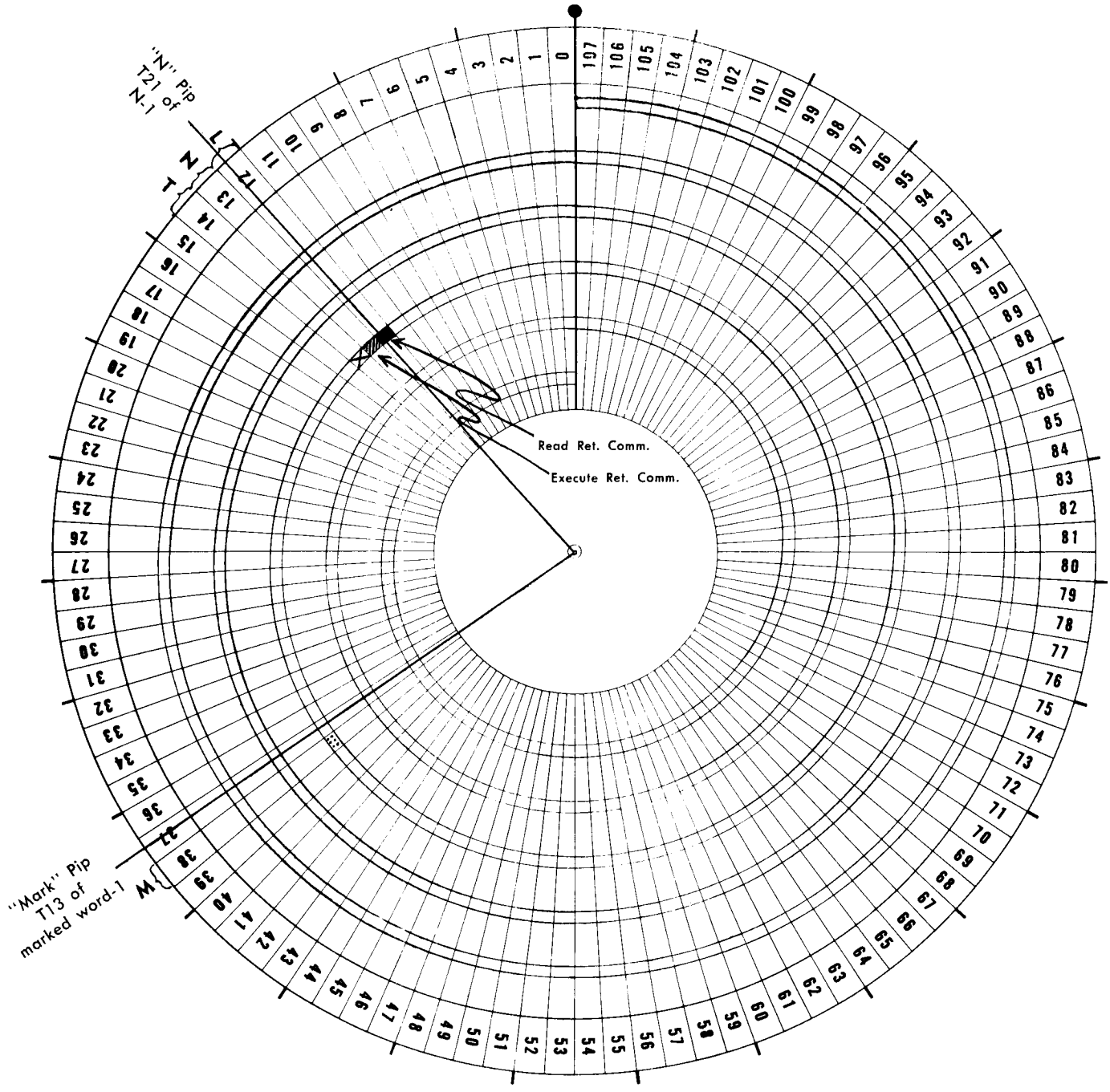
Drawing 2 shows pictorially the operation of the return command when coded in a different manner. In this case the command is executed dur- word-times L + 1 through N - 1. Shown in the drawing, L = 12, N = 27, and the word-times of execution are 13 through 26. The location of the next command to be taken from the return command line cannot be determined by any signal occuring during execution time, unless that signal occurs during the last word-time of execution. The N signal does occur during word-time 26, because N = 27; therefore, the location = N will arrive earlier than any existing marked location, unless the mark is also present during word-time 26. In any event, the next command will be taken from the return command line at word-time N. This is a convenient way to program a transfer of control to line C, word N, without setting a new mark, and therefore allowing an already existing mark to remain.

In drawing 3, where no care has been used in formulating the return command, L = 12, T = 44, and N = 20. The command will execute from 13 through 43, so the first mark or N signal occuring at 43 or later will determine the word-time at which the next command will be taken in the return command line. The drawing shows an existing mark at word-time 37, and we know that the N signal will be, as shown in the drawing, at 19. It is evident, then, that the next such signal will not occur until the following drum cycle, at word time 19. This will be the N signal, and so the next command in the return command line will be at 20 (N). This third case makes it apparent that care should be exercised in making up return commands. The first method, T = L + 2, N = L + 1, is the best.

Up to this point we have been speaking rather blithely about setting the return command in a subroutine; now we will see how. Every subroutine requires, as an input, a return command. Most subroutines require it to be placed in AR. Regardless of the location specified, the return command must be placed there prior to transferring control to the subroutine. One of the first steps in any subroutine, then, is to pick up the word containing this return command and transfer it to the proper location in the subroutine, so that, when that location is finally reached, in the course of the subroutine, the proper return command will be there to be read and executed. In the case of the square root subroutine, the return command must be placed in AR prior to entering the subroutine.

Notice that a command, in this case the return command, can be treated as data. If it is read during execution time, rather than during read command time, the computer will be unable to tell the difference between

Drawing 1:



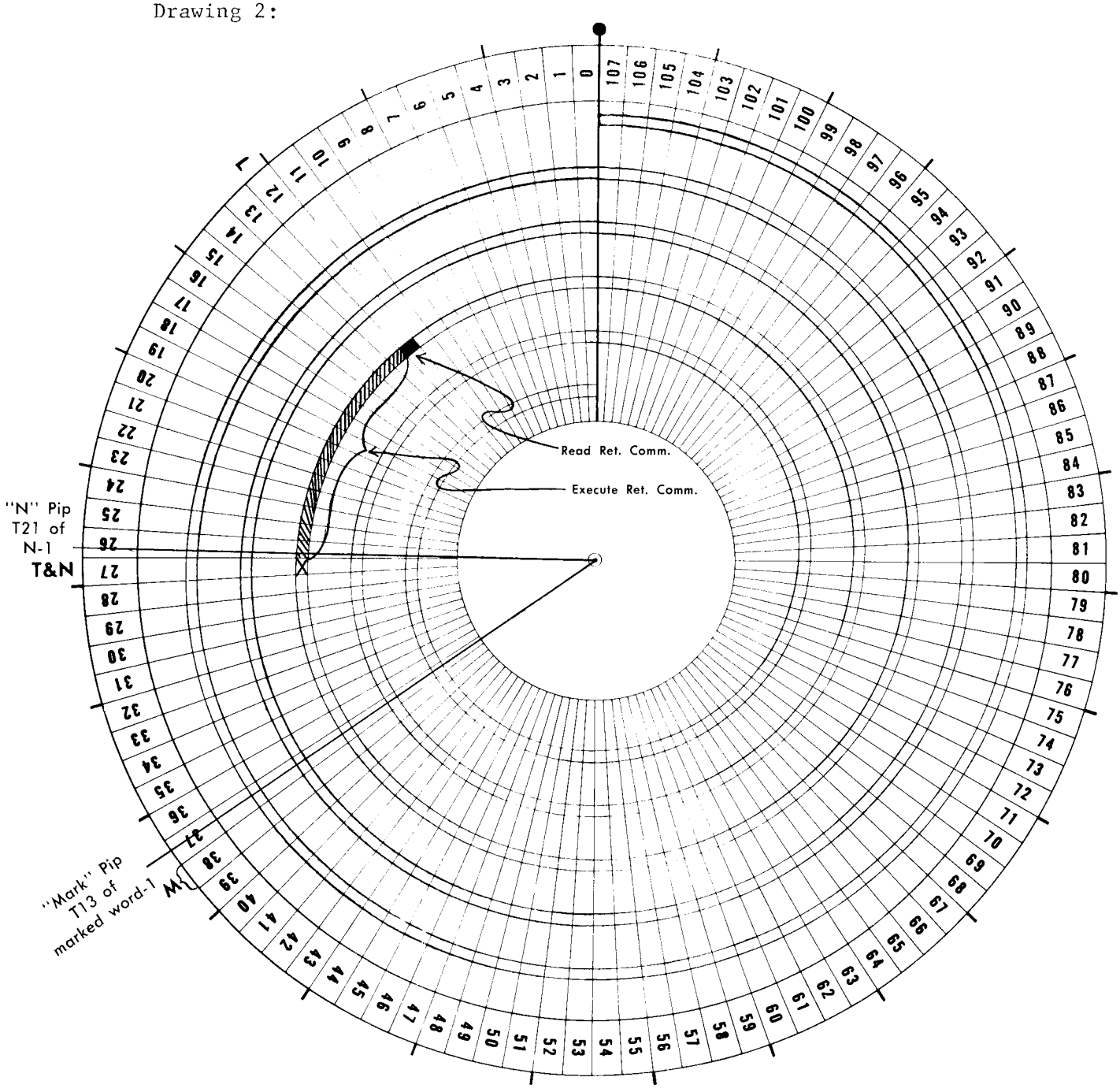
L
12

I/D	T	BP	N
0	14	0	13

CH	S	D	S/D
Line #	20	31	Line #

- NOTES -

Drawing 2:

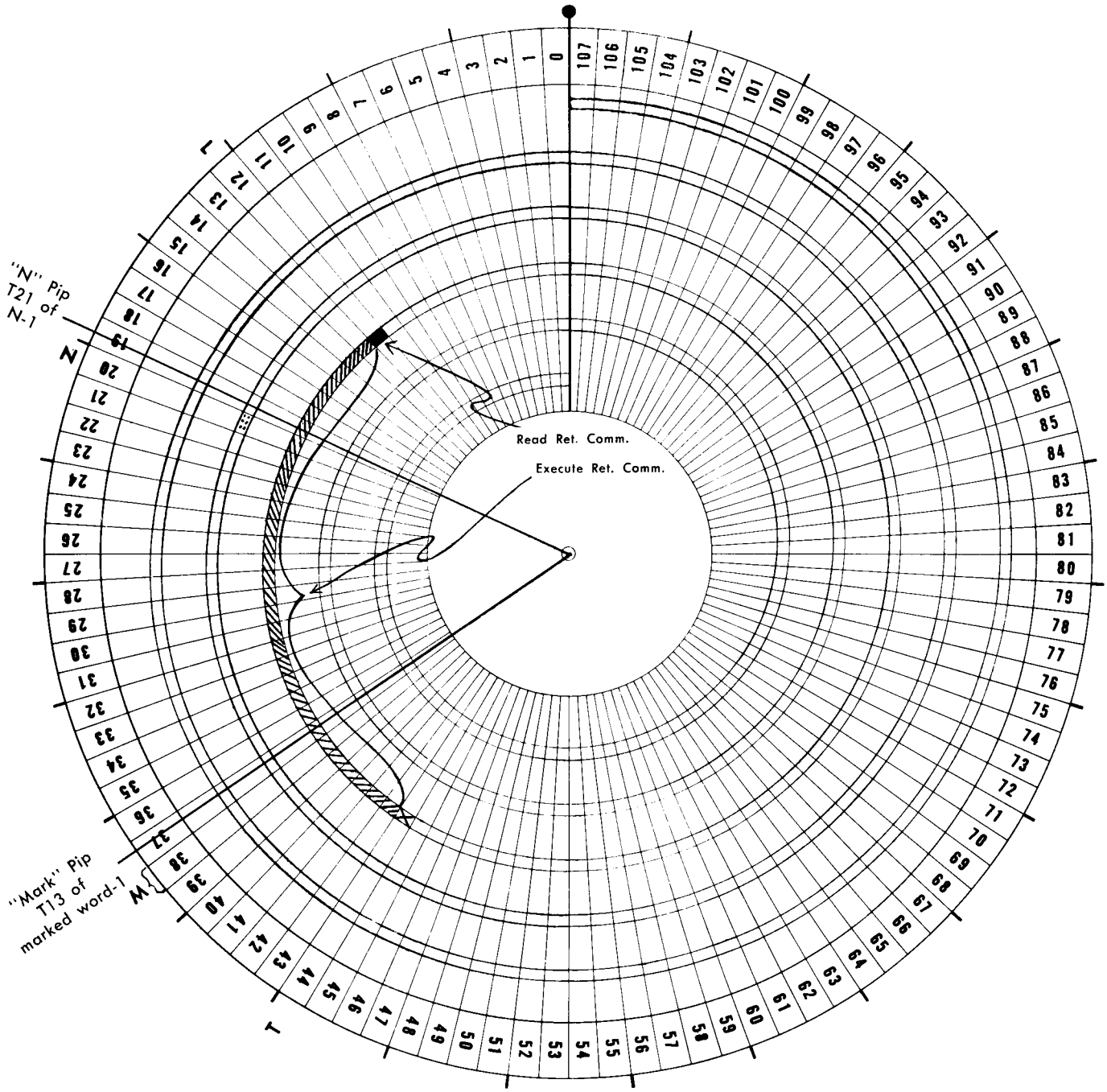


L
12

I/D	T	BP	N	CH	S	D	S/D
0	27	0	27	Line #	20	31	Line #

- NOTES -

Drawing 3:



