"C"	S/D	CH	Meaning
			mented, if that was called for originally, to the second word of the double-precision receiving address.
7	1	11	If 28 or 29 is specified as receiving address, change sign of first word of double-precision operand and transfer it via the inverting gates to AR. Then transfer the second word, all 29 bits, via the inverting gates also, to AR.

* Note: If this table is being used for reference, see page 82.

Notice here that the operations listed so far do not constitute the complete list of operations available in the computer; to name two very important ones which were not included, multiply and divide. We will see later how these operations, and many others, are called for, but first we must complete the description of the basic parts of a command.

So far all we have discussed are operations. Another part of each command is the address of the operand: the computer must be told on which number in its memory the operation is to be performed. Remember that it has been pointed out that an exact address in memory consists of a line number plus a word number, called a word-time. The line in which the operand is located is called the "source", and, in the layout of a command on page 61, this number is referred to as "S". If the line containing the operand is called a source, the obvious name for the line which will receive the transferred word is "destination". This is referred to in the layout of a command as "D". S and D can each range from 00 through 31, although the meaning of 31 has not yet been explained. In binary (all words in memory are in binary), 5 bits are required to represent the decimal number 31.

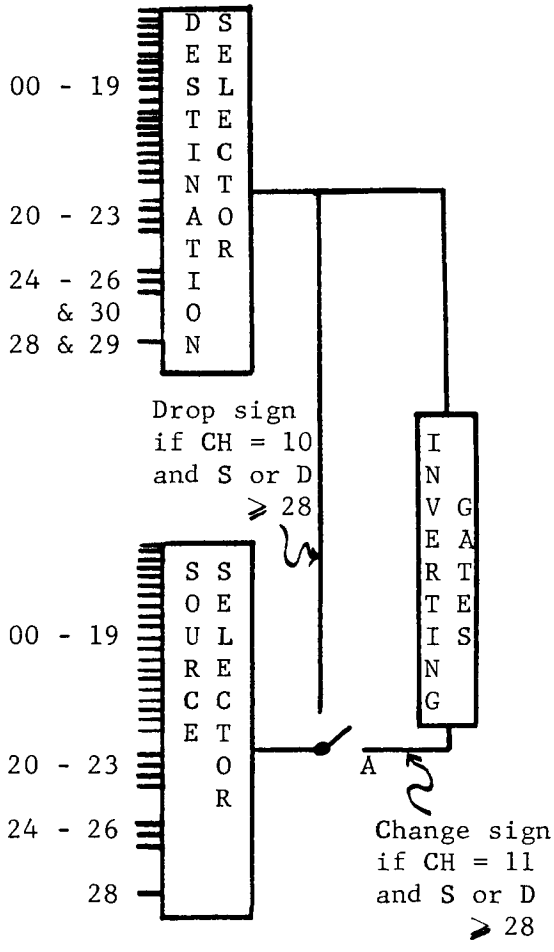
$$31_{(10)} = 11111_{(2)}$$

Notice that five bits have been allotted to both S and D. Since no address is complete without a word-time, there must be an allocation for this in a command. There is; it is "T". There are 108 word-times per long-line. No other line in memory requires more; in fact, no other line in memory requires nearly as many, so the range 00 - 107 (u7) will be sufficient.

$$107_{(10)} = 6v_{(16)} = 1101011_{(2)},$$

so seven bits will be sufficient for T; notice that seven have been allowed. Now, is T combined with S to form an address, or with D? The answer is: with both! So we see that, if the straight transfer

of a single word from one long line to another is called for, the word being transferred will, upon execution of the command, occupy the same word-time in the new line as it does in the old. The drawing below will clarify this.



As long as no command is being executed, the read-heads for each line in memory (there are 28 of them plus the number track) are connected to the corresponding write-heads, and bits, words, and whole lines are recirculating merrily along. This is all going on behind the drawing to the left. But then, when a command is exeuted, things begin to happen. If no commands were ever executed, it would be an easy matter to understand what's going on in the computer, but computers, as a rule, don't do much of anything useful if no commands are being executed. When a command is executed, it has a source (S), a destination (D), a word-time (T) during which it is to be executed, and an operation (CH and S/D), and other things we won't mention here. During word-time T a word is entering the source selector from each line in memory, as shown in the drawing. Notice that only one line number for PN is shown (26, not 30); this is because PN can be a source of data in only one capacity (as a storage location).

In the case of AR, 29 is an illegal line number for AR as a source. AR must be referred to as line 28 when being treated as a source. The source selector now has a word from each line, and it must pick out the correct one and ignore the rest. Of course it can do this, because the command has informed it of the line specified by S. The word from memory whose address is S.T leaves the selector, on its way to be processed.

Processing actually consists of being transferred back to memory again, over the proper circuit, which, in some cases, will perform an addition of bits. It also may consist of complementing a negative number. Part of the processing will be called for by the CH in the command. Depending on the value of CH, switch A in the drawing may be in one position or the other, causing the word being transferred to pass around or through the inverting gates. The word then arrives at the destination selector. The destination selector will usually disconnect the read-head for the line chosen as D from the corresponding write-head, pre-

venting recirculation of word T in that line. It will allow all other read-write connections to remain intact, so that word T in all other lines is being recirculated. It will then take the word it has received from the source selector and feed it to the proper line. Thus, what was originally in the destination line at word-time T is lost and replaced with word T from the source line. The source line has recirculated and still contains word T. If the destination is either 29 or 30, the original contents of the destination line at word time T is not lost, but is added to word T arriving from the destination selector.

In order to hold down the size of this book to a single volume, we will leave it to the reader to trace through this procedure for each operation code listed in the preceding table.

IMMEDIATE vs. DEFERRED COMMANDS

A series of commands could be written to perform any of these operations on a sequance of words; S, D, and the operation could be the same in each of the series of commands, with T being increased by 1, in the case of single-precision operation, or by 2, in the case of double-precision operation, in each succeeding command. As an example, a straight single-precision transfer of 04.10 to 05.10 would be coded with a "C" code of 0, a source of 04, a destination of 05, and T = 10. This could be followed by another command with the same "C" code, the same source, the same destination, but with T = 11. Then T could be increased by 1 again, and so on. Up to a whole long line could be transferred, one word at a time, in this way. By this method, it would require 108 commands to transfer 108 words. There is a way of accomplishing this with one command: it is to code the command in such a way that its execution will cover any desired number of contiguous word-times. In the layout of a command, as shown on page 61, bit 29 of the command is a one-bit indicator called "I/D". The "I" stands for "immediate". An immediate command is one which will be executed immediately after it is read and interpreted. Its execution will continue until the computer is told to stop the execution. The T number in such a command serves as a "flag", telling the computer when to stop the execution of the command. The execution will be stopped before T, but after the immediately preceding word-time. To indicate immediate execution of a command, bit 29 of the command is set with 0. In the above example, if it was desired to transfer words 10 - 15 from line 04 to line 05, a command with a "C" code of 0, S = 4, D = 5, T = 16, and I/D = 0, could be located at word 09 of some line out of which it would be read. Which line of memory the command would be in is as yet an open question. The command would be read at word-time 09 and executed immediately, meaning that its execution would start in the very next word-time, 10. It would continue operating through 10, 11, 12, 13, 14, and 15. The T number of 16 would serve as a flag, stopping the operation after word-time 15 and before word-time 16.

An important point to note in the discussion of immediate commands is that an immediate command must execute for at least one word-time before the flag can be effective. If, in the previous example, the immediate command located at word 09 had a T of 10, the flag could not

stop the operation until a complete drum cycle had elapsed, and word 10 was coming up for the second time. This, then, would be the way in which one command could cause the transfer of one whole long line to another: let T of the command be 1 greater than the location of the command itself, and let the command be an immediate command calling for the straight transfer of words.

Any of the previously discussed commands, either single- or double-precision, can be made immediate by setting bit 29 of the command equal to 0.

If an immediate command is not desired, or, to put it another way, it is desired that operation be deferred until some particular word-time, and then be performed for that word-time only, bit 29 of the command must be set to 1, indicating deferred (D) operation. Of course a deferred double-precision command will still obey the rules for double-precision operations: namely, the operation will continue until the next sign-time, which will be two word-times later. In other words, making a double-precision command deferred does not alter the fact that two contiguous words will be operated upon; it merely pinpoints the two words. An immediate double-precision command will operate on contiguous two-word numbers until stopped by a flag.

The immediate commands which can be made from each of the operation codes discussed so far are often referred to as "block" operations, since they operate on blocks of numbers.

SEQUENCING OF COMMANDS

As shown in the layout of a G-15 command on page 61, each command contains in itself the address of the next command (N) to be obeyed, and the word-time portion of this address is in bits 20 - 14. Notice that seven bits are allotted and are sufficient to express any word-time in memory. Commands, like data, may occupy any word. An address consists of more than a specified word-time, however; a line number must also be included. In the case of N, in a command, the line-number is implied to be the same as the number of the line in which the current command is located. In other words, the G-15 will continue looking for commands in the same line, once it has started with a command in that line. Since this is the case, it is only necessary to specify the word-time at which the next command is located in the same line.

COMMAND LINES

Not all lines in memory are connected to the special circuits which interpret commands. Any line which is so connected is called a "command line", and commands located in it can be read and executed. The command lines are 00, 01, 02, 03, 04, 05, 19, and 23. A command can also be executed out of AR, but this special action by the computer must, in turn, be called for by a special command, which will be discussed later. In order to preserve numerical continuity in all references to command lines, line 19 is referred to as command line 06, and line 23 is referred to as command line 07. AR, because of its special nature

in this regard, is not referred to as a command line. Once a command line has been chosen, the computer will continue to obey commands in that line, but how does a command line get chosen originally? What happens when a program must occupy more than one line? These are logical questions, and we will look into their answers just as soon as we complete the discussion of commands, themselves.

The only bit in a command word which remains unmentioned at this point is bit 21. You may now consider it mentioned, although this would be the wrong time in the discussion to describe its function. For our purposes at present, always assume it contains 0.

So far, although many computer operations have been discussed, they do not include all of the operations we will need for the solution for the quadratic equation. Multiplication and division are just two of the operations not supplied through the normal operation codes. It has been pointed out that, although there is no line 31 in the memory of the G-15, this number may be placed in a command as either the source (line) or the destination (line). If 31 is specified as either S or D in a command, the computer will know that no ordinary transfer is being called for.

SPECIAL COMMANDS

Upon discovery of D = 31 in a command, the computer will treat this command as a "special" command, and interpret it in a special way. The S number will be treated as a special operation code, and the three bits which normally specify the operation will usually be interpreted in the light of the special operation called for.

In the example of the quadratic equation, all additions and subtractions can be performed by using normal operations, but the other operations necessary, of which multiplication and division are two, will require special commands.

MULTIPLICATION AND THE TWO-WORD REGISTERS

The multiply command contains: D = 31, S = 24, and "C" code = 0. Before this command is executed, however, the proper numbers to be multiplied together must be in the two-word registers ID and MQ, as mentioned before. Therefore, the multiply command must be preceded by two other commands in the program, which load these two registers. The product, after multiplication, will appear in PN. The programming method for performing a multiplication can be derived from a further study of the two-word registers and how they operate.

Any two-word register can be loaded with either a single-precision number (via a single-precision transfer) or a double-precision number (via a double-precision transfer), but the two-word registers will always word in double-precision when a multiplication is called for. Two 57-bit magnitudes will be multiplied together. If a single-precision multiplication is really desired, it can be achieved by only loading the most significant bits of ID and MQ, making sure that the remaining, least-significant bits are cleared to 0. A 56-bit product (to be expected

when two 28-bit numbers are multiplied together) will appear in PN in double-precision form. If a single-precision product is desired, it will be in the most significant word of PN. So, in the case of a single-precision multiplication, the two-word registers must be cleared to 0 before they are loaded with the multiplier and multiplicand. Of course the product will be the same, regardless of which of the two numbers is treated as the multiplier and which as the multiplicand.

The G-15 is internally wired in such a way that each bit (of the 58 bits) in PN may be cleared as the corresponding bit in ID is set.* Therefore, if all 58 bits of ID are set, regardless of how they're set, prior to a multiplication, all 58 bits of PN will automatically be cleared, and PN will be ready to receive the product. The setting of MQ will affect no other register, nor will it be affected by the setting of any other register.

In a multiplication, although the magnitudes of the two numbers are to be multiplied, we know that the signs must be added, if the laws of signs are to be obeyed. A product is usually worthless if it contains the wrong sign. The G-15 knows this, too. Therefore, when the two-word registers are being loaded, via a normal operation (transfer),

	+	-
+	+	-
-	-	+

	0	1
0	0	1
1	1	0

if the "C" code is even, (0, 2, 4, 6), the sign of the number is divorced from the magnitude and sent to a special "flip-flop" associated with the two-word registers, called IP. A flip-flop is a two-state device, one state equalling 0, the other equalling 1, and it can remember which state it is in. It can also be read, or "sensed", to determine which state it is currently in. The bit in the two-word register which would normally receive the sign will not; it will be set to 0. When ID is loaded, IP will be set with the sign of the number going into ID. When MQ or PN is loaded, the sign of the number being transferred will be added to the present value of IP, and the result will remain in IP. Similarly, when a number is transferred, via a normal operation, out of a two-word register, and the "C" code is even, the magnitude will come from the register specified as S, but the sign will come from IP. This function of IP is automatic. The only special precaution the programmer must take in order to insure its operation is to transfer numbers to and from the two-word registers with even "C" codes. So, in the setting of ID and MQ prior to a multiplication, the program will have to set ID first, then set MQ, thus insuring the correct sign of the product in IP. Then the multiply command may be given.

* Note: this feature is automatic if ID is set with any even C code (0, 2, 4, 6).

When the computer is commanded to multiply, the following will be the state of affairs in the two-word registers:

ID - Multiplicand

MQ - Multiplier

PN - cleared to 0 and ready for product

IP - correct sign (0 or 1) of product

It has been stated that the two-word registers will multiply in double-precision fashion, regardless of whether or not double-precision operation is really desired. Remember that, in double-precision numbers the most significant bits are in the odd-numbered word (in the case of the two-word registers, we refer to these as ID_1, MQ_1, and PN_1). All 29 bits are magnitude bits. 28 of the bits in the even-numbered word (ID_0, MQ_0, and PN_0) are the least significant bits of the magnitude, and the sign-bit of this word is the sign of the number, or 0, if the sign went to IP.

In the case of double-precision multiplication, then, we would want the initial conditions to be as follows, where x's represent significant bits of magnitude.

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0
MQ:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0
PN:	0000000000000000000000000000	0000000000000000000000000000

IP: 0 or 1, whichever is the correct sign of the product.

To transfer the double-precision multiplicand from its resting place in memory to $ID_{0,1}$, we would use a straight double-precision transfer (C = 4), with D in the command equal to 25 (ID). Because the C code is even, the sign will be disengaged from the magnitude, and sent to IP. Because ID is the destination, IP will be loaded with this sign. Then, to load $MQ_{0,1}$, we would transfer the double-precision multiplier, also with a C = 4, with D in the command equal to 24 (MQ). Because the C code is even, the sign will be disengaged and sent to IP. Because MQ is the destination, IP will add this sign to its present contents, and the result, which will appear in IP, will be the correct sign of the product. When the signs are disengaged, the bits in the two-word registers which would normally have received them are cleared to 0, as shown above. When ID is loaded (each of the 58 bits is set with some value, replacing what was originally there), each corresponding bit (and therefore, all 58 bits) of PN is cleared to 0. Thus the desired initial conditions will be achieved through the execution of two commands, the first of which loads ID, the second, MQ.

In the case of single-precision multiplication, we would want the initial conditions to be as follows, where x's represent significant bits of magnitude.

