High-Speed Arithmetic in Binary Computers
O. L. MacSorley / TR 00.740

IBM

Technical publication

September 14, 1960 TR 00.740

HIGH-SPEED ARITHMETIC IN BINARY COMPUTERS

O. L. Mac Sorley

ABSTRACT

Methods of obtaining high speed in addition, multiplication, and division in parallel binary computers are described and then compared with each other as to efficiency of operation and cost. The transit time of a logical unit is used as a time base in comparing the operating speeds of different methods, and the number of individual logical units required is used in the comparison of costs. The methods described are logical and mathematical, and may be used with various types of circuits. The viewpoint is primarily that of the systems designer, and examples are included wherever doing so clarifies the application of any of these methods to a computer. Specific circuit types are assumed in the examples.

Product Development Laboratory, Data Systems Division
International Business Machines Corporation, Poughkeepsie, New York

CONTENTS

HIGH-SPEED ARITHMETIC IN BINARY COMPUTERS

by

O. L. Mac Sorley

INTRODUCTION

The purpose of this report is to describe various methods of increasing the speed of performing the basic arithmetic operations in such a manner that one method may be readily compared with another, as to both relative operating efficiency and relative equipment cost. It is divided into three parts: Adders, Multiplication, and Division.

Adders

As it is generally recognized that most of the time required by adders is due to carry propagation time, this section deals with methods of reducing this time, together with their efficiency and relative costs. It considers adders both from the standpoint of reducing the length of the carry path when using a fixed-time adder and of recognizing the completion of an addition to take advantage of the short length of an average carry. Circuits shown are in terms of basic logic blocks, and use the transit time of a logical block as a unit to permit the application of conclusions to various types of circuits.

Multiplication

In multiplication, if one addition is performed for each one in the multiplier, the average multiplication would require half as many additions as there are bits in the multiplier. This can be improved considerably by the use of both addition and subtraction of the multiplicand. The rules for determining when to add and subtract are developed, and the method of determining the number of operations to expect from the bit grouping is explained. This results in a variable number of add cycles for fixed-length multipliers. For some applications a fixed number of cycles is preferable. To accomodate this requirement, rules are developed for handling two-and three-bit multiplier groupings.

Multiplication, which involves repeated additions in which the selection of the various addends is not affected by a previous sum, offers the possibility of improved speed by the use of carry-save adders. Condictions under which such improvements will be realized are investigated, and methods that may be used to reduce the amount of equipment required are described.

Division

Working from the premise that a division should require no more additions than would be required if the resulting quotient were used as the multiplier in a multiplication, the development of such a method is traced through several stages. Then another and still faster method is also described. Methods of evaluating the speeds of these various methods are developed in such a manner as to also permit evaluation of the effects of variation in maximum shifter size.

General

For the purpose of illustrating points in the use of these various arithmetical methods which may affect their application to computers, several typical systems circuits are shown, and the use of these is assumed in the numerical examples included. The following is a brief description of the circuits that are assumed available and a definition of terms that will be used.

DC rather than pulse type logic is assumed. Registers, or data storage devices, are assumed to be separate from the adder. The use of a separate shifter rather than a shifting register is assumed. Most registers used are "latch-registers"; this means a register capable of being set from data lines which are in turn controlled by the output of the same register upon the application of a latch-control signal. A gate is a group of two input AND circuits, each having one of its two inputs connected to a common line, and the other input to a data input line. A shifter is a device for transferring all bits in a register a specified number of positions left or right. The term "addition" will be used to include both addition and subtraction, and the same adder will be used for both. Subtraction will always be performed by the use of the two'scomplement of the number to be subtracted from the other. This will be obtained by inverting all bits in the number and also forcing an additional one into the carry position of the low order bit position of the adder when performing the addition.

Logical circuits are shown with inputs on the left and outputs on the right. The bottom output position represents the logical function described in the box, while the top output position represents its inverse. The logical symbols used within the boxes are AND (&), INCLUSIVE OR ($\vee$), and EXCLUSIVE OR ($\not\vee$). When the word OR is used alone, it,means INCLUSIVE OR.

Unless otherwise specified, arithmetic used in examples is assumed to be binary floating point, although the methods described are not limited in their use to this type of arithmetic. When a number is described as normalized, it means that the fraction has been shifted in the register until the high order one in the fraction is located just to the right of the binary point, and the exponent has been adjusted accordingly. Thus a normalized fraction will always have a value less than one and equal to or greater than one-half. In the examples exponent handling is implied, but not described in detail.

BINARY ADDERS

Binary Adders, Fixed Time

The basic binary adder is comparatively simple and quite well-known. It is also comparatively slow. Figure 1 shows one version of one stage of such an adder.

In the discussion of adders the lowest order bit or adder position will be designated as one. The two multi-bit numbers being added together will be designated as A and B, with individual bits being A_1, A_2, B_1, etc. The third input will be C. Outputs will be S (Sum) R (Carry), and T (Transmit).

The conventional ripple-carry adder consists of a number of stages like that shown in Figure 1, connected in series, with the R output of one stage being the C input of the next. The time required to perform an addition in such an adder is the time required for a carry originating in the first stage to ripple through all intervening stages to the S or R output of the final stage. Using the transit time of a logical block as a unit of time, this amounts to two levels to generate the carry in the first stage, plus two levels per stage for transit through each intervening stage, plus two levels to form the sum in the final stage, which gives a total of two times the number of stages.

The usual forms of the logical description of the sum and carry from the nth stage of an adder are $S_n = (A_n \veebar B_n \veebar C_n)$ and $R_n = (A_n B_n \vee A_n C_n \vee B_n C_n)$. Also, from the description of connection between sections, $C_n = R_{n-1}$. If the carry description is rearranged to read $R_n = (A_n \not\vee B_n)\, C_n \vee A_n B_n$, and if T_n is defined as $(A_n \not\vee B_n)$ and D_n is defined as $(A_n B_n)$, then $R_n = D_n \vee T_n C_n$. This separates the carry out of a particular stage into two parts, that produced internally and that produced externally and passed through. The former is called a generated carry and the latter is called a propagated carry. From this the description of the carry into any stage may be expanded as follows.

$$C_n = R_{n-1}$$

$$C_n = D_{n-1} \vee T_{n-1} R_{n-2}$$

$$C_n = D_{n-1} \vee T_{n-1} D_{n-2} \vee T_{n-1} T_{n-2} R_{n-3}$$

$$C_n = D_{n-1} \vee T_{n-1} D_{n-2} \vee T_{n-1}\, T_{n-2}\, D_{n-3} \vee T_{n-1} T_{n-2} T_{n-3} R_{n-4}$$

This can be continued as far as is desired.