L
12

I/D	T	BP	N	CH	S	D	S/D
0	44	0	20	Line #	20	31	Line #

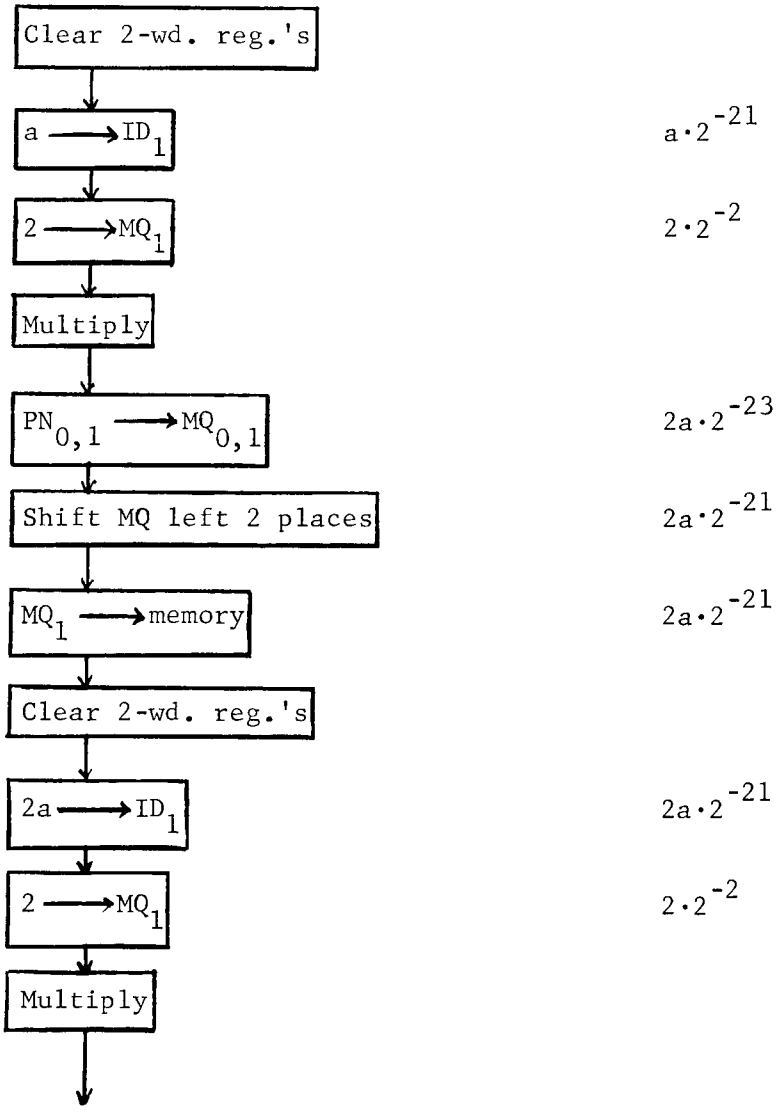
- NOTES -

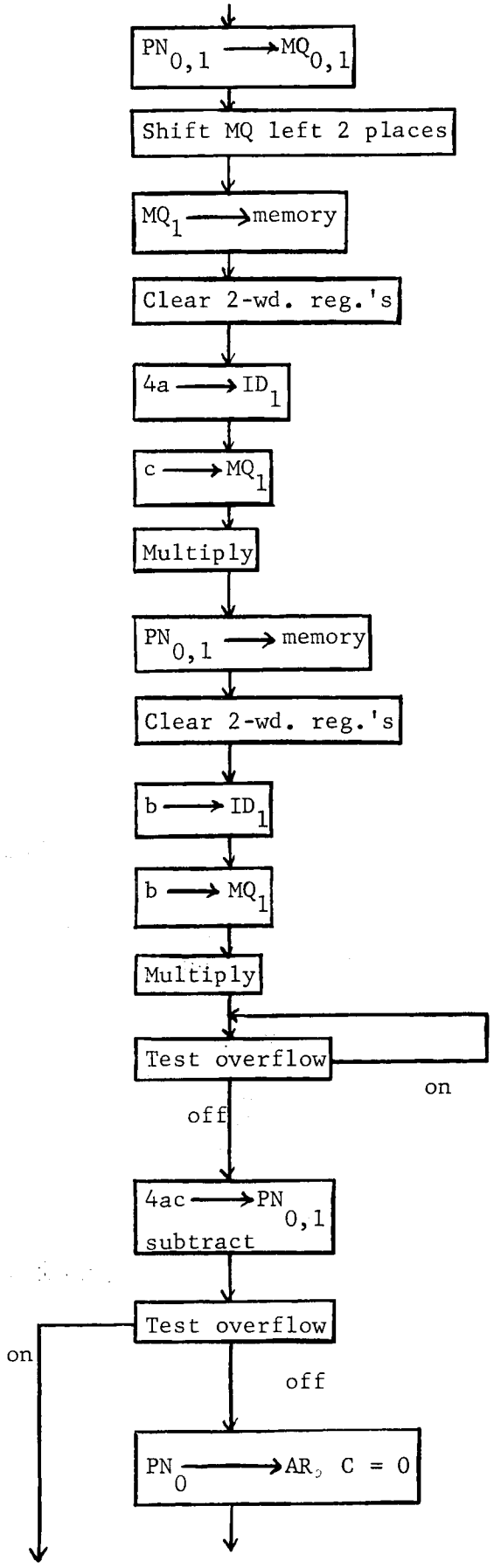
it and legitimate data. Thus, a command could not only be placed in AR, but it could, while there, be modified through the addition of a constant.

This leads us to a brief discussion of another special command, which tells the computer to "take the next command from AR". This, too, is a special command, containing D = 31, S = 31, and C = 0. Thus, a command could be transferred to AR, modified there, and executed out of AR, in its modified form. The N of the "take next command from AR" command will be the word-time at which the command in AR will be read.

REVISED FLOW DIAGRAM

Having determined which test commands we must incorporate, and how to set up for, enter, and exit from, the square root subroutine, we can revise the expanded flow diagram for the solution of the quadratic equation.