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
MQ:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
PN:	00000000000000000000000000000	00000000000000000000000000000

IP: 0 or 1, whichever is the correct sign of the product.

To transfer the single-precision multiplicand from its resting place in memory to ID_1 (the most significant half of the two-word register), we would use a straight single-precision transfer (C = 0), with D in the command equal to 25 (ID). Because the C code is even, the sign will be disengaged from the magnitude, and sent to IP. Because ID is the destination, IP will be loaded with this sign. Then, to load MQ_1, (again, the most significant half of the two-word register), we would transfer the single-precision multiplier, also with a C = 0, with D in the command equal to 24 (MQ). Because the C code is even, the sign will be disengaged and sent to IP. Because MQ is the destination, IP will add this sign to its present contents, and the result, which will appear in IP, will be the correct sign of the product. When the signs are disengaged, the bits in the two-word registers which would normally have received them are cleared to 0, as shown above. These are bit 1 in both ID_1 and MQ_1 (remember, this is a single-precision transfer from an odd-numbered location in memory, thus congruent to 1 mod 2, to word 01 in ID and MQ). When ID is loaded (the first 29 bits are set with some value replacing what was originally there), each corresponding bit (and therefore, the 29 bits of PN_1) of PN is cleared to 0. The desired initial conditions will not be totally achieved unless something further is done to clear the remaining bits of ID, MQ, and PN. It will do no good to follow the two commands which load ID_1 and MQ_1 with two more which load ID_0 and MQ_0, because this will cause the resetting of IP, and destroy the sign of the product. But the two-word registers could be cleared first, and then ID_1 and MQ_1 set with the desired numbers. This is the procedure to be followed, and there is a special command which clears all bits in the two-word registers and also clears IP to 0. Since it is a special command, it has D = 31. Since it is going to operate on two-word registers, it must operate for two word-times. Its source (S) = 23. The C code = 0. Since the C code = 0, it will operate for only one word-time, unless it is made immediate by setting the I/D bit in the command = 0. The T number in this command, as in any other immediate command, then becomes a flag, telling the computer when to stop execution of the command. Remember that an immediate command will be executed starting in the word-time after that in which the command was read, and it will continue to be executed during each succeding word-time up through the one immediately preceding the flag. It will not be executed during the word-time whose number equals the flag. If we refer to the location of a command as L, then, in the case of the

command which clears the two-word registers, T = L + 3. In order to simplify the writing of flags for T numbers, we drop the plus sign, and use the desired number to be added to L as a subscript for L. In the case of the command we are presently considering, then, T = L_3.

Three commands, then, are necessary to establish the desired initial conditions for what we might call a single-precision multiply, although that really is a misnomer. The first will clear the two-word registers and IP, the second will load ID_1, and the third will load MQ_1.

The special circuitry associated with the two-word registers does essentially two things. We have already seen that it enables PN to act as an accumulator. The other feature accomplished through this special circuitry is a "shifting" process. A shift is the movement of bits toward the high-order or the low-order position within a register. In the G-15 it is accomplished one bit-position at a time. ID shifts toward the low-order (T1) position (this is usually referred to as shifting to the right). MQ shifts toward the high-order (T29) position (this is usually referred to as shifting to the left).

Multiplication involves both the shifting and the additive features of the two-word registers, in the following way. The contents of ID are shifted right by one bit-position, moving all 57 magnitude bits to the right one place. The right-most bit (T2 of ID_0) is lost. The left-most bit-position (T29 of ID_1) is filled-in with a 0. Simultaneously MQ is shifted left by one bit-position, moving all bits to the left one place. The left-most bit enters an inspection station, where it is inspected for 1 (it will, of course, be either 1 or 0). The right-most bit-position is filled-in with a 0. After such a simultaneous shift, during a single-precision multiplication, ID and MQ would contain:

	Word 1	Word 0
ID:	0xxxxxxxxxxxxxxxxxxxxxxxxxxxx	000000000000000000000000000000
MQ:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx00	000000000000000000000000000000

Compare these with the initial conditions shown on page 73.

If the bit from MQ which is inspected is a 1, the new contents of ID are added to PN; if it is a 0, the addition is not performed. The first addition in PN will, of course, be to 0, since PN was initially cleared. This process requires two word-times; because it is essentially a double-precision process, it must begin with an even word-time. It can be repeated as often as desired (28 times for a full single-precision multiplication). The multiply command must be immediate, and it will perform the process over and over again, for the indicated number of word-times. T in the command is a "relative timing number". It will be set equal to the desired number of word-times of execution of the command; this should be an even number, and the execution should begin at an even word-time, requiring the immediate multiply command to be located at an odd word-time. If the

process is allowed to continue for 28 times (T = 56), two full single-precision words can be multiplied together, and their product, a series of sums, will appear in PN. Notice that at least one shift is performed prior to the first addition, and the product will actually occupy the 56 most significant bit-positions in PN. In any number system, if two 28-digit numbers are multiplied together, a 56-digit product, counting any leading 0's, will result. If the initial shift in the computer were not performed, it would be possible, in the case of large numbers, to generate an overflow and an erroneous result.

After a single-precision multiplication, the most significant bits of the answer will appear in PN_1, bits 29 - 2. Bit 1 of PN_1 and bits 29 - 3 of PN_0 will contain the least significant bits of the product. Assuming that a single-precision product is all that is required, the least significant bits are merely excess accuracy, and can usually be ignored.

In ordinary pencil-and-paper multiplication, if you were to multiply two 28-bit numbers, you would inspect the multiplier from right to

```
                                    1111111111111111111111111111
                                    1100101010000000001111000001
                                    1111111111111111111111111111
                              1111111111111111111111111111
                             1111111111111111111111111111
                            1111111111111111111111111111
                           1111111111111111111111111111
                    1111111111111111111111111111
                   1111111111111111111111111111
                  1111111111111111111111111111
                1111111111111111111111111111
               1111111111111111111111111111
             1111111111111111111111111111
            1111111111111111111111111111
          1111111111111111111111111111
         1111111111111111111111111111
       1111111111111111111111111111
      1111111111111111111111111111
    1111111111111111111111111111
   1111111111111111111111111111
 1111111111111111111111111111
11001010100000000011110000000011010101111111110000111111
```

left, one bit at a time. If you found a 1, you would add the multiplicand to what you already had in the way of a partial sum. If you found a 0, you would not add the multiplicand. You would then shift the multiplicand to the left one place, and inspect the next bit in the multiplier. You would do this 28 times, once for each bit in the multiplier, and you would generate, as a result, a sum, which represents a product of the two original numbers. The computer does the same thing, in reverse. It starts with the high-order end of the multiplier and inspects toward the low-order end. The shifts are exactly the reverse, therefore; you shifted to the left, but the computer shifts to the right. A double-precision multiply command causes exactly the same sequence of events, but the relative timing number (T) in the command is set to allow the process to continue for 57 times (T = 114). Notice that the multiply command may be allowed to operate for any number of times, merely by setting T = 2k, where k = the number of times desired. The resultant product will always be predictable.

DIVISION AND THE TWO-WORD REGISTERS

Division is somewhat similar to multiplication, in that it also utilizes the shifting and additive features of the two-word registers in order to reach a result.

The divide command contains: D = 31, S = 25, and C = 1 or 5 (these are interchangeable: the setting of the S/D in the command has no bearing on the operation). As in the case of multiplication, the numbers to be divided must be set up in the two-word registers. The rules governing the initial set-up of the two-word registers apply here as well as in the case of multiply, except that the denominator will be loaded into ID, the numerator into PN, and the quotient will appear in MQ. Because PN is cleared as ID is set, the denominator must be loaded first, then the numerator. In order to clear MQ, preparatory to receiving the quotient, the clear command will have to be given. The proper sign of the quotient will be generated in the same manner as it is for a product.

When the computer is commanded to divide, the following will be the state of affairs in the two-word registers:

ID - Denominator

PN - Numerator

MQ - cleared to 0 and ready for quotient

IP - correct sign (0 or 1) of quotient

In division, as in multiplication, the two-word registers will operate in double-precision fashion. The most significant word is the odd-numbered word (ID_1, PN_1, and MQ_1). If single-precision division is required, the single-precision denominator usually will be in ID_1,* bits 29 - 2 (remember its sign will be in IP), followed by insignificant 0's in bit 1 of ID_1 and all 29 bits of ID_0. Wherever the denominator is in ID*, the single-precision numerator should be similarly positioned in PN.

In the case of single-precision division, usually we want the initial conditions to be as follows *, where x's represent significant bits of magnitude.

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
PN:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
MQ:	00000000000000000000000000000	00000000000000000000000000000

IP: 0 or 1, whichever is the correct sign of the quotient.

* The reason for making this indefinite statement will follow a description of the machine's divide process.

In order to get MQ cleared, the first command in the set-up for a division would be the clear command. This would be followed by two straight single-precision transfers from memory to ID_1 and PN_1, in that order. The signs of these two numbers will be disengaged and sent to IP, where the proper resultant sign will be generated. 0's will occur in the bits in ID (bit 1 of ID_1) and PN (bit 1 of PN_1) which would normally have received the signs.

When the desired initial conditions have been established, the divide command may be given. In order to understand the computer's division process, we must first inspect the pencil-and-paper method, to determine what process is involved there; we're so used to doing it, that we usually don't consciously analyze the process as we do it, but there is an underlying, reasonable pattern to the process of "long division". Consider the following long division in binary arithmetic.

```
          0101000
101000/011001000000
       000000
       -------
       0110010
        101000
        -------
        0010100
         000000
         ------
          101000
          101000
          ------
          0000000
           000000
           ------
           0000000
            000000
            ------
            0000000
             000000
             ------
             000000
```

In division we are attempting to find the ratio of one number to another. We call one of these numbers the denominator, and the other, the numerator. The resultant ratio, which we call a quotient, is the ratio of the numerator to the denominator: N/D.

The first step is to subtract the denominator, in its present form ($D \cdot 2^0$), from the numerator. We find that this yields a negative result. (In the decimal system we would inspect each possible multiple of D, starting with $9 \cdot D$, but, in the binary system, the only possible multiples of D are $1 \cdot D$ and $0 \cdot D$). We therefore discard the coefficient of 1, and say that N contains $0 \cdot D$ plus a remainder, R_1. We now shift D to the right one place (in the binary system, this yields $D \cdot 2^{-1}$), and attempt to subtract it from this remainder. This remainder, of course, equals N, since $N - 0 = R_1$. In effect, what we are doing, knowing that N does not contain D, is attempting to discover whether or not N contains $D/2$.

$$N - 1 \cdot D \cdot 2^0 = -r_1$$
$$N - 0 \cdot D \cdot 2^0 = +R_1$$
$$R_1 - 1 \cdot D \cdot 2^{-1} = +R_2$$
$$R_2 - 1 \cdot D \cdot 2^{-2} = -r_3$$
$$R_2 - 0 \cdot D \cdot 2^{-2} = +R_3$$
$$R_3 - 1 \cdot D \cdot 2^{-3} = +R_4$$

It does, and we know that because we get a positive result after the subtraction. R_1 contains $1 \cdot D \cdot 2^{-1}$ plus a remainder, R2. We continue this shifting and subtracting process until we arrive at a remainder of 0 or until we achieve the desired accuracy in the resultant quotient.

$$N = 0 \cdot D \cdot 2^{0} + R_1$$

$$N = 0 \cdot D \cdot 2^{0} + 1 \cdot D \cdot 2^{-1} + R_2$$

$$N = 0 \cdot D \cdot 2^{0} + 1 \cdot D \cdot 2^{-1} + 0 \cdot D \cdot 2^{-2} + R_3$$

$$N = 0 \cdot D \cdot 2^{0} + 1 \cdot D \cdot 2^{-1} + 0 \cdot D \cdot 2^{-2} + 1 \cdot D \cdot 2^{-3}$$

The reason we have taken a close look at the way you divide is that, contrary to popular belief, the designers of digital computers are "just plain folks"; they think the way you do, and when they were faced with the problem of designing a division operation for the Bendix G-15, they followed the same reasoning we have followed here. They noted one important exception to it, however, from the standpoint of the computer: the computer cannot "inspect" prior to a subtraction; it must subtract, and then inspect the result. Since the numerator is in PN, and since the subtraction will also be performed in PN, it is obvious that, after the subtraction, the original numerator will be lost in any event, and either a positive or a negative remainder will be in PN. The computer will have to be able to determine its future course on the basis of the sign of the remainder in PN. The bit that goes in the quotient is easily determined: if the sign of the remainder is negative, a 0 goes in the quotient; if the sign of the remainder is positive, a 1 goes in the quotient. If the remainder is positive, there is no problem: the denominator must be shifted right one more place and a new trial subtraction performed. But, in the case of a negative remainder, the problem is a bit more difficult. We know that the division yielded a 0 at this point, and the remainder actually indicates the quantity by which the denominator exceeded the numerator (or previous remainder, if this is not the first subtraction). If we shift the denominator right one more time, obtaining 1/2 its previous value, and add this to the negative remainder, we will know whether or not an original subtraction of D/2, rather than D, from N would have yielded a positive result. ($N - D + D/2 = N - D/2$). In short, we can devise the following rule: subtract D from N; if the result (R_1) is positive, place a 1 in the quotient, and subtract D/2 from R_1, continuing the process. If the result is negative, place a 0 in the quotient, and add D/2 to R_1, continuing the process.

The designers worried about one other point: the necessity for carrying many insignificant trailing 0's along with N, in order to perform the long division process. They realized that, after subtracting D from N, and arriving at a remainder R_1, the ratio of R_1 to D/2 is the same as the ratio of $2 \cdot R_1$ to D ($R_1 : D/2 :: 2 \cdot R_1 : D$). Therefore, rather than shift D right to obtain D/2, they decided to shift R_1 left to obtain $2 \cdot R_1$, and do this successively, with each remainder, always adding (in the case of a negative R) or subtracting D (in the case of a positive R) from the new value. Although it seems that overflow

might be caused by shifting a remainder to the left (if the remainder has a 1 in the most significant bit position prior to the shift), this will not cause overflow, because of the manner in which the numbers are treated by the circuitry employed during a divide operation. Any temporary overflow condition will right itself in the next step of the continuing process. Such a temporary overflow will not set the overflow indicator. The algorithm upon which this division process is based is that N will never equal or be greater than 2·D. The long division in binary, as shown on page 77, will look like the following, as it is performed by the computer.

```
                 0101000
101000/011001 N
   sub 101000 D
 0) - 001111 R1
    - 011110 2·R1
   add 101000 D
 1)   001010 R2
      010100 2·R2
   sub 101000 D
 0) - 010100 R3
    - 101000 2·R3
   add 101000 D
 1)   000000 R4
      000000 2·R4
   sub 101000 D
 0) - 101000 R5
   -1010000 2·R5
   add 101000 D
 0) - 101000 R6
   -1010000 2·R6
```

Notice that the overflow caused by $2 \cdot R_5$ is corrected by the next addition. This is a temporary overflow.

At the beginning of the division process, MQ is shifted left one bit-position, while D is subtracted from N. MQ, then, if it were not cleared prior to the division, would look like this, where Y's represent the original contents of MQ.