Figure 2 illustrates the application of this principle to a section of a carry propagate adder to increase its speed of operation. By allowing n to have successive values starting with one and omitting all terms containing a resulting negative subscript, it may be seen that each stage of the adder will require one OR stage with n inputs and n AND circuits having one through n inputs, where n is the position number of the particular stage under consideration.

It is obvious that circuit limitations will put an upper limit on the number of stages of an adder than can be connected together in this manner. However, within this limit the maximum carry path between any two stages is two levels, or six levels for the complete addition.

Assume that five stages represent a reasonable number of adder stages to be connected in this manner and designate such an arrangement as a "group". The group containing the five low-order positions of the adder will be group 1, etc. A carry into group n will be C_{gn}, while a carry out of the group will be

R_{gn}. If these five-bit groups are now connected in series with $C_{gn} = R_{g(n-1)}$, a carry will require four levels to be produced and reach the output of the first group, two levels to go through each intermediate group, and four levels to reach and be assimilated into the sum in the final group. Thus, for five-bit groups, the maximum carry path length would be $4 + (2n/5)$ as compared to $2n$ for a straight ripple-carry adder. For a 50-bit adder this would give 24 levels as compared to 100.

Since each five-bit group may be considered as one stage in a radix-32 adder, a transmit signal may be generated to take a carry across the group. This will be designated as T_{gn}, and will be defined as $T_g = T_1T_2T_3T_4T_5$, where the numbers 1, 2, etc., refer to positions within the group rather than within the adder. At the same time D_{gn}, which includes only carries originating within the group, may replace R_{gn}, which includes the effect of C_{gn}, whenever a higher level of look-ahead than the one under consideration is being used with it. The use of R_{gn} where D_{gn} is called for will not produce an error, but will add unnecessary components.

This process may be continued by designating five groups as a section and then using carry speed-up circuits between the sections. Carries into a section will be C_{sn} and carries out of a section will be D_{sn}. (If the third level of carry look-ahead is not used, R_{sn} must be used in place of D_{sn}.) The maximum path length for a carry to be generated within a section and reach the output D_{sn} is six levels. The maximum path length for a carry appearing at the input to a section as C_{sn} to affect the sum is also six levels. The maximum path length for a carry originating within a section to affect a sum within the same section is ten levels.

Carry look-ahead between bits within a group is called level one look-ahead, between groups within a section is called level two, and between sections is called level three. The following table gives a comparison of speed improvement for different amounts of look-ahead. Five bits to the group and five groups to the section are assumed. The time units are logical level transit times.

Look-ahead Levels →	0	1	1 & 2	1, 2, & 3
Adder Bits				
5	10	6	-	-
10	20	8	-	-
25	50	14	10	-
50	100	24	12	-
100	200	44	16	14

The transmit signal has been described as the EXCLUSIVE OR combination of A and B. Correct operation will also be obtained if the INCLUSIVE OR is used instead of or in combination with the EXCLUSIVE OR. The only effect will be a redundant signal at times.

Figures 2 and 3 together illustrate a 100-bit adder with full carry look-ahead. In Figure 2, part 1 shows the details fo the basic sum generation unit, while part 2 shows the basic carry look-ahead unit. Figure 3 shows the method of combining the parts to give the complete adder. The complete circuit shown on Figure 2 represents one group on Figure 3.

Various modifications may be made to the circuit shown in Figure 3 if smaller size or less than maximum speed is required. Some of the possibilities which are likely to be of particular use to the computer designer are listed below, and their relative speeds and costs will be included in the comparison table. Some minor variations which these modifications may cause andwhichwould be obvious to anyone considering the problem will not be described in detail. Comparisons will be made on the basis of 50-bit and 100-bit adders.

(1) Eliminate the look-ahead within groups, but retain it between groups and between sections.

(2) Retain the look-ahead within groups, but use ripple carry between groups.

(3) Use the very elementary carry speed-up circuit used with the Completion Recognition adder (Figure 4). This can be used with any adder, and will give almost a four-to-one increase in speed over that of a full ripple-carry adder of 100 bits for only about 2.5% increase in equipment. It provides a carry bypass circuit within rather than around the group. Its principal merit is the high percentage improvement per unit increase in cost.

The following table summarizes the comparative costs and speeds for five different adder versions for 50-bit and 100-bit adders. The 50-bit ripple-carry adder is usedas a reference for cost comparison. The types being compared are (1) full ripple carry, (2) full carry look-ahead, (3) ripple carry within five-bit groups, look-ahead between groups, (4) look-ahead within five-bit groups, ripple carry between groups, (5) carry bypass within five-bit groups, ripple carry between groups.

Adder Type	50-Bit Adder Logical Units	Comp. Cost	Time	100-Bit Adder Logical Units	Comp. Cost	Time
1	400	100. 0	100	800	200. 0	200
2	636	159. 0	12	1294	323. 4	14
3	466	116. 5	24	954	238. 4	26
4	580	145. 0	24	1160	290. 0	44
5	410	102. 5	36	820	205. 0	52

Binary Adders, Variable Time

It can be shown that for a large number of binary additions the average length of the longest carry of each addition will not be greater than $\log_2 N$, where N is the number of bits in the numbers being added together. Random distribution of bits within the numbers is assumed. This gives an average maximum carry length of not greater than 5. 6 for a 50-bit sum or 6.. 6 for a 100-bit sum.

In a ripple-carry adder a six-position carry would represent twelve units of time, as compared to fourteen units maximum for a 100-bit adder with full look-ahead. Also, the twelve units represent actual transit time, while the fourteen units represent predicted time with safety factor. In addition, the carry look-ahead adder represents 60% more equipment thanthe basic ripple-carry adder.

The variable time (completion recognition) adder must contain additional equipment that will permit the recognition of the completion of carry propagation. Ideally this equipment should have three characteristics. It should be inexpensive. It should not add to the time needed to complete the addition.

It should not indicate completion, even momentarily, when an addition is still incomplete, and if an input changes after an addition has been completed, the completion signal should immediately go off and remain off until the new result is completed.