$4a \cdot 2^{-23}$

$4a \cdot 2^{-21}$

$4a \cdot 2^{-21}$

$4a \cdot 2^{-21}$

$c \cdot 2^{-21}$

$4ac \cdot 2^{-42}$

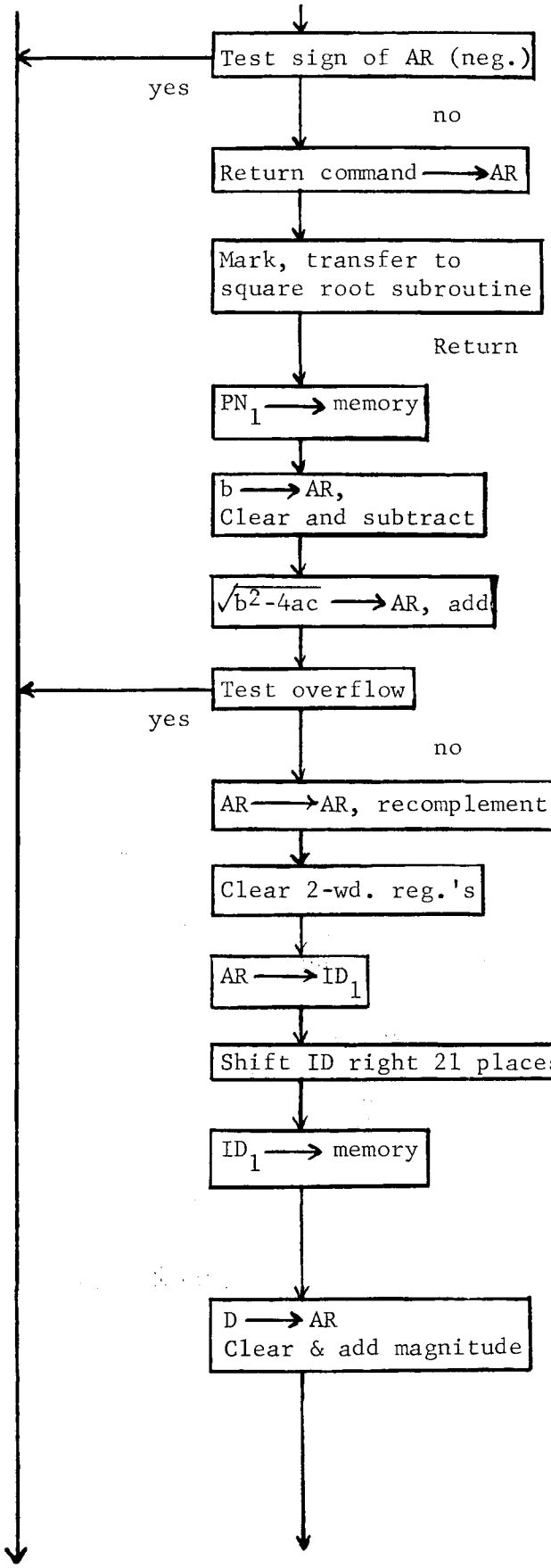
$b \cdot 2^{-21}$

$b \cdot 2^{-21}$

$b^2 \cdot 2^{-42}$

$(b^2-4ac) \cdot 2^{-42}$

sign of radicand in AR



$(\sqrt{b^2-4ac})\cdot 2^{-21}$

$-b\cdot 2^{-21}$

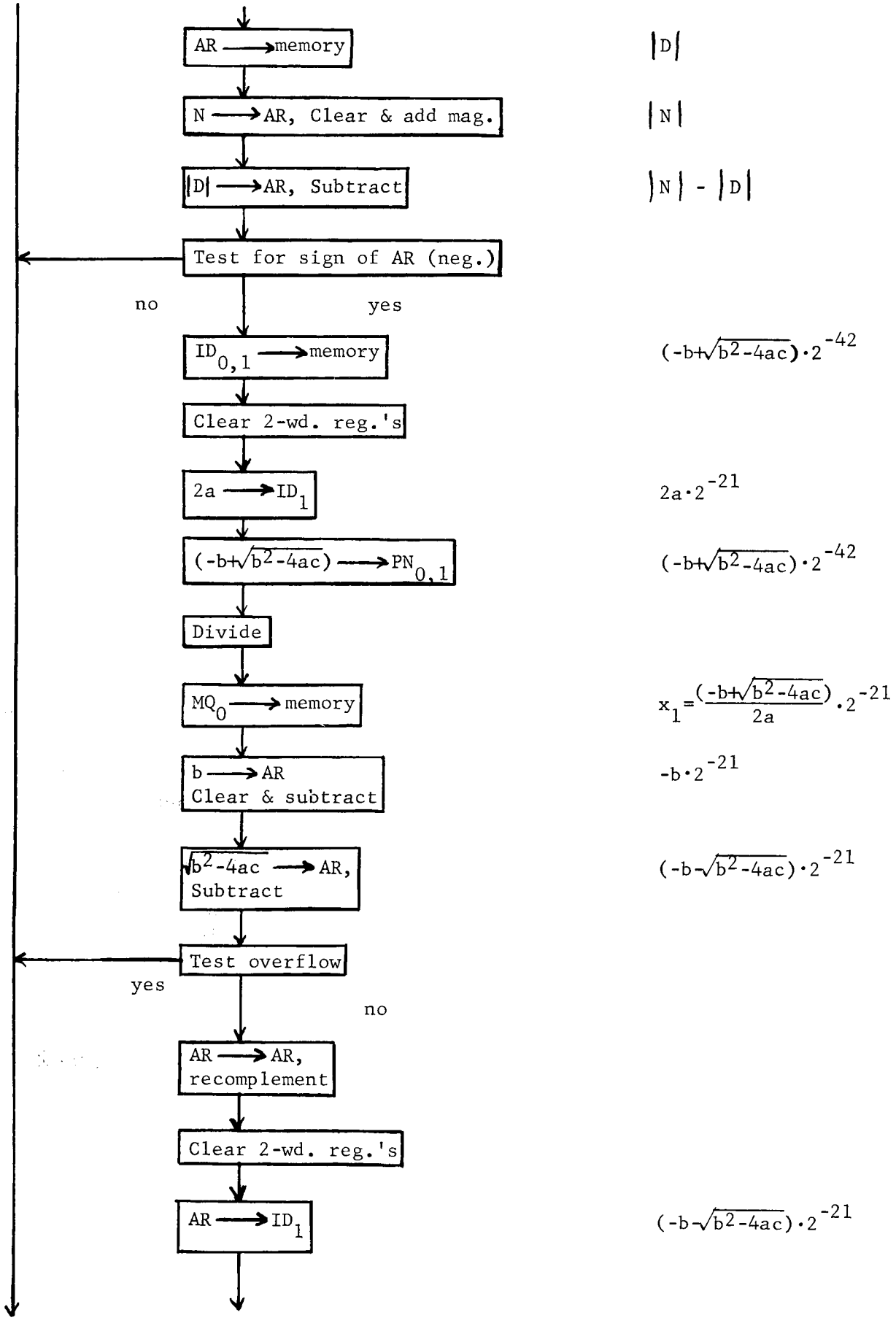
$(-b+\sqrt{b^2-4ac})\cdot 2^{-21}$

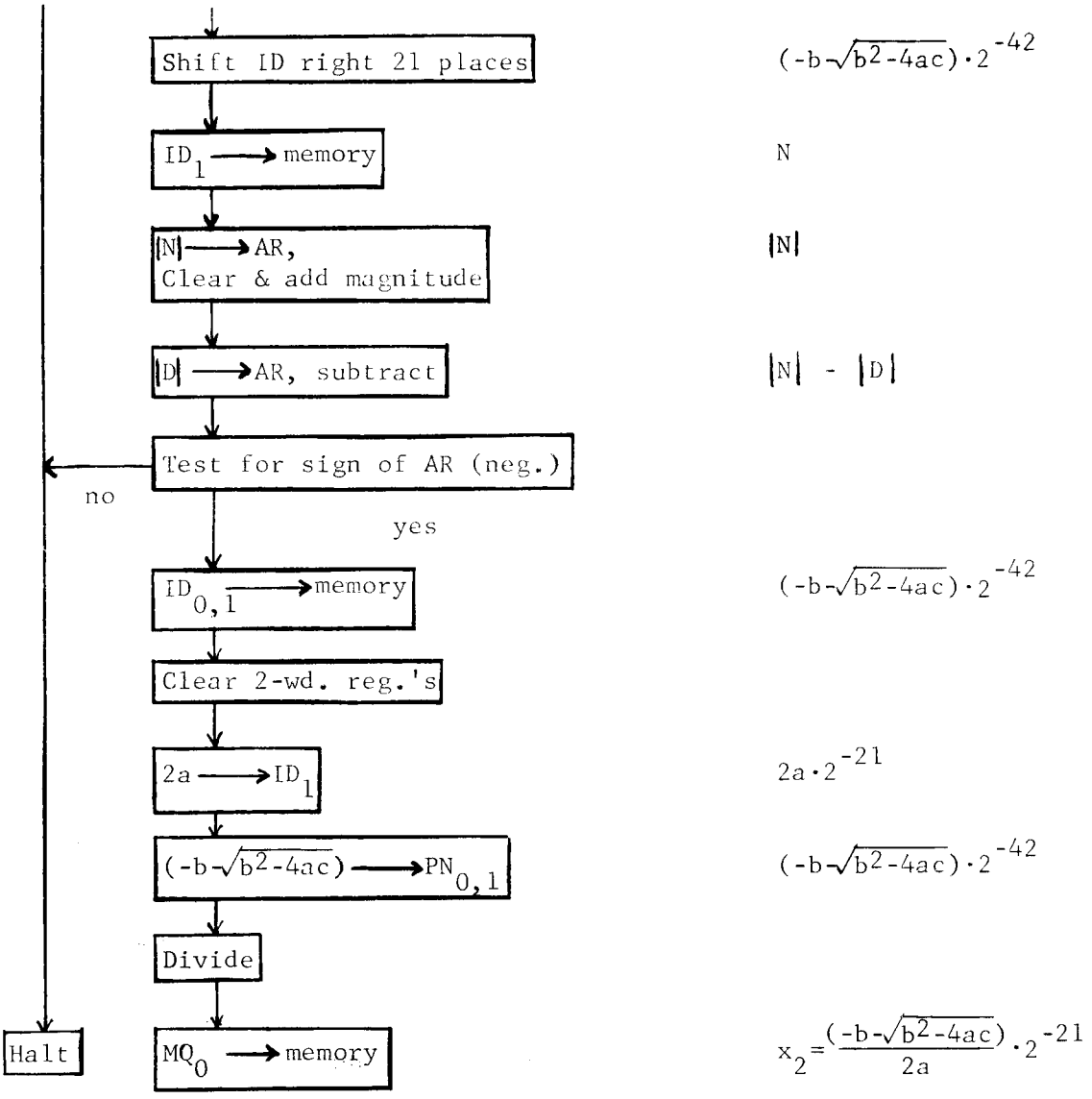
$(-b+\sqrt{b^2-4ac})\cdot 2^{-21}$

$(-b+\sqrt{b^2-4ac})\cdot 2^{-42}$

N

|D|


AR ⟶ memory
|D|
N ⟶ AR, Clear & add mag.
|N|
|D| ⟶ AR, Subtract
|N| - |D|
Test for sign of AR (neg.)
no
yes
ID0,1 ⟶ memory
(-b+√(b2-4ac))·2-42
Clear 2-wd. reg.'s
2a ⟶ ID1
2a·2-21
(-b+√(b2-4ac)) ⟶ PN0,1
(-b+√(b2-4ac))·2-42
Divide
MQ0 ⟶ memory
x1=((-b+√(b2-4ac))/2a)·2-21
b ⟶ AR
Clear & subtract
-b·2-21
√(b2-4ac) ⟶ AR,
Subtract
(-b-√(b2-4ac))·2-21
Test overflow
yes
no
AR ⟶ AR,
recomplement
Clear 2-wd. reg.'s
AR ⟶ ID1
(-b-√(b2-4ac))·2-21



This is the complete flow diagram for the main computation part of the program: it will occupy line 00. The square root subroutine will occupy line 01. However, the program still lacks a method or "scheme" of input, and any provision for output. We call for a, b, and c from memory, but as yet have made no provision for initially storing them there. Similarly, we generate two answers, x_1 and x_2, the two roots of the quadratic equation, but we have no provision as yet for communicating these carefully derived answers to the outside world. They're still stored away inside the computer. We have also made no provision for stopping the computer, or in any way terminating the main body of the program, although we do halt the computer in the case of error.

The next step in development of the program, now that we know the exact form in which we want the inputs, is to devise an input scheme; it, too, will be flow diagramed, and we will treat it almost as a separate program, although the input scheme is really an integral part of any program.

Because of their similarities, we will discuss inputs and outputs together, and then flow-diagram each method chosen for this particular program. They will, of course, be much shorter than the main body of computation.

INPUTS/OUTPUTS

A general-purpose computer is worthless unless it can receive inputs and yield outputs. The requirements of any input or output system are:

1. compatibility with the central portion of the computer,
2. ability to handle any type of information that may be needed or yielded by the computer,
3. accuracy,
4. speed.

These four requirements are listed in their relative order of importance.

Certainly the input system must be compatible with the central portion of the computer. It must be able to convert, if necessary, incoming information into a form recognizable by the computer. In the case of the G-15, the information must be in the form of electrical pulses which can generate magnetized spots on the surface of the drum, called "bits". In special cases, certain inputs to the G-15 may be electronic "signals", capable of activating a specific circuit or component directly. Such signals might, for example, cause an operation within the computer similar to that which could be caused by the execution of a command in a program. Most signals of this sort, you will see later, will call for an input or output, in the same manner it might be called for by a command in a program. This will not always be the case, however. In any event, whether an input of pulses or signals is necessary, a human operator is not anatomically equipped to supply them directly. He is therefore supplied with a set of buttons and switches, which he can manipulate, and which close and open circuits, supplying the computer with the pulses and signals it needs.

In some cases, the computer can be linked directly to some other system. In such a case, the input system receives electrical inputs, rather than manual inputs, and it must convert these to other pulses and signals in the form needed by the computer. This type of operation is sometimes referred to as "on-line" operation because the computer is on a line with the external system. A typical example of this type of operation might be one in which radar receivers are linked to the computer through some type of device which converts the range pulses and angle of deviation from North of the set itself to binary form. Cases of this type, where the computer is used "on-line" require special peripheral equipment for the computer, which is not supplied as standard equipment. In some cases, actual modification of the computer itself might be necessary.

The inputs with which we will concern ourselves at present are those which can be handled by the standard input equipment, supplied with the computer. These will fall into the "operator" category. We will hereafter refer to them as "normal" inputs.

NORMAL INPUTS

Normal inputs to the G-15 can come from either of two sources:

1. an electric typewriter, which supplies the operator with the buttons and switches he needs, and

2. a photo-electric tape reader, which reads punched tape somewhat similar in appearance to teletypewriter tape.

Since the second source is merely a speedier substitute for the first (a reel of tape is punched in codes which simulate the activation of a typewriter key), and since the typewriter contains a button or switch for every possible pulse and signal, we will discuss the typewriter first.

TYPEWRITER INPUTS

The electric typewriter is connected to the G-15 by a cable, which contains many individual lines. Over these lines pass the various inputs from the keyboard and switches on the typewriter. As has already been mentioned, there are essentially two types of inputs that can be supplied to the computer. One type is electrical pulses, which set up information in the memory of the computer; the other type is "signals", which cause the computer to act, these signals having the same general effect as a programmed command.

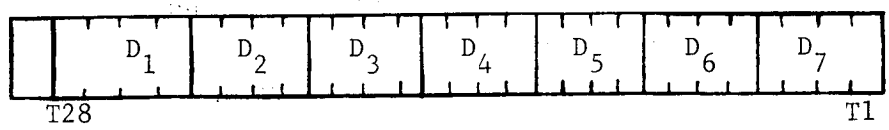
We will consider first the inputs from the typewriter which enter the memory of the computer. They are supplied by certain of the typewriter keys shown in the illustration on page 130. Notice they include all of the sexadecimal digits, 0 through z, the minus sign, the tab, the carriage return, and the slash (/) key. Inputs which enter the memory of the computer, enter at word-times, of course, and they become parts of words. Therefore, they must be in binary notation. The hex number system, as was pointed out earlier, is merely a short cut for binary representation. These inputs could just as well be entered from only two keys on a differently wired typewriter and with some modification of the input-system, one key for "0" and another for "1". It would take 29 punches of these keys to enter one complete word into memory. With the use of hex digits, a complete word can be entered through striking only 7 or 8 keys: seven hex digits and a sign. If the sign is positive, no key is struck; if it is negative, the minus sign key is struck.

Mounted within the typewriter, beneath the key-board, are a set of switches, one for each of these keys. As a key is struck, the associated switch is activated. The corresponding 4-bit code is generated and transmitted to the computer.

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
u	1010
v	1011
w	1100
x	1101
y	1110
z	1111

The minus sign, tab, slash (reload), and carriage return key do not enter 4-bit codes, although they do affect what is stored in memory, as will be explained shortly.

