Bendix Computer — Los Angeles 45, California

G-15D
PROGRAM PROBLEM: Computation

Prepared by _____

Page 4 of 6
Date: _____
Line 00

0	1	2	3	L	P	T or L_k	N	C	S	D	BP	NOTES
4	5	6	7	u2		u3	u4	0	23	28		N = (23.03) ⟶ ARc
8	9	10	11	u4		u5	u6	3	23	29		N = (23.01) $\xrightarrow{-}$ AR+
12	13	14	15	u6		00	01	0	22	31		Test for sign of AR (neg.)
16	17	18	19	01		03	00	0	16	31		Halt
20	21	22	23	02		04	10	4	25	21		N = $(ID_{0,1})$ ⟶ 21.00,01
24	25	26	27	10		13	15	0	23	31		Clear 2-wd. Registers
28	29	30	31	15		17	21	0	20	25		2a = (20.01) ⟶ ID_1
32	33	34	35	21		24	29	4	21	26		N = (21.00,01) ⟶ $PN_{0,1}$
36	37	38	39	29		57	89	1	25	31		Divide
40	41	42	43	89		30	u5	0	24	28		MQ_0 ⟶ ARc
44	45	46	47	u5		u7	24	0	28	19		AR ⟶ 19.u7
48	49	50	51	24		25	28	3	22	28		0 = (22.01) $\xrightarrow{-}$ ARc
52	53	54	55	28		31	33	3	20	29		$\sqrt{\ }$ = (20.03) $\xrightarrow{-}$ AR+
56	57	58	59	33		35	37	0	29	31		Test Overflow
60	61	62	63	37		38	41	1	28	28		AR $\xrightarrow{+}$ AR
64	65	66	67	38		40	00	0	16	31		Halt
68	69	70	71	41		44	45	0	23	31		Clear 2-wd. Registers
72	73	74	75	45		47	49	0	28	25		AR ⟶ ID_1
76	77	78	79	49		42	92	0	26	31		Shift ID right 21 bits
80	81	82	83	92		93	95	0	25	28		ID_1 ⟶ AR
84	85	86	87	95		96	97	2	28	28		|N| = |(AR)| ⟶ ARc
88	89	90	91	97		u1	u3	3	23	29		|D| = (23.01) $\xrightarrow{-}$ AR+
92	93	94	95	u3		u5	51	0	22	31		Test for sign of AR (neg.)
96	97	98	99	51		53	00	0	16	31		Halt
u0	u1	u2	u3	52		54	56	4	25	21		N = $(ID_{0,1})$ ⟶ 21.02,03
u4	u5	u6		56		59	59	0	23	31		Clear 2-wd. Registers

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
59		61	62	0	20	25		2a = (20.01) ⟶ ID_1
62		66	69	4	21	26		N = (21.02,03) ⟶ $PN_{0,1}$
69		57	25	1	25	31		Divide
25		26	31	0	24	28		MQ_0 ⟶ ARc
31		u6	48	0	28	19		AR ⟶ 19.u6
48		50	50	2	21	31		Mark, Transfer ⟶ 02.50

FORM BCD 108-5.0

Bendix Computer — Los Angeles 45, California

G-15D
PROGRAM PROBLEM: Input/Output

Prepared by _____

Page 6 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 L_k	N	C	S	D	BP	NOTES
02								0000011
03								800000x
00		01	04	2	28	28		Clear AR
04		05	06	3	28	29		
06	u	07	07	0	28	19		Clear line 19
07		07	07	0	28	31		Test Ready
08		10	10	0	12	31		Gate Type-in
10		10	10	0	28	31		Test Ready
11		14	15	0	23	28		a = (23.02) → ARc
15		17	18	0	28	21		a = (AR) → 21.01
18		21	22	0	23	22		b = (23.01) → 22.01
22		28	00	0	21	31		Mark, Transfer → 00.00
50		52	07	0	09	31		Type line 19

FORM BCD 108-5.0

At this point let's discuss the program as it is coded. The first part of the program to be operated will be in the loader. With the entire program tape, generated in a manner already described, loaded on the photo-reader mechanism, you will strike the p key, and one block of tape (the loader) will be read into line 19. Striking the p key also sets the computer to take its next command from word 00 of command line 7 (line 23). This word will, of course, be the same as word 00 in line 19, and, as a matter of fact, so will words 01, 02, and 03. The command located at 23.00 is a "block copy" of all of line 19 into all of line 02. Remember the function of the T number in an immediate command: it is to serve as a flag, stopping the execution of the command after word-time T - 1, and before word-time T. Word-time T, in this case, 01, will be the first word-time following the end of the execution of the command at 23.00, and it therefore is the first word-time available for the reading of another command. No time will be lost while the computer waits to read the next command if it is located in word 01. This, then, explains the choice of word 01 as N in the command at 23.00. Of course, the next command in the program must be located at word 01 in the same command line. It is, and it is a "mark and transfer control" command, causing the computer to take its next command from word 03 in line 02. This is also an immediate command, and must be executed for one word-time, 02. Thus 03 is the best possible location for the next command, and that's why it was chosen as N in the command located at 23.01.

Now, notice that line 19 contains, in words 00, 01, 02, and 03, the same four words that are contained in line 23, and that line 02 contains the same words as line 19. Therefore, words 00, 01, 02, and 03, in line 02 are the same as the four words in line 23. So command 03 in line 02 is the command as shown on the coding sheet for the loader. At 03 a command from line 02 will be read which will call for the reading of a block of tape, and the program will continue to 05, the next command. Up to this point, no time has been lost in either waiting to read the next command or in waiting to execute any command.

The command located at 02.05 will cause the computer to test for the normal input/output system "ready". If it is not ready (there is an input or an output in progress), the answer to the question asked of the computer will be "no", and the program will continue at N. Remember that the G-15 will continue to compute while an input or output is in progress. Since N of the test command is 05, the test will be repeated, over and over again, until the input/output system is ready, which will be the case only when the block of tape has been read. Notice that the computer has not stopped operating; it is merely repeating the same command until a given condition exists, at which point it will go on to a new command, and the only reason it is doing this is that this is the way the program was written. Notice that the ready test is also an immediate command, executing during word-times 06 through 04, when it is stopped by the flag (05) in T. 05 is therefore the next available word-time for reading a command, and N equals 05, so there will be no lost time in waiting to read the next command.

Further study on your own should indicate that, because of the T's and N's chosen in the commands of the loader, there will be no wait-time of either variety involved in the operation of this part of the program. Eventually, at word-time 14, another immediate command will be read, and this one, during word-times 15 through 14, will "block copy" all of line 19 (at this point containing the input/output portion of the program) into all of line 02, destroying the loader, which has, at this point, outlived its usefulness. Here the first wait-time is encountered. The last word-time of execution of this command is 14, so the next available word-time at which the next command could be read will be 15. But the next command is called for from word 00 of line 02 (the program continues in the same command line, although the contents of that line have changed). Word-times 15 through u7 will be lost time during the current drum cycle, while the computer waits to read the next command (at 00).

The next command in the program (02.00) calls for placing the magnitude of AR's present contents in AR. Its execution time will be 01, and no time will be lost in waiting to execute. You might ask yourself, looking at the coding for 00 in the input portion of the program, in line 02, why a drum cycle would not be lost. It has been stated previously (page 151) that PPR will make all commands with D $\neq$ 31 deferred. And a long time ago (page 69) it was stated that, when the computer is directed to defer execution of a command until the word-time indicated in T, it must wait one word-time at least before it can execute the command. If, then, a deferred command is given at word-time 00, to be executed in 01, why is it that the computer will not have to wait until 01 in the next drum cycle to execute the command? The reason is that, the authors of PPR, thinking all the time, were one jump ahead of you. When PPR is directed to make a command deferred (D $\neq$ 31), it first tests to determine the relationship between T and L. If T is one greater than L, PPR increases T by one ($T = L_2$) and makes the command immediate. In other words, the command at 00 of the input part of the program, could have been coded in the following form, and the same effect in the resultant machine-language program would be achieved:

L	P	T	N	C	S	D
00	u	02	04	2	28	28

Although no time will be lost waiting to execute this command, notice that two word-times will be lost waiting to read the next command, since it is called for at 04, even though the next available word-time for its location will be 02. The reason for this is that line 02 must contain, as well as this part of the program, the output format, which has already been discussed, and which in this case, occupies words 02 and 03 of the line. Obviously no command can be stored in either of these locations without disturbing part of this format.

You should be able to carry out the rest of this analysis of the program, as coded, for yourself, spotting any wait-to-execute or wait-to-read command time. Notice that an attempt has been made to cut down on this wasted time as much as possible, although this program still wastes quite a bit of time. You might devise for yourself ways in which to cut down

on this waste, by arranging the incoming data (a, b, and c) differently, performing some of the operations in a different order, using different temporary storage locations, etc.

Notice that, towards the end of the computation portion of the program, in line 00, the amount of wait-time increases, due to the increasing unavailability of storage locations for the necessary commands.

This program is not the most efficient method of arriving at the solution for the roots of a quadratic equation; it is straight-forward, however, and coincides with the original flow-diagram for the solution of the problem, as developed on pages 122 - 126.

It was operated with the inputs:

a = 1 (0000080)

b = 1 (0000080)

c = -6 (0000300-)

and the results were:

x_1 = 0000101 = $2(2^{-21})$ + Princeton round-off (from division)

x_2 =-0000181 =$-3(2^{-21})$ + Princeton round-off (from division).

The Program Preparation Routine, as it appears on punched tape, is very long, consisting of many blocks of information. Of all these blocks, four are basic, in that they form the heart of the routine. With them you can prepare all of the commands in machine language for a long line; you can also enter hex constants into the long line at any desired word-time; you can punch a tape of the line's contents; and you can set up almost all of memory in any way you wish (not all of memory, however, because PPR has been written to protect itself, and, if you command it to destroy part of itself, it will reject your command).

The line in which PPR stores the commands and constants you give it is line 18. Word-times will be fixed in this line. The whole line itself may, however, be placed in any other long line in the memory of the G-15, with the exceptions of lines 05, 15, 16 and 17, since these are the lines occupied by the basic package of PPR. Thus, we could use PPR to enter the proper machine-language commands at proper word-times in line 18, along with the properly located hex constants, and then cause PPR to copy line 18 into line 00, thus setting up the main computation part of our program. We could also cause PPR to punch a block of tape containing this line of program, so that it would be preserved in its machine-language form, relieving us of the necessity of having to re-type all of the individual commands and constants, in case it is ever desired to re-enter the program into the computer.

When PPR has been read into the memory of the computer (p, then GO), its four basic lines will be occupying lines 05, 15, 16, and 17, and the computer will be "hung up" on a ready test, gating type-in. PPR is waiting for a command from you to do something.

The first thing you would probably want PPR to do is to clear any already existing garbage out of line 18; you would probably want to do this just prior to making up any line of program or constants. The command which will tell PPR to do this is x00 (tab) s. The neons on the front panel of the computer will flicker momentarily, as line 18 is cleared.

PPR, after performing an indicated task, will always return to that point where it gates type-in of a new command, eager to do more work. The next thing you might want to have it do is start a series of commands, accepting your commands in their "decimal command" form, converting them to their true binary machine language form, and storing them in line 18 at the proper word-times. PPR can be told to start such a series of commands at word-time ab by the command, yab (tab) s. It will immediately type a carriage return and L = ab, and then gate type-in of the command, itself. When the command has been typed in, followed by (tab) s, PPR will convert the various portions of it, make up the proper machine language command, and store it in word ab of line 18. On the next line, on the typewriter, PPR will then type a new L, equal to the N of the command just entered. PPR will be unable to notice the end of such a sequence of commands, and, at some point, after a new L has been typed out, you will not enter another decimal command to be converted and stored, but, instead, you will enter another command to be interpreted by PPR as telling it what to do next. PPR will be able to recognize this new command; there is no danger of it trying to incorporate it as another machine language command in the program you are making up.

This next command you give PPR might be to reproduce, on punched tape, the present contents of line 18; in this case it would be x06 (tab) s. All of the words in line 18, 108 of them in all, will be added, regardless of overflow, and the sum, called a "check sum", because it can be used to check for accurate read-in of the tape later, will be typed out, after which the contents of the line will be punched on tape. In our previous discussion of typing out or punching the contents of line 19, it was pointed out that the output will end when the end code of the output format is sensed, and a resulting check of all of line 19 indicates that the whole line is clear. PPR block copies all of line 18 to line 19, and then executes a command calling for the contents of line 19 to be punched on tape, under control of an "abbreviated" format (DDDDDDDDDDDDDDDDDDDDDDDDDDDDDE), which calls for four words of output at a time. If each D calls for an output character representing four bits, 29 D's will "cover" 116 bits = four full words. This tape will be relatively unintelligible upon inspection but will serve as interim storage of all bits. When it is read into line 19, line 19's original setting, bit-by-bit, will be reproduced. If words 00, 01, 02, and 03 in line 18 are all clear, but there is some non-zero information in the next higher group of four words, the last word of output would be word 04.

When such a tape is read into the computer, the last word from tape will remain in 00 of line 19, leaving the first word of input in word u3 of line 19. This would not be too good, since all the words in the line would be removed from their correct location by four word-times. In order to avoid such an embarrassing state of affairs, PPR will, when called upon to punch line 18's contents on tape, check words 00 through 03 for non-zero. If they contain only zeroes, PPR will insert two bits in word 03, so that its hex form will be 4400000. Thus, as the output continues, every time the end code of the format is reached, a check of line 19 will yield non-zero, causing output to continue for four more word-times, until, finally, words 03 down through 00 will be punched on tape, at which point line 19 will be entirely cleared to zero, and the output will cease. This will assure us of a tape, which, when read into the computer, will load word 00, and therefore all other words, correctly. Remember that when such a tape is read, word 03, although you originally set it with nothing, will now contain the hex number 4400000.

"PRECESSION", AS USED BY PPR

When you call for the output of line 18's contents, there may be many full words of zeroes at the high end of the line. Output of these words is unnecessary since they need not be filled, during input, to guarantee that the non-zero words will be set correctly. If only the last four words contain non-zero information, only four words will be punched on tape. When this tape is read into the computer, only the last four words of line 19 will be set, but they will be properly set. PPR gets rid of words containing all zeroes prior to punching out the contents of line 18, by four-word groups. As soon as the first non-zero four-word group, counting from u7 down, is encountered, output is initiated. To understand how PPR eliminates words from a line four-at-a-time, you must understand the effect of the following command, where both S and D are less than 28, and C = 2:

L	P	T	N	C	S	D
L	u	L_k	N	2	S	D

Consider first a short line. S = D = 20, T = L_6. A C code of 2 calls for an exchange of AR with memory. During each word-time of execution, (AR) ⟶ D.T and (S.T) ⟶ AR. Let the command be:

L	P	T	N	C	S	D
00	u	06	N	2	20	20

word-time of execution:	AR holds contents of:	this word in line 20 receives:	AR receives:
01	(AR)	(AR)	(01)
02	(01)	(01)	(02)
03	(02)	(02)	(03)
04∼00	(03)	(03)	(00)
05∼01	(00)	(00)	(AR) = (01)

The contents of line 20 have moved up one word-time: what was in 00 is now in 01, what was in 01 is now in 02, what was in 02 is now in 03, and what was in 03 is now in 00. AR's original contents have been restored to AR.

Notice that a fifth word-time of execution is necessary, even though we are moving four words. It is necessary to complete the cycle and restore AR.

In the case of a long line, where 108 words are to be moved up, it is impossible to get a 109th word-time of execution in one immediate command. Yet this word-time of execution is necessary, to complete the cycle and restore AR. Consider the command,

L	P	T	N	C	S	D
00	u	01	N	2	19	19

word-time of execution:	AR holds contents of:	this word in line 19 receives:	AR receives:
01	(AR)	(AR)	(01)
02	(01)	(01)	(02)
.	.	.	.
.	.	.	.
.	.	.	.
u7	(u6)	(u6)	(u7)
00	(u7)	(u7)	(00)

All 108 words in long line 19 have been moved up one word-time with the exception of word 00, which never reached 01, although it would have, had the execution been continued for one more word-time. AR currently holds the contents of word 00, and its original contents are in word 01, where we desire to place the contents of word 00. One more word-time of execution, during 01, would cause the contents of AR and 19.01 to be exchanged, restoring AR with its original contents, and placing the contents of 19.00 in 19.01. Therefore, the command above, in order to achieve the desired effect on the entire long line, must be followed by one more command, which exchanges the contents of AR and 19.01. Why can't one immediate command be executed for 109 word-times? This question will be left for you to answer.

Let the above command be followed by another:

L	P	T	N	C	S	D
01		01	N	2	19	19

These two commands, taken together, will cause all of line 19 to be moved up by one word-time, and AR to be restored. Notice that a whole drum cycle will, of necessity, be lost in wait-time, either waiting to read the next command or waiting to execute a command.

It is essential that no commands affecting either AR or 19.01 intervene between these two. For that reason you should try to keep them together as a pair.