Figure 4 illustrates one version of a completion recognition adder. While it does not meet all of the requirements of an ideal unit, it does appear to be reliable when used with the proper restrictions. This adder requires approximately 1280 logical units for 100 bits, which is essentially the same as the 1294 units for the full carry look-ahead adder. Thus costwise they may be considered the same. However, part of the additional equipment required for the carry-recognition circuits may also be used as part of the checking circuitry. To obtain equivalent checking with the carry look-ahead adder would require considerable additional equipment.

Each stage of the adder generates a carry and a no-carry signal, and these are propagated through the adder along separate paths. If these signals are designated as C and N, completion of the addition is recognized by the existence of the condition [(C OR N) and not (C AND N)] at the output of every bit position in the adder.

The operation of this adder will be more readily understood if it is recognized that $C_n = A_n B_n \mathbf{V} T_n C_{n-1}$ and that $N_n = \bar{A}_n \bar{B}_n \mathbf{V} T_n N_{n-1}$. At the start of an addition the inputs to the adder must be cleared. This sets the N output of each block to <u>one</u> and the C output to <u>zero</u>. The desired inputs are then entered, which changes the N outputs to <u>zero</u> for those positions which have a <u>one</u> in either or both inputs. This turns off the completion signal. The C output is changed to one for those positions having an input of 11 and the T signal is changed to one for those positions having 01 or 10. The latter positions have zero on both the C and N lines. Signals will then ripple down either the C or N lines from positions having either 00 or 11 inputs until all positions have either the C or the N output energized, at which time a completion signal will be generated. To prevent false indications of completion, the two inputs must enter the adder simultaneously; once the operation has started, no changes may be made in the inputs; and both inputs must be changed to zero before the next addition may be performed. An alternative to this is to force ones into all input positions by using an additional input to the OR circuits that are usually present at the input to adders. The restriction here would be that the correct inputs are present at the input to the OR circuits at the time the forcing inputs are turned off.

No general statement can be made as to whether fixed-time or variable-time adders are better. The use of a completion recognition adder offers many attractions to the systems designer, particularly if his circuits have a large spread between average and maximum transit time. On the other hand, the limitations on data handling required to prevent ambiguities in the control signals may nullify some or all of the theoretical advantages. The best choice can only be made by a careful consideration of all of the factors involved for the particular application.

BINARY MULTIPLICATION

Multiplication Using Variable Length Shift

Multiplication in a computer is usually performed by repetitive addition. For constant circuit and adder speeds, the time required to perform a multiplication is proportional to the number of additions required. The slowest way would be to go through one add cycle for each bit of the multiplier. Substituting shift cycles for add cycles when the multiplier bit is a zero can reduce this time; providing the ability to shift across more than one position at a time when there are several zeros in a group can reduce the time still further. Assuming random distribution with equal numbers of ones and zeros in the multiplier, this should result in a 50% reduction in time. This is as much improvement as is obvious from normal methods of performing multiplication.

Further improvements may be secured by taking advantage of some of the properties of the binary system. The rules for handling multiplication to obtain this improvement will be developed.

A binary integer may be written in the following form:

$$A_n 2^n + A_{n-1} 2^{n-1} + A_{n-2} 2^{n-2} + \text{------} + A_2 2^2 + A_1 2^1 + A_0 2^0$$

The actual number, as written, consists of the characteristics only and would be written $A_n A_{n-1} A_{n-2} \text{---} A_2 A_1 A_0$, where each A would have a value of either one or zero. If such a number contained the coefficients ---0111111110---, this part of the number would have the value $2^{n-1} + 2^{n-2} + \text{-----} + 2^{n-x}$, where n is the position number of the highest order one in the group for which the lowest order position in the number is designated zero, and x is the number of successive ones in the group. The numerical value of this last expression may also be obtained from the expression $2^n - 2^{n-x}$, where n and x have the same values as before. For example, in the binary number 0111100, n is 6 and x is 4. The decimal equivalent of the number is given by $2^5 + 2^4 + 2^3 + 2^2 = 32 + 16 + 8 + 4 = 60$. It is also given $2^6 - 2^2 = 64 - 4 = 60$. Thus for any string of ones in a multiplier, the necessity for one addition for each bit can be replaced by one addition and one subtraction for each group. The only additional equipment required is a means of complementing the multiplicand to permit subtracting and, of course, some additional control equipment. To illustrate this a typical multiplier is shown below with the required operations indicated. Each group of ones is underlined.

$$\begin{array}{*{26}{c}}
 & \underline{1} & \underline{1} & \underline{1} & \underline{1} & 0 & 0 & 0 & 0 & \underline{1} & \underline{1} & \underline{1} & 0 & \underline{1} & \underline{1} & \underline{1} & 0 & \underline{1} & 0 & \underline{1} & 0 & 0 & 0 & \underline{1} & 0 & \underline{1} \\
+ & & & & - & & & & + & & & - & + & & & - & + & - & + & - & & & + & - & + & -
\end{array}$$

Additional improvement may be obtained by using the fact that $+2^n - 2^{n-1} = +2^{n-1}$ and $-2^n + 2^{n-1} = -2^{n-1}$. This is illustrated by applying it to the above example. The original results are given first, with the operations to be combined underlined.

$$\begin{array}{*{26}{c}}
 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
+ & & & & - & & & & + & & & \multicolumn{2}{c}{\underline{-\;+}} & & & \multicolumn{2}{c}{\underline{-\;+}} & \multicolumn{2}{c}{\underline{-\;+}} & - & & & \multicolumn{2}{c}{\underline{+\;-}} & \multicolumn{2}{c}{\underline{+\;-}} \\
+ & & & & - & & & & + & & & & - & & & & - & & - & - & & & & + & & + \\
 & & & & & & & & & & & & & & & & & & & & & & & & & \\
+ & & & & - & & & & + & & & \multicolumn{2}{c}{\underline{-\;+}} & & & - & \multicolumn{2}{c}{\underline{+\;-}} & \multicolumn{2}{c}{\underline{+\;-}} & & & \multicolumn{2}{c}{\underline{+\;-}} & \multicolumn{2}{c}{\underline{+\;-}} \\
+ & & & & - & & & & + & & & & - & & & - & & + & & + & & & & + & & +
\end{array}$$

Two different arrangements are shown. Both will give the correct result, and the number of cycles required is the same. The first is that obtained by starting at the high order end, and the second by starting at the low order end.