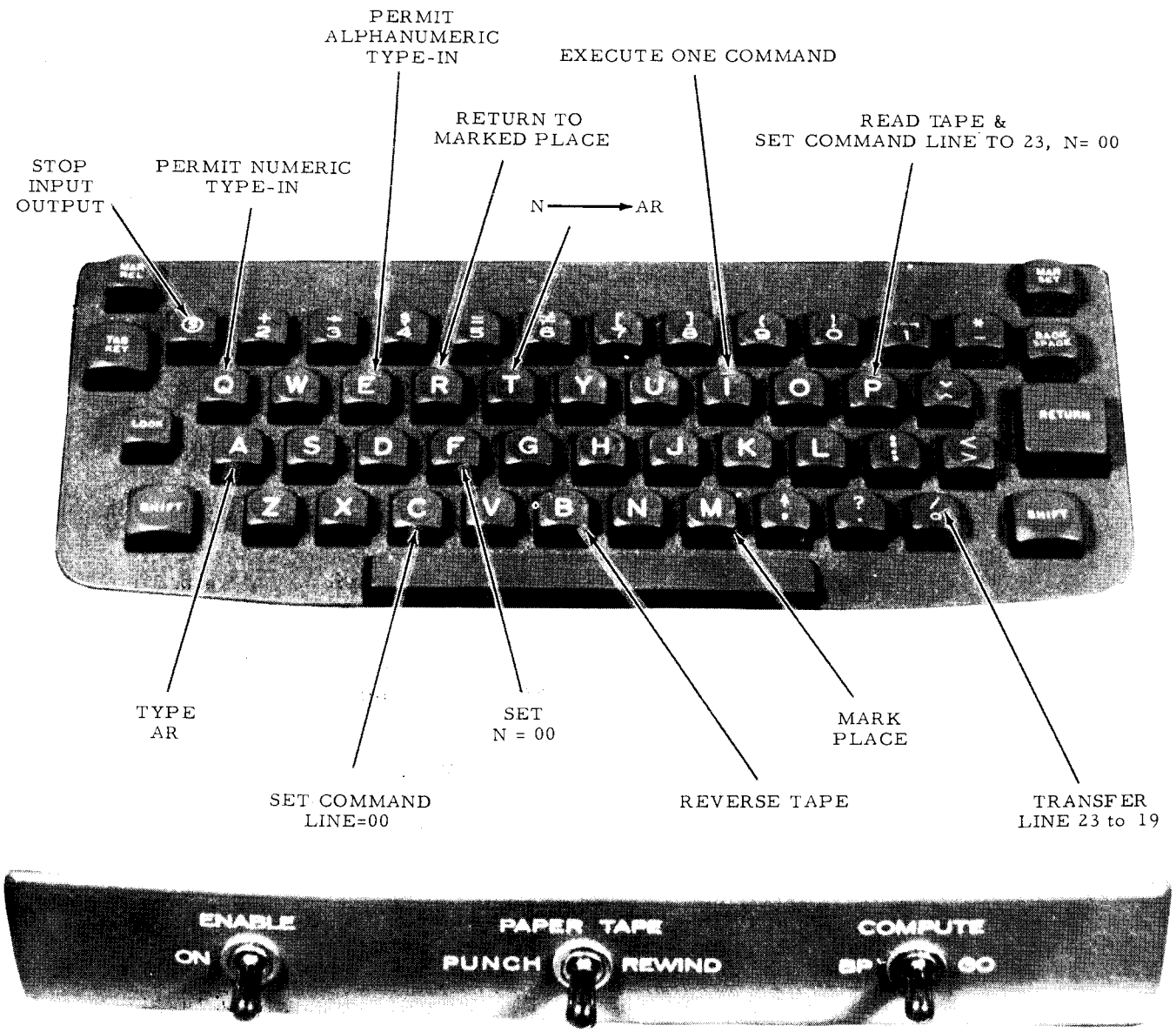
The 4-bit code, when it is generated, is transmitted into word 00 of short-line 23, at the least significant end, so that it occupies bits T1 through T4 of that word. When the next key is struck, assuming for the moment that it, too, will be a hex digit, a new 4-bit code is entered into these same bits, and the preceding one is shifted to the left, into the next four bits, so that the first "character" of input will occupy bits T5 through T8, and the second character will occupy bits T1 through T4, of word 00 in line 23. This process will continue as long as you keep on striking hex digit keys. Finally, after seven of these keys have been struck, word 00 of line 23 will look like this:



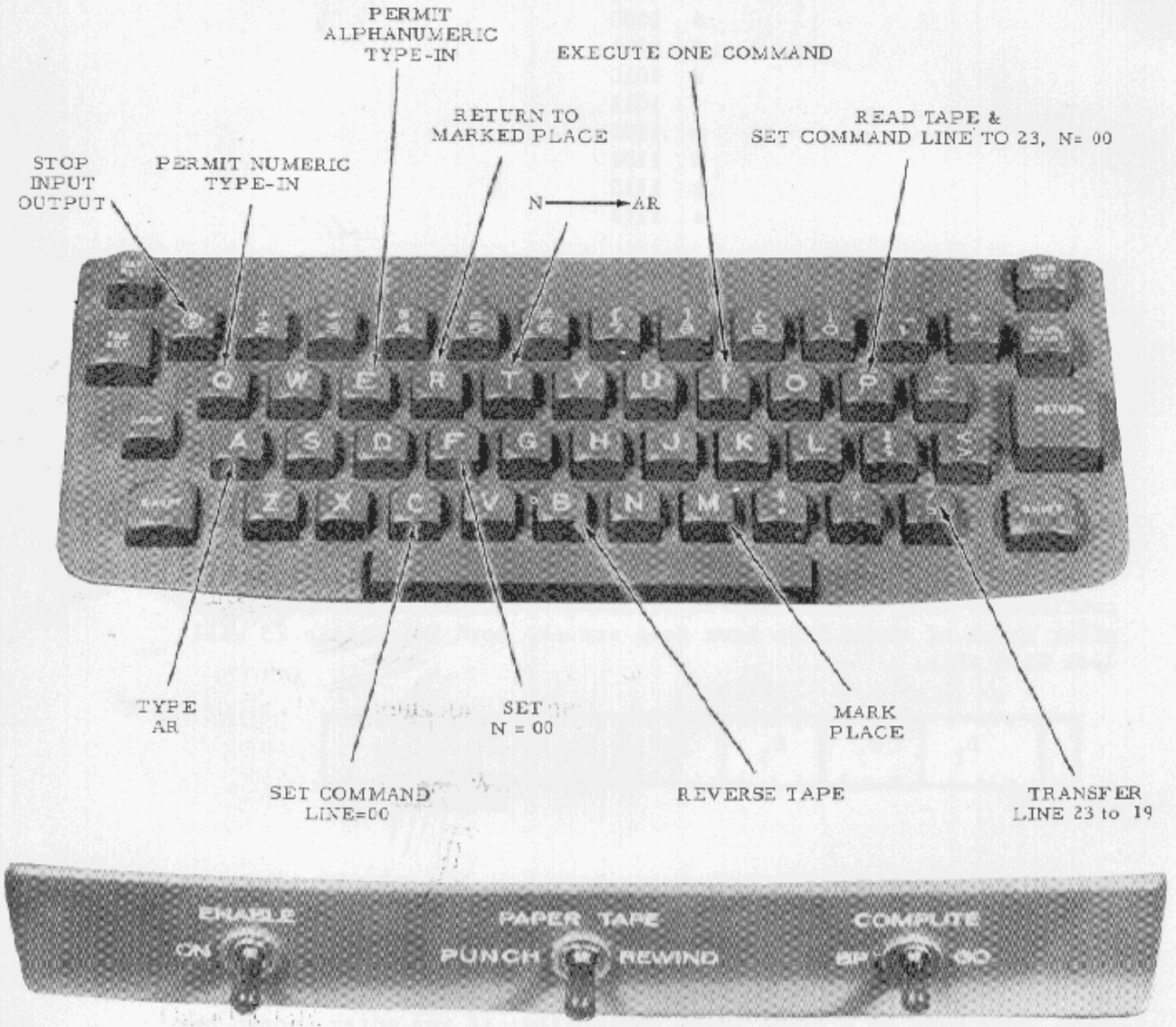
All of short line 23 will have been shifted to the left 28 bits. T1 of 00 will be in T29 of 00, the rest of the bits from 00 will be in 01, and so on, and 28 bits from word 03 will have been lost.

If you continue to enter hex digits, line 23 will continue to shift left, four bits at a time, until eventually, if you enter enough hex digits, the first characters of input will begin to "fall out" of 23.03, and will be lost. There must be a stopping-point. But, before we discuss that, let's continue with the input for a moment, where we have seven hex digits in 23.00.

Notice that, in this case, T1 of 23.00 (the sign-bit) contains the least significant bit of the seventh 4-bit code entered. A shift of one bit is necessary if we want to merely complete word 23.00, without losing any bits from the first code entered.



TYPEWRITER CONTROL KEYS AND SWITCHES



TYPEWRITER CONTROL KEYS AND SWITCHES

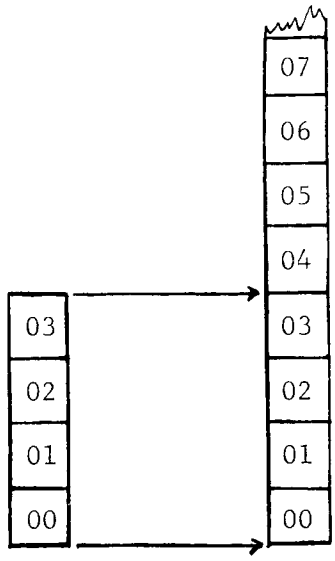
Associated with the input system is a flip-flop, which, like all flip-flops, can be in either of two states. It can contain a 0 or a 1. This particular flip-flop is referred to as the "sign flip-flop". It can be set to 0 (cleared) by striking either the tab key or the carriage return key on the typewriter. It will also be cleared after termination of any input. Once cleared, it can only be set to 1 by the striking of the minus sign key on the typewriter. When either "tab" or "carriage return" is struck, line 23 is shifted by one bit, rather than four, and the content of the sign flip-flop is "dumped" into the sign-bit (T1) of 23.00.

Therefore, if we desire to shift the seven hex digits we have just set up in 23.00 to the left one bit-position, causing them to occupy bits T2 through T29 of 23.00, and place a sign in T1, we strike either the tab or the carriage return key. The present content of the sign flip-flop will become the sign of the binary (hex) number in 23.00.

If the tab key was struck twice in succession, the first input would cause all of line 23 to shift left one bit, and the content of the sign flip-flop (call it x, since it could be either 0 or 1, depending on whether or not the minus key was struck previously) would be entered into T1 of 23.00. The second input would again shift line 23 to the left one bit, placing x in T2 of 23.00, and the content of the sign flip-flop (now known to be 0, since the sign flip-flop was cleared) would be entered into T1 of 23.00. Any succession of inputs is permissible, and the result in line 23 will be predictable. Notice that a minus sign can be struck at any time, but will not enter the computer until the tab or the carriage return key is struck.

Finally, when line 23 is filled as desired, it seems we have run out of space for the storage of inputs. But this is not correct. There remains one typewriter key in the illustration, which has not yet been discussed: the slash (reload) key. Striking this key will cause all of long line 19 to be shifted towards its upper end by four full words, and cause the transfer of all four words in line 23 into the now vacated low-order four words of line 19 (23.00 - 03 $\longrightarrow$ 19.00 - 03).

These four words will also remain in line 23. Input may continue, and eventually line 23 will be refilled with new inputs (the old words will be pushed out of the high-order end of the line; where they go, nobody knows. It has been said they go where a light goes when it goes "out".), and the slash key may be struck again. Line 19 will shift by four words again, and the four words in line 23 will be transferred into the four low-order words of line 19. Thus eight words have entered the computer. If we number them in reverse of the order in which they were entered (call the first word entered 07; the last, 00), then the situation in the computer will look like this:



Line 23 Line 19

Eventually, with 27 reloads, we could fill line 19 with input. If, at that time, another reload is struck, the first four words of input will be lost. Provided this is not allowed to happen, and line 19 is filled with input, the first word of input will be in 19.u7. Four more words of input can be accommodated in line 23, but no reload should be given after they have been entered. This is the limit of one input. No more information can be absorbed by the computer during one input.