It is interesting to note that a long line can be moved up two word-times, with the cycle completed and AR restored, in exactly the same number of commands and time:

L	P	T	N	C	S	D
00	u	01	01	2	19	19
01	u	02	N	2	19	19

word-time of execution:	AR holds contents of:	this word in line 19 receives:	AR receives:
01	(AR)	(AR)	(01)
02	(01)	(01)	(02)
.	.	.	.
.	.	.	.
.	.	.	.
u7	(u6)	(u6)	(u7)
00	(u7)	(u7)	(00)
---	---	---	---
02	(00)	(00)	(02)=(01)
03	(02)=(01)	(01)	(03)=(02)
.	.	.	.
.	.	.	.
.	.	.	.
00	(u7)=(u6)	(u6)	(00)=(u7)
01	(00)=(u7)	(u7)	(01)=(AR)

All words in the long line have been moved up two word-times, and AR has been restored.

Therefore, since PPR desires to eliminate words containing all zeroes by groups of fours, from the high-order end of line 19, it is done through pairs of commands of the type shown above. Two such pairs move all words in any long line up by four word-times. By assumption, if PPR tests and finds such may be done, four words containing zeroes will move "up" from 19.u4 - u7 to 19.00 - 03.

The movement of words through a line in this fashion is called "precession".

OTHER PPR OPERATIONS AVAILABLE

Before commanding PPR to punch a tape containing the contents of line 18, you may want to place some hex constants in the line at strategic locations. The command which tells PPR to accept a seven-digit hex number and store it at location ab in line 18 is: zab (tab) $\pm$ $d_1d_2d_3d_4d_5d_6d_7$ (tab) s. Of course, this must be repeated for each constant to be so entered.

If, in accepting a sequence of decimal commands, PPR finds that a location which is about to receive a new command is already filled, PPR will type out the contents of that location before accepting the new command. You may disregard this typeout and proceed to enter the new command into the location, if you desire to destroy the word currently contained there. If you desire to keep that word, you may, of course, choose a new location for the command you desire to enter, merely by giving PPR a new yab (tab) s command, specifying a new ab. If the last yab, which started the current sequence of commands, was followed by a minus (-) prior to the (tab) s, the contents of any location called for which is found to be non-zero, will be converted by PPR to decimal command form prior to the typeout. If, in the last yab (tab) s command, there was no minus inserted, the contents of any such location will be typed out as a hex number.

If you cause PPR to enter a hex number in any location, it will enter the number, and then it will type out the location into which the number was entered, followed by a typeout of the original contents of that location, if non-zero.

After a line of program and/or constants is made up in line 18, PPR may be commanded to block copy all of line 18 into any other line of memory, excepting lines 05, 15, 16, and 17, all of which are occupied by PPR itself (PPR will refuse to destroy itself). The command which will cause PPR to do this is klx03, where kl stands for the desired line number.

If you find a mistake in a line of your program and wish to correct it through the use of PPR, you can cause PPR to block copy any line of memory into line 18, and, as a bonus, to type out any given word in that line. The command to do this is: klijx02 (tab) s, where line kl will be copied into line 18, and word ij will be typed out. If a minus precedes the (tab) s, the word typed out will first be converted to decimal command form; otherwise, it will be typed out as a hex number.

PPR may also be commanded to type out the entire contents of line 18: x05 (tab) s. All words will be typed out as hex numbers.

Line 18 will be typed out, and, at the same time, punched on tape, if PPR is given the command, x07 (tab) s.

PPR will punch a block of tape containing the number track if it is given the command x01 (tab) s.

PPR will read a block of tape which you have mounted on the photo-reader mechanism if given any one of the following commands:

w00 (tab) s ..... read tape, type check sum

w01 (tab) s ..... read tape, type check sum, type out

w02 (tab) s ..... read tape, type check sum, punch tape

w03 (tab) s ..... read tape, type check sum, type and punch

In any of these cases, the contents of the block of tape read will be block copied into line 18. In all of these cases, if the compute switch is on GO, after the operation called for has been performed on one block of tape, a new block will be read and the same operation will be performed on it. This will continue until all the tape has been read, and, eventually, the photo-reader will be activated, even though no tape is passing through it, in which case, the input will never cease, since no stop code will be read. If the compute switch is thrown to BP, PPR will stop at a breakpointed command in the read tape sequence of commands which follows the reading of the current block.

If PPR is stopped at any time in this manner, remember that, if the compute switch is thrown to GO, it will immediately continue in the same sequence of steps. To return to that portion of PPR which gates the type-in of PPR commands, strike sc5f. As a matter of fact, if, for any reason, you are not currently in PPR, but in some other program, and you know that PPR remains in memory intact, you can always return to that portion of PPR which gates type-in of PPR commands through this keyboard action.

Finally, PPR can be commanded to transfer control to any line from 0 through 4, at any word-time, by the command, cijx04 (tab) s, where c = 0 through 4, and ij is the location of the desired command in that line.

Other operations are possible with PPR, including certain auxiliary operations, of primary interest in three main areas:

1. automatic compilation of loader programs;

2. automatic compilation of output formats;

3. check-out and correction of programs, including listing of commands and constants.

PPR and all its auxiliary routines are thoroughly discussed in the Bendix G-15 operating manual.

DECIMAL NUMBER INPUTS AND SCALING

If PPR is capable of taking decimal numbers (for the various parts of a command), and generating the proper binary equivalents in the machine, it stands to reason that, we, too, could write our program to do this, so that we would not have to convert each decimal input to binary and then to hex prior to typing it in. Let's figure out a method for accepting decimal inputs, rather than requiring hex inputs.

Obviously, the keys on the typewriter, since there is one for each hex digit, will be sufficient for the decimal digits 0 - 9. If a digit is struck, we know that four bits will enter line 23, word 00. If a

complete seven-digit decimal number and its sign are entered, the signed number will appear in 23.00. But, unlike a hex entry, it will not be in the form of a binary magnitude and sign; it will be in a code, each four bits corresponding to a digit of the decimal number. The four most significant bits in 23.00 will contain a four-bit code for the most significant decimal digit; the next four bits will contain a similar code for the next decimal digit entered, and so on. This four-bit code for decimal digits is called "Binary-Coded-Decimal". Remember that a complete BCD (Binary-Coded-Decimal) number, signed, as it occupies 29 bits of a word in the computer, is not the binary equivalent of the represented decimal magnitude, signed. As an example, suppose we entered the decimal number, (7196323-) into the computer, digit-by-digit. In 23.00 we would have:

01110001100101100011001000111

representing

7 1 9 6 3 2 3 -

The problem now is to convert this code into a true, corresponding, binary magnitude. Let's treat the seven-digit decimal number entered as an integral number, so that

$$7196323 = 7\cdot10^6 + 1\cdot10^5 + 9\cdot10^4 + 6\cdot10^3 + 3\cdot10^2 + 2\cdot10^1 + 3\cdot10^0.$$

Now consider the first four bits of the BCD number in 23.00: they represent a multiple of 10^6. If we had, stored in the computer's memory, as a constant, a binary value for 10^6, we could multiply that value by the four-bit multiplier (in this case, $7_{(10)}=0111$). Similarly, we could multiply each digit times the corresponding binary equivalent of a power of 10, and, when we added all these products together, the sum would be the binary equivalent of the original decimal number. Because the decimal number was integral, so would the binary equivalent be integral. In order to prevent overflow in the additions, we would have to be sure to scale each individual product in such a way that their sum could not possibly overflow out of the most significant bit-position in the accumulator. The maximum decimal number which could be specified is 9999999. Since this is less than 2^{24}, twenty-four bits would be sufficient to express it. Therefore, if the individual products are scaled 2^{-24} in the computer, no overflow can result from their sum; of course they will all have to be scaled similarly. It is convenient, however, to scale a converted decimal integer 2^{-28}, for other reasons, which will be discussed later. This will be alright; it merely means that the binary equivalent will have at least four leading 0's.

The first four bits of the BCD number to be converted to binary are, of course, scaled 2^{-4}. They represent, in our example, 7. If we scale the binary equivalent of 10^6 by 2^{-24}, the resultant product, the binary equivalent of $7\cdot10^6$, will be scaled 2^{-28}, according to the rules discussed previously.

The binary (hex form will be used) equivalents of the powers of 10 involved in this case are:

10^6 = 00z4240

10^5 = 00186u0

10^4 = 0002710

10^3 = 00003y8

10^2 = 0000064

10^1 = 000000u

10^0 = 0000001

Scaled 2^{-24}, rather than 2^{-28}, these become:

10^6 = 0z42400

10^5 = 0186u00

10^4 = 0027100

10^3 = 0003y80

10^2 = 0000640

10^1 = 00000u0

10^0 = 0000010

} As these magnitudes would appear in the machine, for our purposes, scaled 2^{-24}.

Now, suppose we clear the two-word registers, and then we store the BCD number (7196323-), which we wish to convert, in MQ_1. Its sign will go to IP. Since we are loading MQ, it will be added to what is already there, but we know that is 0, since we previously cleared the two-word registers. The 28 bits of BCD code will be in the 28 most significant bits of MQ_1. Now we load $10^6 \cdot 2^{-24}$ into ID_1, but with a C of 1, rather than 0. Since this C is not even, the sign of the number will accompany the magnitude, and will not be sent off to IP; of course the sign will be 0. If ID is loaded with an odd C, the setting of it will not cause PN to be cleared. We know that PN will contain 0's, though, because we previously cleared the two-word registers. Now suppose we multiply these two numbers, but allow the multiplication to last only eight word-times, rather than the usual 56. We said previously, we could cut a multiplication short at any time, provided an even number of word-times has been allowed, and be able to predict the result in PN. Eight word-times would be just sufficient for four bits from the most significant end of MQ_1 to be inspected. The effect, in PN, will be to generate the product of these four bits times the contents of ID. In other words, in PN we will have $7 \cdot 10^6$ scaled 2^{-28}, which is what we want. MQ will have been shifted left by four bits. Thus the four-bit binary code for the first decimal digit is gone, and the four most significant bits in MQ_1 contain the next decimal digit, scaled 2^{-4}.

Suppose now, we reload ID_1, again with a C equal to 1, with $10^5 \cdot 2^{-24}$. The sign in IP and the product in PN will remain undisturbed. Now we can again multiply for eight word-times. The effect will be to generate a sum in PN equal to the old product plus a new one which is, in our example, $1 \cdot 10^5 \cdot 2^{-28}$. The sum will equal $(7 \cdot 10^6 + 1 \cdot 10^5) \cdot 2^{-28}$.

We can do this seven times in all, arriving at a sum in PN equal to

$$(7 \cdot 10^6 + 1 \cdot 10^5 + 9 \cdot 10^4 + 6 \cdot 10^3 + 3 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0) \cdot 2^{-28}$$

which is the desired answer. It is a binary integer, scaled 2^{-28} in the machine, representing the decimal integer we started with.

This number could be manipulated satisfactorily by a program; as far as the computer is concerned it is a binary fraction, with the machine binary point preceding the most significant bit. But, since the decimal number may have been scaled, itself, in the decimal system, containing fractional digits, this procedure would eventually lead to misery, since both a decimal and a binary scale factor would have to be remembered by the programmer. We would like to treat it as a decimal fraction in the machine, represented in binary, and, just as we previously spoke of the machine treating every number as a fraction, scaled 2^{-28}, we would now like to speak of the machine treating every number as a fraction, scaled 10^{-7}. If only seven decimal digits are allowed per number during input, then the machine value will always be a fraction, if the machine treats this number as scaled 10^{-7}.

But there is something further that must be done to the converted number, as we left it, before we can say it is scaled 10^{-7} in the machine. We have a decimal integer scaled 2^{-28}; if we divide it by $10^7 \cdot 2^{-28}$ (which, in hex, would be represented in the machine as 0989680), we will truly have in the machine the binary equivalent of a decimal fraction, and from here on out, we can drop all references to binary scaling, and speak only of decimal scaling. Therefore, in our conversion process, one more major step is required: we have an integer scaled 2^{-28} in PN_0; it is already properly located as the numerator for a division. If we load the denominator (0989680_{16}) into ID_1 with a C of 1, leaving PN undisturbed, we can perform a single-precision divide, and MQ_0 will contain the quotient. IP will still contain the sign of the number, which is what we want. The quotient will equal the original decimal integer, call it "D", divided by 10^7, or $D \cdot 10^7 \cdot 2^0$. We have thus completely eliminated binary scaling, and have substituted decimal scaling, which jibes more closely with the way in which we are used to handling numbers.

Applying the same rules here to decimal scaling that we applied earlier to binary scaling, if we say that

$$A^* = A \cdot 10^{-7},$$

where A* is the machine representation of A, we are saying that there are no fractional digits in A; its true decimal point is seven decimal

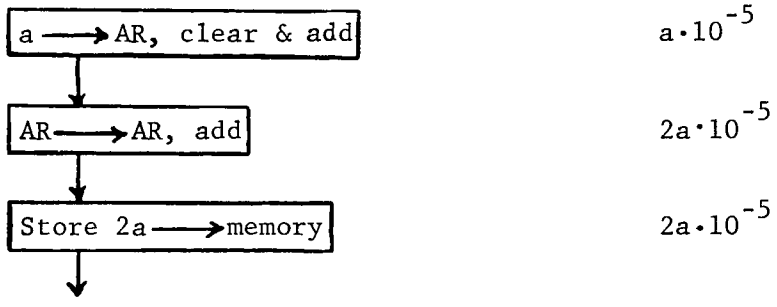
places to the right of the machine decimal point, or, following the least significant decimal digit.

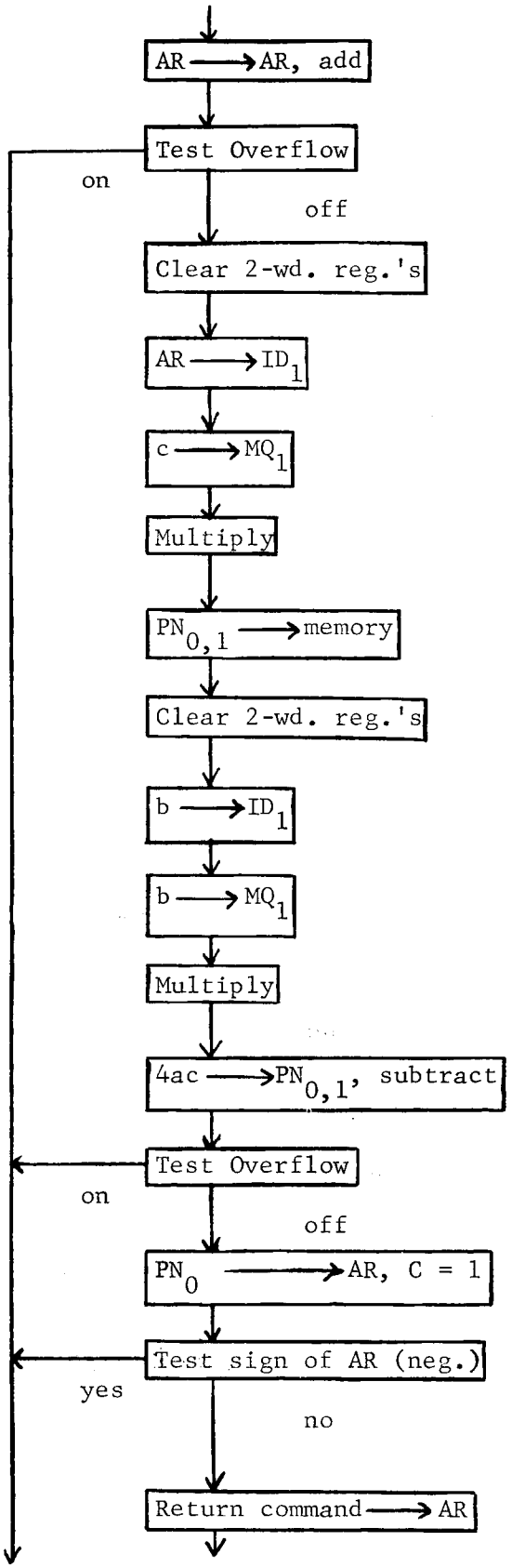
In our problem, we originally chose to allow seven binary digits for fractional accuracy, in order to carry accuracy at least to the nearest 1/100th. Now we can carry accuracy exactly to the nearest 1/100th, merely by rescaling a, b, and c. Rather than enter them as seven-digit decimal numbers with the true decimal point following the least significant digit, we can understand the true decimal point to be to the left two places allowing fractional accuracy to the nearest 1/100th. We are saying, then, that the machine will contain a, b, and c, all scaled $10^2 \cdot 10^{-7}$, or 10^{-5}. In other words, the true decimal point is five decimal places to the right of the machine decimal point, and the following two decimal digits are fractional, in the true sense of the word.

This will make the inputs easier (we haven't yet discussed the outputs, but they will be easier, too), and it will reduce the scaling problems to a form with which we are familiar. But notice what else it will do; it will greatly restrict the range of values we can have for these three numbers, and therefore, for the answers. If seven decimal digits are allowed for a number, and two of them must be interpreted as fractional, the range of values for any number is $-99999.99 \leq N \leq 99999.99$. Compare this with the ranges we were able to accommodate when we used binary scaling and did our own conversions of inputs and outputs.