For a given multiplier, the number of additions that will be required may be computed as follows. Define a group of ones as a series of bits containing not more than a single zero between any pair of ones within the series, containing at least one pair of adjacent ones , and starting and ending with a one.

Then the number of add cycles is equal to the following: Two times the number of groups, plus the number of zeros contained within groups, plus the number of ones not contained within groups. This may be illustrated with the previous example.

1 1 1 1 0 0 0 0 1 1 1 0 1 1 1 0 1 0 1 0 0 0 1 0 1

There are two groups. The first group contains no zeros, the second contains three. There are two ones not contained in any groups. This gives (2 x 2) + 3 + 2 = 9, which is the number **of operations** that was obtained. Within the limitation of using only multiples of the multiplicand that can be obtained directly by shifting and using only one of these at a time, it is believed that this represents the least number of additions with which a binary multiplication can be performed.

The rules for performing a multiplication may now be given. It is assumed that the multiplier and the partial product will always be shifted the same amount and at the same time. The multiplier is shifted in relation to the decoder, and the partial product with relation to the multiplicand. Operation is assumed starting at the low order end of the multiplier, which means that shifting is to the right. If the lowest order bit of the multiplier is a one, it is treated as though it had been approached by shifting across zeros.

(1) When shifting across zeros (from low order end of multiplier), stop at the first one.

(a) If this one is followed immediately by a zero, add the multiplicand, then shift across all following zeros.

(b) If this one is followed immediately by a second one, subtract the multiplicand, then shift across all following ones.

(2) When shifting across ones (from low order end of multiplier), stop at the first zero.

(a) If this zero is followed immediately by a one, subtract the multiplicand, then shift across all following ones.

(b) If this zero is followed immediately by a second zero, add the multiplicand, then shift across all following zeros .

A shift counter or some equivalent device must be provided to keep track of the number of shifts and recognize the completion of the multiplication.

If the high order bit of the multiplier is a one and is approached by shifting across ones, that shift will be to the first zero beyond the end of the multiplier, and that zero along with the bit in the next higher order position of the register will be decoded to determine whether to add or subtract. For this reason, if the multiplier is initially located in the part of the register in which the product is to be developed, it should be so placed that there will be at least two blank positions between the locations of the low order bit of the partial product and the high order bit of the multiplier. Otherwise the low order bit of the product will be decoded as part of the multiplier. An alternative to this is for the fact that the shift counter indicates the end of the multiplication to force the last operation to be an addition.

It should be noted that whenever the shifting is across groups of ones the partial product will be in complement form, which means that the shifter must contain provision for inserting ones in all high order positions that would normally be left blank by the shifting.

If the multiplication is performed starting from the high order end of the multiplier, the partial product will always be in true form,but any operation may result in a carry traveling the full length of the partial product. The shifting rules are a little more complicated, as may be seen below.

(1) When shifting across zeros (from high order end of multiplier)

- (a) If the first one following the zeros is followed immediately by a second one, stop shifting at the last zero and add the multiplicand, then shift across following ones.
- (b) If the first one following the zeros is followed immediately by a zero, stop shifting at the first one and add the multiplicand, then shift across following zeros.

(2) When shifting across ones (from high order end of multiplier)

- (a) If the first zero following the ones is followed immediately by a second zero, stop shifting at the last one and subtract the multiplicand; then shift across the following zeros.
- (b) If the first zero following the ones is followed immediately by a one , stop shifting at the first zero and subtract the multiplicand, then shift across the following ones.

The high-order one of the multiplier is treated as though there were at least two zeros immediately preceding it.

As was previously stated, there two methods of decoding the multiplier will yield the same number of add cycles. This number is dependent on the number and distribution of ones within the multiplier. If random distribution is assumed, it can be shown that the average shift for each addition will be 3.0 bit positions when using an infinite shifter, or 2.9 bit positions for a shifter having a limit of six.

Multiplication Using Uniform Shifts

For some applications a method of multiplication which uses shifts of uniform size and permits predicting the number of cycles that will be required from the size of the multiplier is preferable to a method that requires varying sizes of shifts. The most important use of this method is in the application of carry-save adders to multiplication, although it can also be used for other applications. The use of carry-save adders will be discussed in a later section.

Two methods will be described. The first requires shifting the multiplier and partial product in steps of two, the second in steps of three. Both methods require the ability to shift the position of entry of the multiplicand into the adder in relation to its normal position. The latter is designated as the one-times-multiplicand position and used as a reference position in all descriptions. This small shifter will be the length of the multiplicand rather than of the partial product. Both methods may be used starting from either end of the multiplier, but because of the reduced requirements on the size of the adder, are usually used starting from the low-order end. The latter will be assumed for any operating descriptions, but for easier explanation the rules of operation will be developed assuming a start from the high order end.

Uniform Shifts of Two

Assume that the multiplier is divided into two bit groups, an extra zero being added to the high order end if necessary to produce an even number of bits. Only one addition or subtraction will be made for each group, and, using the position of the low order bit in the group as a reference, this addition or subtraction will consist of either two times or four times the multiplicand. These multiplies may be obtained by shifting the position of entry of the multiplicand into the adder one or two positions left from the reference position. The last cycle of the multiplication may require special handling. Rules for this will be considered after the general rules have been developed.

The general rule is that, following any addition or subtraction, the resulting partial product will be either correct or larger than it should be by an amount equal to one times the multiplicand. Thus, if the high order pair of bits of the multiplier is 00 or 10, the multiplicand would be multiplied by zero or two and added, which gives a correct partial product. If the high order pair of bits is 01 or 11, the multiplicand is multiplied by two or four, not one or three, and added. This gives a partial product that is larger than it should be, and the next add cycle must correct for this.

Following the addition the partial product is shifted left two positions. This multiplies it by four, which means that it is now larger than it should be by four times the multiplicand. This may be corrected during the next addition by subtracting the difference between four and the desired shift.

Thus, if a pair ends in zero, the resulting partial product will be correct and the following operation will be an addition. If a pair ends in a one, the resulting partial product will be too large, and the following operation will be a subtraction.

It can now be seen that the operation to be performed for any pair of bits of the multiplier may be determined by examining that pair of bits plus the low order bit of the next higher order pair. If the bit of the higher order pair is a zero, an addition will result; if it is one, a subtraction will result. If the low order bit of a pair is considered to have a value of one and the high order bit a value of two, then the multiple called for by a pair is the numerical value of the pair if that value is even and one greater if it is odd. If the operation is an addition, this multiple of the multiplicand is used. If the operation is a subtraction (the low order bit of the next higher order pair a one), this value is combined with minus four to determine the correct multiple to use. The result will be zero or negative, with a negative result meaning subtract instead of add. The following table summarizes these results.