Now we come to the method for stopping any normal input. Any normal input is stopped by a "stop" code. The "s" key on the typewriter will supply this code. This brings us to the second type of input from the typewriter: signals which control the computer directly.

The "s" key supplies the computer with a "stop" code, which is a signal capable of controlling the computer directly. The computer is capable of handling only one normal input or output operation at one time. When one is in progress, the input/output system will be in a "not ready" status, which can be determined by inspecting the neon lights on the front panel of the computer. (See page 208. The bottom row of neons contains a group of five lights pertaining to inputs and outputs. One of these is marked "R"; it is the "ready" light.) If the "ready" light is not on, the input/output system is not ready, because an input or an output is in progress. If the computer's input/output system is not ready, the process currently being carried out must be stopped before another is begun. The stop code will do this.

The only way the stop code can be provided to end a typewriter input is via the striking of this key. It may be done at any point during the input, even before any information has been placed in line 23. In that case, of course, line 23 will retain its original contents.

In order for an input or output to be processed by the input/output system, it must be called for. This can be done by command in a program, but this will be discussed later. It can also be done from the keyboard of the typewriter.

We now seem to be on the horns of a dilemma. We have just said that a typewriter input, or, for that matter, any normal input, cannot be processed until it is called for. We then proceeded to say that we would originally call for it through an input from the typewriter.

"ENABLE" ACTIONS

Notice again the difference between the two types of input from the typewriter: one, already discussed, places information from the keyboard into the memory of the computer; the other, which we are now discussing, supplies control signals directly from the keyboard to the computer. The latter type is called for in only one way: through a switch action. It cannot be called for by a program. In the preceding drawing of the typewriter, you will notice a switch mounted on the front of the base of the typewriter, called the "enable" switch. This switch has only two positions: to the left, it is on; in the center position, it is off. When the enable switch is on, the control keys on the keyboard are enabled to send control signals to the computer; when it is off, these keys are not connected to the computer.

There is only one exception to this rule, and we have discussed it; when a type-in of information for the memory of the computer is called for, the stop code to end it can be, and must be supplied by striking the s key. In this case, the enable switch need not be on. In all other cases, including use of the s key to stop any other input or output, the enable switch must be on in order to activate the control keys on the keyboard. Because this is the only way these inputs can be called for, and because they are not really inputs, in the sense that they don't place information in the memory of the computer, they are not usually referred to as inputs. Rather, they are referred to as "enable actions" or "control actions". The custom adopted as a short-hand for specifying one of these actions is to underline the appropriate letter (e.g., s). We will drop the reference to these as inputs, and adopt the name, "enable actions".

As seen in the diagram on page 130, q will call for a typewriter input. We have already mentioned that s will stop any normal input or output, and set the input/output system ready.

Two other enable actions should be mentioned here: they are c and f. Much earlier in this book, a question was deliberately left unanswered, with the excuse that it would be covered later. The question was: How do we initially select a command line, and how do we change control from one command line to another during the operation of a program, if that is necessary? The latter part of the question has been answered: we change command lines under program control through use of either the mark and transfer control command or the return command. The former

part of the question will be answered now. We can initially select a command line through the use of the two enable actions, c and f. You will notice in the drawing that c will signal the computer to set the command line equal to the following number, or, if no number follows, to command line 00. This implies that c would be followed by the typing of a number from 0 through 7, corresponding to the desired command line. This is correct. A 0 may follow c or not; there will be no difference in the effect of the signal.

Selecting a command line, however, does not fully establish the address of the first command to be obeyed. There still remains the word-time portion of the address. In the two commands that transfer control to a specified command line, this is accounted for, either in the command itself, or in the timing of the command, combined with the existence of a "mark" in the computer. In the case of enable actions, a word-time must be supplied by another enable action, f. This signals the computer to take the next command from word 00 of the selected command line. Notice, there is no provision to specify any word-time other than 00. If however, this is not specified, and the computer is allowed to start operating, it will take the first command in the selected command line at whatever word-time it received as the N of the last command read, which, in most cases, will lead to an erroneous result, since it is a good bet that, when you choose to change command lines, the desired word-time for the start of the new sequence will be different.

Because of this feature of the f action, it has been the experience of many programmers of the G-15 that it is best, wherever possible, to start a program at word-time 00.

As you can see, in the drawing on page 130, there are many other enable actions which, as yet, remain uncovered. They will be discussed, one by one, as they arise during the further discussion of inputs and outputs.

Discussion of punched tape input requires some knowledge of what punched tape contains. Therefore, we will next bring up punched tape output, followed by punched tape input.

PUNCHED TAPE OUTPUT AND OUTPUT FORMAT

The use of punched tape for output serves two purposes. It preserves information in a form in which it can be retained for later use as an input for the computer; in other words, it acts as an interim storage device. The second purpose is to speed up the output operation of the computer (the other normal output is via the typewriter, and is a good deal slower), and yields an output in a form which can be processed off-line (not involving the computer), on any suitable tape-reading device which can read this type of punched tape and type out the contents, much on the order of some teletypewriters.

When a punched tape output is called for, the output information will be taken from line 19 in a manner prescribed by a format stored in the memory of the computer.

A format is a series of binary codes, each of which calls for a type of output character. The types of characters, their abbreviations, and the related format codes are shown below:

Type of Output Character	Abbreviation	Format Character
Digit	D	000
End (stop)	E	001
Carriage Return	C	010
Period (point)	P	011
Sign	S	100
Reload	R	101
Tab	T	110
Wait (skip one digit)	W	111

The complete format for the punching of tape is contained in four words in memory, 02.00 - 02.03. The desired format characters are placed end-to-end, beginning with T29 of word 03 in line 02, and working backwards, ignoring word-boundaries, towards T1 of word 00. The format may be any desired length within the limit of four words. Since all of line 19 may contain information to be transferred to the tape punch, it is readily apparent that not enough 3-bit format characters can be placed in the available bits in four words to call for every digit and every sign of the output. The reload code in the format will cause all of the preceding format characters to be reinspected, as the processing of line 19 continues. At this point we must investigate the processing of line 19.

Each D in the format will call for the output of bits T29 down through T26 of word 19.u7 (the most significant four bits of the word, and therefore, the most significant hex digit of the word) as a hex digit. Line 19 will then be shifted up four bits, losing the four which have just been inspected, and vacating the four least significant bits (T4 down through T1) of word 19.00. Thus, successive D's in the format will cause a succession of hex digits in the output, and they will also cause a succession of shifts in line 19.

Each S in the format will cause an inspection of bit T1 of word 19.u7, and an output of either a plus, or a minus sign. No shift will occur in line 19. Notice that the sign of a number will have to be called for at the time it is in T1 of 19.u7.

Each W in the format will cause a 4-bit shift of line 19, but there will be no output of the corresponding hex digit; instead, an output which will be treated as a blank character is substituted.

Each T in the format will cause a special tab code to be punched on tape. Line 19 will be shifted one bit.

Each C in the format will cause a special carriage return code to be punched on tape. Line 19 will be shifted one bit.

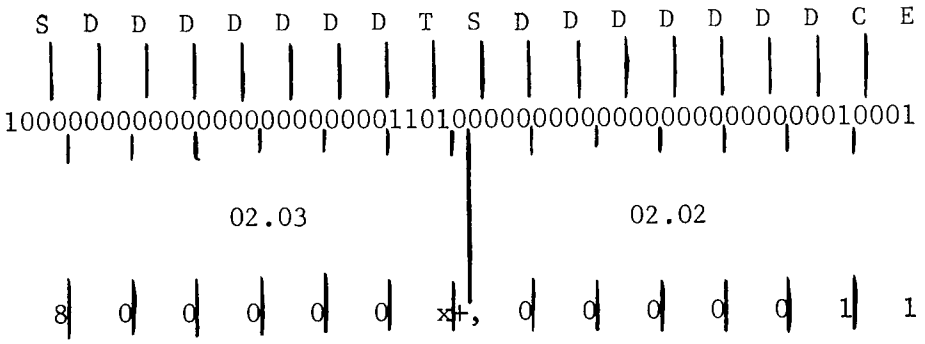
Each P in the format will cause a period to be punched on tape. Line 19 will not be shifted.

The only two remaining format characters are R and E. Each R will cause a special reload code to be punched on tape. There will be no shift of line 19. The inspection of the entire format, beginning at T29 of 02.03 will be repeated. Thus, once an R has been placed in a format, the format is essentially closed in a loop. Any remaining bits in the allotted four words in line 02 will never be inspected, and the output will never end. Use of R in a format requires caution. A way to stop an output under control of such a format by program command is available, and will be discussed later. Striking s on the typewriter keyboard will also stop it.

Use of an E character in a format is the normal method of stopping an output. In addition to punching a stop code on the tape, it supplies a stop code for the output. But the operation of this character of format is very special, and requires closer scrutiny. You have noticed that, as characters (hex numbers and signs) in line 19 are used up, during an output, they are shifted out of the line (processing of a sign does not accomplish this), and bits are vacated at the low end of the line, in word 00. In any shift, the vacated bit-positions are filled with 0's. When line 19 contains nothing but 0's, we want output to cease. Thus, by clearing line 19 and then properly positioning the output data in line 19, prior to the output, we can control the duration of the output. This is made possible through the computer's interpretation of an E character in a format. When the E character is encountered, as the format is inspected, character-by-character, line 19 is searched for at least one non-0 bit. If a 1 is found anywhere in the line, the E character is automatically interpreted as an R, and causes the same sequence of events as is caused by an R character. Eventually line 19 will contain all 0's. When the E character is encountered, the search of line 19 is performed, it is found that the entire line is clear, and the E character is interpreted as calling for a stop code to be generated on the tape, and for the output to be stopped. At this point, the input/output system will be "ready".

Consider, as an example, the case of the program we have already developed, in which two answers will be generated, each a signed single-precision number. We could first clear line 19. The best method for this is an immediate command, allowed to work for one complete drum cycle, with S containing 0's in each of its words, and D = 19. We could, for example, clear the two-word registers (and IP) with the clear command, then use any one of them as S, with C = 0, and D = 19. We would make this an immediate command (I/D = 0), and set T (flag) equal to L_1. During each word-time either the even or the odd half of the specified two-word register will be copied into the specified

word-time of line 19. After execution of this command, every word in line 19 will contain 29 0's. After line 19 has been cleared, we could transfer x_1, generated by our program, into 19.u7, and x_2 into 19.u6. Then we set up the following format in line 02.



After both the data and the format are properly loaded, we initiate the output to the tape punch. Let's examine the results.

The first character of the format is inspected, and found to call for a sign. Therefore, bit T1 of 19.u7 is inspected, and a code for plus or a code for minus is punched on tape.

The second character of the format is inspected, and found to call for a digit. Therefore, the hex digit representing the four bits T29 - T26 in 19.u7 is punched on tape. Call this the first hex digit. Line 19 is shifted up four bits, so that T29 - T26 of 19.u7 receive the bits which previously occupied bits T25 - T22 of the same word; call this the second hex digit. Bits T4 - T1 of 19.u6 receive 0's from the previous word (all of line 19 has been cleared to 0, except words u6 and u7).

The third character of the format is inspected, and found to call for a digit. The same sequence of events occurs again, the third hex digit of the number x_1 arriving in T29 - T26 of 19.u7, while the second hex digit is punched on tape. The shift of line 19 leaves bits T8 - T1 of 19.u6 cleared.

The same thing happens for the fourth, fifth, sixth, seventh, and eighth characters of the format, which all call for digits. After the eighth format character has been processed, and the character it calls for has been punched, line 19 is shifted again. The result, up to and including this shift will be: the sign and seven hex digits representing x_1 have all been punched on tape, sign first, followed by the most significant digit down through the least significant digit. Line 19 has been shifted so that 28 bits from 19.u7 have been lost, T29 of 19.u7 contains what was originally the sign-bit of that same word, bits T28 down through T1 of 19.u7 contain the magnitude bits of x_2 (originally in 19.u6), and T29 of 19.u6 contains the bit that was originally the sign-bit of the same word. Bits T28 through T1 of 19.u6, like the rest of the line, contain 0's.

The ninth character of the format is inspected, and found to call for a tab. A tab code is punched on tape, and line 19 is shifted by one bit. 19.u7 now contains all of x_2, while the rest of the line is cleared.