Whether you decide to use decimal or binary scaling will depend to a great deal on the requirements of your particular problem, but you should always be aware that operation of the computer is essentially designed for work with binary numbers representing binary quantities. Once you become familiar with the new number system, you will find it no harder to use than the decimal number system; as a matter of fact, you will find it easier.

The flow diagram of our computation will be simplified, if we treat the numbers in the machine as decimal numbers. Notice that we will again resort to double-precision numbers in order to retain both wide range and the necessary accuracy. Notice also the mental gymnastics we play in order to avoid repositioning an answer by shifting. Shifting rescales a number in a binary manner. It will not usually be adaptable to dealing with decimally scaled numbers.





$4a \cdot 10^{-5}$

$4a \cdot 10^{-5}$

$c \cdot 10^{-5}$

$4ac \cdot 10^{-10}$

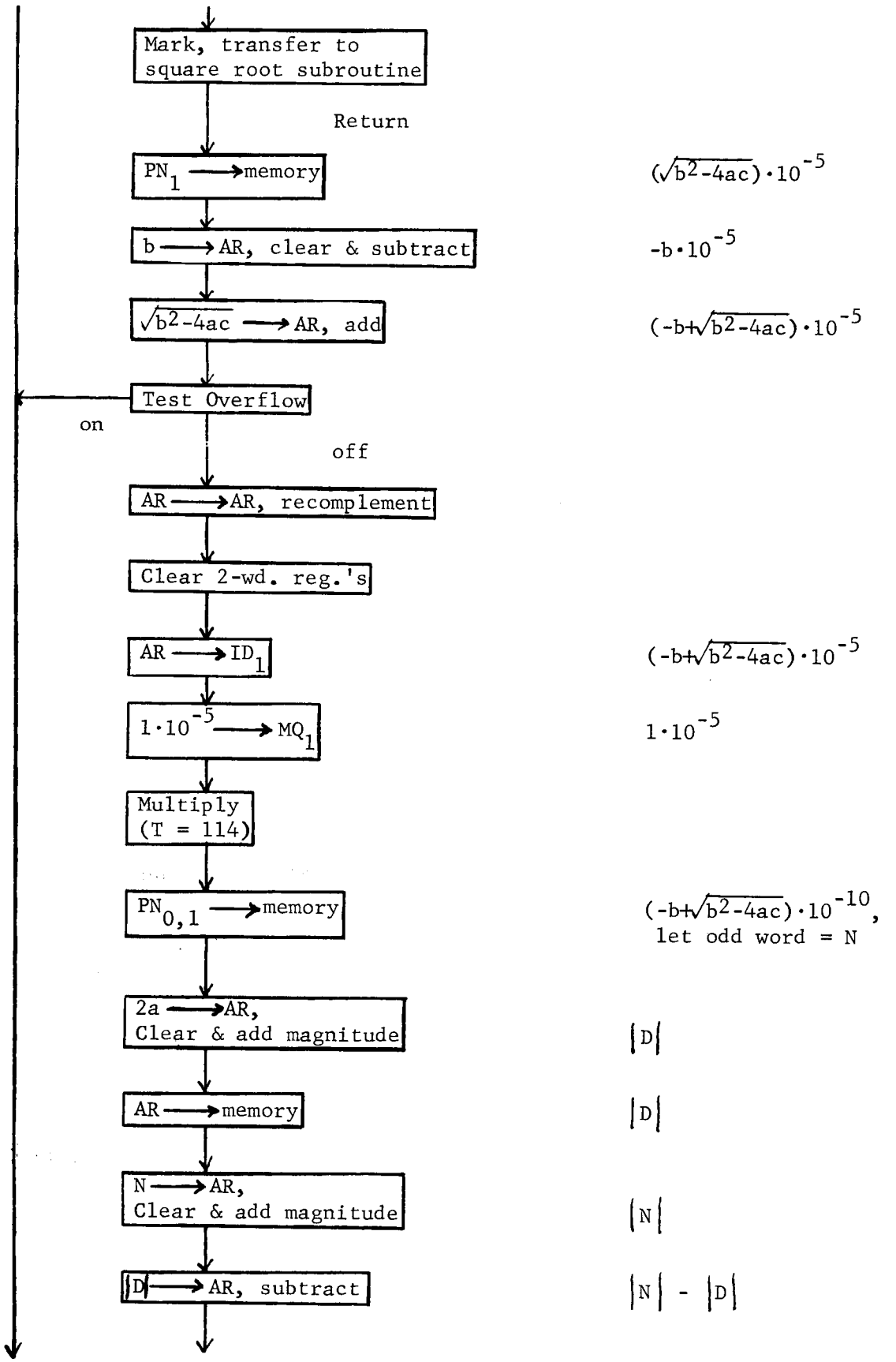
$b \cdot 10^{-5}$

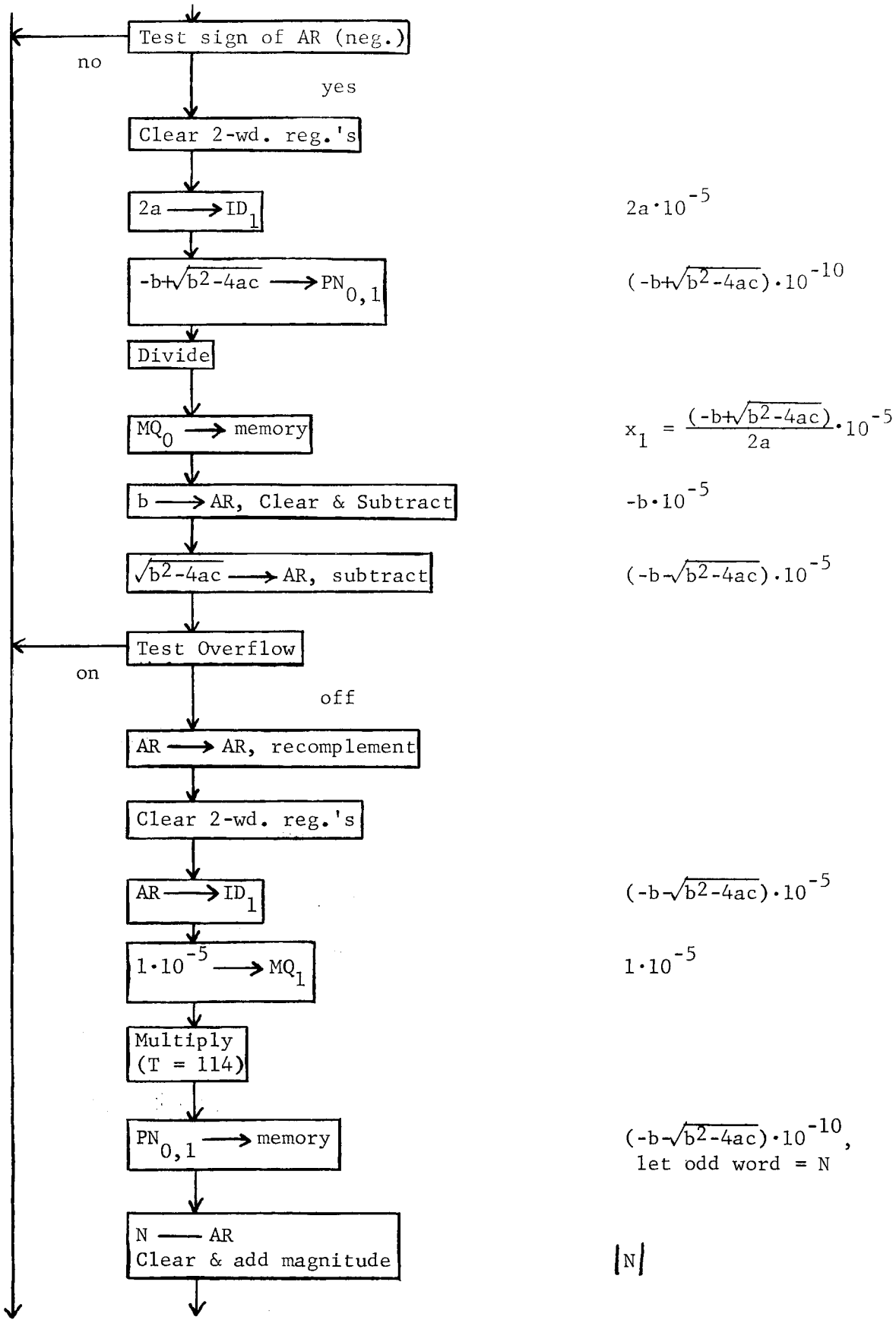
$b \cdot 10^{-5}$

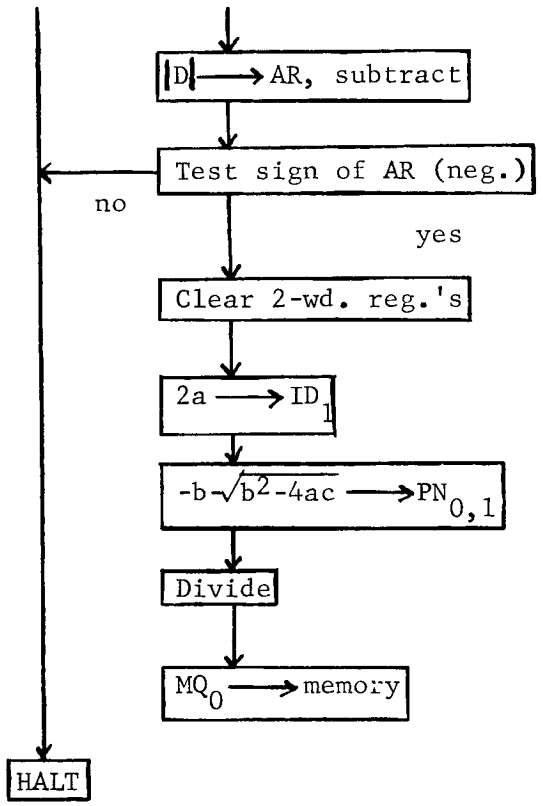
$b^2 \cdot 10^{-10}$

$(b^2-4ac) \cdot 10^{-10}$

sign of radicand in AR


Mark, transfer to square root subroutine
Return
PN1 → memory
(√(b2-4ac))·10-5
b → AR, clear & subtract
-b·10-5
√(b2-4ac) → AR, add
(-b+√(b2-4ac))·10-5
Test Overflow
on
off
AR → AR, recomplement
Clear 2-wd. reg.'s
AR → ID1
(-b+√(b2-4ac))·10-5
1·10-5 → MQ1
1·10-5
Multiply (T = 114)
PN0,1 → memory
(-b+√(b2-4ac))·10-10, let odd word = N
2a → AR, Clear & add magnitude
|D|
AR → memory
|D|
N → AR, Clear & add magnitude
|N|
|D| → AR, subtract
|N| - |D|


Test sign of AR (neg.)
no
yes
Clear 2-wd. reg.'s
2a ⟶ ID1
-b+√b2-4ac ⟶ PN0,1
Divide
MQ0 ⟶ memory
b ⟶ AR, Clear & Subtract
√b2-4ac ⟶ AR, subtract
Test Overflow
on
off
AR ⟶ AR, recomplement
Clear 2-wd. reg.'s
AR ⟶ ID1
1·10-5 ⟶ MQ1
Multiply (T = 114)
PN0,1 ⟶ memory
N ⟶ AR
Clear & add magnitude
2a·10-5
(-b+√b2-4ac)·10-10
x1 = (-b+√b2-4ac)/2a ·10-5
-b·10-5
(-b-√b2-4ac)·10-5
(-b-√b2-4ac)·10-5
1·10-5
(-b-√b2-4ac)·10-10, let odd word = N
|N|



$|N| - |D|$

$2a \cdot 10^{-5}$

$(-b-\sqrt{b^2-4ac}) \cdot 10^{-10}$

$x_2 = \frac{(-b-\sqrt{b^2-4ac})}{2a} \cdot 10^{-5}$

We have already discussed the manner in which we treat the input data, in order to supply this computation program with the numbers it needs, but, before we flow diagram a new input scheme, we should consider output. We will write an input/output program for this case, just as we did before.

We will, upon completion of the computation, have two answers, each scaled 10^{-5}, in 19.u7 and 19.u6. These will be binary numbers, but they will be meaningless to us in their binary form. We must convert them back to decimal and type out decimal answers. We can indicate the proper scaling of the answers by the positioning of the decimal point in the type-out. We know, if these answers are scaled 10^{-5} in the computer, each of them should be typed out in the form: SDDDDDPDD.

The method we developed for the conversion of a BCD number to binary grew directly out of the inspection process: we developed the binary equivalent for each multiple of a power of ten, and then added these binary equivalents. It stands to reason that, from the inspection process, as it was defined in the Introduction, we can develop a reverse conversion process for our program, to convert a binary number to its decimal equivalent.

The binary scaling of our numbers was eliminated by generating a binary scale factor of 2^0, and leaving remaining only a decimal scale factor. If each number, as it appears in the machine actually has a binary scale factor of 2^0, it is truly a fraction. The general rule for converting a binary fraction to its decimal equivalent is: multiply the binary

fraction by the binary equivalent of 10, which is 1010. Retain the integral portion of the product as the coefficient of 10^{-1}. Perform the same operation on the fractional portion of the product. Retain the integral portion of the new product as the coefficient of the next power of 10 in the decreasing series, which would be 10^{-2}. Repeat the process as often as necessary (the fractional portion of some product equals 0), or until the desired accuracy in the converted form of the original fraction is achieved.

As an example, consider the conversion to decimal of the binary fraction,

```
                .10111000001110011101010000011
                                          1010
              ------------------------------------
                1 01110000011100111010100000110
               101 11000001110011101010000110
              ------------------------------------
7(10)         0111.00110010010000100100100111110
                                          1010
              ------------------------------------
                0 01100100100001001001001111100
               001 10010010000100100100111100
              ------------------------------------
1(10)         0001.11110110100101101110001011100
                                          1010
              ------------------------------------
                1 11101101001011011100010110000
               111 10110100101101110001011000
              ------------------------------------
9(10)         1001.10100001111001001101101110000
                                          1010
              ------------------------------------
                1 01000011110010011011011100000
               101 00001111001001101101110000
              ------------------------------------
6(10)         0110.01010010111100001001001100000
                                          1010
              ------------------------------------
                0 10100101111000010010011000000
               010 10010111100001001001100000
              ------------------------------------
3(10)         0011.00111101011001011011111000000
                                          1010
              ------------------------------------
                0 01111010110010110111110000000
               001 11101011001011011111000000
              ------------------------------------
2(10)         0010.01100101111111001011011000000
                                          1010
              ------------------------------------
                0 11001011111110010110110000000
               011 00101111111001011011000000
              ------------------------------------
3(10)         0011.11111011101111100011100000000
```

The decimal equivalent of the above binary fraction is .7196323. Compare the binary equivalent of this fraction with the BCD number corresponding to this fraction.

We can use the above method to convert each answer generated by our program to its BCD equivalent. Then we can type out this BCD number as the answer, so that, on the typewriter, a decimal number will appear which can be read directly. Of course, the BCD equivalent will be in the form of a fraction: it will be a decimal number scaled 10^{-7}.

If we choose to interpret it, as we do, scaled 10^{-5}, we can move the decimal point to the right five decimal places, arriving at a number, as we have already seen, of the form SDDDDDPDD. Thus, we can use the output format itself to properly scale the answer, as it is typed out, so that it can be read directly.

We will place the answer in ID_1 and the multiplier, $1010_{(2)}$, in MQ_1, in the four most significant bits. When we multiply a number scaled 2^0 (the answer) by a number scaled 2^{-4} (the multiplier), we get an answer scaled 2^{-4} (the integral portion of the product will be in the first four bits of PN_1). If we had a way of "extracting" these four bits and saving them, we could then reload ID with the remaining bits from PN, which constitute the fractional portion of the product, reload MQ_1 with the same multiplier (the first one was shifted out as it was inspected), and perform the process all over again. Eventually, after seven such operations, each time saving the integral portion of the product, we would have the seven BCD digits corresponding to our answer. If we had a way of recombining them, end-to-end in one word, we would have exactly the number we wish to type out as an answer.

EXTRACT, AND ITS USE IN NUMBER CONVERSION

There is an extract operation available in the G-15. It is called for by a special command of slightly different form from the other special commands we have thus far considered. S in this command equals 31; again the computer will know, since there is no line numbered 31, that a special operation is being called for. C equals 0. Any destination may be specified. The command will operate during whatever word-time(s) is (are) specified; it may be either immediate or deferred. The number out of which certain bits are to be extracted must be in line 21, in whatever word you desire (of course, you must be sure that it will be in the word available during the word-time of execution of this command). In the corresponding word of line 20, there will be a "mask", which will specify to the computer which bits you wish extracted. The mask in line 20 will contain a 1 in each bit-position to be extracted, a 0 in each bit-position you wish left behind. For instance, the mask in our case above, would be: 11110000000000000000000000000, causing only the first four bits of the product (the integral portion, a BCD digit) to be extracted. The result of the extraction, containing 0's in all those bit-positions not extracted, will be transferred to the destination during the same word-time. The number from which the bits were extracted will remain intact in line 21, while the mask will also remain intact in line 20.