	Word 1	Word 0
MQ:	YYYYYYYYYYYYYYYYYYYYYYYYYYYY	YYYYYYYYYYYYYYYYYYYYYYYYYYYY0

In the case of division, although MQ shifts left, ID does not shift right, so $D \cdot 2^0$ remains in ID. R_1 is inspected: if it is positive, a 1 is placed in T2 of MQ_0; if it is negative, a 0 is placed in the same bit. PN, containing R_1, is shifted left one bit-position, so that it now contains $2 \cdot R_1$. The sign of R_1 is used to control the inverting gates during the next transfer of D to PN for addition or subtraction. (Notice that D will pass through the inverting gates because the C code of the divide command contains a 1.) If the sign of R_1 is positive, it will be reversed and combined with D from ID on the next pass, so that, as D passes through the inverting gates on its way to PN, the effect will

be to subtract D from $2 \cdot R_1$. If the sign of R_1 is negative, it will be reversed, combined with D from ID, and cause the addition of D to $2 \cdot R_1$. A complete step such as the one described above will require two word-times, since division is essentially a double-precision operation, even though single-precision numbers may actually be involved. The next step in the process will begin with the shifting of MQ left by one bit-position again, so that the first bit in the quotient will occupy T3 of MQ_0, and T2 will be ready to receive the next bit. During the second step, $2 \cdot R_1$ will be in PN, and D will be added to, or subtracted from it. This process will continue for as many word-times of execution as are allowed by the divide command. The command will be immediate, and the relative timing number in T will be set to allow 57 word-times of execution (T = 57) for a single-precision divide.

After 56 word-times of execution, at 2 per step in the division process, 28 bits of quotient will be generated in MQ_0, and MQ will look like this, where Y's represent original bits, and x's represent quotient bits.

	Word 1	Word 0
MQ:	YYYYYYYYYYYYYYYYYYYYYYYYYYYY	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0

If only 56 word-times of execution are allotted, the first x in the drawing above (in T29 of MQ_0) will represent $x \cdot D \cdot 2^0$, while the remaining bits in MQ_0 will represent a fractional quotient. If a 57th word-time of execution is called for, during that word-time MQ_0 will be shifted left one bit-position, and a new bit will be placed in T2 of MQ_0, so that MQ will look like this.

	Word 1	Word 0
MQ:	YYYYYYYYYYYYYYYYYYYYYYYYYYYY	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0

Notice that MQ_1 did not move, while MQ_0 did. The first bit in MQ_0 ($x \cdot D \cdot 2^0$) is shifted into a flip-flop which detects overflow. The bit now in T29 of MQ_0 represents $x \cdot D \cdot 2^{-1}$, and the whole word is a fractional quotient. This is the normal form for a ratio, and it is the form most desirable when programming the G-15. Overflow will be indicated if the quotient actually equals or exceeds 1, since, in that case, a 1 will reach the overflow flip-flop during this last shift. If T = 56 in the divide command, the overflow indicator may be erroneously set, and should never be depended upon.

The rule, then, for a single-precision division is:

1. Never divide an N which is greater than, or equal to D.

2. Use a T = 57 in the divide command.

3. As in the case of multiply, the divide command must be located at an odd word-time.

You can see that there are exceptions to this rule, but that a thorough knowledge of computer logic and much experience are required. In no case will division work if N is greater than or equal to 2·D.

Double-precision division involves exactly the same operations as does its single-precision counterpart. Of course, the execution time of the command must be greater. With 57 bits of quotient to be generated, 114 word-times would be necessary. If this is the time allotted, the first bit of the quotient (T29 of MQ_1) will represent $x \cdot D \cdot 2^0$. In the case of double-precision, two more word-times are necessary to shift the entire quotient one bit-position to the left. If T = 116, all 57 bits of the quotient will be fractional, T29 of MQ_1 representing $x \cdot D \cdot 2^{-1}$, and T2 of MQ_0 representing $x \cdot D \cdot 2^{-57}$. In this case, the overflow flip-flop will be set with $x \cdot D \cdot 2^0$. If the quotient equals or exceeds 1, x will equal 1, and overflow will be indicated. If T = 114, erroneous overflow may be indicated.

The rule, then, for double-precision division is:

1. Never divide an N which is greater than, or equal to D.

2. Use a T = 116 (v6) in the divide command.

Again, exceptions are possible, but thorough knowledge of computer logic and much experience are required. In no case will division work if N is greater than or equal to 2·D.

Remember that a quotient is nothing more than a ratio of one number to another. It stands to reason that, if 2/17 = 4/34, etc.,

000000000000000000000000000000 000000000000000000000000000100

divided by

000000000000000000000000000000 000000000000000000000000100010,

equals

000000000000000000000000000000 000000000000000000000000001000

divided by

000000000000000000000000000000 000000000000000000000001000100,

which, in turn, equals

000000000000000000000000000000 010000000000000000000000000000

divided by

000000000000000000000000000010 001000000000000000000000000000,

and any of these quotients equals the quotient derived by dividing

000100000000000000000000000000 000000000000000000000000000000

by

100010000000000000000000000000 000000000000000000000000000000.

So long as the denominator and the numerator occupy corresponding bit-positions in ID and PN, respectively, which bit-positions they occupy will have no effect on the quotient. For each denominator and numerator, regardless of their positions in these two registers, provided the above requirement is met, there will be one correct quotient appear in MQ_0 for single-precision division, and one correct double-precision quotient, if that is called for, in all of MQ.

It would be desirable to round off a quotient to the nearest figure in the last digit carried, but this is not possible, because there is no way of examining the remainder in the G-15. Truncating the quotient, i.e., ignoring the value of digits beyond the one carried, will result in the quotient reported being somewhere between correct and one unit low, and averaging a half unit low. This would give a systematic error which could seriously affect accuracy after a large number of operations. One way of avoiding this is to add an average of one half to all quotients. This is simply done by making the last bit always 1, i.e., adding one unit to half of the uotients: those originally ending in zero. This is called the "Princeton Round-off". A truncated quotient is between zero and one unit low, while a Princeton-rounded quotient is between one unit low and one unit high.

There is no way to program the elimination of the Princeton round-off; it will always occur. If you desire a single-precision quotient without this round-off, you can carry the division out as a double-precision operation, making T = 116 (v6) in the divide command, in which case the single-precision quotient will be in bits 29 - 2 of MQ_1, and the rest of the bits in MQ will contain the remainder. The Princeton round-off will occur at T2 of MQ_0.

Notice that, in either single-precision multiplication or single-precision division, four commands are necessary to set up for, and perform, the operation. One command clears the two-word registers, two others load each of the two-word registers containing the numbers to be operated on, and the fourth command actually calls for the desired operation. The clear command can sometimes be eliminated by utilizing special circuitry connecting AR and the two-word registers.

In the table of normal operations (pages 64 - 66), it was pointed out that, if both S and D are less than 28, codes 2 and 6 call for exchanges of AR with memory. So do codes 3 and 7, under similar circumstances, except that, in these two cases, complementation of negative numbers is involved. If any one of these four commands is used to load ID (25), MQ (24), or PN (26) from some other location in memory (S less than 28), during all even word-times AR cannot load any of these three registers.

In other words, the contents of AR will be blocked off from ID_0, MQ_0, and PN_0. In the place of the contents of AR, the even half (word 00) of the specified two-word register will receive 29 0's. If the C code = 6, during the following odd word-time, the contents of AR will be transferred to the odd half of the specified two-word register. Because the C code is even, the sign of the double-precision number will go to IP, according to the rules discussed earlier. Notice that during the even word-time of execution, the original contents of AR attempt to reach the even half of the specified two-word register, but are blocked off, and 0's are transferred instead. During the same word-time, the even-numbered word from memory goes to AR. During the following odd-numbered word-time, the contents of AR (originally an even-numbered word from memory) goes to the odd half of the specified two-word register. The fact that this word was delayed one word-time because of its transfer via AR does not alter the fact that it is the first word of a double-precision number. Therefore, even though it reaches the two-word register at an odd word-time, its sign will be divorced, and sent to IP, in accordance with the rules already mentioned.

Consider, then, a single-precision multiplication: A·B, where A is stored in an even-numbered word. If a transfer of A to ID via the AR register is called for (C = 6), during the first word-time (an even numbered word-time), the original contents of AR attempts to reach ID_0, but is blocked off, and all 29 bits of ID_0 are cleared to 0. During the same time, A is transferred to AR. During the next word-time (an odd word-time), A is transferred from AR to ID_1, but, since A is the first half of what the computer believes to be a legitimate double-precision number, its sign, being treated as the sign of the number, is disengaged from the magnitude bits, and it is transferred to IP. T1 of ID_1, which normally would have received this sign, is cleared to 0. Since every bit in ID has been set during this operation, every bit of PN has been cleared. The only initial condition remaining to be satisfied is the placing of the multiplier, B, in MQ_1. If B is in an odd word in memory, a straight single-precision transfer to MQ_1 will accomplish this. Since the C code for this is 0 (therefore, it is even), the sign will be disengaged from the magnitude portion of B, and it will be sent to IP, to be combined with IP's present contents. Notice that the initial conditions in this case will be:

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
MQ:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx0	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY
PN:	00000000000000000000000000000	00000000000000000000000000000

IP: 0 or 1, whichever is the correct sign of the product.

In the above layout of the two-word registers, the Y's in MQ_0 represent the original contents of that word, remaining after MQ_1 has been set. Since, during a multiplication, MQ is shifted to the left one bit-position

at a time, each succeeding bit being inspected to determine whether or not the contents of ID should be added to the contents of PN, and since, if a single-precision multiplication has been called for, only 28 bits from MQ will be inspected, the remaining "garbage" in MQ will have no effect on the multiplication. There is no need to clear MQ_0 prior to a single-precision multiplication. However, if the multiplier were in an even-numbered word in memory, it would be perfectly permissable to use a transfer via AR (C = 6) to get it into the odd half of MQ. In this case, of course, MQ_0 would be cleared.

In the case of a single-precision division (N/D), if D is stored in an even word in memory, and transferred into ID_1 via AR, this will succeed in properly preparing ID for the division and setting up IP for the addition of signs. But setting ID clears PN, and not MQ. PN could be set with the proper value (N), either by a straight single-precision transfer, or by a transfer via the accumulator, and, in either case, it would also be set up properly. If PN_1 were set by a straight single-precision transfer (C = 0), PN_0 would still have been cleared because each bit in ID_0 was set (to 0). MQ will remain unaffected, containing its original contents.

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
PN:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
MQ:	YYYYYYYYYYYYYYYYYYYYYYYYYYYY	YYYYYYYYYYYYYYYYYYYYYYYYYYYY

IP: 0 or 1, whichever is the proper sign of the quotient.

The Y's in the above layout of MQ represent its original contents, remaining after both ID and PN have been set. This is perfectly all right, however, since MQ is shifted to the left one bit-position at a time, as each bit of the quotient is placed in T2 of MQ_0. T1 of MQ_0 is cleared by the initial shift, preparatory to the placement of the first quotient bit in MQ_0. If a full 28-bit quotient is generated, all the Y's shown above in MQ_0 will be shifted out to the left, and all 28 magnitude bits of that word will contain quotient bits.

REVIEW

At this point, we pause to review what has been covered. We first pointed out that, in order to effectively use the computer, the programmer must analyse the problem:

1. determine the formula(s) by which a solution can be reached, or in some way define exactly what is called for;

2. discover the form, magnitude, and ranges in values, of the data which will be available as input for the program;

3. choose an appropriate method of solution;

4. outline, very briefly, the logical path to be followed in this method, such an outline being called a flow diagram.

We then analysed a sample problem, that of solving for the roots of a quadratic equation of the form, $a \cdot x^2+b \cdot x+c$. After we developed a flow diagram for its solution, we saw that we needed more thorough knowledge of available computer operations, especially the arithmetic operations.

This lead us to a discussion of commands as they appear, in binary, in the computer. The first part of a command we studied was the C code, consisting of a two-bit characteristic and a one-bit indicator for either single- or double-precision operation. We saw that the various possible C codes in three bits run the gamut of 0 - 7, where 4, 5, 6, and 7 are essentially the double-precision counterparts of the single-precision codes 0, 1, 2, and 3, respectively.

Each of these operations is a transfer of some type, and it may call for the use of special circuitry during the transfer, such as inverting gates, or circuitry which can change a sign, drop a sign, or add. The various types of transfers that were seen to be available were:

1. straight single- or double-precision transfer from one place in memory, including either the single- or double-precision accumulator, to another, also including the accumulators; C = 0, 4;

2. single- or double-precision transfer via the inverting gates (to accomplish complementation of negative numbers, preparatory for addition) from one place in memory, including either of the accumulators, to another, also including either of the accumulators; C = 1, 5;

3. exchange of memory with AR, both single-precision words, and the transfer in each direction, a straight single-precision transfer; C = 2;

4. transfer of a double-precision number via AR, in which, during the even word-time, the even word of the double-precision operand goes to AR while the original contents of AR go to the even word of the double-precision destination; during the odd word-time, the new contents of AR go to the odd word of the destination, while the odd word of the operand goes to AR; C = 6;

5. transfer either a single- or a double-precision magnitude to or from the appropriate accumulator, the sign being dropped during the transfer; C = 2, 6;

6. exchange memory with AR, both single-precision words, the transfer from AR to memory being a straight single-precision

transfer, but the transfer from memory to AR being via the inverting gates; C = 3;

7. transfer of a double-precision number via AR, as in (4), above, except that each word of the double-precision operand as it enters AR, enters via the inverting gates; C = 7;

8. transfer either a single- or a double-precision number to or from the appropriate accumulator, but with a change of sign, and subsequent passage through the inverting gates, for complementation, if necessary, preparatory to addition (changing the sign of a number and adding it accomplishes the same end result as subtracting it); C = 3, 7.

After discussing the normal operations (each one actually a different type of transfer of words available in the G-15), we examined the various addresses contained in a command.

One of these is the address of the operand, the word(s) to be transferred. All addresses in the memory of the computer are composed of a line number and a word number, or word-time, within that line. An address is denoted in the following manner, where LL stands for line number, and TT, for word-time: LL.TT. The line containing the number to be transferred is called the Source, and the address of the operand is SS.TT, usually written S.T. The address of the word(s) receiving the number to be transferred is also composed of a line number and a word-time, and is written D.T, where D stands for Destination. In a command, the word-time (T) involved in both these addresses is the same, and is given only once. Therefore, a transfer of a word(s) from one line to another will place the number being transferred in the Destination at the same word-time it occupies in the Source, or (in the case of transfers between lines of different lengths) in a word-time congruent to the word-time it occupies in the Source.

The functions of S and D were described. They control selectors which, in turn, modify the normal recirculation of memory at the proper word-time in the proper line.

We then discovered that a series of individual commands, each with the same S, D, and C, but with successively increasing T's, can be replaced by one immediate command, in which the T number is a flag, telling the computer when to stop the operation. In such a case, the operation commences in the very next word-time after the command has been read, so the location of an immediate command helps to determine how many, and which words will be transferred. We called these immediate commands "block" commands, since they work on blocks of congruent words in given lines.

If it is not desired that a command be immediate, it can be made deferred, in which case it will operate only on the word (or two words, in the case of double-precision) indicated by the T number. A bit indicating whether the command is immediate or deferred is included in the command, itself. It is the I/D bit, (T29).

It was pointed out that, when we say a G-15 command contains within itself the address of the next command to be obeyed, and thus the program sequence is determined by the programmer when he makes up the individual commands in his program, we are only partly correct. Each command contains the word-time at which the next command is to be read, but the line number in which that next command is located is not contained within the current command. The reason it is not, is that, once a sequence of commands is started in any "command" line, the line will remain the same, and thus, need not be specified from command to command. Only word-times need be specified. It was also pointed out that not all lines in the memory of the G-15 are "command" lines. Commands can only be read out of lines 00, 01, 02, 03, 04, 05, 19, and 23. These are called "command" lines 0, 1, 2, 3, 4, 5, 6, and 7, respectively.

Two points remained open, although they were discussed:

1. how a command line is initially chosen, and how a program can switch from one line to another, should that be necessary; and

2. the meaning of the BP bit in a G-15 command, this being the only bit not defined.

After the discussion of the various parts of a command, the concept of special commands was introduced. It was pointed out that not all of the operations necessary for the solution of the quadratic equation were, as yet, described. The two most apparent of these omitted operations were multiply and divide.