The tenth character of the format is inspected, and found to call for a sign. Bit T1 of 19.u7 is inspected, and the proper sign code is punched on tape.

The eleventh through the seventeenth characters of the format are inspected, and, since they also call for a series of digits, are processed in the same fashion as were the second through the eighth format characters. After the seventeenth character of the format has been processed, its corresponding character of output has been punched, and the corresponding shift of line 19 has been carried out, the result will be: the sign and seven hex digits representing x_2 have all been punched on tape, sign first, followed by the most significant digit down through the least significant digit. Line 19 contains only one of the original data bits, in T29 of 19.u7, which was originally the sign-bit of x_2 in 19.u6. The rest of line 19 is clear.

The eighteenth character of the format is inspected, and found to call for a carriage return. A carriage return code is punched on tape, and line 19 is shifted one more bit. Now the entire line is cleared to 0.

The nineteenth character of the format is inspected, and found to call for a stop code. Line 19 is searched, and, since it is found to contain nothing but 0's, this character is treated as an E character. The stop code called for is punched on tape. A stop code is also generated which stops the output and sets the input/output system "ready" for another input or output.

At this point there is a tape hanging out of the computer (top, front). There will be no doubt in your mind where the punch is after you have once activated it. A toggle switch on the face of the computer allows you to feed blank tape through the punch until you can tear off the piece of tape containing the entire contents of the output you called for. How will you know when you reach the end of valuable information? You know which character was punched last; it was the stop code. Therefore, if you can recognize a stop code, you can tell the end of the information on the tape. The following table shows the codes punched on tape corresponding to each character which can be punched. In the punched codes, as shown, a 1 represents punch; a 0, no punch. It will be seen that there are five "channels" on the tape.

The length of tape so generated is referred to as a "block" of tape. Every block is ended by a stop code. Its length will be determined by the lowest-ordered word in line 19 containing non-0 data when the output is called for which generates the block of tape.

Output Character	Code Punched on Tape
0	10000
1	10001
2	10010
3	10011
4	10100
5	10101
6	10110
7	10111
8	11000
9	11001
u	11010
v	11011
w	11100
x	11101
y	11110
z	11111
Space	00000
Minus	00001
CR	00010
Tab	00011
Reload	00101
Period	00110
Stop	00100
Wait	00111

The particular block of tape generated in our example will contain the equivalent of two words, since, by the end of the first inspection of the format, all of line 19 will contain 0's. The order of punched codes on the tape would be:

1st	sign
2nd	digit
.	.
.	.
.	.
8th	digit
9th	tab
10th	sign
11th	digit
.	.
.	.
.	.
17th	digit
18th	CR
19th	stop

Notice that the tape now contains the same characters that you might choose to supply, were you to "gate" type-in, and enter the two numbers x_1 and x_2 as typewriter inputs.

A block of punched tape can be read (by a photo-reader mounted on the front of the computer), upon command to the computer; this is the other normal input. Were this block of tape to be mounted on the drive mechanism of the photo-reader, and a p action taken (see drawing on page 130); it would be read as a computer input. The rules governing the entry of its information are the same as those governing a type-in. The first character entered, being a sign, would not immediately enter memory, but would enter the sign flip-flop. Then seven digits would be entered into word 00 of line 23. The next character of input, the tab, would shift line 23 one bit, placing the seven previously entered digits in bits T29 - T2 of 23.00. The sign of the number would be dumped into T1 of the same word from the sign flip-flop, and that flip-flop would be cleared. The next eight characters would be entered in the same way, the first complete 29-bit number (x_1), being shifted into bits T28 - T1 of 23.01 and T29 of 23.00, while bits T28 - T1 of 23.00 receive seven new digits. The following character, a carriage return, will have the same effect as the tab, and the result will be x_1 in 23.01 and x_2 in 23.00. The next character, the stop code, automatically reloads, shifts line 19 by four complete words, and places words 23.00 - 03 in 19.00 - 03 *. These words also remain in line 23. Then it terminates the input, and sets the input/output system "ready" for another operation. This, then, is the pattern for the entry of a block of tape into the memory of the G-15. All of line 19 could be loaded in this manner, and the last four words to be entered into line 19 would remain in line 23. This fact will be important to us a little later. Punched tape input is preferable to typewriter input in one respect, at least: speed.

TYPEWRITER OUTPUT

The format which controls the output from line 19 to punched tape also controls the output from line 19 to the typewriter, in exactly the same way, except in the case of typing, keys, tabs, and the carriage return on the typewriter are affected, rather than punch-heads. You can actually see the keys move as the contents of line 19 are typed out.

The contents of AR may also be typed out, under control of a format made up in exactly the same way as the format for line 19. The AR output format must be placed in line 03, words 02 - 03, prior to calling for the type-out of AR. Again, the inspection of the format will begin with T29 of word 03 and move toward the low-order end of the line. During a type-out of its contents, AR will be shifted in the same manner as line 19.**

---

* Note: This automatic reload feature of the stop code is not true when the s key is used to stop a type-in.

** Note: Because four words' worth of format should, in all cases, be sufficient to "cover" one word of output, it should be unnecessary for an "end" code in the format to be automatically changed to a "reload" code. For this reason, the "end" code in an AR format will never be changed to cause a reload.

The type-out of the contents of AR brings us to another topic. You will notice, from inspection of the drawing on page 130, a will cause the contents of AR to be typed out. You will also notice that this is the only output of the three mentioned (and these are all of the normal outputs) which can be called for through an enable action.

Consider for a moment the function of the whole class of enable actions. It is to enable the operator to give the computer commands directly, not in the normal binary command form. When would this be useful? Primarily, when the computer does not have loaded in its memory the desired program. These actions enable the operator to get a program into the memory of the computer, either one command at a time, through type-in, or a block of tape at a time, through the reading of tape. Once the program has been loaded, control can be given to it, within the computer, and it will operate the machine. For instance, when the computer is first turned on, perhaps in the morning, there will be no information in its memory. Turning it off the night before cleared memory. The enable actions enable an operator to start the computer. At such a time it is hard to conceive of the need for an output. Quite the contrary, when outputs are required, a program will have generated them, and that same program can call for them with commands, none of which we have yet defined.

DEBUGGING

The reason for providing for this one output, the type-out of the contents of AR, through enable action, is to assist the programmer in "debugging" his program. It has been painfully established by almost all the programmers who have preceded you, no matter what computer or programming system has been employed, that very few programs work successfully in all respects as originally written. There are usually a few flaws, perhaps stemming from carelessness, or from lack of knowledge, or from a change in requirements. Finding these flaws by inspection of the program is sometimes almost impossible, especially in very long and complicated programs. In such cases, the programmer will usually resort to making up a "test case", for which he will calculate the correct answer(s). He will then enter his program into the computer, and allow it to operate with the inputs of the test case. He will cause the program to halt temporarily at various strategic points, and inspect the partial results he has achieved. In this way he can eventually isolate the steps in the program which are causing the trouble. How does he inspect these results? He stops the program at points where AR contains vital information, and inspects AR. Thus, the provision for type-out from AR.

BREAK-POINT

Now we come to the only remaining question which was intentionally left open earlier, and answer it. In the machine form of a command, bit T21 was left undefined. This bit in a command is called the BP bit. BP stands for Break-Point. If a command contains a 1 in this bit, the

computer will halt upon execution of the command, provided a switch action has been taken previously. (Do not break-point a return command.)

On the front of the typewriter base is a switch called the "compute" switch. Other than performing an enable action, the computer will not operate until this switch is on. The center position for this switch is the off position. The switch is on when thrown either to the left or to the right. If thrown to GO, it will cause the computer to continue operating until either a halt command is reached or the compute switch is moved back to the off position. If thrown to BP, it will cause the computer to operate until a halt command is reached, the switch is thrown back to the off position, or a command with BP = 1 (called a "break-pointed" command) is reached. If you want the computer to be sensitive to these inserted break-points, then, you must move the compute switch to BP rather than GO to operate your program. A rule is that the enable switch and the compute switch should never be on simultaneously. If you have stopped at a break-point, turn compute off before turning enable on.

The entire process of debugging encompasses far too many techniques and far too much effort to be thoroughly discussed here, but one of the important facets of it is this periodic inspection of AR. Combined with the ability to type out the contents of AR, are certain other enable actions. For instance, t, will place in AR the address of the next command the computer will obey if the compute switch is turned back on. This enable action, followed by a, should help you determine whether or not your program is following the predicted path.

Notice that the contents of AR will shift as it is typed out. This means that, following the type-out, AR will no longer contain what it did. If, after inspection of AR, you wish to return to your program, it is quite conceivable that this destruction of AR's contents will cause errors in the rest of the program. Prior to a, you should take the m action, which will save the contents of AR and mark the location of the next command. Following a, r will restore this information.

"SINGLE CYCLE"

Another enable action which is of help in the debugging process is i, which causes only one step to be executed. You could operate a whole program through a long enough series of i's. If you will look at the drawing on page 208, you will see that among the neons on the front of the computer, there is a set for S and another for D. These lights will contain the S and D number, respectively, of the command being executed. By following your program, as it is written on paper, and these lights through a series of i's, you can often spot errors in the path of your program. Do not single cycle return commands.

INPUT/OUTPUT COMMANDS

It is necessary for most programs to call for their own inputs and outputs, by command. The commands which will do this are:

"Gate" type-in: D = 31, S = 12, C = 0.

Read punched tape: D = 31, S = 15, C = 0.

Type AR: D = 31, S = 08, C = 0.

Type line 19: D = 31, S = 09, C = 0.

Punch line 19 on tape: D = 31, S = 10, C = 0.

These inputs and outputs will behave in the manner described.

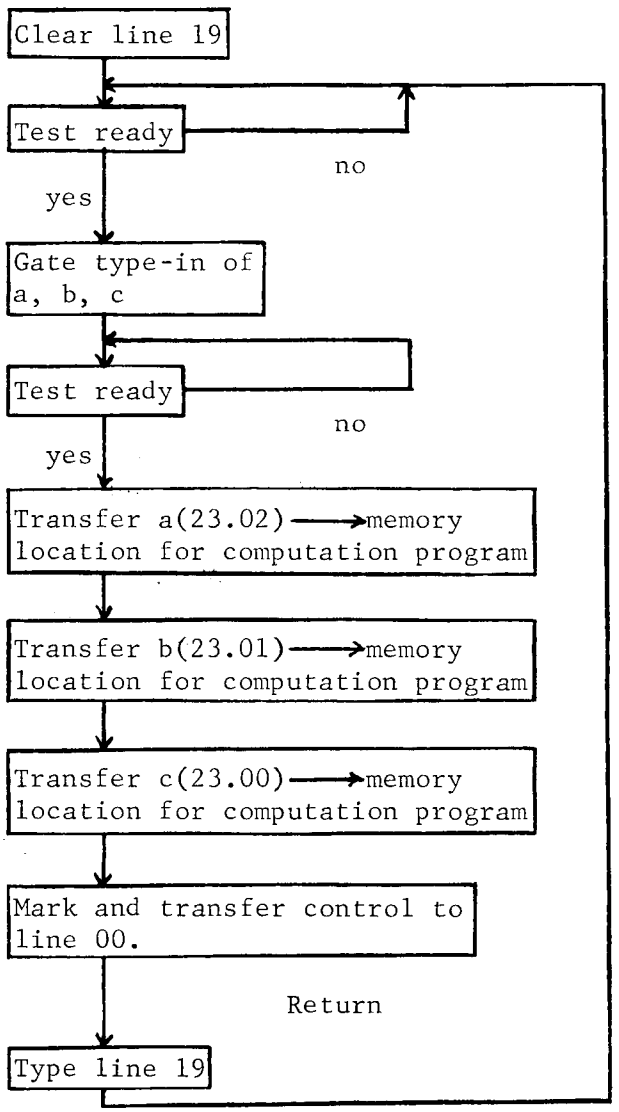
The G-15, although it can handle only one normal input at a time, has no interlock to prevent the initiation of another before the input/output system is "ready". In such a case, the input or output called for will be a logical sum of the two special (S) codes of the conflicting commands. In short, the results of allowing your program to make this mistake are disastrous. For example, if, during a type line 19 operation (S = 09), you executed a "gate" type-in command (S = 12), you would suddenly find you were, as far as the computer is concerned, requesting an input/output operation with S = 13, their logical sum. It just so happens that this special code for an input/output operation calls for "read magnetic tape". You might not even have a magnetic tape drive at your installation, but nevertheless, the computer would attempt to read from one. Of course it would never receive a stop code, and thus the attempted input would never terminate, to say nothing of the fact that neither your desired input nor your desired output will be accomplished.

$$\begin{array}{rl} 09_{(10)} & = 1001 \\ 12_{(10)} & = \underline{1100} \\ & 1101 = 13_{(10)} \end{array}$$

In order to prevent such distressing occurrences, you are equipped with a test command which can be incorporated in your program. It was mentioned earlier, but not defined: "test for 'ready'". If the input/output system <u>is</u> "ready", the next command will be taken from N + 1; if <u>not</u>, the next command will be taken from N. The most common use of this test is to set N = the location of the test itself, so that the test will be repeated until the input/output system is ready for a new operation, at which time the test will be met, and the program will proceed at N + 1. At this point another input or output might be called for. Another use of the ready test so programmed is to prevent the program from trying to use a set of inputs until they have been completely received. In order to achieve the most benefit from the ready test you will usually want to make it immediate, and let it be executed as often as possible. If N is set equal to L, you want the last word-time of execution to be L - 1, so that no delay will be involved waiting to take the next command from N. If the last word-time of execution is to be L - 1, the flag (T) in the test command must also equal L. The ready command then, programmed in this recommended way, will contain D = 31, S = 28, C = 0, T = L, N = L, and the command will be immediate. This command should precede all normal input or output commands.