There is a second extract operation available in the G-15, which causes exactly the same sequence of operations to occur, with the exception that the bits extracted from the number in line 21 will be those bits corresponding to 0's in the mask, while the others, corresponding to 1's in the mask, will be left behind. In short, the extract operation is the same, but interpretation of the mask is exactly reversed. The command for this is: S = 30, C = 0, D may be any line.

Consider now, the first product we arrived at, on page 176.

01110011001001000010010010011110

This requires more than 29 bits, and therefore, we would have to use two words in lines 20 and 21. This means we would have to make the extract operations immediate for two word-times of execution ($T = L_3$). We could load this number in line 21, and a mask in line 20, as shown below (word-boundaries have been ignored):

```
01110011001001000010010010011110    Line 21 (number)
11110000000000000000000000000000    Line 20 (mask)

01110000000000000000000000000000    Result 1

00000011001001000010010010011110    Result 2
```

The first extraction, S = 31, yields result 1, in which we have only the first binary-coded-decimal digit, in its proper position. We could now store this in memory.

The second extraction, S = 30, yields result 2, in which we have only the remaining fractional portion of the product, similar to that shown in the first product on page 176, except that here, there are four leading 0's. This is fine; if we transfer this to ID (double-precision) and multiply it by $1010 \cdot 2^{-4}$, we will get the second product shown on page 176, except that it will be removed four places to the right. In short, this second product will be: 000000011111011010010110111000101100. If we load this into line 21, and a mask into line 20, as shown below, we can perform the extractions all over again:

```
00000001111101101001011011100010110 0    Line 21 (number)
00001111000000000000000000000000000 0    Line 20 (mask)

00000001000000000000000000000000000 0    Result 1

00000000111101101001011011100010110 0    Result 2
```

The first extraction, S = 31, yields result 1, in which we have only the first binary-coded-decimal digit, in its proper position (the second group of four bits). We could now store this in memory.

The second extraction, S = 30, yields result 2, in which we have only the remaining fractional portion of the product, similar to that shown in the second product on page 176, except that here, there are eight leading 0's. This, again, is fine; if we transfer this to ID and multiply it by $1010 \cdot 2^{-4}$, we will get the third product shown on page 176, except that it will be removed eight places to the right.

We could continue this process until seven multiplications have been performed, with all of the accompanying extractions. At that point, we would have saved seven results from the first extractions, and they would be,

eliminating trailing 0's, and expressing them as single-precision 28-bit magnitudes:

```
0111000000000000000000000000
0000000100000000000000000000
0000000010010000000000000000
0000000000000110000000000000
0000000000000000001100000000
0000000000000000000000100000
0000000000000000000000000011
```

If we now add these together, we will get:

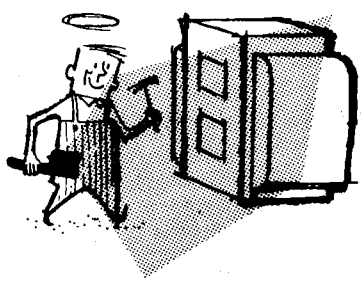
0111,0001,1001,0110,0011,0010,0011

If this were typed out, it would yield:

7 1 9 6 3 2 3

which is the number we expected.

This method will work, and we could program it, but it involves many, many commands. Prior to each set of extractions, the product must be transferred from PN to line 21, and a new mask must be transferred into line 20. Prior to each multiplication, the two-word registers must be re-set-up. This method will also require quite a few storage locations in memory.



The Bendix Computer Division engineers, always aiming to make life easier for the programmer, built into the G-15 a special extract command, designed especially for the purpose of saving bits in PN and resetting ID for further multiplication. It is a special command: D = 31, S = 23, C = 3. It may be either deferred or immediate. In our case, we will make it immediate, operating for two word-times, thus covering both halves of PN and ID.

During each word-time of execution of this command, the word in that word-time of line 02 will serve as a mask for an extraction. The bits in those bit-positions in PN which correspond to the bit-positions in the mask containing 0's, will remain in PN. The bits in PN for which there are corresponding 1's in the mask in line 02 will be transferred to ID. PN will retain only the results of the extraction which treated the mask in reverse (extracting bits corresponding to 0's in the mask); ID will receive only the results of the extraction which treated the mask in the normal manner (extracting bits corresponding to 1's in the mask). Thus, if we set up masks in line 02 which will have 0's corresponding to the integral portions of PN, and 1's corresponding to the remaining bits (the fractional portions

in PN), we will, through execution of this one command, generate the sum of the integral portions of the products in PN_1, while we directly reload ID for the next multiplication. Remember that the resultant product, in each case, is moved to the right by four more places, so that the integral portion of PN, as it grows longer, will never be disturbed by the succeeding multiplication.

The extractor, or mask, used after the first multiplication will be:

odd word: 0zzzzzz-
even word: zzzzzzz

After the second multiplication:

odd word: 00zzzzz-
even word: zzzzzzz

After the third multiplication:

odd word: 000zzzz-
even word: zzzzzzz

After the fourth multiplication:

odd word: 0000zzz-
even word: zzzzzzz

After the fifth multiplication:

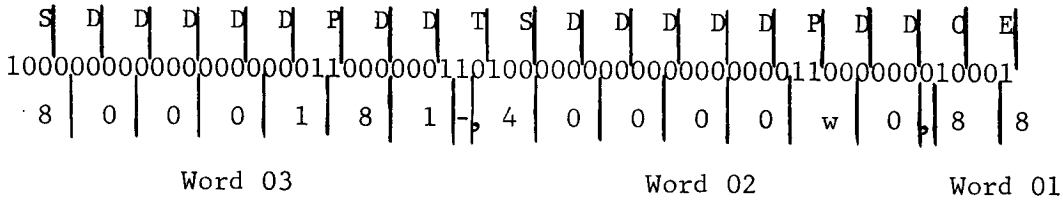
odd word: 00000zz-
even word: zzzzzzz

After the sixth multiplication:

odd word: 000000z-
even word: zzzzzzz

After the seventh multiplication, no extraction will be necessary: the seventh BCD digit will occupy the desired four bits, and no further multiplications are necessary.

The BCD number which results will be the desired answer in decimal. It can be placed properly in line 19 for type-out under control of the output format. Several times we have mentioned that we want the form of the decimal answers, as they are typed out, to be SDDDDDPDD. Our output format, therefore, for two such answers, will be:

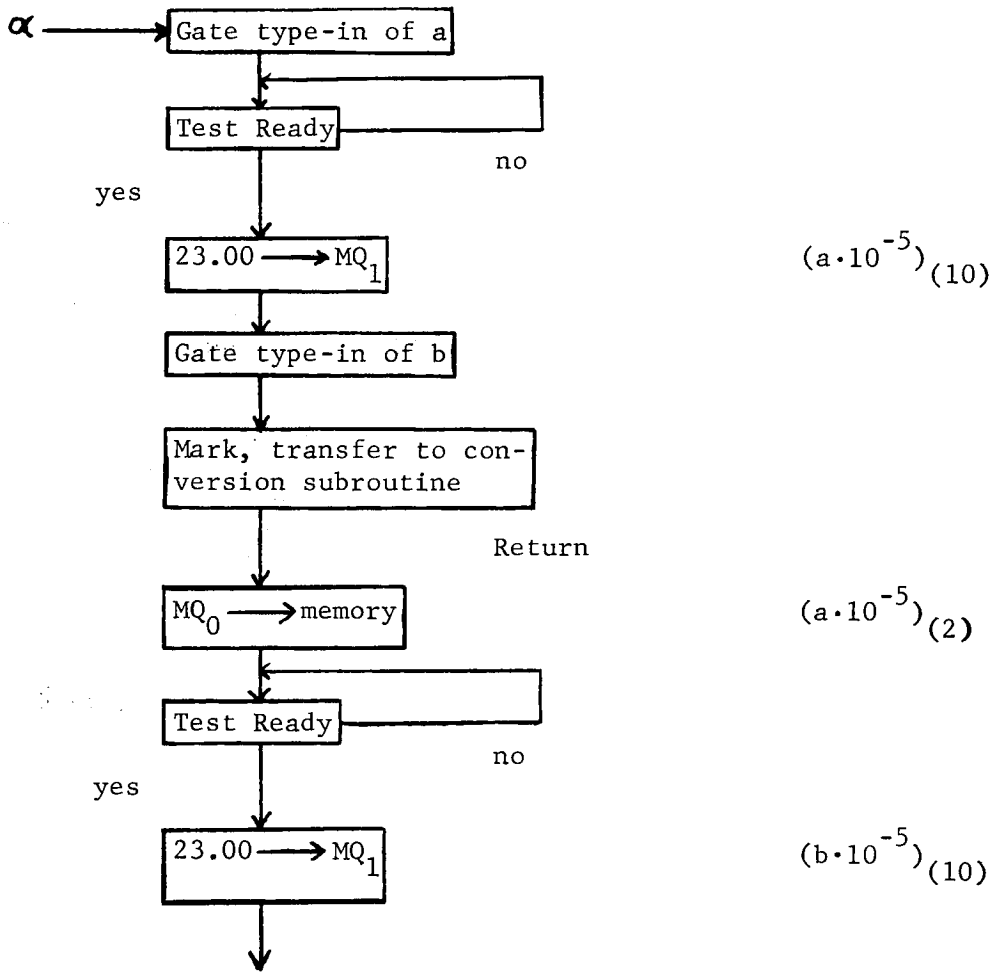


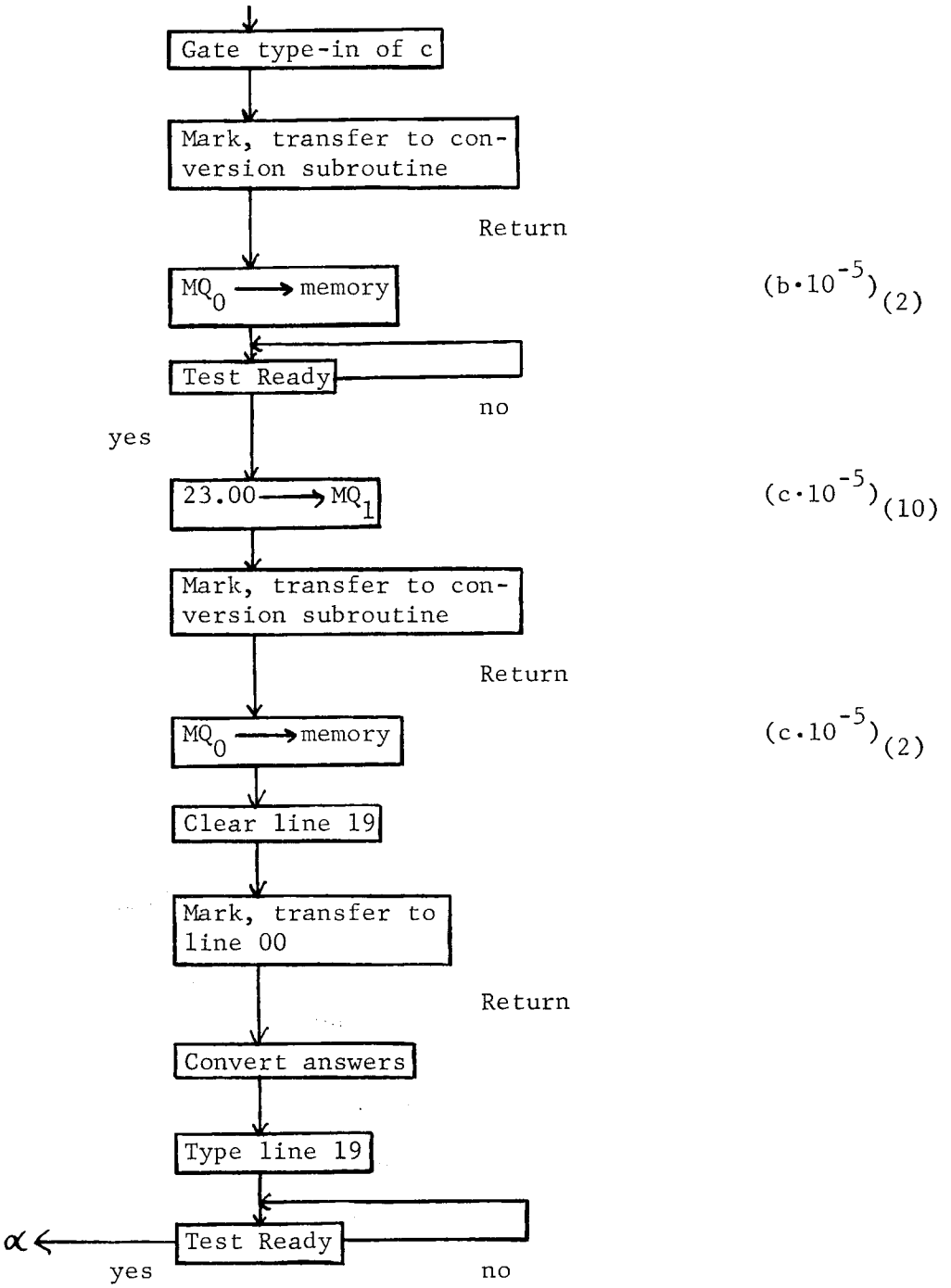
One further point remains, before we flow-diagram this new input/output program. We have mentioned, up to this point, that we will reload the multiplier into MQ_1 before each multiplication during the conversion for output. The multiplier is 1010. Why not place seven multipliers in MQ_1 all at once, and multiply for only eight word-times (therefore four bits) each time? In that case, our multiplier will look like this: 01010101010101010101010101. Cutting short a multiplication is fine, and this will work. But, if we are going to do this, notice that each multiplication will end with a multiplication by 0. This merely means to the computer that it is not to add the shifted contents of ID to what it already has in PN. Why must we tell the computer that? There is no good reason for it, so we can eliminate the last 0 in each group of four bits, making our multiplier look like this:

10110110110110110110100000000.

We will limit the word-times of execution of the multiplication, in each case, to six, meaning that only three bits from MQ_1 will be inspected. The hex equivalent of this binary number is v6xv680.

Now let's flow-diagram the new input/output program, for line 02.

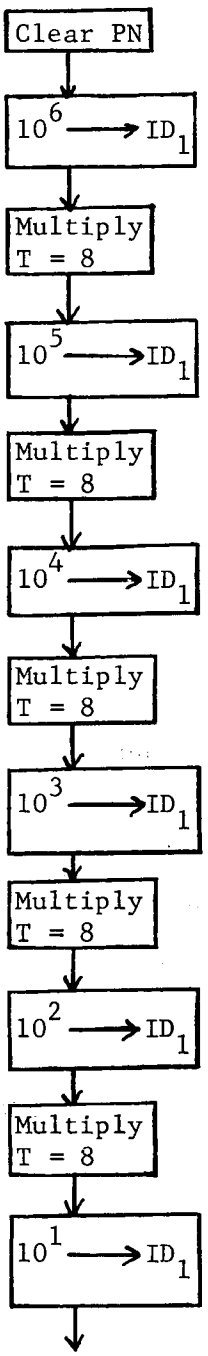




Notice that, in the preceding flow diagram, we made reference to a conversion subroutine. This we are also going to write, since we now know how. It will be in line 02, along with the input/output program. The mark, transfer command need not cause control to be transferred from one line to another; it may simply transfer control within the same line to another word-time, as it will do in this case. We need not supply this subroutine with a return command each time we enter it, because we will code the subroutine with a return command already in it.

The reason for that provision, as we mentioned earlier, is to allow very general use of a subroutine by any number of users, all of whom may not wish to return to the same line upon completion of the subroutine.

The following is a flow diagram of the conversion subroutine for BCD to binary, written assuming that the BCD number to be converted is already in MQ_1.



$10^6 \cdot 2^{-24}$

$D_1 \cdot 10^6 \cdot 2^{-28}$

$10^5 \cdot 2^{-24}$

$(D_1 \cdot 10^6 + D_2 \cdot 10^5) \cdot 2^{-28}$

$10^4 \cdot 2^{-24}$

$(D_1 \cdot 10^6 + D_2 \cdot 10^5 + D_3 \cdot 10^4) \cdot 2^{-28}$

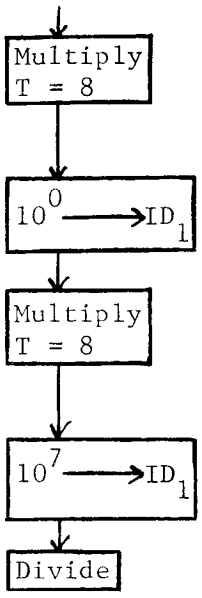
$10^3 \cdot 2^{-24}$

$(D_1 \cdot 10^6 + D_2 \cdot 10^5 + D_3 \cdot 10^4 + D_4 \cdot 10^3) \cdot 2^{-28}$

$10^2 \cdot 2^{-24}$

$(D_1 \cdot 10^6 + D_2 \cdot 10^5 + D_3 \cdot 10^4 + D_4 \cdot 10^3) \cdot 2^{-28}$

$10^1 \cdot 2^{-24}$



$$(D_1 \cdot 10^6 + D_2 \cdot 10^5 + D_3 \cdot 10^4 + D_4 \cdot 10^3 + D_5 \cdot 10^2 + D_6 \cdot 10^1) \cdot 2^{-28}$$

$$10^0 \cdot 2^{-24}$$

$$(D_1 \cdot 10^6 + D_2 \cdot 10^5 + D_3 \cdot 10^4 + D_4 \cdot 10^3 + D_5 \cdot 10^2 + D_6 \cdot 10^1 + D_7 \cdot 10^0) \cdot 2^{-28}$$

$$10^7 \cdot 2^{-28}$$

$$\frac{(D_1 10^6 + D_2 10^5 + D_3 10^4 + D_4 10^3 + D_5 10^2 + D_6 10^1 + D_7 10^0) \cdot 2^0}{10^7}$$

Another conversion subroutine is also necessary in the program, and it, too, will be in line 02. This is the subroutine for converting the binary answers to their BCD equivalents. When we discussed this conversion process earlier, we omitted one point at that time, which now must be mentioned, since you will see provision for it in the flow diagram which follows.