If D is set equal to 31 in a command, since there is no line referred to by that number, the G-15 treats this command as a special command. In this case, the S number in the command will become a special operation code, and the C code will usually be treated in the light of the special operation called for. Having thus defined special commands, we proceeded to discuss two of them, multiplication and division.

We saw that the command calling for a multiplication contains D = 31, S = 24, C = 0, and T = a relative timing number, which indicates for how many word-times the execution of the command is to be carried out, where two word-times are necessary for each bit in the product. The duration of operation of this command can be of any length, provided T is a multiple of 2, and, in any case, the results will be predictable. This command must be an immediate command, and, because its operation is always double-precision in nature, it must be located at an odd word-time, so that the first word-time of execution will be an even word-time.

We also saw that it is necessary to place the multiplicand and the multiplier in the two-word registers, ID and MQ, respectively, prior to giving the multiply command. Certain clearing of the two-word registers is also necessary. The rules for setting up these registers, and how they operate during the multiplication were discussed, but suffice it to say here that the product will appear in PN: if a full single-

precision multiplication is called for (T = 56), the product will be in PN_1; if a full double-precision multiplication is called for (T = 114), the product will be in $PN_{0,1}$. In any case, the correct sign of the product will be generated in IP. In this regard, we saw that, if a number, either single- or double-precision, is transferred to any two-word register with an even C code (0 is treated as even), the sign will be divorced from the magnitude, the sign going to IP, and the magnitude going to the magnitude bits of the appropriate word(s) in the appropriate two-word register. Similarly, when a number is transferred out of any two-word register, if the C code is even, the magnitude bits of the number will be picked up from the register itself, while the accompanying sign will be picked up from IP. Although this makes ordinary use of the two-word registers for storage slightly confusing, it is necessary for proper operation during multiplication and division.

After multiplication, we discussed division, and saw that it, too, utilizes the two-word registers and IP. The divide command contains D = 31, S = 25, C = 1 or 5 (the operation will be exactly the same, regardless of which is used), and T = a relative timing number. For a single-precision divide, T must equal 57; for a double-precision divide, T must equal 116. Exceptions to this rule are possible but require thorough knowledge of the internal logic involved and extreme care in treatment of the quotient. This command must be immediate, and, because its operation is always double-precision in nature, it must be located at an odd word-time, so that the first word-time of execution will be an even word-time.

The denominator and the numerator are placed in ID and PN, respectively, prior to giving the divide command. Certain clearing of the two-word registers is also necessary. The quotient will appear in MQ; a single-precision quotient will appear in MQ_0; a double-precision quotient will appear in $MQ_{0,1}$. The correct sign of the quotient will be generated in IP. The least significant bit in a quotient will always be 1; this is called the Princeton round-off.

A quotient represents the ratio of one number to another. In the G-15, this ratio should be in the form of a proper fraction, less than 1. If a quotient less than 1 is to be obtained, care must be taken to insure that, prior to the division, the numerator, as it appears in the machine, is less than the denominator, as it appears in the machine. Since a ratio is desired in a division, the location of the numbers to be divided, in ID and PN, is immaterial, provided they occupy corresponding bit-positions in those two registers.

Because of a unique circuit connecting AR and the two-word registers, use of a C code equal to 6 in the transfer of a single-precision number from an even location in memory (S less than 28), via AR, and into the odd half of ID (ID_1), will accomplish all the clearing necessary for a single-precision multiplication or division, eliminating the necessity for a clear command. This same circuit will cause the same clearing to occur whenever S is less than 28, D equals 24, 25, or 26, and the C code equals 2, 3, 6, or 7.

MACHINE FORM OF A NUMBER AND SCALING

Several times reference has been made to the machine form of a number, and, in the Introduction to the G-15, it was implied that the following numbers in machine form are equal to the decimal numbers shown below:

Binary number in machine	Decimal equivalent
000000000000000000000000000001|0	1.
000000000000000000000000000010|0	2.
000000000000000000000000000011|0	3.
000000000000000000000000000011|1	-3.

This implies that the binary point in a machine number is usually between the least significant magnitude bit and the sign bit.

$$000000000000000000000000000001.+_{(2)} = +1._{(10)}$$

This would mean that usually a machine number is entirely integral, and has no fractional bits. But if this were the case, the result of a multiplication or of a division is most disconcerting, for a multiplication will result in a product smaller than either the multiplicand or the multiplier, and a division will result in a quotient larger than the numerator, and, in some cases, also larger than the denominator. These statements are verified by the fact that multiplication of even the very largest possible numbers cannot cause overflow, and division of a number by another can cause overflow.

In short, it would seem, from inspection of these results, that numbers in the computer are actually fractional rather than integral. If two fractions are multiplied together, the resultant product will be smaller than either of the original numbers, and if a fraction is divided by another fraction, the result will in all cases exceed the value of the numerator, and may exceed the value of the denominator.

Actually, the computer treats every binary number in its memory as a 28-bit fraction with a sign. The binary point in the machine, sometimes referred to as the machine-point, precedes the most significant bit of a number. If this is the case, then:

$$.000000000000000000000000000001|0_{(2)} = 1 \cdot 2^{0} \cdot 2^{-28}{}_{(10)} = 1 \cdot 2^{-28}{}_{(10)}$$

$$.100000000000000000000000000000|0_{(2)} = 1 \cdot 2^{27} \cdot 2^{-28}{}_{(10)} = 1 \cdot 2^{-1}{}_{(10)}$$

We can interpret any 28-bit binary value in the machine in any way we want; that is to say, we can understand the true binary point for our purposes to follow T2 if the machine holds $1/2^{28}$th of what we intended. The maximum value we can express in 28 bits is:

$$1111111111111111111111111111._{(2)},$$

which equals

$$1\cdot2^{0}+1\cdot2^{1}+1\cdot2^{2}+\ldots+1\cdot2^{26}+1\cdot2^{27},$$

and this is 1 less than $1\cdot2^{28}$. Therefore, the machine will always be able to make a fraction out of whatever we give it, not by changing any bits, but merely by assuming the binary point to always precede T29 of the word.

Of course addition and subtraction will yield the same, familiar results, as we interpret them, since these operations will be performed on fractions all of which have the same denominator.

If $a + b = c$, then $\frac{a}{2^{28}} + \frac{b}{2^{28}} = \frac{c}{2^{28}}$.

Regardless of the fact that we might be considering a and b as integers, and the computer is considering them as fractions, the fraction which results in the computer must represent the integer c, as we interpret it. We will be able to convert the bits of the result as a binary integer and arrive at the decimal integer equivalent.

```
a  000000000000000000000000000010
b  000000000000000000000000000100
---------------------------------
c  000000000000000000000000000110
```

If a = 1, b = 2, c will equal 3. We can so place a in a word in memory that it will be of the form shown above. We interpret this as $a\cdot2^{-28}$. We can similarly place b in another word, knowing that it will be similarly interpreted. We can then transfer a to AR (D = 28), and b to AR (D = 29), and they will be added in AR. The sum in AR will appear as shown above. We know it is really $c\cdot2^{0}\cdot2^{-28}$, and we interpret it as $c\cdot2^{0}$, or 3.

Suppose, in the example above, we wanted to add numbers containing fractional as well as integral bits. Let a = 1.5 and b = 2.75. The sum of these, c, must equal 4.25. We must first determine how many bits will be required to express the longest fraction in the binary equivalents of these numbers. $.5_{(10)} = .1_{(2)}$, which is 1/2. $.75_{(10)} = .11_{(2)}$, which is 1/2 + 1/4 = 3/4. $.25_{(10)} = .01_{(2)}$, which is 1/4. The most bits required by any fraction involved are two. Therefore, if two fractional bits are reserved, each number involved can be fully expressed. Note that not all decimal fractions will convert evenly to binary fractions. Sometimes a programmer must decide what accuracy is needed, and let this determine the number of fractional bits needed.

When we speak of fractional bits, as in the paragraph above, we are referring to bits which we will interpret as fractional, as opposed to those which we will interpret as integral. When we say that the computer holds $a\cdot2^{-28}$, we are saying that we know that a, as it appears in the machine, is in the form, $a\cdot2^{0}\cdot2^{-28}$, meaning that the true binary point, for us, is 28 bits to the right of the machine binary point.

This means there are no bits left to express any fractional value for a. If we were to place a in a word in memory in such a way that two fractional bits followed the true binary point, and do the same for b and c, then the conditions of the problem in the above paragraph would be met. This means that a, as it appears in the machine, must be in the form, $a \cdot 2^2 \cdot 2^{-28}$. For a = 1, the value in the machine would appear as

$$000000000000000000000000010\overset{\uparrow}{0}0,$$

where the true binary point would be known to be at the point shown by the arrow. In such a case, we say that the machine holds $a \cdot 2^{-26}$ (note that this equals $a \cdot 2^2 \cdot 2^{-28}$), or, that the true binary point is 26 places to the right of the machine binary point.

Now, using the same process for placing b, and for interpreting c, let's add, using our new values, and check the result.

$$\begin{array}{l} 000000000000000000000000011\,|\,00 = a \cdot 2^{-26} \\ \underline{000000000000000000000000010\,|\,11\,0 = b \cdot 2^{-26}} \\ 000000000000000000000000100\,|\,01\,0 = c \cdot 2^{-26} = 100.01_{(2)} = 4.25_{(10)}, \end{array}$$

which is correct.

If we let A* be the machine representation of any binary number A, then

$$A^* = 2^n \cdot A,$$

where n may range from 0 through -28, depending on the fractional accuracy being carried with A.

The entire subject we have just discussed is called "scaling". In the above expression, 2^n is called the "scale factor" of A. Every number is "scaled" somehow when it is placed in the computer. A is said to be scaled 2^n in the machine. From now on, we won't have to make long, cumbersome references to the true binary point in a number as it appears in the machine; we can simply say the number is scaled in a certain manner, and you will immediately know where the true binary point is.

As you can see, the concept of scaling is really quite simple and straightforward; scaling is a short-hand method of interpreting the numbers that appear in a digital computer.

In ordinary decimal arithmetic, you know that you cannot add

$$\begin{array}{r} 100.962 \\ \underline{6.75401}, \end{array}$$

because the decimal points are not lined up correctly. Rather, you can add them, but you shouldn't; the result will be meaningless. Similarly, in ordinary binary arithmetic, you cannot, or at least you sould not, add

$$\begin{array}{r} 110.111 \\ \underline{1.00011}, \end{array}$$

because the binary points are not lined up correctly. In the computer, you should not add two numbers which are scaled differently, for the same reason. You can, and occasionally programmers have, but the result is meaningless, as they have found out, much to their chagrin.

Suppose we are to add a + b, where a is scaled 2^{-15}, and b is scaled 2^{-13}. It's obvious that one or the other of these numbers will have to be moved, in order to line up the true binary points prior to the addition. You already know how numbers are moved back and forth within a word in the computer: they're shifted in one direction or the other. In the case of pencil-and-paper arithmetic, the job of lining up the base points of two numbers, in order to add them, is simple: we rewrite the numbers. In the previous binary addition, we would rewrite the numbers as

$$\begin{array}{r} 110.111 \\ \underline{\ \ 1.00011}, \end{array}$$

and proceed to add them. Unfortunately, as we shift numbers in a computer, we must lose bits. 29 bits are allotted to each single-precision number; after it is shifted, there will still be only 29 bits allotted to any single-precision number. Thus, if the number is shifted to the left, bits will be lost from the most significant end; if the number is shifted to the right, bits will be lost from the least significant end. In the case under consideration, a can be shifted left two places, increasing it by a factor of 2^2, and thus rescaling it from 2^{-15} to 2^{-13}, and making it compatible with b. Or b can be shifted to the right two places, decreasing it by a factor of 2^{-2}, and thus rescaling it from 2^{-13} to 2^{-15}, making it compatible with a.

Which would be the better scheme can be determined from consideration of a number of factors:

1. the desired scaling of the answer, if any particular scaling is desired;

2. the number of integral bits that must be allowed to insure that overflow will not occur when the numbers are added (this can be determined by considering the largest possible sum of a and b, and in all events, this number of bits must be allowed, regardless of what shifting is necessary to insure it; otherwise, the answer will be erroneous);

3. the fractional accuracy desired in the sum.

From these considerations and perhaps others, unique to a given problem, you will determine the shifting that is required prior to the addition. It may be that both numbers will have to be shifted. In any event, once you have decided that shifting is necessary prior to an addition in your program, you will, of course, need a command which will direct the computer to do it.

The shift command is another special command, with D = 31, S = 26, C = 1 (or any other non-zero number), and T = a relative timing number (similar in function to T for a multiply or divide command). The command will be immediate, and, like multiplication and division, it is double-precision in nature. Two word-times are required for each shift of one bit-position. Therefore, two times the number of bit-positions desired in the shift = T. If you wished to shift a number 10 bit-positions in either direction, T of the shift command would equal 20. Because this operation is immediate and double-precision in nature, it must be located at an odd word-time.

You have already seen that shifting can take place in the two-word registers, and this is where the shifting caused by this command will occur. When this command is executed, ID will shift to the right the indicated number of bit-positions, and, simultaneously, MQ will shift to the left the same number of bit-positions. If you have one number you want to shift, prior to giving the shift command you must place that number in the appropriate two-word register. Either half of the register will do for a single-precision number. You might have a number in each of these registers, one moving to the right, the other to the left.

Notice, if you have a single-precision number you wish to shift to the right, and you load that number in ID_1, then execute the shift command, the number will move to the right, and the vacated bits will be filled in with 0's, which, of course, would be fine. But, under the same conditions, if you loaded that number in ID_0, the vacated bit-positions in ID_0 would be filled in with bits from ID_1, and unless ID had been previously cleared, the single-precision word containing your number would receive "garbage", which could very well contain 1's. Of course this would change your number, making it erroneous. A similar situation, but in reverse, holds true for the shifting of a single-precision number to the left in MQ.

You are probably wondering why any non-zero C is permissible in the command discussed above, and why a C of zero is not permissible. The only function of a C in this command is to distinguish it from a similar command, with D = 31, S = 26, and C = 0. The latter is also a shift command, calling for the exact operation described above, but if C = 0, a tally of the shifts performed will be kept in AR. For each complete shift of the registers by one place, $1 \cdot 2^{-28}$ will be added to the present contents of AR.

Of course the operation called for by this second shift command will cease at the end of the indicated number of word-times, just like any other command. But it will also cease if an end-around-carry is generated in AR, regardless of whether or not the indicated number of word-times have been consumed. In other words, this shift is performed under control of AR.

An example of the usefulness of such a command might be the following: rescale the binary number x, in the computer, by a factor of $2^{(a-b)}$, where a and b will also be available in the computer. Assume all numbers are single-precision. When you originally write your program, you won't know how many shifts to call for to be performed on x. As a matter of fact you won't even know in which direction x is to be shifted. All this depends on the current values of a and b.

You could subtract b from a in AR, and, depending on the sign (+ or -) of the answer, you could load x into the proper half of the proper two-word register: ID_1 if x is to be shifted to the right, because the sign of (a - b) is negative; MQ_0 if x is to be shifted to the left, because the sign of (a - b) is positive. There is an implication here that some provision is available to programmers to cause their programs to automatically determine which of two alternate logical paths to follow, based on inspection of a given condition in the computer. This is correct, and the method available for doing this will be discussed shortly, in pages 105 - 109. For the moment, you may assume that such a decision has been made, and x is in the proper two-word register.

The problem now is to use the number in AR to control the shifting process. We know that the shift command we want has D = 31, S = 26, C = 0. It's operation will cease either when an end-around-carry has been generated in AR or when the number of word-times called for by T has been consumed, whichever occurs earlier. We will set T with some maximum number, so that, unless (a - b) is useless (due to the fact that it calls for so many shifts that all of x will be lost), (a - b) will effectively control the process. Assuming that we want only a single-precision answer, $x \cdot 2^{(a-b)}$, from either ID_1 or MQ_0, the maximum number of shifts that can be performed in either direction, without losing all significance, will be twenty-seven. On the twenty-eighth shift in either direction, all twenty-eight magnitude bits of the original x will be lost. We will therefore set T = 54 (= 2·27). And thus we have the shift command that will be included in our program.