Multiplier	Operation	Multiplier	Operation
0 - 0 0	+ 0	1 - 0 0	- 4 + 0 = - 4
0 - 0 1	+ 2	1 - 0 1	- 4 + 2 = - 2
0 - 1 0	+ 2	1 - 1 0	- 4 + 2 = - 2
0 - 1 1	+ 4	1 - 1 1	- 4 + 4 = - 0

It is obvious from the method of decoding described that the multiplier may be scanned in either direction. When starting from the high-order end, the partial product will always be in true form, but starting from the low order end will result in a complement partial-product part of the time. This means that the main shifter must be designed to handle the shifting of complement numbers.

The possibility that the low-order bit of the multiplier will be a one presents a special problem. For operations starting at the high order end of the multiplier this may be handled in either of two ways. One requires an additional cycle only when the low-order bit is a one, and consists of adding the complement of one-times the multiplicand following a zero shift after the completion of the last regular operation. The other method adds an additional add cycle to every multiplication by always treating the multiplier as though it had two additional low-order zeros. The two extra zeros which this introduces into the product are then ignored.

When operating from the low order end of the multiplier this problem may be handled more easily. On the first cycle there is no previous partial product. Therefore zeros are being entered into one side of the adder. If the low order bit of the multiplier is a one, enter the complement of one times the multiplicand into the adder by way of the input usually used for the partial product. At the same time, the multiple of the multiplicand selected by decoding the first pair of bits of the multiplier is entered at the other adder input. This does not require any additional cycles.

Uniform Shifts of Three

This method of handling three bits of the multiplier at a time requires being able to obtain two, four, six, or eight times the multiplicand. One times may also be required to handle the condition of a one in the low order bit position of the multiplier. One, two, four, and eight times can all be obtained by proper positioning of the multiplicand, but the six times must be generated in some manner. This can be done by adding one times the multiplicand to two times the multiplicand, shifting the result one position, and storing it in a register.

The development of the decoding rules for this method follows the same basic requirements already described for handling two bit groups. This is evident from the table given below, and will not be repeated.

Multiplier	Operation	Multiplier	Operation
0 - 0 0 0	+ 0	1 - 0 0 0	- 8 + 0 = - 8
0 - 0 0 1	+ 2	1 - 0 0 1	- 8 + 2 = - 6
0 - 0 1 0	+ 2	1 - 0 1 0	- 8 + 2 = - 6
0 - 0 1 1	+ 4	1 - 0 1 1	- 8 + 4 = - 4

(Continued)

Multiplier	Operation	Multiplier	Operation
0 - 1 0 0	+ 4	1 - 1 0 0	- 8 + 4 = - 4
0 - 1 0 1	+ 6	1 - 1 0 1	- 8 + 6 = - 2
0 - 1 1 0	+ 6	1 - 1 1 0	- 8 + 6 = - 2
0 - 1 1 1	+ 8	1 0 1 1 1	- 8 + 8 = - 0

There are some general facts that apply to both the two-shift and the three-shift methods of multiplication.

(1) The choice of true or complement entry of the multiplicand into the adder is dependent only on the condition of the low-order bit of the next higher order group of the multiplier.

(2) Special provision must be made for the condition of a one in the low order bit position of the multiplier. Procedure is the same for both methods.

(3) Whenever complement inputs are used for multiplicand multiples, there must also be provision for entering a low order one into the adder to change the one's complement to a two's complement. This includes the complement of one-times the multiplicand used because of a low order multiplier one. This can result in a design problem, since odd numbers in the two low order groups of the multiplier may call for the entry of two additional ones into the low order position of the adder, making a total of four entries. A solution to this is to decode the low-order group of the multiplier to call for the desired multiple or one less instead of one more. Then the true value of one times the multiplicand can be used in the partial product position on the first cycle when the multiplier has a low order one. This may be done very easily, on the first cycle only, by forcing the low-order bit of the group to enter the decoder as a zero, but using its actual value to determine whether or not to add one-times

the multiplicand. The justification for this may be seen from either table. This modification of the decoding will not work for any cycle except the first, and only when operating from the low order end of themultiplier.

To permit a comparison, the illustrative multiplier used previously to show decoding for the variable shift method will be shown below for variable shift, two-position shifts, and three-position shifts.

```
(1)     0 0 1 1 1 1 0 0 0 0 1 1 1 0 1 1 1 0 1 0 1 0 0 0 1 0 1
          +       -       +       -     -   +   +       +   +

          +2  -0   -2 +0  +2  -0  -2 -0  -2  -2  -4 +2  -4+
(2)     0'0 1'1 1'1 0'0 0'0 1'11'0 1'1 1'0 1'0 1'00'0 1'0 1'
          +       -       +      -        -   - -   + -   +

          +2     -0    -8   +4    -2     -2    +6    -8   +4+
(3)    '0 0 1'1 1 1'0 0 0'0 1 1'1 0 1'1 1 0'1 0 1'0 0 0'1 0 1'
          +       -       +       -     -  + + -        +   +
                                           ---
```

All decoding shown is based on starting at the low order end of the multiplier. Multiplier groupings are indicated in (2) and (3). The use of multiples of four in (2) and of eight in (3) places the effective location of the operation under the low order bit of the next higher group. An underline under a pair of operations in (3) indicates the use of the previously prepared three-times multiple. The (+) following the multiple figure for the low order group indicates that one-times the multiplicand is also used in the partial product entry position. The decoding for this particular group is assumed modified as previously described.

Multiplication Using Carry-Save Adders

Figure 5 shows a brief outline of a system capable of performing multiplication in the manner just described. At the start of the operation the multiplier is entered in the right half of the MQ register, the multiplicand into the MD register, one more than the multiplier size into the shift counter register, and two into the shift control register, and also the "use" trigger is set OFF. (It is assumed that the multiplier is initially entered into the same position of the MQ register as the low order end of a double precision number would be, which would place its high order bit immediately adjacent to the low order position of the partial product. The initial shift of two separates these by two bit positions, the necessity for which was previously described. The initial shift counter register setting is adjusted for this. The decoder is located to give correct operation with this offset.)