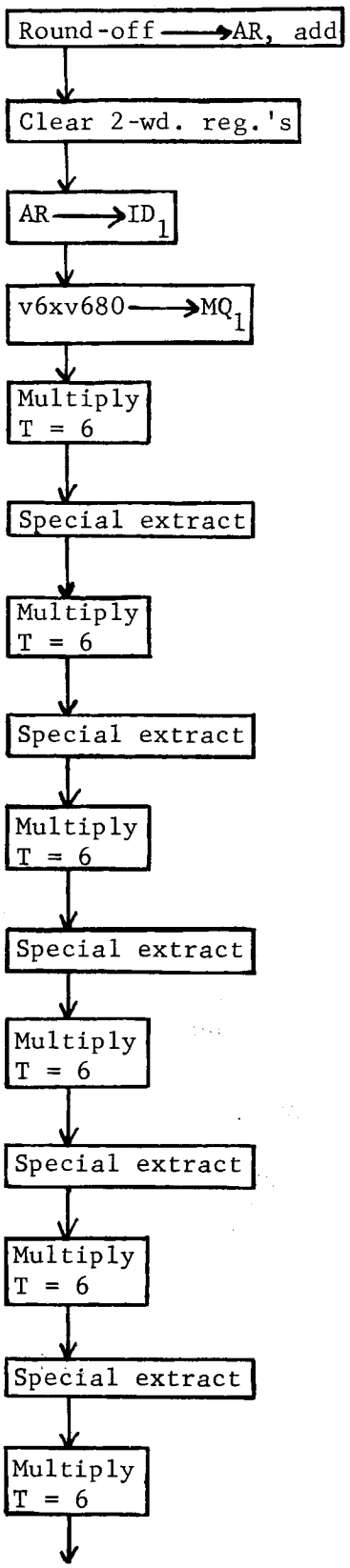
The least significant BCD digit generated will be 10^{-7}, or $1/10^7$. This quantity is a good deal greater than 2^{-28}, or $1/2^{28}$. For this reason, the last bit in the binary number prior to conversion can have no effect on the BCD result. As a matter of fact, $2^{-23} > 10^{-7} > 2^{-24}$. In order to arrive at the seven-digit decimal number closest to the value represented in 28 bits, we must round off the binary number prior to conversion.

$$\begin{aligned} 2^{-28} &= .000000003725 \\ 13 \cdot 2^{-28} &= .000000048425 \\ 14 \cdot 2^{-28} &= .000000052150 \end{aligned}$$

$13 \cdot 2^{-28}$ is very close to $.5 \cdot 10^{-7}$; so is $14 \cdot 2^{-28}$. But the latter exceeds $.5 \cdot 10^{-7}$, and, faced with a choice, we choose to round up to the next higher decimal digit only if we are at least within 1/2 of it. We will therefore choose the smaller of the two round-off numbers, since it will require a value in the original binary number of more than $.5 \cdot 10^{-7}$. Our round-off, expressed in hex, will be:

.000000x

Now we can flow-diagram the binary - BCD conversion subroutine, assuming the number to be converted has already been placed in AR, with a C of 1.



$(X_{(2)} + .000000x) \cdot 2^0$

$X_R \cdot 2^0$

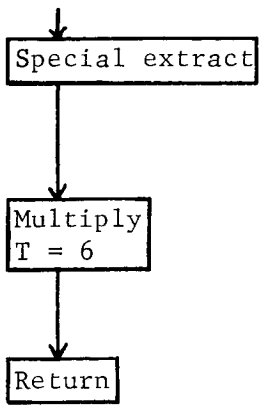
$D_1 \cdot 10^{-1}$

$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2}$

$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3}$

$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3} + D_4 \cdot 10^{-4}$

$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3} + D_4 \cdot 10^{-4} + D_5 \cdot 10^{-5}$



$$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3} + D_4 \cdot 10^{-4} + D_5 \cdot 10^{-5} + D_6 \cdot 10^{-6}$$

$$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3} + D_4 \cdot 10^{-4} + D_5 \cdot 10^{-5} + D_6 \cdot 10^{-6} + D_7 \cdot 10^{-7}$$

The loader program and the square root subroutine will remain the same. The square root subroutine just "cranks out" square roots of binary numbers. What those numbers represent makes no difference at all to the subroutine.

The program tape thus prepared can be read into the computer with an p switch action, followed by moving the compute switch to GO. The program will gate type-in, at which time a must be typed in. Type-in will be gated again, and this time b must be entered. Type-in will be gated a third time, and c must be entered. Notice that the program has been written to make as much use as possible of the time required to type in a number. You will be unable to type in the number so fast that the computer will not have to wait for you. After the third type-in, computation will take place, and two decimal answers will be typed out. These can be read directly, since their decimal points will be properly positioned. Type-in will again be gated, this time for a new set of values.

OTHER PROGRAMMING TECHNIQUES

A commonly needed programming technique remains undiscussed because the program we have been considering up to this point does not require it. However, because it is such a common technique, it cannot remain unmentioned any longer. It is called looping. Consider the case in which you are given 50 random positive numbers in 19.00 - 49, and you must sort them, placing the least number in 00, and the greatest number in 49.

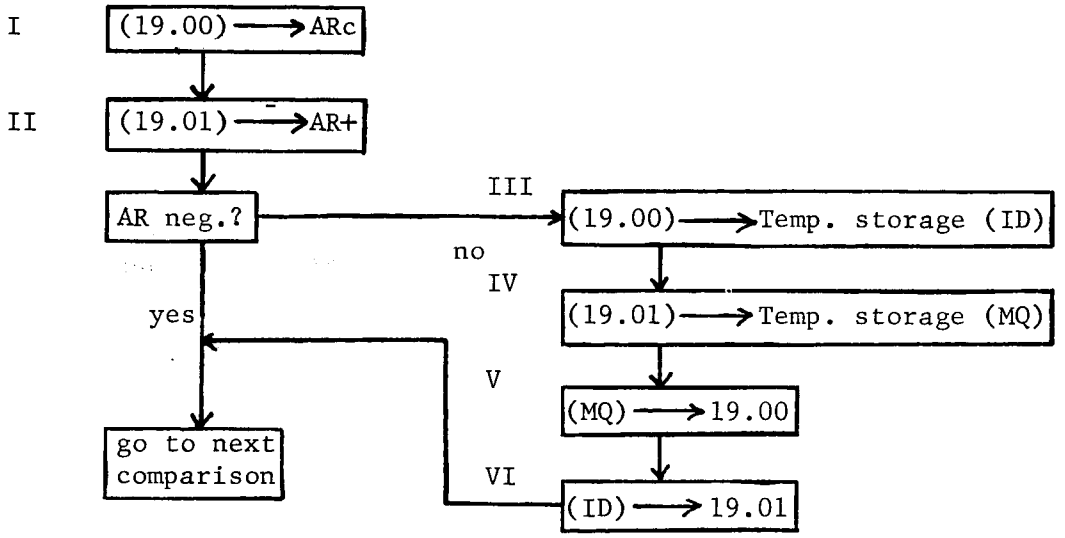
We know that commands are in the same binary form as constants and data in the memory of the computer. In fact, we know that the only distinction the computer can make between commands and constants or data is based on when it reads the words: if a word is read during RC time, it will be treated and interpreted as a command; if it is read during EX time, it will be treated and interpreted as data. Thus, any command could be treated as data by the computer, if it were read during EX time. A command could be called into AR, and have something added to it, or subtracted from it. We also know that a command can be executed out of AR, the computer being told to do this by a special command, D = 31, S = 31, C = 0. When this special command is executed, the computer is set up to take the next command from AR at time N of the special command. After

the command in AR is executed, the computer will go to N (of the command executed from AR) in the same command line from which it was previously taking commands.

The requirements of the proposed problem are to pick a number, say 19.00, and compare it with all 49 others, each time exchanging if necessary, to assure that, at the end of the series of comparisons, the least number of the 50 is in 19.00, and then repeat the process in order to get the smallest number in 19.01, etc.. The total number of comparisons necessary will be:

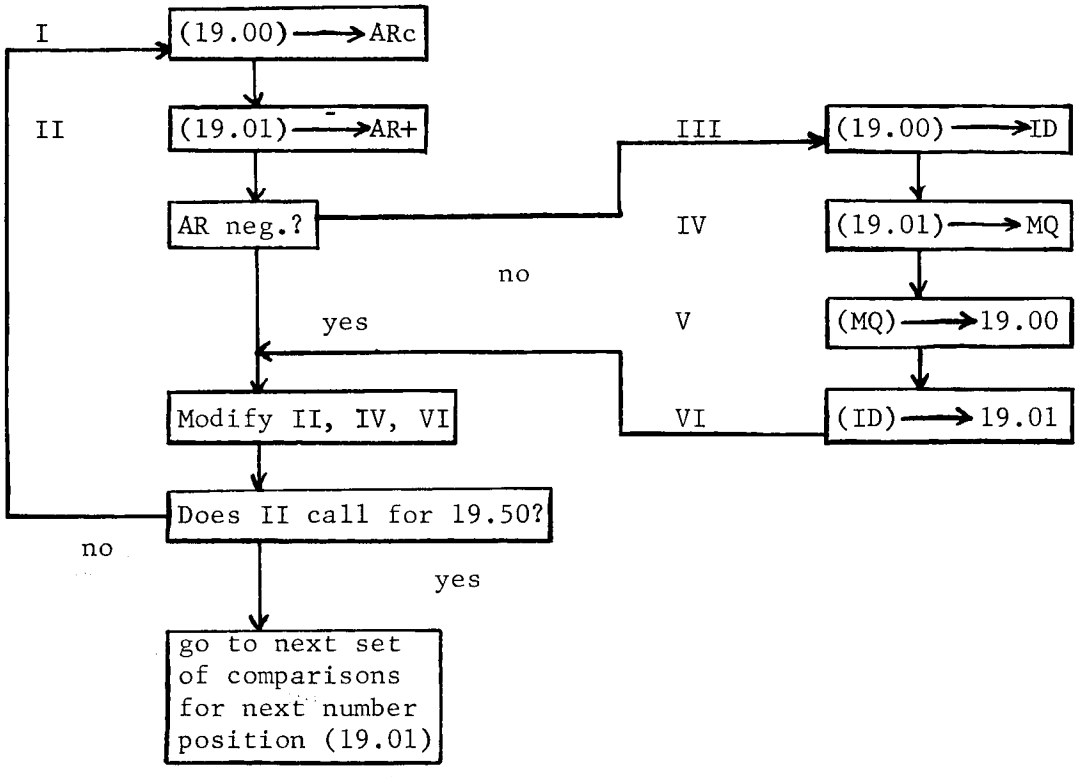
for 00:	49
for 01:	48
for 02:	47
.	.
.	.
.	.
for 48:	1
for 49:	0
	1225 = total number of comparisons necessary.

It's an easy job to flow-diagram the comparison and exchange, if an exchange is necessary:

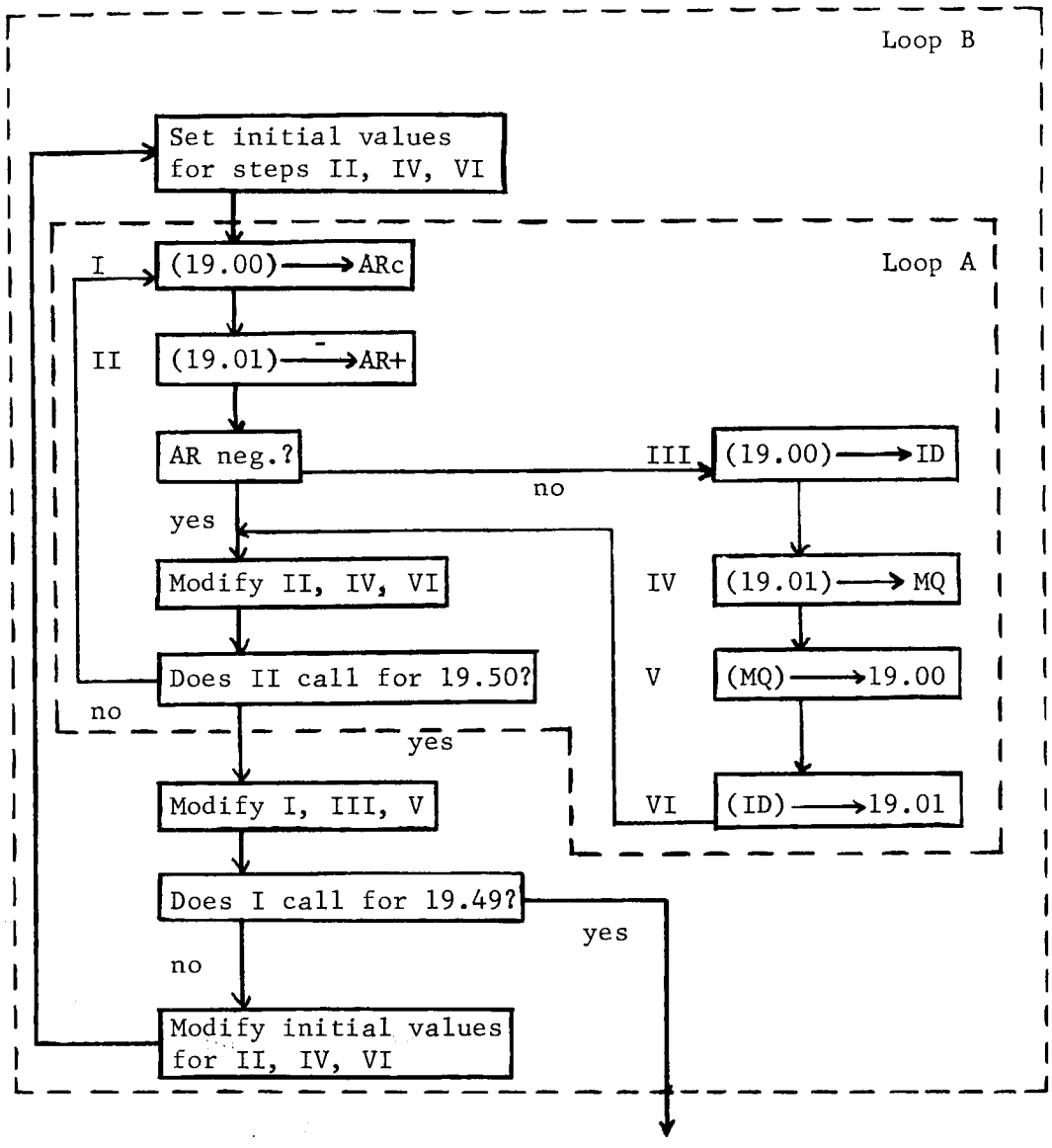


It would be an easy job to write this same sequence 1225 times, each time for the comparison of a new pair of numbers, but the resultant program would be too long to store in the memory of the computer. In the flow diagram above, the next comparison would be of 19.00 against 19.02. The same diagram would be sufficient for our purposes if boxes II, IV, and VI were modified to affect 19.02, rather than 19.01. We could then modify them again, to affect 19.03, etc.. Eventually they would affect 19.49, and after that, we could be sure that the least

number of the 50 would be in 19.00. If, instead of an arrow in the preceding flow diagram pointing to the next comparison, we inserted steps to modify steps II, IV, and VI, and then drew an arrow going back to step I, we would be indicating a loop which would be repeated over and over again, each time comparing 19.00 with a new word from line 19. This loop would be unending, and this, of course, would be disastrous. We will therefore establish a limit beyond which the loop will not continue; that limit will be after 19.00 has been compared with 19.49. At that point we will continue with the program, and the flow diagram will be as follows:



Now 19.01 must be compared with each succeeding location, resulting in the placement of the next least value in 19.01, according to the terms of the problem. This could also be done with a loop involving the above flow diagram. After the above flow diagram is followed up through the limit for step II, we can insert steps to modify I, III, and V to affect the next higher location in line 19. But, notice that steps II, IV, and VI will all be set to affect 19.50. They must all be reset, but not to their original values. Every time the above flow diagram is entered for a new set of comparisons, steps II, IV, and VI must be set to initially affect a word in line 19 whose number is one greater than the word affected by steps I, III, and IV. New initial values for II, IV, and VI will have to be set in the program every time I, III, and V are modified.