The problem now is to so set AR that, after (a - b) shifts have been performed, and $(a - b) \cdot 1 \cdot 2^{-28}$ has been added to AR, an end-around-carry will be generated. Any positive number plus its negative complement will yield +0 in the computer. Therefore, if we start with the negative complement of $|(a - b)|$ in AR, and if we add $1 \cdot 2^{-28}$ to it $|(a - b)|$ times, we will have generated, in AR, the quantity,

$$-|(a - b)| \cdot 2^{-28} + |(a - b)| \cdot 2^{-28},$$

and this must be +0. Because we cannot know that (a - b) will always be positive, we use its absolute value, which of course will be positive. After the last shift, there will be an end-around-carry, and the sign will be changed to +. The end-around-carry will halt the shifting process.

Of course the same scaling rules that apply to addition of single-precision numbers apply as well to the subtraction of single-precision numbers.

Multiplication is a slightly different case. If a·b is desired, a is scaled 2^{-15}, and b is scaled 2^{-13}, ab will be scaled 2^{-28}, in accordance with the rule of exponents.

$$a \cdot 2^{-15} \cdot b \cdot 2^{-13} = a \cdot b \cdot 2^{-28}.$$

A very simple rule governing the scaling of a product in PN is, following the multiplication of one number by another, the product will be scaled by a factor equal to the product of the scale factors of the two numbers.

Consider now a multiplication of $1 \cdot 2^{-28}$ by $4 \cdot 2^{-28}$. The product will be 4, scaled 2^{-56}. The initial condition of the two-word registers would be:

	Word 1	Word 0
ID:	00000000000000000000000000010	0000000000000000000000000000
MQ:	00000000000000000000000001000	YYYYYYYYYYYYYYYYYYYYYYYYYYYY
PN:	00000000000000000000000000000	0000000000000000000000000000

IP: 0 or 1, whichever is the correct sign of the product.

Each bit in MQ, starting with T29 of MQ_1, will be checked for 0 or 1. If it is 0, nothing will be added to PN; if it is 1, the present contents of ID will be added to PN. The T number of the command will be 56, allowing the inspection of 28 bits from MQ. Thus the Y's in MQ_0, shown above, representing the original contents of MQ_0 before the multiplier was loaded, will have no bearing on the process. The bits in MQ are made available for inspection by being shifted out of MQ, to the left, to an inspection station. A shift is performed before the first bit from MQ_1 can be inspected. For each shift to the left of MQ, ID is shifted to the right. 25 0's will be inspected before the 1 in MQ_1 is sensed at the inspection station. Then the 1 will be shifted into the inspection station, making a total of 26 shifts to the left, before ID is to be added to PN for the first time. This means that ID will have been shifted to the right 26 times before it is added to PN. ID will then look like this:

	Word 1	Word 0
ID:	00000000000000000000000000000	0000000000000000000000001000

Since this is the only 1 that will be found in MQ, this is the only addition to PN that will take place. Therefore, upon completion of the multiplication, PN will also look like the above. Notice that, in the full double-precision magnitude of the two-word register (we previously stated that, during both multiplication and division, the operation of the two-word registers is essentially double-precision in nature, even if single-precision numbers are actually involved in the operation), the answer is $4 \cdot 2^{-56}$. This is to be expected;

the computer was given two fractions to multiply together, each of which was very small. Naturally, the resultant fraction will be even smaller, and, in fact, it is so small that, if you demand a 28-bit expression of its value (take the single-precision answer from PN_1), the nearest value to it that can be expressed in 28 bits is 0.

Notice, then, that multiplication can result in an answer whose scale factor will require more than the 28 bits of PN_1 for expression. As long as you are aware of this, and can devise methods for using the answer, fine. But, if you want the answer expressed in 28 bits, re-scale the two numbers entering into the multiplication before you multiply, in such a way that the scale factor of the product (equal to the product of the scale factors of the two numbers) will lie in the range $2^0 - 2^{-28}$.

In division, the scaling rule is: the quotient, in MQ, will be scaled by a factor equal to the quotient of the scale factors of the numbers being divided. For example, if a/b is desired, a is scaled 2^{-15}, and b is scaled 2^{-13},

$$\frac{a \cdot 2^{-15}}{b \cdot 2^{-13}} = \frac{a}{b} \cdot 2^{-2}$$

Another example:

$$\frac{a \cdot 2^{-28}}{b \cdot 2^{0}} = \frac{a}{b} \cdot 2^{-28}$$

And finally, one more example:

$$\frac{a \cdot 2^{-28}}{b \cdot 2^{-28}} = \frac{a}{b} \cdot 2^{0}$$

In each of the above examples, there is a basic assumption that a appears in the machine to be smaller than b, in accordance with the rule for division. Notice that a can appear smaller than b, in the machine and yet, as in the second example above, we interpret it as being of greater magnitude than b. The scaling we associate with a number in the machine is unknown to the computer; it merely aids us in interpreting the numbers the computer works with. In the second example, we know that a, as it appears in the machine, represents a 28-bit binary integer, counting any leading 0's (because the true binary point is 28 places to the right of the machine binary point), while b, in the machine, represents a 28-bit binary fraction, counting any trailing 0's, (because the true binary point coincides with the machine binary point). In reality, then, as we interpret these numbers in the machine, we are dividing a relatively large magnitude by a relatively small magnitude. The computer, not realizing this, will perform the division correctly, without generating overflow, as long as the 28-bit value in the machine representing a is less than the 28-bit value in the machine representing b.

We have discussed scaling in the light of single-precision numbers in order to minimize the number of bits you have to keep track of. All of the principles and rules of scaling that have been mentioned apply equally well to either single- or double-precision numbers.

Now all four basic arithmetic operations (+, -, x, ÷) are available, and you know how to arrange numbers in the computer to suit your purposes. You also know how to interpret the results. In the solution of the quadratic equation, there is one operation that has not, as yet, been described: it is taking the square root of a number ($\sqrt{b^2-4ac}$). There is no one command that will cause the computer to do this, because the computer is not wired to do it directly. We can generate the square root of any number through a combination of the four basic arithmetic operations, repeated over and over again, but we will, for the present, postpone a discussion of this.

We are ready to expand the original flow diagram of our solution of the quadratic equation, as it appears on page 4, but first, we must decide on what ranges of values we will allow for a, b, and c. Remember the formula is:

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

We previously decided to use single-precision, so the scale factor for each value must lie in the range 2^0 to 2^{-28}. Let's arbitrarily allow 7-bit fractional accuracy in the binary numbers, so that a, b, and c will be scaled 2^{-21}. In terms of decimal equivalents, this means that our program will be able to process values accurate to the nearest 1/100th, since $1/2^7 = 1/128$, and this is even a smaller value than 1/100. Notice that 6 fractional bits would not give accuracy to the nearest 1/100, since $1/2^6 = 1/64$. Now let's also assume that we want x to the same accuracy, scaled 2^{-21}. We could do whatever shifting is necessary to insure that the numerator, prior to the division, is scaled 2^{-21}. The question, therefore, is, how do we determine what scaling the denominator needs? We will find the answer from the following equation:

$$x \cdot 2^{-21} = \frac{N \cdot 2^{-21}}{D \cdot 2^n}$$

The solution of the above equation is: $n = 0$. This means that the true binary point of D would have to coincide with the machine binary point, meaning that $-1 < D < 1$. Since $D = 2a$, $-1 < 2a < 1$, or $-1/2 < a < 1/2$. This is, of course, too great a restriction on a; our program could, in no sense, be called a general program.

Let's go in the other direction:

$$x \cdot 2^{-21} = \frac{N \cdot 2^{n}}{D \cdot 2^{-21}}$$

The solution of the above equation is: n = -42. We know that we can position N in such a way as to meet this requirement, through shifting. This would seem to work out quite well, so let's do it.

This means that the integral portion of 2a is going to be expressed in 21 bits. 2a is scaled 2^{-21} in the machine. Since this is so, we better make sure that the integral portion of a does not, in any case, exceed 20 bits, even though a will originally be scaled 2^{-21} (in other words, a, as originally stored in the machine, will always have at least one leading 0). In 20 bits we can express all integral values up to, and including, 2^{20} -1; this, then, becomes the limit for a. $2^{20}=1048576_{(10)}$. Therefore, $-1048575 \leqslant a \leqslant 1048575$. Now that a set of limits has been found for a, let's find a similar set for b and c. Notice that we are going to generate 4a. If 2a requires 21 bits, $2^{1} \cdot 2a$ will require 22 bits. We would like to shift the product 4ac in such a way as to scale it 2^{-42}. The reason for this is, if b is scaled 2^{-21}, as it was agreed it would be, b•b will be scaled 2^{-42}. If 4ac is scaled the same way, we can subtract immediately, without having to rescale b^2. If 4ac is to be shifted to be scaled 2^{-42}, its integral value must not, under any conditions, require more than 42 bits for expression. We have already seen that 22 bits will be necessary for 4a. If $c = 2^{20}$, $4a \cdot 2^{20}$ will require 42 bits for expression. Therefore c cannot exceed $2^{20} = 1048576_{(10)}$.

It is possible that what looks like a subtraction in the formula, b^2-4ac, might very well become an addition, if either a or c, but not both, is negative. Therefore, it might be possible, if we allow 42 bits for the integral portion of both b^2 and 4ac, that the combination of b^2 and 4ac will cause overflow. Since this is undesirable, we must prevent it. We can do this by limiting the integral value possible, in the generation of 4ac, to 41 bits, thus being sure that in all cases, as it is expressed in 42 bits, it will have at least one leading 0. If we similarly limit b^2, no overflow will be possible when we add b^2 and -4ac. Therefore, we will revise our limit for c. Whereas we originally suggested that c not exceed 2^{20}, we will now say that c may not exceed $2^{19} = 524288_{(10)}$.

If we limit both b^2 and -4ac to 41 integral bits, the result of b^2-4ac will be limited to 42 bits. When we take the square root of that number, we will get a number whose integral value is limited to 21 bits, and this number will be scaled 2^{-21}, since the square root of 2^{-42} is 2^{-21}. When this is combined with b, however, to form the final numerator, overflow might result. The square root will have to be limited, in its integral portion, to 20 bits, scaled 2^{-21}, meaning that it will have a leading 0. If this is so, the radicand, b^2-4ac, will have to be limited, in its integral portion, to 40 bits. This means that b^2 and -4ac will have to be limited to 39 integral bits, to assure no possibility of overflow when they are combined. The limits on a and c,

up to this point, will limit the integral value of 4ac to 41 bits. To reduce this to 39 bits, we could further cut down on c, but it would be preferable to cut the limit on a, assuming that, in the equation, ax^2+bx+c, greater values will be desired for c than for a. We have previously seen that presently 4a will require 22 integral bits. If we cut this down to 20, and c retains its limit of 19, the limit of the integral bits in 4ac will be the desired 39. Since $4a = 2 \cdot 2 \cdot a$, to get a result limited to 20 integral bits, we must limit a to 18 integral bits, meaning that the maximum a expressable will be $2^{18}-1$. $2^{18} = 262144_{(10)}$; therefore $-262143 \leqslant a \leqslant 262143$.

Similarly, the integral portion of b^2 is limited to 39 bits. If b contains 20 integral bits, b^2 may contain 40. If b contains 19 integral bits, b^2 may contain 38, which meets our requirement. So we will limit b to 19 integral bits, the maximum b then being $2^{19}-1$. $-524287 \leqslant b \leqslant 524287$.

Now no overflow will be possible in either the generation of b^2-4ac or the generation of $-b \pm \sqrt{b^2-4ac}$.

We have thus set up the following limits and scale factors, for this program:

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

where:

$-262143 \leqslant a \leqslant 262143$, or

$-(2^{18}-1) \leqslant a \leqslant (2^{18}-1)$, where a is scaled 2^{-21};

$-524287 \leqslant b \leqslant 524287$, or

$-(2^{19}-1) \leqslant b \leqslant (2^{19}-1)$, where b is scaled 2^{-21};

$-524288 \leqslant c \leqslant 524288$, or

$-(2^{19}) \leqslant c \leqslant (2^{19})$, where c is scaled 2^{-21};

x will be scaled 2^{-21}.

With these ranges of values for a, b, and c, we can truly say that this program can be used in almost any application, in order to solve for the roots of a quadratic equation. There is one further restriction:

if $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$, then

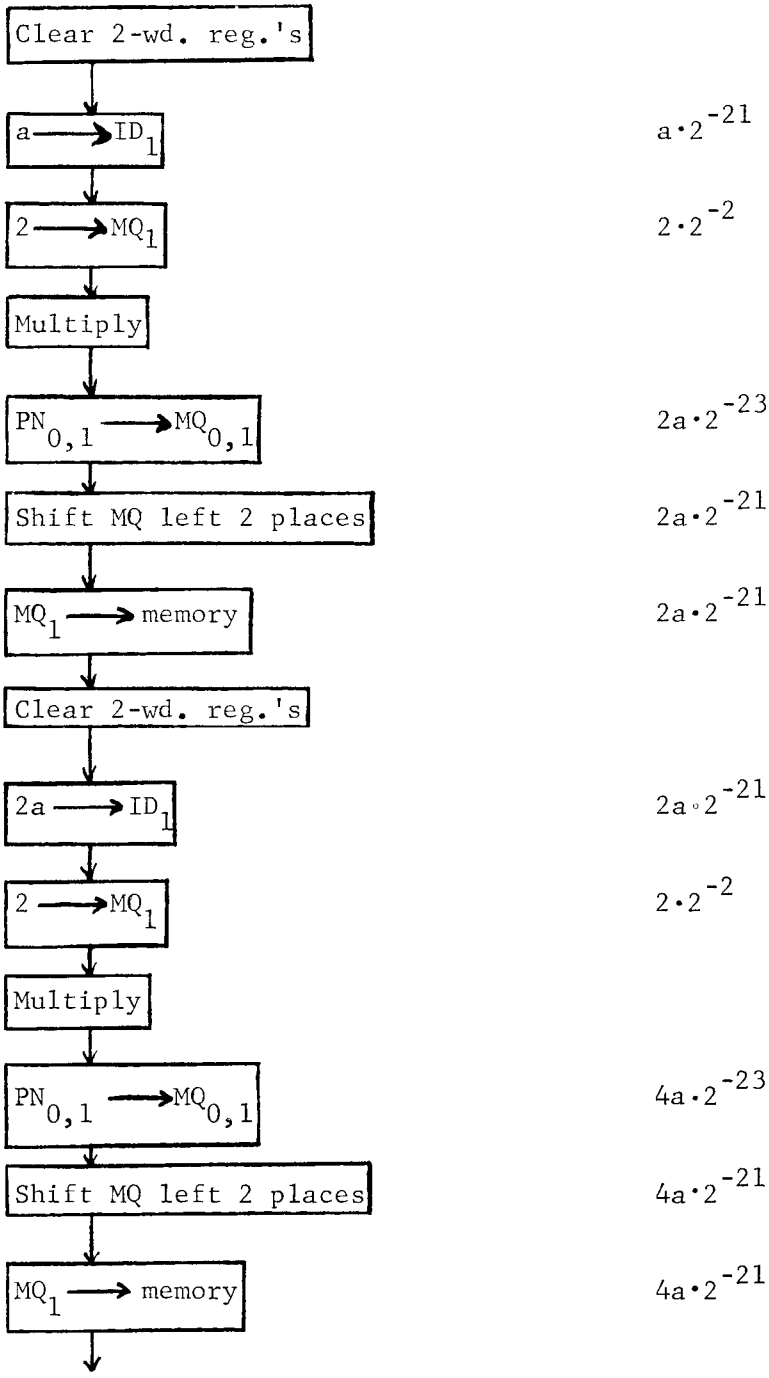
as a approaches 0,

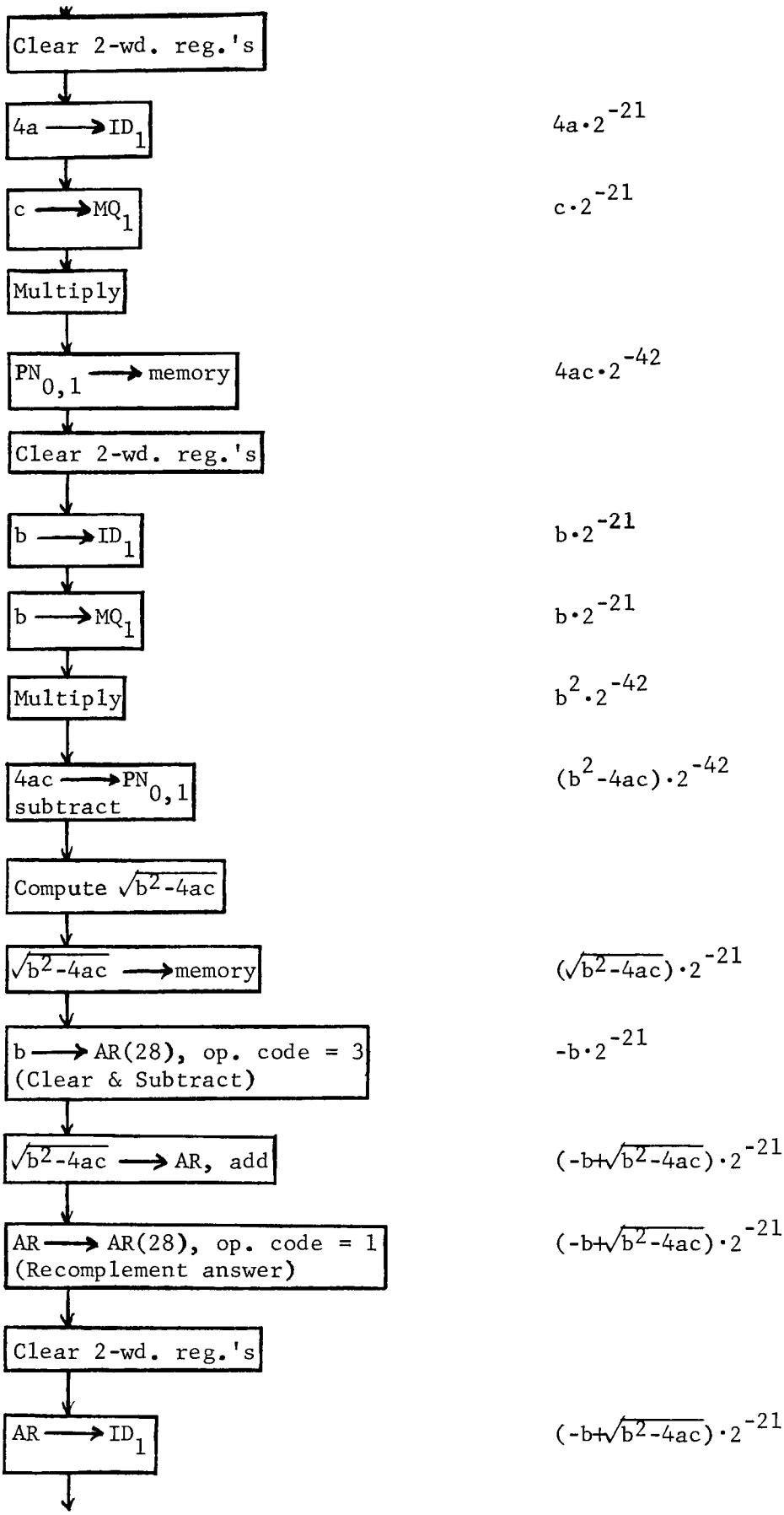
x approaches $\frac{-b+b}{0} = \frac{0}{0}$ or $\frac{-2b}{0}$,

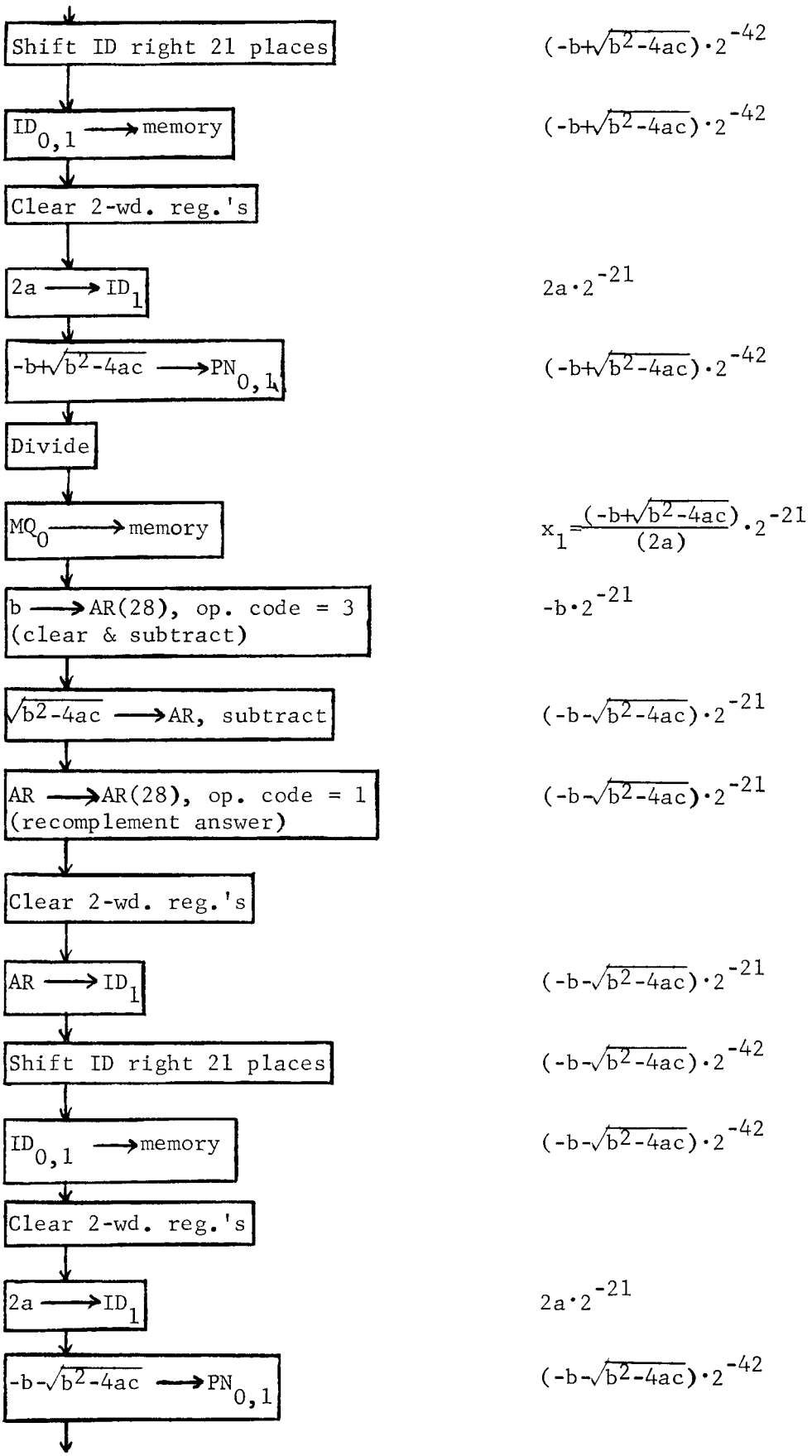
and the division will yield an erroneous result. In any case, a must be unequal to 0. If $|a| < 1/2$, the limit for $|b|$ will decrease in proportion.

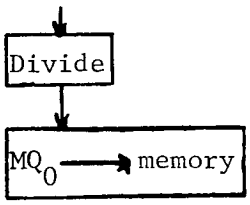
FLOW DIAGRAM

We can now go on to expand the original flow diagram, as it was developed on page 4.




Clear 2-wd. reg.'s
4a → ID1
4a·2^-21
c → MQ1
c·2^-21
Multiply
PN0,1 → memory
4ac·2^-42
Clear 2-wd. reg.'s
b → ID1
b·2^-21
b → MQ1
b·2^-21
Multiply
b^2·2^-42
4ac → PN0,1
subtract
(b^2-4ac)·2^-42
Compute √(b^2-4ac)
√(b^2-4ac) → memory
(√(b^2-4ac))·2^-21
b → AR(28), op. code = 3
(Clear & Subtract)
-b·2^-21
√(b^2-4ac) → AR, add
(-b+√(b^2-4ac))·2^-21
AR → AR(28), op. code = 1
(Recomplement answer)
(-b+√(b^2-4ac))·2^-21
Clear 2-wd. reg.'s
AR → ID1
(-b+√(b^2-4ac))·2^-21


Shift ID right 21 places
$(-b+\sqrt{b^2-4ac})\cdot 2^{-42}$
$ID_{0,1} \longrightarrow$ memory
$(-b+\sqrt{b^2-4ac})\cdot 2^{-42}$
Clear 2-wd. reg.'s
$2a \longrightarrow ID_1$
$2a\cdot 2^{-21}$
$-b+\sqrt{b^2-4ac} \longrightarrow PN_{0,1}$
$(-b+\sqrt{b^2-4ac})\cdot 2^{-42}$
Divide
$MQ_0 \longrightarrow$ memory
$x_1=\frac{(-b+\sqrt{b^2-4ac})}{(2a)}\cdot 2^{-21}$
$b \longrightarrow$ AR(28), op. code = 3 (clear & subtract)
$-b\cdot 2^{-21}$
$\sqrt{b^2-4ac} \longrightarrow$ AR, subtract
$(-b-\sqrt{b^2-4ac})\cdot 2^{-21}$
AR $\longrightarrow$ AR(28), op. code = 1 (recomplement answer)
$(-b-\sqrt{b^2-4ac})\cdot 2^{-21}$
Clear 2-wd. reg.'s
$AR \longrightarrow ID_1$
$(-b-\sqrt{b^2-4ac})\cdot 2^{-21}$
Shift ID right 21 places
$(-b-\sqrt{b^2-4ac})\cdot 2^{-42}$
$ID_{0,1} \longrightarrow$ memory
$(-b-\sqrt{b^2-4ac})\cdot 2^{-42}$
Clear 2-wd. reg.'s
$2a \longrightarrow ID_1$
$2a\cdot 2^{-21}$
$-b-\sqrt{b^2-4ac} \longrightarrow PN_{0,1}$
$(-b-\sqrt{b^2-4ac})\cdot 2^{-42}$



$$x_2 = \frac{(-b-\sqrt{b^2-4ac})}{(2a)} \cdot 2^{-21}$$

In this expanded flow diagram you can see the program begin to take shape. The arrows connecting the boxes indicate the logical path of the program, from step to step. Each box in this flow diagram represents one command, with perhaps one or two exceptions, one of which is the box containing "Compute $\sqrt{b^2-4ac}$". A former statesman once said, "My job is to reduce problems to manageable proportions." That is exactly the function and purpose of a flow diagram. Initially, a problem may seem quite complicated and unmanageable, but, when it is disected into individual little parts, each part becomes easily understandable and manageable. Each programmer develops his own method of flow-diagramming. The diagram above is perhaps a little more detailed than need be, but limiting the logical size of the boxes in a flow diagram to approximately one command per box is a fine idea when starting out as a programmer.

Notice that arrows have been used inside boxes to indicate the direction of a transfer.

Certain transfers are usually named in order to simplify references to them. The transfer of a number into AR or PN, replacing what was originally there (D = 28 for AR, 26 for PN), prior to an addition (the characteristic will usually equal 0 or 1), is called "clear and add". The transfer of a number into either of the same two registers, to be combined with their present contents (D = 29 for AR, 30 for PN), thus performing an addition, is called "add". The transfer of the magnitude of a number into AR (C = 2), replacing what was originally there (D = 28), prior to an addition, is called "clear and add magnitude". The transfer of a number into AR or PN with the same characteristic and D = 29 or 30 is called "add magnitude". The transfer of a number into AR with a C = 3, and D = 28, is called "clear and subtract". The transfer of a number into AR or PN with the same characteristic and D = 29 or 30, is called "subtract". The transfer of the result of any of these operations from either AR or PN to some storage location in memory is referred to as "storing" the result.

Remember that if any two-word register, with the exception of PN = 30, (which is greater than or equal to 28), is the destination of a transfer whose C code is 2, 3, 6, or 7, the operation called for will be a transfer via AR, and the even half of the two-word register will be cleared. Therefore, "clear and add magnitude" into PN, which would require a destination of 26, is out. But a transfer of a number into PN with C = 4 will divorce the sign from the magnitude and load it into IP. This leaves the magnitude of the number in PN, with a positive sign (0), in T1 of PN_0, where we want the sign of a double-precision number which is to be involved in an addition.

Similarly, "clear and subtract" into PN is out, since its C code is 7, and PN must be referred to as 26. But the same thing can be accomplished by first clearing PN, and then subtracting a number from 0.

THE NEED TO AUTOMATICALLY CHECK COMPUTATIONS

There are certain conditions which might arise in the operation of this program, as it is written, which would result in erroneous answers. Despite the scaling and the limitations on a, b, and c that we have chosen, in either of the two divisions we have incorporated, the numerator could exceed the denominator in apparent value in the machine. As has been pointed out, this would cause an overflow and an erroneous quotient. But how will the person using the program know when this has occurred? How will he know which answers can be trusted, and which cannot? We must include something in our program which will prevent the output of an erroneous answer.

We have been very careful, in our choice of scaling and limitations, to prevent the possibility of overflow in any of the necessary additions or subtractions. Does this mean that no overflow can occur as a result of any of these? Ideally, yes; practically, no. If the limitations, as we set them, on a, b, and c, are obeyed, no overflow will occur. But, never trust anyone else to follow your limitations when using your program. Anyone who knows <u>how</u> to operate the computer, without knowing <u>why</u> it does what it does, might try to use this program for his purposes. To him, because he might not understand <u>why</u> the limitations have been imposed, they may be meaningless. If he attempts to use values outside the prescribed ranges, overflow might result. He will, of course, be unaware of this, and treat the answers he receives as accurate, unless they are obviously wrong. To prevent this sort of thing from happening, even though we have taken steps to prevent it, we must include in the program something which will prevent an output in the case of overflow resulting from addition or subtraction.

Our program, as diagramed, includes a computation of a square root. It is possible that the radicand might be negative. We will assume that we do not want imaginary numbers as answers. We must, therefore, make sure the radicand is not negative before proceeding to compute and put out an answer.

In short, there are two types of deleterious conditions that might arise during the operation of almost any program. One is that type of situation that cannot adequately be prevented through an "ounce" of caution, because it might arise from given data which, on the face of it, seems to be perfectly acceptable. The other is that type of situation that arises when someone other

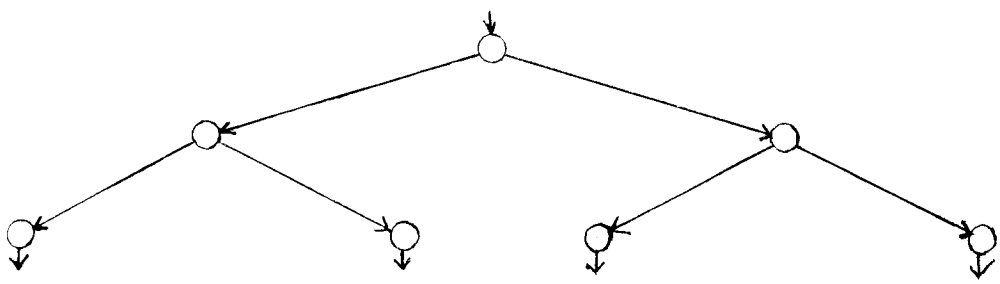
than the programmer, himself, attempts to twist the program to suit his own needs, heedless of warning. In this case, the programmer might very well like the output to consist of a few well-chosen four-letter words, but, for our purposes, we will be content with merely frustrating the offender by refusing to give him an answer.