Since the "use" trigger is OFF and the partial-product in the MQ register is also zero, the output of the main adder will be zero. The two in the shift-control register causes two to be subtracted from the contents of the shift counter register in the shift counter adder. The low-order end of the shifted multiplier goes into the decoder and is decoded to give the next shift required and to determine whether the next operation will be add-true, add-complement, or neither (if shift called for is larger than shifter can give.) When sufficient time has been allowed for these operations to be completed, a latch control signal sets the results into the proper registers, and the next cycle starts. These cycles are repeated as many times as required, the shift called for as a result of decoding being compared each time with the contents of the shift counter register to determine when sufficient cycles have been taken.

To determine the time required for a cycle, three data paths must be considered and the longest used. They all include time to power the latch control signal and set information into the proper trigger, plus any safety factor that must be allowed because of variation in transit times. One path is from the MQ register, through the shifter to the decoder, through the decoder to the shift control register or to the multiplicand true-complement control trigger. A second path is from the shift control register or shift counter register through the shift counter adder and back to the shift counter register. The third path is from the MQ register through the shifter to the main adder, and through the main adder back t the MQ register. It will be assumed initially that the third path is the longest.

It has already been shown that most of the time required in a adder is required for propagation of carries, and various methods have been described for reducing this. The most efficient of these reduced the time to 12 transit time units for a 50-bit adder for a component increase of 59%. Four of the 12 units are due to the basic adder, and 8 are due to carry propagation.

When successive additions are required before the final answer is obtained, it is possible to delay the carry propagation beyond one stage until the completion of all of the additions, then let one carry propagate cycle suffice for all the additions. Adders used in this manner are called carry-save adders.

A carry-save adder consists of a number of stages, each similar to the full adder shown in Figure 1. It differs from the ripple-carry adder in that the carry (R) output is not connected directly to the next higher order stage of the same adder, but goes to an intermediate register or other device in the same manner as the sum (S) output. Thus a carry-save adder has three inputs which, as far as use is concerned, may be considered **identical**, and two outputs which are not identical and must be treated in different manners.

The procedure for adding several binary numbers by using a carry-save adder would be as follows. Designate the inputs for the n'th bit as A_n, B_n, and C_n, and the outputs for the same bit as S_n and R_n, where S_n is the sum output and R_n is the carry output. In the first cycle enter three of the input numbers into A, B, and C. In the second cycle enter the S and R obtained from the previous cycle into A and B and the fourth input number into C. In this operation S_n goes into A_n, but R_n goes into B_{n+1}, where B_{n+1} is in the next higher order bit position than B_n: This is in accordance with the customary rule for addition that a carry resulting from adding one column of figures is added into the next higher order column. The third cycle is the same as the second, etc. This is continued until all of the input numbers have been entered into the adder.

Carry propagation may be performed in either of two ways. Since each add cycle advances all carries one position, add cycles as already described may be continued with zeros being entered into the third input each time until the R outputs of all stages become zero. The alternative is to enter S and R into a carry-propagate adder and allow time for one cycle through it. This carry-propagate adder may be completely separate from the carry-save unit with a control line for selecting either carry-save or carry-propagate operation.

Before carry-save adders can be used in the multiplication loop, it is necessary to know the answers to these questions: (1) How should they be used? (2) How much additional equipment is required? (3) How much time will be saved? Assume that the circuit shown in Figure 5 is modified by changing the adder to a CP/CS adder which is so designed that the ability to operate as either a carry-save or a carry-propagate adder does not cause it to be any slower when operating in the carry-propagate mode than is a comparable adder without this feature. Such an adder can be constructed at an additional component cost of about 50% of the number of components in the corresponding ripple-carry adder. Also, since the partial product will now become a partial sum and a partial carry, and since the latch-register and shifter presently shown can only handle one of them, a duplicate latch-register and shifter must be provided for the other.

Figuring in necessary gates and mixing circuits, and allowing the equivalent of four levels for rise time, skew, and uncertainties in the latch driver power circuits, the data path loop contains fourteen levels besides those in the adder. Also, for the system shown in Figure 5, no speed advantage is gained by making the main adder faster than the path through the decoder and shift-counter-adder. The latter will be in the neighborhood of eleven levels, seven for the adder and four for the complete decoder. Eleven levels, however, can be obtained at considerably less cost in equipment with the carry-propagate adder with full look-ahead. From this it may be concluded that there would be very little, if any, time gain and considerable additional expense if the adder in Figure 5 were changed to a CP/CS adder with the necessary associated changes.

The above does not mean that faster multiplication cannot be obtained through the use of carry-save adders. It merely indicates that that particular method of applying it would not produce the desired result.

In Figure 5 the high-speed main adder represents probably about half of the equipment in the complete data path. Figuring the adder as twelve, and the remainder of the path as fourteen, the total loop path is the equivalent of 26 logical levels. If a carry-save adder were connected in series with the present adder, then the total path length would be fourteen plus twelve plus four, or thirty; however two additions could be performed in each cycle, which would halve the number of cycles. This is, of course, an oversimplified description of the method and its results, but its proper application will permit profitable use of carry-save adders in multiplication.

When two or more adders are operated in series in the performance of multiplication, an attempt to have a variable shifter ahead of each of them will result in a more complicated decoder, longer path length, and considerable additional equipment. For this reason a fixed-shift type of operation, such as one of those already described, is more desirable than the variable-shift methods. The comparative merits of and requirements for two-and three-bit shifts have already been described, together with the decoding rules for each. The application of carry-save adders will be described in terms of the two-bit shift. Necessary variations in using the three-bit shift will be readily apparent from the previous description.

Figure 6 illustrates a system that will handle eight bits of the multiplier at a time. It shows three carry-save adders operating in series, with the two outputs of the last of these going to a carry-propagate adder. One of the three input to CSA 1 is the partial product from the previous cycle. The other two are multiples of the multiplicand determined by decoding two groups of multiplier bits. Two of the three inputs of CSA 2 are required for the two outputs of CSA 1, leaving one for a multiple of the multiplicand obtained by decoding the third group of the multiplier. In a similar manner CSA 3 provides an input for a fourth multiple. The two outputs of CSA 3 go to the inputs of the carry-propagate adder, and the single output of the CPA goes to the main latch-register as the partial product for the next cycle. The modification of the decoding of the first group for the first cycle is used as was described, so that the true value of one-times the multiplier can be used when the low order bit of the multiplier is a <u>one</u> Entry for this is shown as G13.