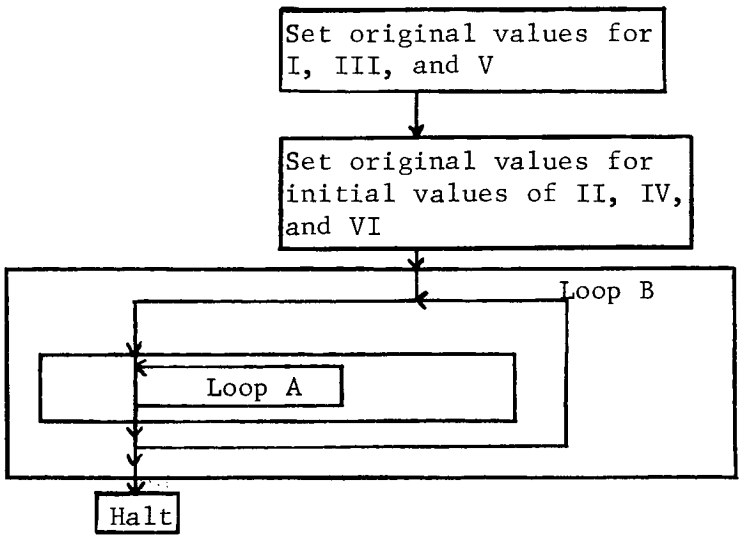
Notice that we have established a limit for the looping back to step I after modifying it, in order to prevent undesired continued operation of the program after a certain point. Only 50 numbers are to be ordered; the fiftieth is in 19.49. When all the preceding ones have been ordered, the one remaining in 19.49 must be in its correct position and need not be compared at all. So, when step I would call for 19.49, we wish to leave the loop, having accomplished all that was originally asked for.

We have, then, in the above flow diagram, two loops, one within the other. We will call the smaller one (which is operated a varying number of times per each operation of the larger one) loop A. The larger loop we will call loop B; it will be operated 49 times. A pass through a loop we will call an "iteration". The two loops are shown in the above flow diagram.

After we leave loop B, we are done, and the 50 numbers are ordered as desired. We could halt at this point, giving the HALT command an N equal to the starting location of the entire program, so that it could be repeated again, if desired.

There is one remaining difficulty with this program, as flow-diagrammed on the preceding page. After it has operated once, steps I through VI will be modified, and the program would not operate successfully in another complete pass, from the beginning, without some restoring. Every program which modifies itself should also restore itself to its initial condition, so that it can be operated as many times as desired without having to be reloaded, in its entirety, into the memory of computer. Such restoration by the program itself is called "housekeeping".

Housekeeping should be done initially in a program. We will therefore, further alter the preceding flow diagram, as follows:



The only remaining question is, "how do you modify a command in a program?"

Take step II in the above flow diagram, for instance. It calls for the transfer of 19.01 to AR+, with the sign changed and via the inverting gates. A command corresponding to this might be:

L	P	T	N	C	S	D
u6		01	50	3	19	29

To modify this command, transfer it into AR: (19.u6)$\longrightarrow$ARc. Use a C of 0 to do this. If the command itself is negative, that is an

indication of the fact that the command calls for a double-precision operation; this does not mean that we want to complement the binary number, however. All we want to do is to modify it in its present form, by adding a 1 to T. After (19.u6) is in AR, add the following constant ("dummy command") to it:

L	P	T	N	C	S	D
	u	01	00	0	00	00

This is called a "dummy command" because it will never be read and interpreted as a command; it is merely a constant, entered in decimal command form when making up the program using PPR, for the sake of convenience. Transfer this constant into AR+. The result in AR will be:

L	P	T	N	C	S	D
		02	50	3	19	29

Now store the present contents of AR into word u6 of the command line containing step II, and the next time this word is interpreted as a command it will be executed at word-time 02.

Notice that the dummy command has a prefix of u. Why? (Hint: will this make the command immediate or deferred? What will be the effect of this on the binary number generated by PPR?)

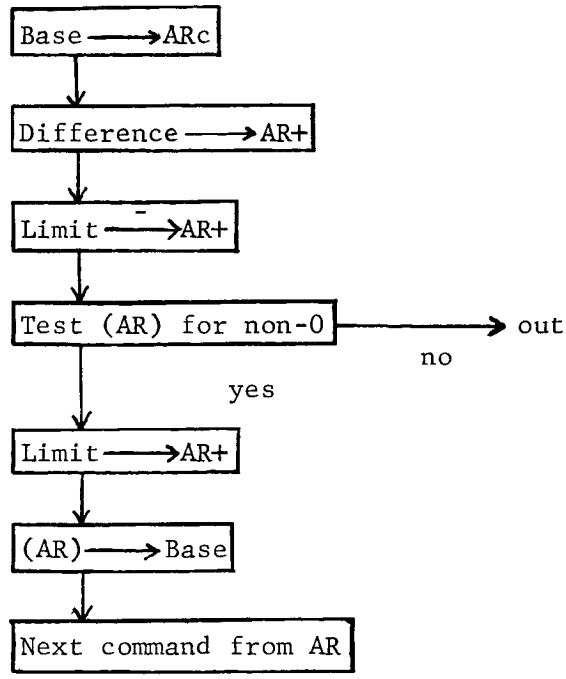
Rather than actually modify commands as they appear in a long line (which would necessarily entail great time delays, since the command would have to be picked up from a word in a long line, and then, in its modified form, be restored into the same word-position in the same long line), we would like to operate the modified command, in each case, out of AR, which is always available. Now the problem arises, if step I places a number in AR, how can we then modify a command there and execute it from there without destroying the number which was originally placed there by step I? The answer is that we will have to store the number called for by step I in a short line, then operate step II, with a destination of ARc, rather than AR+. Then we will place a command in our program which calls for the original number from its short-line temporary storage location. The destination of this command will be AR+.

Always be careful, when executing commands from AR, that you don't destroy valuable data already residing in AR.

INDEXING

"Indexing" is probably the easiest and most convenient way of modifying and operating a command out of AR. It involves a "Base" command, which is modified by a "Difference" (dummy command), and the result is restored into the Base, so that, on the next pass, the new base will again be modified, and so on. Usually there will be a "Limit" associated with such

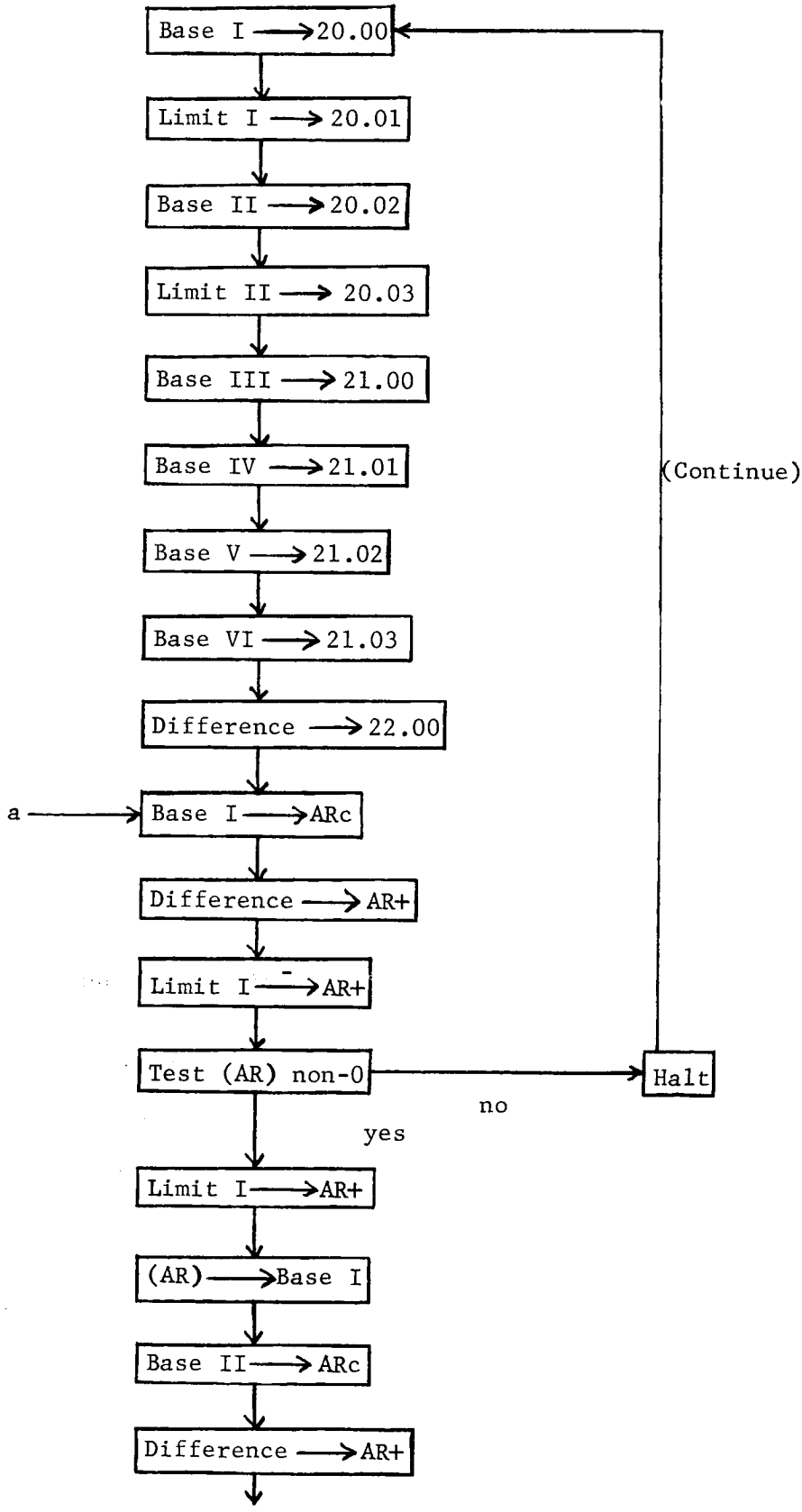
an index. If there is a limit, the modified version of the base will be checked against it each time, in order to determine whether or not the limit has been reached. All of this modification and checking will be done in AR, and the final contents of AR will then be operated as a command (unless the limit has been reached, of course). The sequence of steps might be:

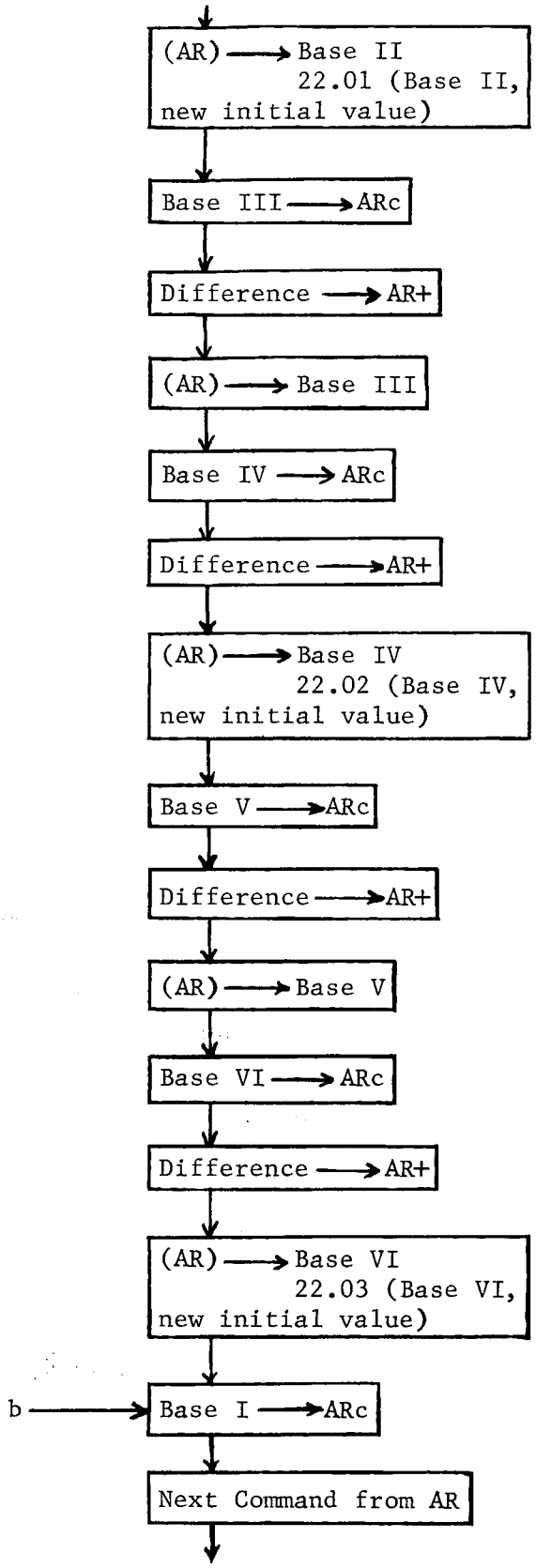


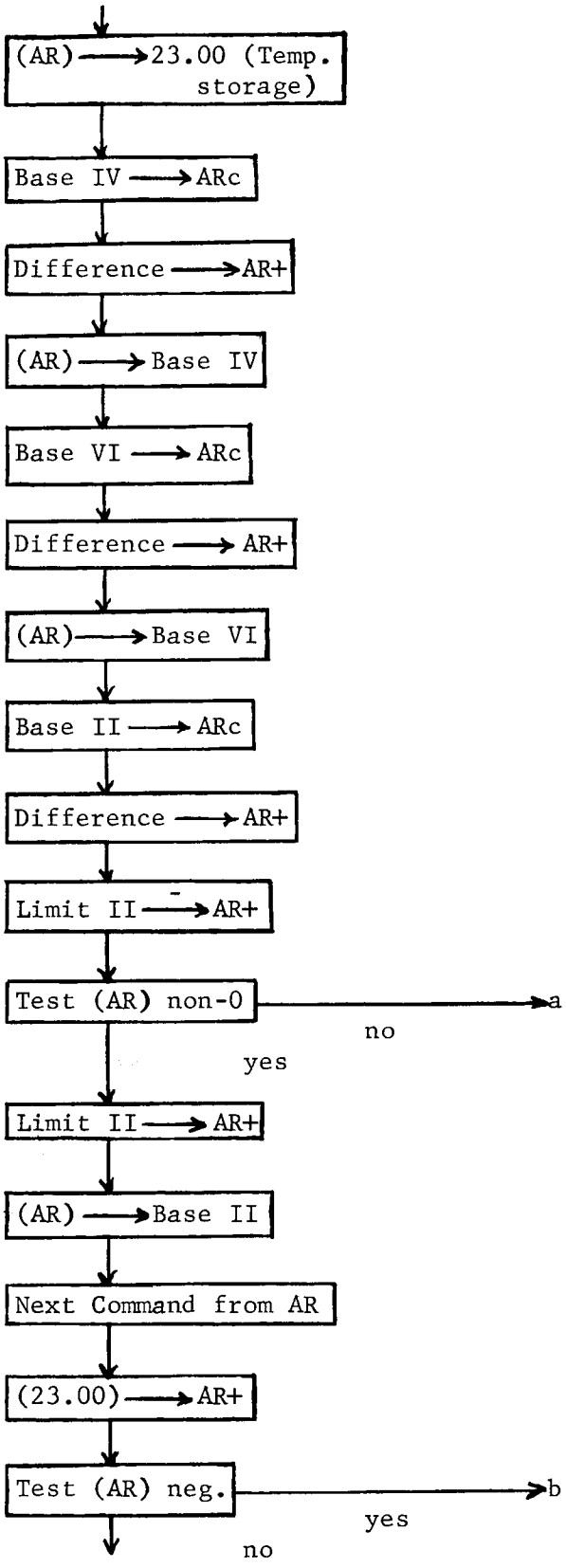
Such an index for step II might be:

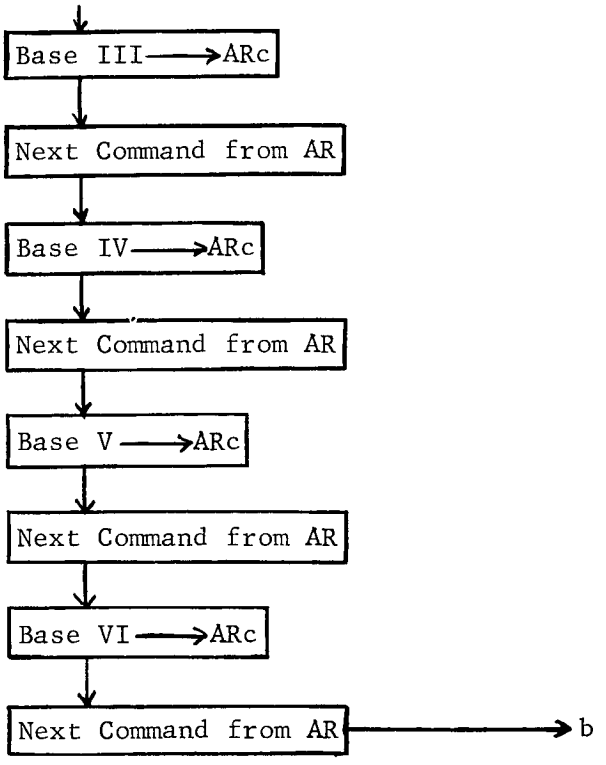
Base.....		00	50	3	19	28
Difference.....	u	01	00	0	00	00
Limit.....		50	50	3	19	28.