TEST COMMANDS

The G-15, like many other digital computers, has the ability to determine the presence or absence of any one of several conditions, and a program can be written with two alternate logical paths, either of which will be followed, during operation of the program, depending on the decision made by the computer. The program itself will tell the computer when to "test" a particular condition, and the commands which do this are called "test" commands. Depending on which state the tested condition is in (off or on), the computer will take its next command from N (as it usually would) or N + 1, respectively. Always remember that only one of two answers to the test is possible: there is no "maybe" in the computer.

This simple "decision-making" power of computers is what has led laymen to use the term "electronic brain", and other equally erroneous terms, when referring to computers. You can see that, actually, the G-15 does not "think"; it merely tests, upon command, the condition of a circuit or component as to "on" or "off", and, in this respect only, it can answer "Yes" or "No" to a particular, properly chosen, question or "test".

A limitless number of tests can be included in any program, each with two alternate paths, so a program's flow diagram, unlike the straight, unswerving one we generated, can take on the shape of a "tree".



The following tests are available in the G-15:

1. test for overflow, D = 31, S = 29, C = 0;

2. test for sign of AR (neg.), D = 31, S = 22, C = 0;

3. test for "ready", D = 31, S = 28, C = 0;

4. test for punch switch on, D = 31, S = 17, C = 1;

5. test for non-zero, D = 27, S = any memory line.

Test for overflow:

You can see that this is a special command (D = 31). It commands the computer to test the condition of the overflow flip-flop. If it is off (no overflow), the next command will be taken from N (as usual). If it is on (overflow), the next command will be taken from N + 1 (thus changing the path of the program). In our program, this path will not contain further computation, but will halt the program (the command to halt computer operation has not yet been discussed).

We will use this test immediately following additions and subtractions.

The manner in which the computer determines the existence of an overflow condition is somewhat indirect, and should be understood. We will discuss it in relation to a single-precision addition in AR.

Three questions are automatically asked by the computer when the addition is performed:

1. Is the intermediate sign of the result 0 (= +)?

2. Did the inverting gates complement the last number to enter AR?

3. Was there an end-around-carry out of bit T29 of AR?

The computer uses the answers to these questions in order to determine whether or not an overflow was generated in the following way:

1. If the answer to question (1) is "no", the intermediate sign of the result is 1 (= -), the two numbers added were of unlike sign, and overflow could not result. If the answer is "yes", the intermediate sign of the result is 0 (= +), the two numbers added were of like sign (both + or both -), and overflow could result. In this case only, proceed to (2).

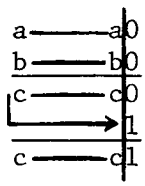
2. If the inverting gates did not complement the last number to enter AR, both numbers were positive; if the inverting gates did complement the last number to enter AR, both numbers were negative. In either case, proceed to (3).

3. If both numbers were positive, and an end-around-carry did occur, overflow is present; if both numbers were positive, and an end-around-carry did not occur, overflow is not present.

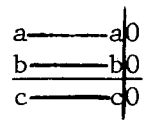
Examples:

if a + b = c,

Overflow



No Overflow

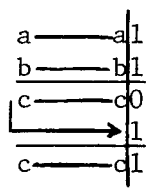


If both numbers were negative, and an end-around-carry did occur, overflow is not present; if both numbers were negative, and an end-around-carry did not occur, overflow is present.

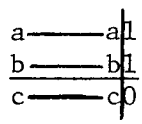
Examples:

if a + b = c,

No Overflow



Overflow



The importance of understanding the method in which the computer determines the presence or absence of overflow is pointed up by the following example:

Assume we wish to double a negative number by adding it to itself:

$$2\cdot(-a) = -a + (-a) = -2a.$$

We could transfer the number (-a) into AR, replacing the original contents of AR, with a properly coded command containing C = 1 and D = 28. The number will be complemented on its way to AR, and it will be ready for addition. Now we could transfer the contents of AR to AR, calling for an addition (S = 28, D = 29). In this latter command, however, C must equal 0, so that the complement form of the negative number will be retained. We would therefore write a command with C = 0, S = 28, D = 29.

But look at what happens to us when we attempt to check overflow.

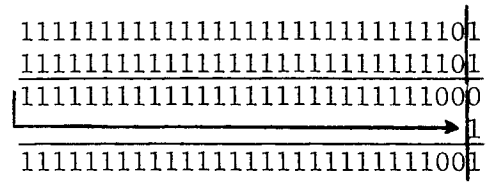
Assume the number was $-2\cdot2^{-28}$:

00000000000000000000000000101.

After execution of the first command mentioned above, AR will contain:

11111111111111111111111111101,

which is the complement of the original negative number, ready for addition. Because of the C = 0 in the second command discussed above, the following addition will be performed in AR:



and this is a valid answer, being the complement of $-4 \cdot 2^{-28}$. We can see that no overflow occurred. But the computer believes that two positive numbers were added, because (1) the intermediate sign of the result is 0 (= +), and (2) the inverting gates did not complement the last number to enter AR (indeed they could not, because C = 0, and the number did not pass through them at all). The computer is aware that an end-around-carry has occurred in AR, through the addition of two positive numbers, and overflow is indicated, the overflow flip-flop being automatically turned on. Therefore, if we test for a possible overflow after this addition, the test will be answered "yes", even though, in reality, no overflow occurred.

Therefore, if we have a number, a, whose sign could be either + or - at the time the program is operated, and if we wish to generate 2a by adding a to itself, and if this could result in a true overflow, necessitating an overflow test following the addition, the best method would be to transfer a from its storage location in memory to AR twice, each time with C = 1. In the first transfer D will be 28, and in the second transfer D will be 29. Now the sum can be checked reliably for overflow.

Test for sign of AR (neg.):

This, too is a special command. It commands the computer to test the sign-bit of the number in AR.

If it is off (0), the next command will be taken from N (as usual). If it is on (1), the next command will be taken from N + 1 (thus changing the path of the program). In our program, this path will not contain further computation, but will halt the program.

We will use this test to determine the sign of the redicand prior to taking the square root; we want to halt rather than take the square root of a negative number.

Test for "ready":

This is a special command. It commands the computer to test for the presence of the "ready" state of the input/output system which we have not, as yet, discussed. We will explain the effect and use of this test later, when we discuss inputs and outputs.

Test for punch switch on:

This is a special command. It commands the computer to test the setting of an external switch. Again, since this test is associated primarily with outputs, we will discuss it later.

Test for non-zero:

Notice that D = 27; this is the only possible line number that has not yet been discussed. This is the only case in which this number is permissible as a destination.

If C = 0, 29 bits of S.T will be tested for non-zero.

If C = 4, 58 bits from the double-precision number contained in S.T and T + 1 will be tested for non-zero.

If C = 1, 29 bits of S.T, after passing through the inverting gates, will be tested for non-zero.

If C = 5, 58 bits of the double-precision number contained in S.T and T + 1, after passing through the inverting gates, will be tested for non-zero.

If S = 28, and C = 2, the magnitude of the number will be tested for non-zero.

If S < 28, and C = 2, all 29 bits of the original contents of AR will be tested for non-zero, and the contents of S.T will be placed in AR.

If C = 6 (S < 28), during the first word-time of execution (even), all 29 bits of the original contents of AR will be tested for non-zero, and S.T will be placed in AR. During the next word-time of execution (odd), AR's contents (S.T) will be tested for non-zero and S.T + 1 will be placed in AR.

If S = 28, and C = 3, the sign of AR will be changed and all 29 bits, after passing through the inverting gates, will be tested for non-zero.

If C = 7 (S < 28), the operation will be the same as for C = 6, except that numbers entering AR will enter via the inverting gates.

In the case of two-word registers, IP will never be tested for non-zero.

. . . . .

Now that we know the test commands that are available, we can incorporate them into our program at strategic places, in order to prevent the output of erroneous answers.

One situation we want to prevent is the division of a number by another which appears to be smaller in value in the computer. In the case of each of the two divisions we have called for, we generate the numerator by shifting $(-b \pm \sqrt{b^2-4ac})2^{-21}$ right 21 places, arriving at a double-

precision value in ID equal to $(-b \pm \sqrt{b^2-4ac})2^{-42}$. We want to assure ourselves that this value is less than the double-precision value, $2a \cdot 2^{-21}$. To do this we must subtract the absolute value (positive magnitude) of $2a \cdot 2^{-21}$ from the absolute value of $(-b \pm \sqrt{b^2-4ac})2^{-42}$, and inspect the sign of the result.

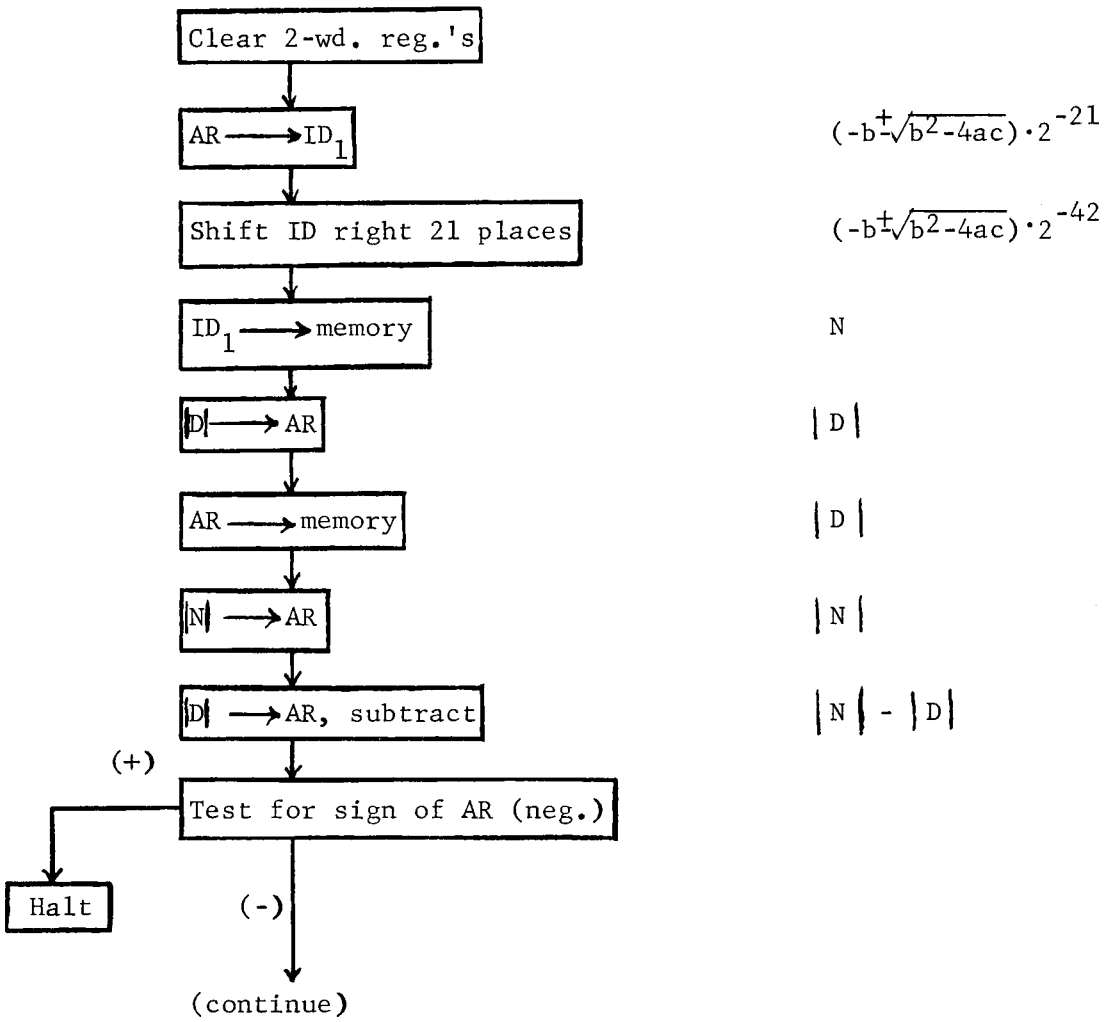
TO SUBTRACT A MAGNITUDE

You probably noticed, in the discussion of the various transfers possible in the G-15, there was no mention of "subtract magnitude" or of "clear and subtract magnitude", although mention was made of "add magnitude" and "clear and add magnitude". The reason for this is that no single transfer available will cause the subtraction of a magnitude. This is most easily accomplished through a series of transfers. We can clear and add the magnitude of D into AR, then store AR's contents in memory. In memory, then, we will have $|D|$. Now we can clear and add the magnitude of N into AR. AR will now contain $|N|$. Now we subtract from AR the number in memory which equals $|D|$. The result in AR is $|N| - |D|$. This number may be either positive or negative, depending on whether $|D|$ exceeds $|N|$ or not. Notice that the result in AR will be positive if the two magnitudes are equal (+0). If we subtract the absolute value of the denominator from the absolute value of the numerator, and get a negative result, we know that the magnitude of the denominator exceeds that of the numerator, and division is permissible. If the result is positive, we know that the magnitude of the numerator either exceeds, or is equal to, that of the denominator, and in either case division is not permissible.

We therefore want to clear and add the absolute value of the numerator into AR. But, the numerator is a double-precision value, scaled 2^{-42}. The denominator, likewise, will be treated as a double-precision value, scaled 2^{-21}, but we know that, when D is in ID, all the least significant magnitude bits, from T1 of ID_1 through T2 of ID_0, will be 0's, because all we do to generate it is to transfer the single-precision number, $2a \cdot 2^{-21}$ into ID_1, being careful to clear the rest of ID. If the most significant 28 bits of magnitude of the numerator equal or exceed the most significant 28 bits of the denominator, we can be sure that the numerator at least equals the denominator, and we cannot divide. We can pick up the first 28 magnitude bits of the numerator from ID_1, following the shift, and leave T1 of that word behind, by transferring out of ID_1 with a C = 0. Carried with the 28 magnitude bits will be the original sign of the numerator, from IP. We'll store this number in memory. Then we'll clear and add the magnitude of D (= $2a \cdot 2^{-21}$) into AR, and store it back in memory, calling it $|D|$. Now we can clear and add $|N|$ (C = 2) into AR, and subtract $|D|$. The result, in AR, will be $|N| - |D|$. If this result is positive, we cannot divide; if it is negative, we can. Therefore, we will use the "test for sign of AR (neg.)" command. Our computation will proceed with the command located at an address one greater than the N of the test command. The command at the location with an address equal to N of the test command will call for a halt.

The "halt" command is another special command; D = 31, S = 16, C = 0. This command is very easy to explain: its execution causes the computer to stop.

Now we can rewrite those two portions of the flow diagram preceding the divide operations.



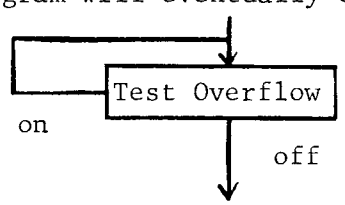
As for overflow errors, unfortunately we cannot test for overflow after shifting MQ left in order to rescale 2a and 4a, since the overflow flip-flop is not connected to MQ during a shift operation, although it is during a divide. If the limit for a is exceeded, an overflow may occur at either or both of these times, and we will be unable to detect it. But we can prevent computation from proceeding if any of the additions or subtractions causes an overflow.