The details of one cycle of the multiplication of two 16-bit binary numbers are illustrated in Figure 7. During the first add cycle a 16-bit number is being multiplied by an 8-bit number. This may give a true result not exceeding 24 bits in length. Therefore a one in position 25 will indicate a complement partial product. One-times the multiplicand, when required, goes into positions 1-16 of the A input of CSA 1. Decoding of the low order group of the multiplier calls for zero, two, or four times the multiplicand to be entered at the B input of CSA 1. This multiple is referenced to position 1 of the adder, which means that two times the multiplicand would go to positions 2-17, while if four times were called for, it would go to positions 3-18. All other positions of this adder input get zeros if the input is true and ones if it is complement.

Since the low order bit of group 2 of the multiplier is two positions to the left of the corresponding bit of group 1, the reference position for dete.mining entry into the adder is also two positions to the left of that for group 1, that is, position 3 instead of position 1. This means that a two-times multiple for group 2 will go into positions 3-19, while a four-times multiple will go into positions 4-20. Again, unused positions get zeros for true and ones for complement.

For CSA 2 the A_2 input is the sum outputs (S_1) from CSA 1 carried down in the same columns. The B_2 input is the carry outputs (R_1) of CSA 1, each shifted one column left, which leaves column 1 for the complement forced carry input for group 2. The C_2 input is obtained from decoding group 3, and is referenced to column 5.

For CSA 3 the A_3 input is the sum output of CSA 2 brought straight down, and the B_3 input is the carry output of CSA 2 shifted one position left, which leaves column 1 of B_3 for the complement forced carry entry due to group 3. The C_3 input is obtained by decoding group 4, and is referenced to column 7. The sum outputs of this adder go into the corresponding columns of one of the inputs of the carry-propagate adder, while the carry outputs go into the carry-propagate adder shifted one position left. This leaves one entry in column 1 available for the forced carry input associated with group 4. The forced carry associated with group 1 can also be entered into the carry-propagate adder by way of the carry input circuit of position one. Rather than use a special adder connection, this can be done by entering an input into both sides of position zero when the carry input is desired.

For all of the adders, carry outputs from column 25 that would normally go into column 26 of the next following adder are ignored and lost, as it would serve no useful purpose to retain them. Column 25 supplies the required information as to whether the partial product is in true or complement form.

Figure 7 assumes that each carry-save adder has a length equal to the length of the partial product developed in each cycle. Means for reducing each of these to approximately the length of the multiplicand will be described following a summary of the operating sequence. The sequence is essentially the same for either version.

Step 1.

Enter the multiplier into the right half of the MQ register and the multiplicand into the MD register. Set the shifter to shift the right half of the MQ register eight positions to the right, keeping it at this setting throughout the multiply operation. Clear multiplicand selection register. Set first-cycle trigger to cause proper treatment of the low-order bit of the multiplier.

Step 2.

Energize latch-control signal. This sets decoder results into the multiplicand selection register that controls the gates into the carry-save adders, shifts the multiplier right eight positions to discard the low order eight bits and bring the next group of bits into the decoder, and sets the output of the CPA adder (zero in this case) into the MQ register.

Step 3.

Energize latch-control signal (after sufficient time has elapsed for the data to have passed through all of the adders). This sets the results of decoding the second set of eight bits of the multiplier into the multiplicand selection register, shifts the multiplier eight positions right, and enters the data from adder output positions 1-25 into positions 9-33 of the MQ register. The low order eight bits of this partial product are in their final form. These are in positions 9-16 of the register. Therefore, on this cycle, the entire adder group is effectively shifted eight positions, which means that data from register positions 17-33 will go to the A_1 input of CSA 1 positions 1-17. Since position 33 contains a zero if the partial product is true and a one if it is complement, input positions 18-25 of A_1 will be set to agree with the input to position 17.

Step 4

Energize latch-control signal. This sets decoder output into multiplicand selection register (has no meaning since multiplier was shifted out of register by Step 3, but no advantage is agained by suppressing it), shifts partial product that was inpositions 9-16 of MQ register into positions 1-8, and enters the remainder of the product from the carry-propagate adder into positions 9-33.

Note that the data that was in positions 17-33 is replaced, and not shifted elsewhere. This completes the multiplication.

Component Reduction with Carry-Save Adders

A carry-save adder takes three signals in and gives two out. If the number of inputs is reduced to two, the number of outputs still remains at two. Therefore, when two or more carry-save adders are used in series, any bit positions which always have zeros for one of the three inputs may be omitted. This eliminates two outputs from the omitted adders, thus vacating inputs to two positions farther down the adder chain. The two inputs that would have gone to the omitted adder positions can then go to these two positions. An input may be moved from any one place in the chain of adders to any other place as long as it is always kept in the same column.

When the two's complement of a binary number is desired, the one's complement is obtained, then a one is added to this in the column of the lowest order bit. The column into which the one is entered may vary from this if the column selected is the same as, or of a lower order than, the column containing the lowest order one in the true value of the number, and also provided that the zeros to the right of the selected column are not inverted when forming the one's complement of the number.

The application of these two principles will permit the elimination of a number of low order positions from the adders shown in Figure 7. This is illustrated in Figure 8.

Since the input C_1 never needs to have anything except zeros in positions 1, 2, and 3, and since nothing needs to be added into these columns in any other adder, the inputs for these columns that would normally go to A_1 and B_1 may be shifted down to the CPA inputs and all carry-save adder positions for these columns eliminated. The forced carry input for group 1 remains the two CPA inputs in column zero. In Figure 8 terminations for the adders are indicated by double vertical lines. Positions outside these terminations are designated by numbers in circles, and the position to which these are transferred is designated by the same number in a hexagon.

The three inputs for CSA 2 are the sum and carry from CSA 1 and the multiple obtained by decoding group 3. The lowest order column required by the latter is six, which means that the inputs to columns 4 and 5 may be transferred. It should be noted that with the group 2 multiple ending at column 4, the forced carry for this was moved to column 4 of B_2, and is now being transferred to the same column of CPA input B. CSA 3 is then treated in a similar manner. Altogether, these modifications have eliminated fifteen adder positions from the low order ends of the adders.