These could all be stored in a short line, and thus be available without undue delay during the modification process. Notice, in the preceding flow diagram, modification takes place prior to execution of the command, not after. If this is the case, and if we want to use the same steps for all passes through the loop, including the first, the Base must start out with a T number one lower than the first word-time at which we want to execute the command. Notice above that the Base has a T = 00, even though step II should initially contain T = 01. The previous flow diagrams for this problem must now be revised slightly. Notice in the revision that not all indices have been assigned a limit, because we know that some bases can never be increased too far, due to the fact that their modification is dependant upon modification of other bases, where limits have been imposed.


Base I ⟶ 20.00
Limit I ⟶ 20.01
Base II ⟶ 20.02
Limit II ⟶ 20.03
Base III ⟶ 21.00
Base IV ⟶ 21.01
Base V ⟶ 21.02
Base VI ⟶ 21.03
Difference ⟶ 22.00
a
Base I ⟶ ARc
Difference ⟶ AR+
Limit I ⟶ AR+
Test (AR) non-0
Halt
no
yes
Limit I ⟶ AR+
(AR) ⟶ Base I
Base II ⟶ ARc
Difference ⟶ AR+
(Continue)


(AR) → Base II
22.01 (Base II, new initial value)
Base III → ARc
Difference → AR+
(AR) → Base III
Base IV → ARc
Difference → AR+
(AR) → Base IV
22.02 (Base IV, new initial value)
Base V → ARc
Difference → AR+
(AR) → Base V
Base VI → ARc
Difference → AR+
(AR) → Base VI
22.03 (Base VI, new initial value)
b
Base I → ARc
Next Command from AR


(AR) ⟶ 23.00 (Temp. storage)
Base IV ⟶ ARc
Difference ⟶ AR+
(AR) ⟶ Base IV
Base VI ⟶ ARc
Difference ⟶ AR+
(AR) ⟶ Base VI
Base II ⟶ ARc
Difference ⟶ AR+
Limit II ⟶ AR+
Test (AR) non-0
a
no
yes
Limit II ⟶ AR+
(AR) ⟶ Base II
Next Command from AR
(23.00) ⟶ AR+
Test (AR) neg.
b
yes
no



The following coding sheets contain the individual commands in this program.

Notice that, in the Bases, T = w7 (127), and the command, so written, has deliberately been made immediate. The eight most significant bits in such a command will be:

(1) 01111111...

All of the Bases are initially modified, prior to being executed, through the addition of a Difference, whose eight most significant bits are:

(2) 00000001...

When (2) is added to (1), the result will be:

(3) 10000000...,

which will be a deferred command with T = 00. This is what is initially desired. Bases II, IV, and VI are further modified by the Difference, so that T = 01 prior to their execution for the first time.

This program, as written, requires slightly less than three minutes to sort numbers which are already in their proper order, and it requires eight minutes to sort numbers in the worst possible arrangement

initially. It is not by any means, a "good" program, because it is inefficient. Notice that it will exchange numbers which are equal, and it will "sort" fifty numbers which are already in proper order.

It was written in this manner to demonstrate the use of indexing in as straight-forward, and uncomplicated, a way as possible.

The "Notes" column on each coding sheet has been left blank, for you to fill-in, as a review.

Bendix Computer — Los Angeles 45, California

G-15D

PROGRAM PROBLEM: 50-Word Sort

Prepared by _____

Page 1 of 3

Date: _____

Line 00

0	1	2	3	L	P	T or L_k	N	C	S	D	BP	NOTES
4	5	6	7	00		01	04	0	00	00		
8	9	10	11	04	u	09	09	0	00	20		
12	13	14	15	05		49	49	0	19	28		Limit I
16	17	18	19	06	u	w7	51	3	19	28		Base II
20	21	22	23	07		50	51	3	19	28		Limit II
24	25	26	27	08	u	w7	49	0	19	28		Base I
28	29	30	31	09	u	14	14	0	00	21		
32	33	34	35	10	u	w7	60	0	24	19		Base V
36	37	38	39	11	u	w7	u4	0	25	19		Base VI
40	41	42	43	12	u	w7	52	0	19	28		Base III
44	45	46	47	13	u	w7	58	0	19	28		Base IV
48	49	50	51	14	u	19	19	0	00	22		
52	53	54	55	15								Base VI
56	57	58	59	16	u	01	00	0	00	00		Difference
60	61	62	63	17								Base II
64	65	66	67	18								Base IV
68	69	70	71	19		20	21	0	20	28		a
72	73	74	75	21		24	25	0	22	29		
76	77	78	79	25		29	30	3	20	29		
80	81	82	83	30		31	32	0	28	27		
84	85	86	87	32		34	00	0	16	31		
88	89	90	91	33		37	38	0	20	29		
92	93	94	95	38		40	41	0	28	20		
96	97	98	99	41		42	43	0	20	28		
u0	u1	u2	u3	43		44	45	0	22	29		
u4	u5	u6		45		46	47	0	28	20		

FORM BCD 108-5.0

Bendix Computer

Los Angeles 45, California

G-15D

PROGRAM PROBLEM: 50-Word Sort

Prepared by _____

Date: _____

Line 00

0	1	2	3	L	P	T or L_k	N	C	S	D	BP	NOTES
4	5	6	7	47		49	50	0	28	22		
8	9	10	11	50		52	53	0	21	28		
12	13	14	15	53		56	57	0	22	29		
16	17	18	19	57		60	61	0	28	21		
20	21	22	23	61		65	66	0	21	28		
24	25	26	27	66		68	70	0	22	29		
28	29	30	31	70		73	74	0	28	21		
32	33	34	35	74		78	79	0	28	22		
36	37	38	39	79		82	83	0	21	28		
40	41	42	43	83		84	85	0	22	29		
44	45	46	47	85		86	87	0	28	21		
48	49	50	51	87		91	92	0	21	28		
52	53	54	55	92		96	97	0	22	29		
56	57	58	59	97		99	u0	0	28	21		
60	61	62	63	u0		u3	u4	0	28	22		
64	65	66	67	u4		00	02	0	20	28		b
68	69	70	71	02		04	u6	0	31	31		
72	73	74	75	u6		xx	49	0	19	28		
76	77	78	79	49		52	54	0	28	23		
80	81	82	83	54		58	59	0	22	28		
84	85	86	87	59		60	62	0	22	29		
88	89	90	91	62		66	67	0	28	22		
92	93	94	95	67		71	72	0	22	28		
96	97	98	99	72		76	77	0	22	29		
u0	u1	u2	u3	77		79	80	0	28	22		
u4	u5	u6		80		81	82	0	22	28		

FORM BCD 108-5.0

Bendix Computer — Los Angeles 45, California

Page 3 of 3

G-15D

PROGRAM PROBLEM: 50-Word Sort

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
82		84	86	0	22	29		
86		87	88	3	20	29		
88		89	19	0	28	27		
20		23	24	0	20	29		
24		25	02	0	28	22		
u6		xx	51	3	19	28		
51		52	55	0	23	29		
55		57	u3	0	22	31		
u3		u4	02	0	21	28		
u6		xx	52	0	19	28		
52		54	56	4	28	25		
56		58	02	0	22	28		
u6		xx	58	0	19	28		
58		60	63	4	28	24		
63		66	02	0	21	28		
u6		xx	60	0	24	19		
60		63	02	0	22	28		
u6		xx	u4	0	25	19		

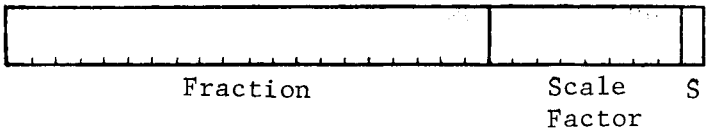
FORM BCD 108-5.0

FLOATING-POINT OPERATION

When the range of numbers which is anticipated during a calculation is either very large or unpredictable, it may become extremely difficult to work with fixed-point data. In such situations, floating-point arithmetic is usually used.

In floating-point operation a data word is divided into two major portions, similar in function to the characteristic and the mantissa of a logarithm. One portion contains a fraction; the other, an integral power of 2 by which the fraction is to be multiplied. The sign-bit of the data word is associated with the fraction, as its algebraic sign. Any arbitrary decision as to how many bits of a word will be used to express the fraction and how many will be reserved to express the accompanying exponent of the base of 2 is permissible.

The Bendix programmers have generally adopted the following rule. The fraction part of the data word is contained in bit positions T10 through T29. The sign of the fraction is contained in bit T1 of the data word and the scale factor is contained in bits T2 through T9.



As T1 of the data word indicates the algebraic sign of the fraction and, since signed exponents are desirable, the scale factor is formed by adding 128 to the power of 2. Thus, the range of the scale factor is $0 \leq$ scale factor ≤ 255, while the range of the power of 2 is $-128 \leq$ power of $2 \leq 127$. Therefore, the larger the scale factor, the smaller the number that is represented in the fraction or mantissa of the data word.

In floating-point operation it is often desirable to keep as many significant bits as possible in the mantissa of a data word and to use the scale factor to express all leading zeroes. A special command is available in the G-15 that will allow the programmer to keep all significant bits of a data word and will also keep a count of how many leading zeroes were dropped. This command is the "normalize" command. It's normal form is:

L 54 N 0 27 31.

The "normalize" command operates in the following manner:

1. $MQ_{0,1}$ and AR are the only registers involved in the execution of this instruction.

2. $MQ_{0,1}$ will shift left until a 1 bit appears in T29 of MQ_1. For every leading zero that is lost from MQ_1 a 1 is added to AR in

the 2^{-28} position. A zero will be inserted in T1 of MQ_0 after each shift.

3. As in a shift command, the location for the "normalize" command must be odd.

Unlike some other machines, floating-point arithmetic operations are not automatic in the G-15. It is up to the programmer to keep track of the scale factors and fractions that he is working with. Thus multiplication and division become rather simple programming problems in floating-point because all the programmer must do is separate the fractional portions of the numbers from the scale factors, do the multiplication or division with the fractions, either add or subtract the scale factors to get the new scale factor of the product or quotient and then combine the product or quotient with the new scale factor for storage.

Addition and subtraction are now more difficult because the programmer must compare the scale factors of the numbers to be added or subtracted, and if they are unequal, must shift one of the numbers until the scale factors do become equal. He must keep track of the number of shifts to get the new scale factor of the number. A shift command is available to help the programmer keep a count of the number of shifts that have been made. This is a special command and its normal form is:

L 54 N 0 26 31.

This command will operate on ID and MQ in the same manner as the shift command mentioned earlier with the following exception, for each bit-position shift in ID and MQ a one will be added to AR scaled 2^{-28}. If at any time during the shift the one added to AR causes an end-around-carry in AR, the shift will terminate and no more bits will be shifted in ID and MQ. Therefore, by using this command the programmer can either count the shifts in AR or use AR, loaded with the complement of the number of bit-positions that he desires shifted, to control the number of shifts. The location of this command must be odd so that execution will start at an even word-time.

The two extract commands previously discussed enable the programmer to separate the fraction and the scale factor so that he can operate on them separately. After the operations have been performed on both the scale factors and the fraction, the programmer will want to combine the new fractions and scale factors for storage. Another extract command is available for this purpose. Its normal form is:

L T N 0 27 D.

This extract command operates in the following manner:

1. Where there are one bits in the mask at word-time T in line 20, the corresponding bits in line 21 are extracted to word-time T of the destination.

2. Where there are zero bits in the mask at word-time T in line 20, the corresponding bits in AR are extracted to word-time T of the destination.

Thus, by use of this extract command the programmer can unite his new scale factor with the new fraction for storage.

From the preceding paragraphs, it can be seen that floating-point operation in the G-15, although it presents a somewhat more difficult programming effort, can operate with very large or very small numbers that fixed-point operation could not handle.

MISCELLANEOUS TOPICS TO BE COVERED BEFORE CLOSING

So far, in discussing outputs, we have mentioned the possibility of either punching or typing the contents of line 19. If it is desired to get both a tape and a typed copy of the line's contents, two separate outputs would have to be called for.

On the base of the typewriter, as shown on page 130, there is a punch switch. If this switch is on when a type-out of line 19's contents is called for, the characters of output, as well as activating the typewriter, will also activate the punch, and the two outputs will proceed simultaneously as the result of one command (Type line 19). Of course the speed of the punch will be slowed down to the speed of the typewriter, which is considerably slower than the normal speed of the punch, when used alone. This punch switch is merely a physical connection enabling the pulses which reach the typewriter to also reach the punch.

Of course, the punch switch must be on prior to execution of the "type line 19" command. And so the question arises, "How can you be sure the punch switch has been manually turned on?" A test command is available which tests for this condition. It is a special command, D = 31, S = 17, C = 1. If the switch is on, the next command will be taken from N + 1; if the switch is off, the next command will be taken from N.

Normally, you would use this test prior to calling for a type-out of line 19's contents, if you want to be sure that a tape will also be punched. The "type line 19" command would be available only after the test was answered affirmatively, the next command coming from N + 1.

If the answer is "no", you would normally want to repeat the test until the switch is turned on. In such a case, it would be desirable to call the operator's attention to the fact that he is to throw the punch switch on. It is reasonable to assume that the operator will not be aware of this desire of yours; he might not even be at the computer (coffee-break, of course). What then?

There is a special command available (D = 31, S = 17, C = 0) which rings a bell inside the computer once each time it is executed. At N you could give this command, and then go back to the test again. The bell would thus be rung once each time the test is executed and answered negatively. Presumably this continuous bell-ringing would cause somebody to come to

the computer. There would be the operating instructions for your program, containing one all-important sentence: "If the bell continuously rings, turn on the punch switch."

The ringing of the bell requires a physical action on the part of the computer: the movement of a solenoid, striking the rim of the bell. It is a safe bet that, whenever physical action is involved, timing problems occur. In this case, it is safe to allow one complete drum-cycle execution time for the command. This will be sufficient to cause the solenoid to ring the bell. Since D = 31 in the "ring bell" command, PPR will make the command immediate. Set T (the flag) equal to L + 1, allowing a complete drum cycle of execution.

Solenoids require a recovery time, and, if the solenoid which rings the bell is not allowed to recover after each ring, it will merely vibrate against the bell, causing a buzz, rather than a series of individual rings. Recovery time for this solenoid is three drum cycles. Therefore, three drum cycles must elapse between executions of the "ring bell" command. These can be achieved through purposeful "bad" coding of commands, requiring "maximum access-time". For instance, the following command will waste two drum cycles:

L	P	T	N	C	S	D
51		51	50	0	00	00
50		..	..	.	..	..

Notice that both the "ring bell" and "test punch switch" commands have a special S code of 17.

ring bell:	T	N	0	17	31
test punch switch on:	T	N	1	17	31.

The punch switch test will also ring the bell, if a full drum cycle of operation is allowed (T = L + 1). Of course, recovery time for the solenoid is still necessary.

Recovery time is also necessary in one other case already discussed in this text. When type-in is called for, the stop code of the input is supplied by striking the "s" key. There is a physical contact involved in this action, and that contact will remain closed for approximately 1/10 second (3 drum cycles). If another input or output is initiated prior to the opening of that contact, the stop code pulse will still be present, and that input or output will immediately stop.

Therefore, rule: after completion of typewriter input (ready test is successfully met), allow three drum cycles to elapse before initiating any other input or output.

Concerning punched tape output, one point should be made quite clear. The reloading of the format will cause a reload code to be punched on tape. If the tape being punched is later to be read into the computer

(interim storage) you must be sure that it is originally punched under control of an output format which calls for four full words prior to the reload.

The number track, mentioned previously, is a timing channel physically located on the surface of the drum. It occupies a long line similar to the long lines already discussed, and this long line recirculates once per drum cycle in the same manner as all other long lines. There is no way to program the loading of this channel: it is loaded automatically when the computer is turned on. Two blocks of punched tape will automatically be read: the computer will automatically load the number track with the information from the first; the second should be a loader program designed to read in a test routine, in the normal manner. The contents of this block of tape will occupy line 19. Turn-on procedure, including use of test routines, is fully discussed in the Operating Manual.

The function of the number track is to affix specific word-times to all words in memory. At each word-time, in the number track, the T and N portions of the word contain the number of the next word-time to come up under the read-heads. The computer compares T's and N's of commands being interpreted with the T and N available from the number track, and in this way is able to determine when to execute or read a command.

Page 206 contains a type-out of the number track. The words are typed in four-word groups, reading from left to right, one group per line. The first word typed out is u7, and the last is 00. Notice that in all words except u7, the I/D bit is set equal to 1.