There is a very important implication in what has just been said. The G-15 continues to operate your program during any normal input or output. In many computers, input and output time is "dead" time as far as computation goes, but this is not so in the G-15. While you are typing out one answer, for example, you can be computing the next. As a matter-of-fact, experienced programmers write programs in which almost none of the input/output time required is left unused as far as computation goes.

At this point we can develop another program, not very long, which will handle the inputs and outputs for our main computation program. Let's assume we want to type the answers we derive, x_1 and x_2, out of line 19. Since we will have to use a line 19 format (we already developed it; see page 137), we might just as well use command line 02 for this program. Thus, the program and its output format will occupy the same line.



In addition, we must include a command at the end of the program in line 00, a mark and transfer control command, which will transfer control back to this program at the correct point, for the type-out of the answers.

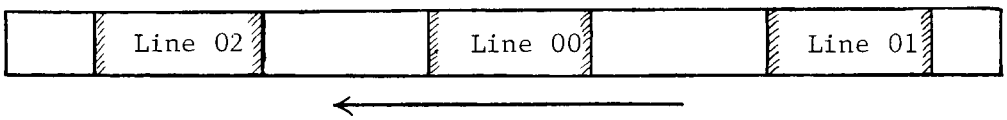
In the computation program, when we generate x_1, we will store it in 19.u7. Similarly, we will store x_2 in 19.u6, so that these numbers will be properly positioned, ready for output, prior to returning to the output portion of the program above.

Notice that the first thing this input/output program does is to clear line 19, so that it will always be clear when we attempt to type out our answers. If this were not done, and line 19 contained some garbage, we would type out this garbage as well as our valid answers during the first output.

The input/output program continues in a "loop". After completion of one run of the whole program, it immediately returns to an earlier command and eventually calls for type-in of a new set of a, b, and c. (N of last command = L of test.)

In this way, our program will continue forever, always calling for a new set if inputs after typing out the last set of answers. Of course we can stop this at any time we want by turning off the compute switch on the typewriter and walking away from the computer. More will be said later concerning loops and their uses.

Now we know that we can enter our program a line at a time, into the computer and punch out that line on tape. Suppose we punch line 01 (the square root subroutine) on tape. Then, on the same tape, we follow it with line 00, the main computation program. Then, still on the same tape, we follow line 00 with line 02. Finally when we run the tape out of the punch, we will have a long piece of punched tape containing three blocks, as shown in the drawing below, where the arrow indicates the direction in which the tape would be read.



BLANK "LEADER"

Notice that a blank space is located before the first block, between blocks, and after the last block. When a tape is mounted on the photo-tape-reader, it must have some leader which can be fed through the mechanism and onto the winding-spool, similar to the loading of a movie film. After an input, the drive mechanism coasts to a stop, and we don't want valuable information from the next block to slip past the photo-reader during this coast-time. The blank tape at the end will result simply from manually feeding the tape out of the machine, prior to tearing off the desired length, after it has been punched. Approximately 9" to 1' of blank tape should be left before the first block

as "leader", for initial winding purposes. Approximately 6" to 9" of blank tape should be left between blocks, to allow the tape drive mechanism to coast to a stop without allowing any information in the next block to slip past the photo-reader. This means that, when an input from punched tape is called for, an indeterminate length of blank tape will be "read" prior to the reading of any valuable information. The "reading" of blank tape will cause no input to the computer.

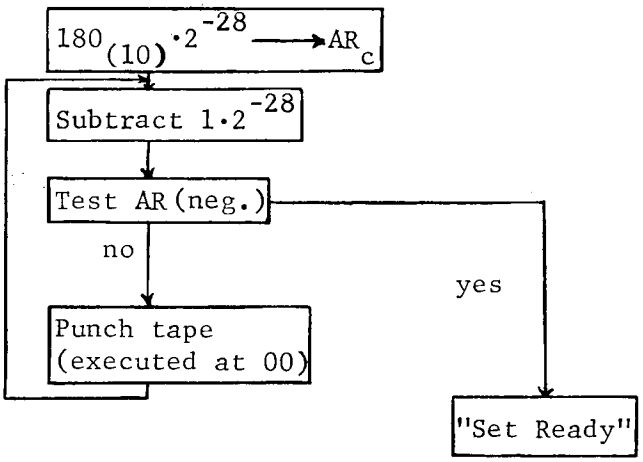
There is an acceptable method for generating blank leader automatically, under program control. This method is based on the fact that the "punch line 19" command not only initiates the punch operation, but also reloads the line 19 format (in line 02) into a four-word inspection buffer.

If the punch command is given before the end of the line 19 format has been reached during its inspection, the format will be automatically reloaded, and its inspection will begin anew, from the first format character. In this way, the end code in the format might never be reached, and the output would continue indefinately. As a matter of fact, if the punch command is repeated often enough, only the first character of the output format will ever be inspected.

Therefore, as tape is punched, a series of characters will be transmitted to tape; it will be a series of whatever is called for by the first format character. Of all data transmitted to the tape punch, the only one which causes no punch is a + sign. We therefore will cause a series of + signs to be transmitted, thus causing blank tape to be fed out of the punch. The first format character in the line 19 format (contained in line 02) must be a sign character, and the sign-bit of 19.u7 must be 0 (= +). In this manner, for as long as punching continues we will get only blank tape.

Ten strokes of the punch will yield one inch of tape. Two drum cycles are necessary for each punch stroke. Therefore, 20 drum cycles are necessary to generate one inch of blank tape. The generation of nine inches of blank tape would require 180 drum cycles.

In order to achieve this, the punch tape command should be executed at word-time 00 of every drum cycle for 180 drum cycles.

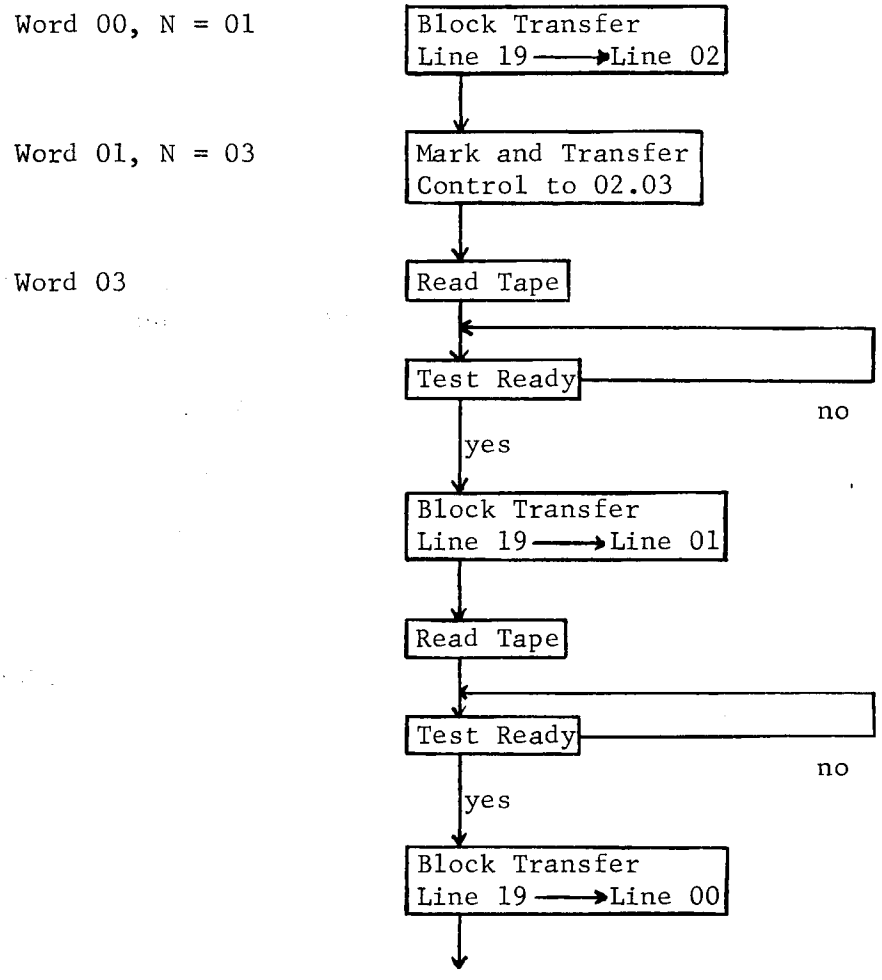


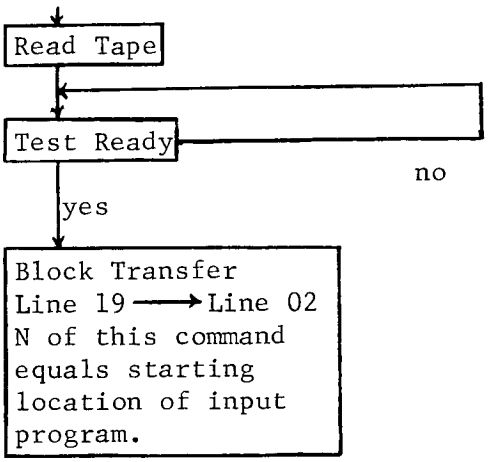
In the program flow-diagrammed on the preceding page, N in the punch tape command will equal the word-time in which the subtract command is located. The program will continue looping and counting the elapsed drum cycles by subtracting 1 from 180 for each drum cycle. Eventually AR will contain +0, and when 1 is subtracted from it, it will contain -1, the answer to the test will be "yes", and the program will exit from the loop. At this point the proper length of blank tape will have been punched.

You have noticed the use of a "set ready" command in the flow diagram. This is a special command with D = 31, S = 00, and C = 0. When this command is executed, whatever input or output is in progress will be automatically and arbitrarily stopped. No "stop" code will be punched on the tape. This command must be used with caution; it may shift the contents of line 19. Do not place the valid outgoing information in line 19 until after the set ready command has been executed.

LOADER PROGRAM

Suppose now, we entered the following program, and punched it on tape, and then we spliced this block of punched tape onto the other, preceding line 01.





This is called a "loader" program. Its value is derived from the fact that, if p is used to cause the reading of a block of tape, and if, after the input has ceased, the enable switch is turned off and the compute switch is turned to GO, the computer will take its next command from 23.00, which, as you know, will be the same as 19.00. Now we can trace the operation of this program.

The loader will be in line 19, and its first four words will also be in line 23, following the input caused by p.

When the compute switch is thrown to GO, the next command is taken from 23.00.

This command, word 00 of the loader program, causes the transfer of all of line 19 into line 02. The next command to be executed, still in the same command line, (command line 07, which is line 23) is at word 01.

This command is a mark and transfer control command, and control is transferred to line 02. The N number of this command selects the word-time of the first command to be read from line 02: it is word-time 03.