The first point in the program where we can detect an overflow is the generation of b^2-4ac in PN. If we follow this subtraction with an overflow test, the computer will test for the overflow flip-flop being set. Unfortunately, it could have been set earlier, by another program. Once the overflow flip-flop has been set, it can only be turned off in either

of two ways. One is to turn off the computer (not with a halt command, but through an actual switch action which turns off the power). The other way it can be turned off is by the overflow test itself. In addition to testing the flip-flop, this command resets it to the "off" position if it was on. A previous program run in the computer may have turned on the overflow flip-flop and never tested it. In such a case it will remain on.

In order to be sure the overflow flip-flop is off prior to the execution of those steps in our program which could turn it on, we will precede them with an overflow test whose only function is to turn off the flip-flop. The fact that this test command will start either of two alternate paths through our program now becomes a hindrance rather than a help, because we want to continue with the same sequence, regardless of the condition of the overflow flip-flop. We can solve this problem by placing the same command at both N and N + 1, so that, no matter which will be taken as the next command, the same operation will be performed. These two commands will have the same N, so that, following either of them, the same path will be followed through the logic of our program.

Another method is commonly used, however, to achieve the same net effect. We will use it in this program, in order to familiarize you with it. We know that, if the overflow flip-flop is on, the next command will be taken from N + 1. Suppose we choose an N for the test command such that N + 1 = the location of the overflow test command itself. If the overflow flip-flop is off, the program will continue with the command located at N, which is one word-time earlier than the test command. This is fine; the test command will not be read and interpreted again. If the overflow flip-flop is on, the next command will be taken from N + 1, which is the location of the overflow test command itself. This means the test will be repeated. But, the first time the test was made, the flip-flop was reset to the "off" condition. Therefore, this time, when it is tested, it will be found to be off, and the program will continue at N. No matter which condition the flip-flop is in when the test is initially given, the program will eventually continue at N.



If we precede the generation of b^2-4ac with this operation, we can follow the subtraction with another overflow test. This time, the program will halt if overflow is found (N + 1), and it will continue if overflow is not found (N). Notice that, from now on, if the program continues, we need not reset the overflow flip-flop in the manner described above, prior to testing it after a series of arithmetic operations.

Other points at which we want to test for overflow will be following the generation of:

1. $-b + \sqrt{b^2-4ac}$;

2. $\frac{-b + \sqrt{b^2-4ac}}{2a}$;

3. $-b - \sqrt{b^2-4ac}$;

4. $\frac{-b - \sqrt{b^2-4ac}}{2a}$.

We want to include one other test in the program; a test of the sign of (b^2-4ac) prior to attempting to compute the square root of it. If this difference is negative, we want to halt. We will, of course, use the command which tests the sign of AR (neg.). If the answer is yes, the next command will be taken from N + 1, where we will place a "halt". If the answer is no, the program will continue at N.

This completes the use of test commands in the computation.

SUBROUTINES

Up to this point we have very neatly evaded the issue of computing a square root in a computer not wired to do it directly. It must be done through a series of basic arithmetic operations. We can no longer evade it, however; it's the only portion of the computation remaining unplanned. How are we going to do it? A mathematician-turned-song-writer-and-performer has answered our question in one of his songs:

> "Plagiarize, plagiarize, plagiarize.
> Let no one else's work evade your eyes."

Bendix Computer Division, of course, does not recommend or condone plagiarism, but it does supply a standard "package" of programs designed to make life easier for its customers. This package is standard equipment with every G-15 computer. Each of these programs is designed to perform a commonly needed function among computer users. Typical examples are calculation of square roots and trigonometric functions. These programs are called "subroutines". A subroutine operates out of a prescribed command line in memory, with certain inputs, which are also prescribed, and it is usually designed to generate a solution, or set of solutions, which will appear at a prescribed location in memory.

There is a square root subroutine available. The command line prescribed for its execution is line 01. The input, the number whose square root is desired, must be placed in $PN_{0,1}$, prior to execution of the subroutine. The first command is at word-time 94. The answer will appear in $PN_{0,1}$. All of these facts, and others, can be found on a specifications sheet which accompanies a write-up of the subroutine. All subroutines are written-up, and specifications similar to those above are supplied.

If this subroutine is to occupy line 01, certainly our program should not. Assume that our program will occupy line 00. The question, then, is, at the right point in our program, after we have the number whose square root we desire in $PN_{0,1}$, how do we cause the computer to change command lines from 00 to 01, and take its next command from word 94 in the new command line?

There is a special command (D = 31), whose function is to cause the computer to change command lines. This is called the "mark and transfer control" command. In it, S = 21, and C = the line number to which control is to be transferred. The line number specified could be 0, 1, 2, 3, 4, 5, 6 (referring to line 19), or 7 (referring to line 23). The word in the new line at which the new sequence is to start will be located, as usual, in the N portion of the mark and transfer control command.

The mark and transfer control command that we want to incorporate in our program, then, will contain D = 31, S = 21, C = 1, and N = 94.

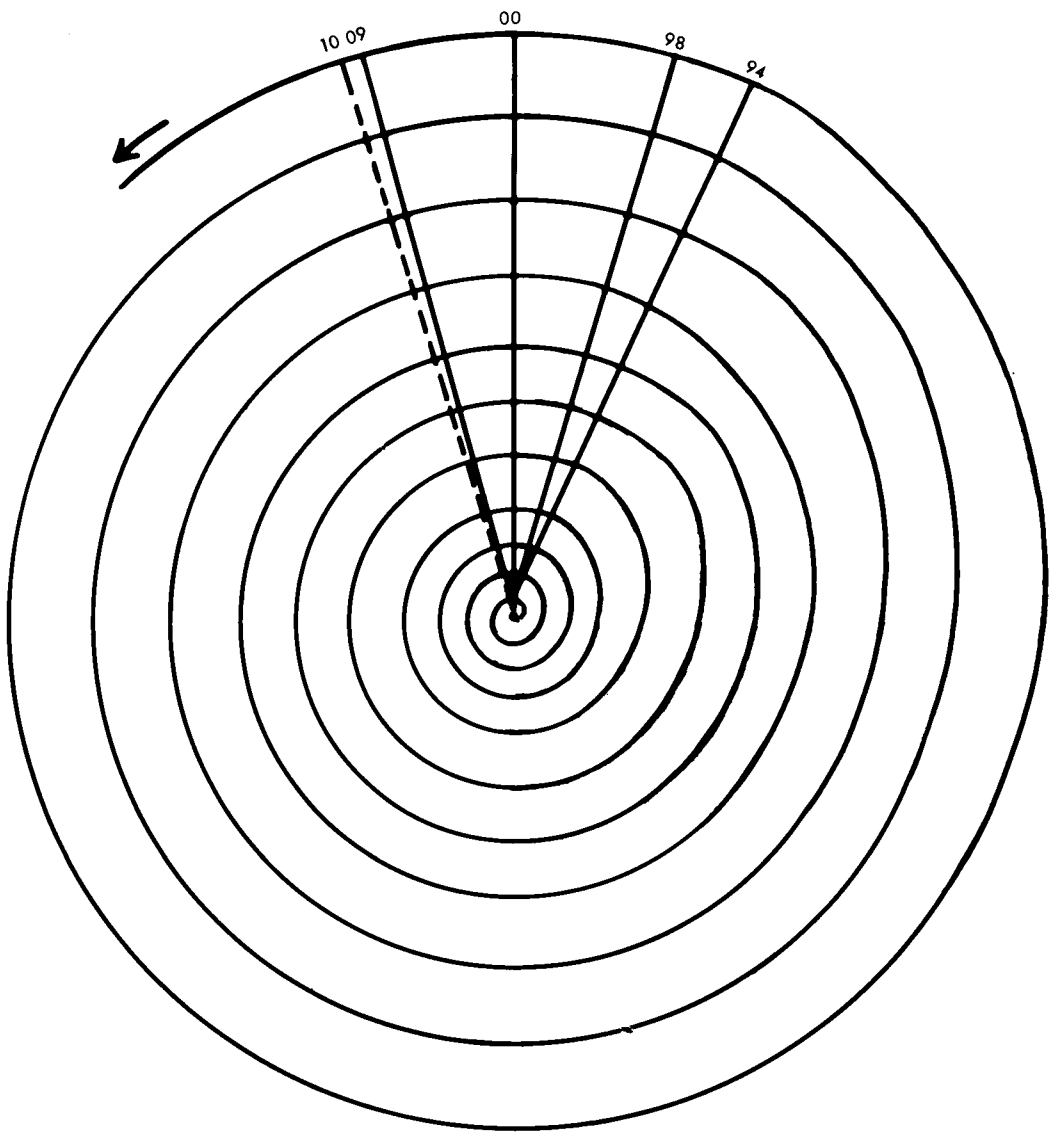
The word "mark" in the name of this command has special significance, other than just making the command sound complicated. In the case of a subroutine, be it one from the "standard package", or one that you write for yourself, there will come a time, after the subroutine has done its work, when you would like it to return to your own main program, in whatever line that might be (in this case, line 00), at a given word-time. Each of the subroutines in the standard package is equipped with a "return" command, which is similar in nature to a mark and transfer control command. In the return command, which is also a special command, D = 31, S = 20, and C = the line number to which control is to be returned. We will call this the "return line". The determination of the word-time at which the sequence in the return line will begin, however, is a bit different than in the preceding case.

When a mark and transfer control command is executed, a "mark" is generated electronically in the computer, at the word-time immediately preceding the word-time of execution. In other words, if the command is executed at word-time 10, the mark will be at word-time 09. If the return command is properly made up, it will cause the computer to sense this mark, after the return line has been selected, and the computer will take its next command, in the return line, from the next location. If the mark is at word-time 09, the next command will be taken in the return line from word-time 10. Notice that this was the time of execution of the original mark and transfer control command in that same line: it was not the word-time in which that command was located.

Thus, if the mark and transfer control command and an accompanying return command are properly made up, a transfer of control to a new line will be effected at the proper word-time in the new line, and, while still taking commands from the new line, the computer gets a command directing it to return to a return line (which usually will be the line from which control was originally transferred, in our case, line 00, although this does not have to be the case); it will return control to the command line specified, and take its next command at

a word-time corresponding to the word-time of execution of the original mark and transfer control command.

In the following drawing the spiral indicates the passage of time, in the direction shown by the arrow, which is outward. As the spiral passes the heavy vertical line, which represents word-time 00, a complete drum cycle has elapsed. Ten complete drum cycles are shown, in the center of the spiral a fraction of another cycle is shown, and, on the outside of the spiral, a fraction of still another drum cycle is shown.



Let us refer to that drum cycle of which only a fraction is shown in the center of the spiral as drum cycle 1. Then, counting outward, drum cycles 2 through 11 are completely shown, followed by a fraction of drum cycle 12.

During drum cycle 1, assume control of the computer is in command line 00. It remains there up through word-time 10 of cycle 2. Sometime before word-time 10 of cycle 2, a mark and transfer control command is read in line 00, calling for a transfer of control to line 01 at word-time 94. This command is executed at word-time 10 of drum cycle 2, shown in the drawing. Beginning with word-time 11 in drum cycle 2, control is in command line 01, and the computer is waiting for the next command, which it will read at word-time 94 of drum cycle 2. This command will come from word-time 94 in command line 01. An "electronic mark" was generated by the mark and transfer control command, at the word-time whose number is one less than that of the word-time of execution of the command itself.

If the command is executed, as we say, at word-time 10, this mark will be at word-time 09, as shown in the drawing. Such a mark will last indefinitely, being turned off, or erased, only by either the creation of another "mark" by a similar command, or turning off the computer. The square root subroutine, in line 01, is now operating, starting at word-time 94 of drum cycle 2. It continues for approximately 9 drum cycles (stated in the specifications), through word-time 99 of drum cycle 11, in the drawing. At word-time 98 in command line 01, which is finally reached in the last drum cycle of execution during the square root subroutine, a return command is located. This is specified in the write-up of the subroutine, and will be, for all subroutines. This command specifies the command line to which control is to be transferred (returned), in our case, line 00, and it enables the computer to sense the mark (currently coming up at the next word-time 09). Beginning at word-time u0 (100) in drum cycle 11, control has been returned to line 00, and the computer is looking for the next command. It will find the next command at either of two locations: the location specified by the N of the return command, or the marked location. Which of these will contain the next command the computer will read is determined by which arrives earlier. The early word gets control.

It therefore becomes the programmer's responsibility to see to it that the return command in the subroutine is timed in such a way that the marked location cannot be missed. He cannot place this return command in any location in the subroutine other than the one specified, which, in this case, is word-time 98. But he can set up the command so that the marked word-time will have to come up before the word-time specified by N in the return command.

We have set word-time 10 as the next command in our program, upon the return from the square root subroutine, for purposes of example. We want to be sure that word-time 09, which bears the mark, will come up before N of the return command. We could set N of the return command in the subroutine equal to 10. In that way, the location picked by the mark and the location picked by N of the return command would coincide, and there would be no doubt as to which word in line 00 would be the first to be interpreted as the next command.

But notice, if we could make up a general return command in such a way that we would always return to the marked word-time in the same return line, no matter what that marked word-time might be, we could then use

the same return command in the subroutine, no matter how many times in the course of the program we wanted to enter the subroutine. In our particular example, we only use the square root subroutine once, but it is not inconceivable that some programs would use it literally dozens of times. It is possible to make up a return command for any subroutine in such a way that it can be used over and over again, each time returning control to the same line, but at a different word-time, depending on where the mark is currently located. Remember it was said that a mark is erased by the setting of a new one. Only one mark may exist in the computer. This general return command is ideal, because now the place at which the main program picks up, after receiving control back from the subroutine can, in each case, be picked through the setting of the mark and transfer control command which transfers to the subroutine. One time we could set the T of the mark and transfer control command equal to 10, the next time, to 90, and so on. The main program will pick up at word-time 10 after the first use of the subroutine, at word-time 90 after the second use of the subroutine, and so on.

Such a general return command is made up in the following manner, for the reasons indicated. Make the command immediate, and let it be executed for one word-time. This can be done by setting T equal to the location plus two. In this case, the immediate command will be read at word-time L (location), and the immediate execution will begin in the next word-time, L + 1. The T number will act as a flag, as mentioned previously in the discussion of immediate commands. Since T = L + 2, the operation will be stopped after word-time L + 1, and therefore will last only one word-time. During this word-time, the computer will begin to search for the existing mark. The mark, when found, will be rejected unless it is in the last word-time of execution of the return command or later. Since there is only one word-time of execution of this command when coded in this form, it is also the last. Therefore, the search for a mark will begin at L + 1 and continue until the mark is found, and the next command in the return command line will be specified by the location of the mark. Let N = L + 1 in the return command; it cannot specify the next command in the return command line, because N = L + 1 and cannot be effective for one whole drum cycle. The mark must be found and become effective at some time during the drum cycle; the worst case would be the one in which the mark is at L, determining L + 1 (= N) as the location of the next command. (A detailed description of the occurrence of machine signals in this regard follows: do not attempt to master it on the first reading of this text.)

Drawing 1 shows pictorially what will be the effect of a mark when the return command is coded properly: T = L + 2, N = L + 1. The return command is located at word-time 12 and executes during word-time 13, as shown on the time-spiral. Because word-time 13 is also the last word-time of execution a mark sensed at that time will be effective.

The first possible location of the next command in the return command line, therefore, is 14, as indicated by the X in the drawing. If the mark is not found at 13, the search will continue until it is found, and in the drawing this is seen to be word-time 37. The next command therefore, in the return command line, will be taken at 38. The worst case