Word u7 is unlike any of the others. You would expect its T and N to contain 00, but this is not the case. The counting of T and N is, of necessity, modulo 128 (there are seven bits for each). However, there are only 108 words per long line, and, therefore, only 108 word-times possible for either T or N. Word u7 in the number track contains 20 in each of these positions, so that, when this is added to the respective time counters, they will be cleared to 00 indicating that the next word-time will be 00. The meaning of other bits set in word u7 of the number track would require more engineering background than the reader is assumed to have at this point.

Notice, if you store a command in word u7 of a command line, and if you expect your program to read and interpret this command at word-time u7, it will be interpreted simultaneously with the jumping ahead of the counters by 20 word-times. Therefore, if you want to do this, the T and N portions of your command must equal the desired word-time plus 20 in each case. For example, if, at word-time u7, you desire to call for the transfer of word 10 from line 08 to line 09, and then you desire to take your next command at word-time 11, your command would be coded in the following manner:

L	P	T	N	C	S	D
u7		30	31	0	08	09
11		..	..	.	..	..

Number Track

-1414794	yv6v000	yu6u000	y969000
y868000	y767000	y666000	y565000
y464000	y363000	y262000	y161000
y060000	xz5z000	xy5y000	xx5x000
xw5w000	xv5v000	xu5u000	x959000
x858000	x757000	x656000	x555000
x454000	x353000	x252000	x151000
x050000	wz4z000	wy4y000	wx4x000
ww4w000	wv4v000	wu4u000	w949000
w848000	w747000	w646000	w545000
w444000	w343000	w242000	w141000
w040000	vz3z000	vy3y000	vx3x000
vw3w000	vv3v000	vu3u000	v939000
v838000	v737000	v636000	v535000
v434000	v333000	v232000	v131000
v030000	uz2z000	uy2y000	ux2x000
uw2w000	uv2v000	uu2u000	u929000
u828000	u727000	u626000	u525000
u424000	u323000	u222000	u121000
u020000	9z1z000	9y1y000	9x1x000
9w1w000	9v1v000	9u1u000	9919000
9818000	9717000	9616000	9515000
9414000	9313000	9212000	9111000
9010000	8z0z000	8y0y000	8x0x000
8w0w000	8v0v000	8u0u000	8909000
8808000	8707000	8606000	8505000
8404000	8303000	8202000	8101000

Powers of "2"

k = no. of pre-zeros

2^n	2^{-n}	k	n
1	1	0	0
2	.50000000	0	1
4	.25000000	0	2
8	.12500000	0	3
16	.06250000	0	4
32	.03125000	0	5
64	.01562500	0	6
128	.00781250	0	7
256	.k3906250	2	8
512	.k1953125	2	9
1024	.k9765625	3	10
2048	.k4882812	3	11
4096	.k2441406	3	12
8192	.k1220703	3	13
16384	.k6103516	4	14
32768	.k3051758	4	15
65536	.k1525879	4	16
131072	.k7629395	5	17
262144	.k3814697	5	18
524288	.k1907349	5	19
1048576	.k9536743	6	20
2097152	.k4768372	6	21
4194304	.k2384186	6	22
8388608	.k1192093	6	23
16777216	.k5960464	7	24
33554432	.k2980232	7	25
67108864	.k1490116	7	26
134217728	.k7450581	8	27
268435456	.k3725290	8	28
536870912	.k1862645	8	29

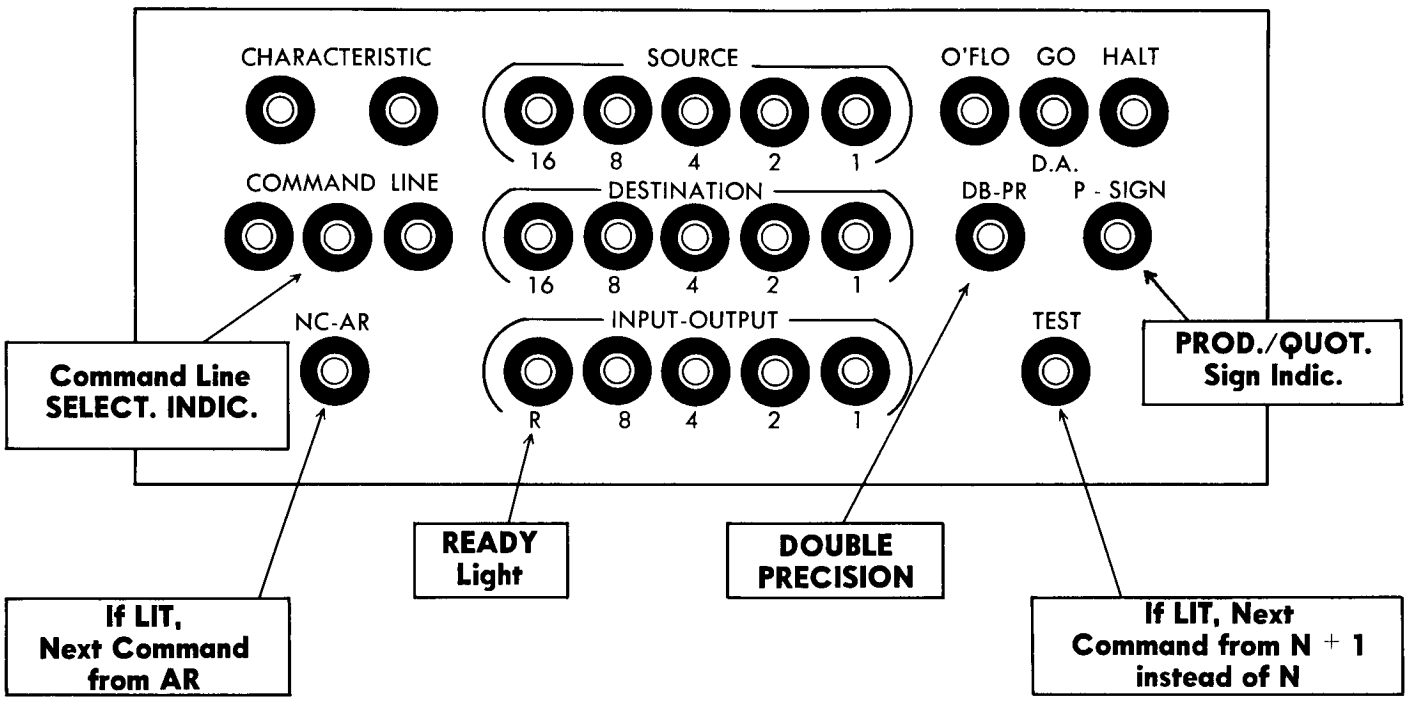
Hex Powers of "10"

k = no. of pre-zeros

10^n(Hex)	10^{-n}(Hex)	k	n
0000001	1	0	0
000000u	.199999u	0	1
0000064	.k28z5w29	1	2
00003y8	.k4189375	2	3
0002710	.k68xv8vv	3	4
00186u0	.ku7w5uw4	4	5
00z4240	.k10w6z7u	4	6
0989680	.k1ux7z2u	5	7
5z5y100	.k2uz31xw	6	8

Constants

π = 3.14159265
π^2 = 9.86960440
$\sqrt{\pi}$ = 1.77245385
e = 2.71828183
log e = 0.43429448
log 2 = 0.30103000
log = 0.49714987


CHARACTERISTIC
SOURCE
O'FLO
GO
HALT
16
8
4
2
1
D.A.
COMMAND LINE
DESTINATION
DB-PR
P - SIGN
16
8
4
2
1
NC-AR
INPUT-OUTPUT
TEST
R
8
4
2
1
Command Line SELECT. INDIC.
PROD./QUOT. Sign Indic.
READY Light
DOUBLE PRECISION
If LIT, Next Command from AR
If LIT, Next Command from N + 1 instead of N

INDEX

Abbreviated Format (see Input/Output System)
Absolute Value (see Magnitude)
Access Time, 10, 150, 152, 159-161
Accumulator
    Double Precision (see PN Register)
    Single Precision (see AR Register)
"Add", 20, 103
"Add Magnitude", 22, 25, 103
Address, 7, 10-11, 15, 66
    of next command, 16, (see also Next Command)
Analysis (see Problem Analysis)
Arithmetic Operations, 20-27, 33-41, 202
AR Register, 11, 20

Bell (see Ring Bell)
Binary-Coded-Decimal (see Decimal Inputs)
Binary Point
    in machine (see Machine Point)
    true, 91
Binary Scaling (see Scaling)
Bits
    ordering of, 6
Blank Leader (see Leader)
Block of Punched Tape, 18, 138, 140
Block Operations, 25-26, 68-69, 163-165
"Bootstrap" (see Loader)
Break-Point Operation, 16, 141-142

"C" Codes, 13-15, 64-66
Characteristic (CH), 61-66
Check Sum, 162
Clear (see Erase)
"Clear & Add", 20, 103
"Clear & Add Magnitude", 22, 25, 103
"Clear & Subtract", 21, 62, 103
Coding Sheets, 152-158
Command
    binary form, 60-61
    decimal form, 13, 150-151, 162
    modification of, 122, 186-191, (see also Indexing)
    ordering of, 69
        next command from AR, 51, 122, 186-187, 191
    parts of with respect to computer operation, 13-17, 61-70, 141-142, 205
    restoration of (see "Housekeeping")
    special (see Special Command)

Command Line, 28, 69-70
    selection of, 134
Complement, 23-24, 63-64
Complementation (see Inverting Gates)
Compute Switch, 17, 130, 142, 167
Control Information (see Timing)
Copy, 13-14
Copy via AR, (see Transfer via AR)
Cycle, drum (see Drum Cycle)

Debugging, 141-142
Decimal Command (see Command)
Decimal Inputs, 31, 167-168
    conversion to binary, 28-30, 168-170
Decimal Scaling (see Scaling)
Decision-Making (see Test Commands)
Deferred Command, 15, 69, 160
Destination, 15, 66-68
Divide, 39-40, 76-84
    considered as a ratio, 81-82
    round-off (see Round-Off)
Double Precision, 11-12
Drum Cycle, 8
Drum Revolution, 6
Drum Memory, 5-13

Enable Actions, 17, 133-134, 142
Enable Switch, 17, 130, 133, 142
End-Around-Carry, 106-108
Erase, 8-9
Erase Head, 8-9
Exchange AR with Memory, 13-15, 61-62, 163-165
    D = two-word register (see Two-Word Registers)
Extract, 44-47, 177-180, 202-203

Fixed-Point Operation (see Scaling)
Flag (see Immediate Command)
Flip-Flop
    sign, 131
Floating-Point Operation, 201-203, (see also Scaling, Floating-Point)
Flow Diagrams
    description of, 4-5
Format (see Input/Output System)

Halt Command, 56, 111
"Housekeeping", 190

ID Register (see Two-Word Registers)
Immediate Command, 16, 18-19, 68-69, 160
Immediate-Deferred Bit, 68
Indexing, 50, 191-196
Input/Output System
    commands, normal, 18, 142-143
    enable actions (see Enable Actions)
    normal inputs, 17, 128-133
        punched tape, 18, 140
        typewriter, 17-19, 30-31, 128-134, 204
            drawing of, 130
    normal outputs, 51-52
        abbreviated format, 162
        format, 52-55, 135-136, 138, 204-205
        punched tape, 54, 134-139, 145-147, 162-163, 203-205
        typewriter, 140-141, 203-204
    ready (see Test Ready)
    requirements, 127
    simultaneous with computation, 19, 144
    stop code, 132, 136, 140
        recovery time for S key, 204
Introduction, 1-3
Inverting Gates, 13, 61, 67, 106
IP Flip-Flop (see Two-Word Registers, use of, in multiplication)

Leader, 145-147
Loader Program, 60, 147-149
Logical Addition, 27, 51, 143
Logical Operations, 42-47
Long Lines, 6-7
Loop
    simple, 47-49, 145
    through command modification and indexing, 50-51, 186-196

Machine Point, 89
Magnitude
    of a number (see following: "Clear & Add", "Add", and "Subtract")
Mark & Transfer, 28, 29, 114-121
Mask (see Extract)
Maximum Access (see Access Time)
Memory (see Drum Memory)
Method of Solution, 3-4
Millisecond, 8
Minimum Access (see Access Time)
MQ Register (see Two-Word Register)
Multiply, 35-38, 70-75
    Round-Off (see Round-Off)

Neons, front panel, 18, 132, 208
Next Command (see Commands, ordering of)
Next Command from AR, 51, 122, 186-187, 191
"Normalize", 201-202
Notes, 16
Number
    BCD (see Decimal Inputs)
    conversions, 27, 28-30, 55, 168-170, 175-180, 182-186
    machine form, 89
    table of powers, 207
Number Track, 12-13, 26-27, 205-206

Operand, 15, 66
Operation Code
    special (see Special Operations)
Output (see Input/Output System)
Overflow (see also Test)
    definition, 33
    indicator, to turn off, 34, 111-112
    resulting from divide, 80
    temporary, 79

Photo Reader, 18
PPR (see Program Preparation Routine)
PN Register (see Two-Word Register)
Precession, 163-165
Prefix, 16, 18, 151
"Princeton" Round-Off (see Round-Off)
Problem Analysis, 3
Problem Method (see Method of Solution)
Program Preparation Routine, 13, 18-20, 59-60, 150-151, 160-163, 165-167
Psuedo-Commands for PPR, 151, 162, 165-167
Punched Tape (see Input/Output System)
Punch Switch, 130, 203-204

Range of Values
    associated with scaling, 97-100, 171
Read Heads, 6, 8-9
Ready (see Test Ready)
Recirculating Memory, 8-9, 67-68
Recovery Time, 204
Relative Timing Number, 36, 39
Rescaling, 91-92
Return Command, 29, 114-121, 142
Return Line, 29, 114
Revolution, Drum (see Drum Revolution)
Ring Bell, 203-204

Round-Off
after division, 82
of binary number prior to conversion to BCD, 184

Scaling, Fixed Point
binary, 89-97
decimal, 32-33, 170-171
in BCD output, 175, 180
Scaling, Floating Point, 201-202
Selector, Source & Destination, 67
Self-Destroying Loader (see Loader)
Set Ready, 146-147
Shift, 42-44, 93-94, 201-202
Short Lines, 9-12
Sign Flip-Flop (see Flip-Flop)
Sign Time (see Bits, ordering of)
"Single-Cycle", 142
Single-Double Precision Bit, 61, 64-66
Solenoid, 204
Special Commands, 16-17, 18, 70
table of, 56-59
Sorting, 186-197
Source, 15, 66-68
Stop Code (see Input/Output System)
"Store", 103
"Store Magnitude", 22
"Subtract", 14-15, 20, 62, 103
"Subtract Magnitude", 110
Subroutines, 27, 113, 118
square root, 40

"T" Numbers
pertaining to bits, 6
Temporary Overflow (see Overflow)
Test Commands, 19, 105-109
non-zero, 41, 109
overflow, 33-34, 106-108
punch switch on, 109, 203-204
and ring bell, 204
ready, 19-20, 108, 143
sign of AR negative, 41, 108
Timing (see Machine Time)
Problems where physical action is required, 204
Timing and Control Information, 12, (see also Number Track)
in immediate commands (see Immediate Commands)
relative timing number (see Relative Timing Number)
"TO" Pulse, 12
Transfer Control (see Mark & Transfer)

Transfer via AR
divide, 82, 84
multiply, 82-23
Two-Word Registers, 10-12
double precision accumulator, PN register, 11, 21
exchange of AR with memory, D = two-word register, 13-15
summary of rules, 37-38
use of, in division, 39-40, 76-84
use of, in multiplication, 35-38, 70-75

Word-Time, 6-7, 8, (see also Number Track)
as part of address, 15, 66
"Working Memory", 7
Write Heads, 6, 8-9

Offices:

BOSTON 16

114 Waltham Street
Lexington 73, Massachusetts
862-7976

CHICAGO 11

919 N. Michigan Avenue
MIchigan 2-6692

CLEVELAND 13

55 Public Square
CHerry 1-7789

DALLAS 1

2626 Mockingbird Lane
FLeetwood 1-9951

DAYTON 2

1900 Hulman Bldg.
BAldwin 6-2341, Area code 513

DETROIT 37

12950 West Eight Mile Road
JOrdan 6-8789

HUNTSVILLE

Holiday Office Center
Memorial Parkway
South 539-8471

LOS ANGELES

291 S. La Cienega Boulevard
Beverly Hills, California
OLeander 5-9610

NEW YORK 17

205 East 42nd Street
Room 1205
ORegon 9-6990

PHILADELPHIA

723 Street Road
Southampton, Pa.
ELmwod 5-0600, Area code 215

WASHINGTON 6, D. C.

1000 Connecticut Avenue, N.W.
STerling 3-0311

CANADA

Computing Devices of Canada

P.O. Box 508
Ottawa 4, Ontario, Canada
TAlbot 8-2711

OTHER COUNTRIES

Bendix International Division

205 E. 42nd Street
New York 17, New York
MUrray Hill 3-1100

Bendix Computer Division
LOS ANGELES 45, CALIFORNIA

July 1961

APR-01601-1