The program in line 02, beginning with word 03, will now be executed. But this is the loader program, itself. Word 03 calls for an input from punched tape. A preceding "ready" test is unnecessary, because it can be firmly predicted that no input or output is already in progress. The block of tape read into line 19 will be the next block on the program tape, following the loader. This is the block containing the square root subroutine, destined for line 01. Therefore, after this input is completed, line 19 is transferred, word-for-word, into line 01. Then the next block of tape is read, still under control of the loader program in line 02. This block of tape contains the main computation program, and therefore, upon completion of the input, line 19 is transferred into line 00. The remaining block of tape is then read into line 19. This block contains the input/output program designed to accompany the computation program, and is destined for

line 02. But line 02 is already in use; it contains the loader program. After the input is finished, the loader program will execute one more command, which will be its last. It calls for all of line 19 to be transferred into line 02. The loader program thus destroys itself, and line 02 contains the input/output program we desire. The command line is still line 02; nothing has been done to change that. The next command, as always, will be taken from the same command line, at a word-time specified as N of the previous command. Therefore, if, as indicated in the flow diagram, the N of the last command of the loader program equals the location of the initial command of the input/output program, during the next read-command time the input/output program will start its normal execution, just as we originally planned it.

A loader of this type is called, rightly enough, a "self-destroying" loader. Its purpose is to set up the memory of the computer for the operation of a given program completely, and yet occupy no part of memory after the set-up has been completed and it is no longer needed.

This type of procedure, involving the use of a loader program, is sometimes given the picturesque name of "bootstrap", for obvious reasons. Once such a tape has been mounted on the drive mechanism of the photoreader, the only actions necessary at the typewriter are:

1. with the compute switch off, put the enable switch on;

2. strike p;

3. after one block of tape has been read into the computer, make sure the enable switch is off, and move the compute switch to GO.

From that point, in our example, the rest of the program will pick itself up by its own bootstraps, enter the computer's memory at the proper locations, and proceed to operate until it reaches the point where it gates the type-in of the first set of a, b, and c. At this point the S and D neons on the face of the computer will not be flickering rapidly as step after step is executed, because always the same step is being executed. It is the "ready" test. The neons will remain steady, indicating D = 31, S = 28, and an input/output code of 12 (the code for a "gate type-in").

You will enter the numbers in the following order, as determined by the way we originally formulated the input program: a (tab) b (tab) c (tab) s. Each number will consist of seven hex digits and a sign. The s will set the input/output system "ready", the test on which the program was "hung up" will be met, and the program will proceed. Provided the numbers entered don't generate erroneous results, the computation program will place the two answers in 19.u7 and 19.u6 and transfer control back to the input/output program, which will type them out in the following order: x_1 (tab) x_2 (carriage return). It will then hang up again on the "ready" test, awaiting a new set of inputs. It will keep on performing this cycle until we simply don't supply any more inputs.

Notice that we will have to convert decimal numbers for a, b, and c, to hex numbers, prior to the input, and that, when these hex numbers are typed in, they will be scaled, to our knowledge, 2^{-21}. The decimal numbers should therefore be converted to binary, rather than hex, 21 bits being allowed for the expression of the integral value, and 7 bits for the fractional value. From the resultant series of bits, a corresponding hex number can very easily be made up, and it will be this number that will be typed in.

The output will also be in hex, representing a binary value scaled 2^{-21}. This binary value must be converted to its decimal equivalent, which can be done quite easily, by inspection. A table of corresponding powers of 10 and 2, as well as corresponding powers of 10 and 16, is located in the back of this book (page 207).

Only two tasks remain to be performed before we can punch a complete program tape of the type described above.

One is to choose word-locations in the appropriate command lines for all of the necessary commands. Because of the nature of the memory in the G-15, as has been pointed out previously, timing becomes a consideration in the writing of a program. We wish to minimize the amount of wait-time preceding both the reading of commands and their execution, in order to enable the program, as a whole, to operate in the shortest amount of time possible. We therefore will have to choose wisely the locations into which we place the commands, the times at which we will execute them, the words in which we store constants used by our program, and its inputs.

The other task is to code, in binary, each command and constant that our program needs. We know the binary make-up of a command, so this will not be difficult. From the binary number, we will have to get a hex number which can be typed into the computer. When this has been done for all the words in a line, that line's contents will be in line 19. We can then punch a block of tape. We must repeat this for each block of tape necessary. Although this is not a difficult task, it is time-consuming.

PROGRAM PREPARATION ROUTINE (PPR)

Fortunately, Bendix Computer Division has developed a program, called the Program Preparation Routine (PPR), which will do this. As inputs, it needs commands composed of decimal numbers for T, N, C, S, and D, in the form shown below.

T N C S D

The decimal number for T will contain two digits, ranging from 00 through u7. Similarly, that for N will also contain two digits, within the same range. The decimal number for C will contain one digit, ranging from 0 through 7. The decimal number for S will contain two digits, ranging from 00 through 31. Similarly, for D, the decimal number will contain two digits, within the same range.

PPR needs to be told the location for the command, as well as being given the "decimal form" of the command. From these two facts, it will cause the proper binary command to be entered into the specified word-time of the line it uses for storage. This is line 18. Finally, line 18 will contain all of the commands we want to appear in, say, line 00, at the appropriate word-times. We can then give PPR a command to punch a block of tape with this line. It will transfer line 18 to line 19, and then punch line 19 on tape. Now we can give PPR another command to clear line 18, and then proceed to set up a new line of our program in the same manner. Eventually, we will generate the entire program tape we desire.

When reference is made to giving PPR a command to do something, the question of how this is done arises. PPR gates type-in after completion of every operation. At that time, the operator types in what we call a "pseudo-command" which can be recognized by PPR, telling it what is desired next. Some of these are:

"accept a decimal command input, and store it in word __";

"punch out the program now stored in line 18";

"clear line 18".

There are many more such pseudo-commands for PPR, and they, along with PPR's functions and capabilities, are discussed elsewhere in this book.

At the present time, assume the availability of such pseudo-commands, and we will proceed to see how the commands of a program, in particular, the program we have developed, are coded, in decimal form, prior to being supplied to PPR.

The decimal form has already been shown, but some additions must be made to it:

P T N C S D BP

P is a "Prefix". PPR does not know whether a command, as coded in decimal form, is intended to be an immediate or a deferred command. It will assume that all commands with D $\neq$ 31 will be deferred, and all commands with D = 31 will be immediate. If this assumption is correct as pertaining to a command, the P is left blank. If it is incorrect, in that you wish to make a command with D $\neq$ 31 immediate, you must supply a P of "u". If it is incorrect in that you wish to make a command with D = 31 deferred, you must supply a P of "w".

BP stands for breakpoint. If you wish a command to be breakpointed, you must supply a minus sign at this point. If you do not wish the command to be breakpointed, you supply no sign, which is tantamount, as we have seen, to supplying a plus sign.

Usually the commands of a program are written on standard, printed forms, called "coding sheets". The following pages contain our program, its loader, and line 02, coded on such sheets. Notice that the location of each command and constant needed by our program is also specified, along with the command or constant, itself. This location will be supplied to PPR, but not as an integral part of the coded command. You will see that most of the commands are explained somewhat in a "NOTES" column on the sheet, just as a matter of convenience.

Pay close attention to the timing numbers, T and N in each command, and how they have been chosen to reduce wait time during the operation of the program.

Following the coding sheets, there will be some discussion of the timing numbers chosen for individual commands.

You will notice that no coding for line 01 is included. This is the square root subroutine. It has already been written, and we will use it in its present form. We will simply reproduce a block of tape containing that subroutine, and include this reproduction in our own program tape.

Bendix Computer — Los Angeles 45, California

G-15 D
PROGRAM PROBLEM: Loader

Prepared by _____

Page 1 of 6
Date: _____
Line 02

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	

L	P	T or Lk	N	C	S	D	BP	NOTES
00	u	01	01	0	19	02		line 19 ⟶ line 02
01		03	03	2	21	31		Mark, Transfer ⟶ 02.03
03		05	05	0	15	31		Read Tape
05		05	05	0	28	31		Test Ready
06	u	07	07	0	19	01		line 19 ⟶ line 01
07		09	09	0	15	31		Read Tape
09		09	09	0	28	31		Test Ready
10	u	11	11	0	19	00		line 19 ⟶ line 00
11		13	13	0	15	31		Read Tape
13		13	13	0	28	31		Test Ready
14	u	15	00	0	19	02		line 19 ⟶ line 02
								(next command 02.00)

FORM BCD 108-5.0

Bendix Computer — Los Angeles 45, California

G-15D
PROGRAM PROBLEM: Computation

Prepared by _____

Date: _____

Line 00

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	

L	P	T or L_k	N	C	S	D	BP	NOTES
00		03	03	0	23	31		Clear 2-wd. Registers
03		05	06	0	21	25		a = (21.01) $\longrightarrow ID_1$
06		07	09	0	00	24		2 = (00.07) $\longrightarrow MQ_1$
09		56	66	0	24	31		Multiply
66		68	71	4	26	24		$PN_{0,1} \longrightarrow MQ_{0,1}$
71		04	76	0	26	31		Shift MQ left 2 bits
76		77	78	0	24	20		2a = (MQ_1) $\longrightarrow$ 21.01
78		81	81	0	23	31		Clear 2-wd. Registers
81		85	86	0	20	25		2a = (20.01) $\longrightarrow ID_1$
86		07	11	0	00	24		2 = (00.07) $\longrightarrow MQ_1$
11		56	68	0	24	31		Multiply
68		70	73	4	26	24		$PN_{0,1} \longrightarrow MQ_{0,1}$
73		04	79	0	26	31		Shift MQ left 2 bits
79		81	82	0	24	28		$MQ_1 \longrightarrow$ AR (4a)
82		85	85	0	23	31		Clear 2-wd. Registers
85		87	88	0	28	25		4a = (AR) $\longrightarrow ID_1$
88		92	91	0	23	24		c = (23.00) $\longrightarrow MQ_1$
91		56	40	0	24	31		Multiply
40		42	44	4	26	20		4ac = ($PN_{0,1}$) $\longrightarrow$ 20.02,03
44		47	47	0	23	31		Clear 2-wd. Registers
47		49	50	0	22	25		b = (22.01) $\longrightarrow ID_1$
50		53	55	0	22	24		b = (22.01) $\longrightarrow MQ_1$
55		56	05	0	24	31		Multiply
05		07	04	0	29	31		Test Overflow
04		06	08	7	20	30		4ac = (20.02,03) $\longrightarrow$ PN+
08		10	12	0	29	31		Test Overflow

FORM BCD 108-5.0

Los Angeles 45, California

G-15D
PROGRAM PROBLEM: Computation

Prepared by _____

Page 3 of 6
Date: _____
Line 00

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	

L	P	T or L_k	N	C	S	D	BP	NOTES
12		14	14	1	26	28		$PN_0 \xrightarrow{+}$ ARc
13		15	00	0	16	31		Halt
14		16	16	0	22	31		Test for sign of AR (neg.)
16		18	19	0	00	28		Return Command±(00.18) ⟶ AR
17		19	00	0	16	31		Halt
18		u0	99	0	20	31		Return Command
19		21	22	0	20	03		2a = (20.01) ⟶ 03.21
22		25	26	0	22	03		b = (22.01) ⟶ 03.25
26	w	20	94	1	21	31		Mark, Transfer ⟶ 01.94
20		21	23	0	03	20		2a = (03.21) ⟶ 20.01
23		25	61	0	03	22		b = (03.25) ⟶ 22.01
61		63	60	0	29	31		Test Overflow
60		00	27	0	00	00		Go to 27
27		29	30	3	22	28		b = (22.01) $\xrightarrow{-}$ ARc
30		31	32	0	20	29		$\sqrt{}$ = (20.03) ⟶ AR+
32		34	34	0	29	31		Test Overflow
34		35	36	1	28	28		AR $\xrightarrow{+}$ ARc
35		37	00	0	16	31		Halt
36		39	39	0	23	31		Clear 2-wd. Registers
39		41	43	0	28	25		AR ⟶ ID_1
43		42	87	0	26	31		Shift ID right 21 bits
87		89	90	0	25	23		N = (ID_1) ⟶ 23.01
90		93	94	2	23	28		|N| = |(23.01)| ⟶ ARc
94		95	96	0	28	23		|N| = (AR) ⟶ 23.03
96		97	98	2	20	28		|D| = |(20.01)| ⟶ ARc
98		u1	u2	0	28	23		|D| = (AR) ⟶ 23.01

FORM BCD 108-5.0