The modification of the high order end of the adders is based on the fact that, since the inputs are staggered, the adders will have a number of high order positions containing either a string of <u>ones</u> or a string of <u>zeros</u>. When two of the three inputs meet this condition, these two inputs may always be replaced by a single input, which reduces the total number of required inputs to two. As has already been shown, when this condition exists, these stages of the adder may be eliminated, and the pair of inputs moved down to the next adder in the chain. The operation of this is illustrated below for the various combinations that may occur.

```
1  1  1  1  '  1  *  X  X  X  X  A1
1  1  1  1  '  1  *  X  X  X  X  B1
D  E  F  G  '  H     X  X  X  X  C1             TWO
                                             COMPLEMENT
1  1  1  1  '  S     S  S  S  S  A2            INPUTS
D  E  F  G  '  R     R  R  R  R  B2
---------------------------------------------------------------
```

$$\begin{array}{cccccccl}
1 & 1 & 1 & 1 & ' & 1 & * & X\,X\,X\,X\ A_1 \\
0 & 0 & 0 & 0 & ' & 0 & * & X\,X\,X\,X\ B_1 \\
D & E & F & G & ' & H & & X\,X\,X\,X\ C_1 \\
\\
\bar{H} & \bar{H} & \bar{H} & \bar{H} & ' & S & & S\,S\,S\,S\ A_2 \\
D & E & F & G & ' & R & & R\,R\,R\,R\ B_2
\end{array}$$

ONE
COMPLEMENT
INPUT

$$\begin{array}{cccccccl}
0 & 0 & 0 & 0 & ' & 0 & * & X\,X\,X\,X\ A_1 \\
0 & 0 & 0 & 0 & ' & 0 & * & X\,X\,X\,X\ B_1 \\
D & E & F & G & ' & H & & X\,X\,X\,X\ C_1 \\
\\
0 & 0 & 0 & 0 & ' & S & & S\,S\,S\,S\ A_2 \\
D & E & F & G & ' & R & & R\,R\,R\,R\ B_2
\end{array}$$

NO
COMPLEMENT
INPUTS

The three inputs shown together represent the inputs as they would be if the complete adder were used. The asterisks in two of the inputs indicate that there are never any high order true bits to the left of this point for these two inputs. The apostrophes indicate the point at which it is desired to terminate the adder shown with three inputs. The two inputs below are two of the three inputs of the next following adder. For columns to the right of the termination point of the first adder, the inputs to the following adder are the sum (S) and carry (R) outputs of the adder above. To the left of the termination of adder 1, the B_2 input of adder 2 becomes what would have been the C_1 input of adder 1 for the same columns. Note that the carry output of the highest order column of adder 1 after it is terminated does not go into the next higher order of column B_2, as this position is occupied by G from C_1. The corresponding A_2 inputs to adder 2 are the same for all bit positions to the left of the termination point of adder 1, and are determined from the three inputs to the highest order column of the terminated adder. 1.

Figure 8 illustrates the effect of applying this method to the adders of Figure 7. In CSA 1, input A_1 is determined by its true or complement condition starting with column 17, B_1 with column 19, and C_1 with column 21. It is therefore possible to terminate this adder with position 19, and move the normal C_1 inputs for columns 20 and 21 to the corresponding columns of C_2.

The normal full adder used for each position of the CSA contains the following logic.

$$S = (A \forall B) \forall C \qquad (1)$$

$$R = (A \forall B)\ C \vee A\ B \qquad (2)$$

For the high order column of the terminated adder, in this case column 19, this is modified to the following.

$$S = (A \forall B) \forall C \qquad (3)$$

$$D = (A \forall B)\ \bar{C} \vee A\ B \qquad (4)$$

In equations(1), (2), and (3), the terms A, B, and C may be applied to any of the three inputs to the adder. This is not true in equation (4), where the terms A and B refer to the two inputs determined by the fact that they are in true or complement form, while C refers to the data input. D describes the input that goes to all higher order positions of the next adder, and for that adder it may be treated as are those positions whose input is determined by knowledge of whether the input is true or complement.

By continuing with this procedure, CSA 2 may be terminated at position 21, the position 21 circuit being modified as described above; and CSA 3 may be terminated with column 23, the position 23 circuit also being modified.

The three carry-save adders as originally described in Figure 7 required a total of 75 individual full adders. The same adders with the modifications described require 45 full adder units plus three modified units, a saving of 27 units.

For the operation described, the length of the carry propagate adder had to exceed the length of the multiplicand by two more than the length of the section of the multiplier handled during each cycle. If this additional length is not required for other operations, and if the main part of the adder uses full carry look-ahead, the reduced path length for the low-order bits in the carry-save adders resulting from the modifications made to save components permits the use of a ripple-carry adder for most of the extension to increase the length of the main adder without causing any loss in speed.

From the information given, the modifications required to permit the use of three-bit multiplier groups instead of two-bit groups are obvious. The question of how many carry-save adders to connect in series is a matter of economics to be decided for a particular application. The example given was intended merely to help describe the general method, and many modifications of it to suit special conditions will be readily apparent.

BINARY DIVISION

There are several methods, of varying complexity and speed, by which division may be performed in a computer. The implementing of a particular method will vary between computers because of differences in circuits and machine organization. It is the intent here to discuss primarily basic methods, and to illustrate these methods, when required for clarity, with a particular type of machine organization. The characteristics of this type were described in the introduction.

The time required to perform a division is proportional to the number of additions required to complete it, and the methods that will be described for increasing speed will be primarily concerned with the reduction of the required number of additions. These methods will all use a variable length shift, and the number of additions required for any particular example will be dependent on bit distribution.

For all methods of division it will be assumed that prior to the start of the actual division the divisor is so positioned in the Divisor register that it has a one in the highest order position of the register. It will also be assumed that the divisor and dividend are binary fractions with the binary point located just to the left of the high order position. Thus the divisor will always have a numerical value less than one but equal to or greater than one-half. These assumptions do not limit the application of the principles of operation to be described, and they simplify the description.

Since all of the methods to be described involve variable shifts, it will always be assumed that a shift counter of some type is included, that this counter is set initially with the number of quotient bits to be developed, and that any shift-determining circuits include means for comparing the shift called for against the number still allowed by the shift counter and then acting on this information according to the rules that will be developed for the particular method.

In all descriptions the term dividend will be used to mean both the initial and partial dividend, while the term remainder will mean the final remainder after the quotient is completely developed.

Figure 5, which was used in the description of multiplication, will also be used as the basic circuit for describing division. Any modifications required by a particular method will be described. All operations start by setting the dividend into the MQ register, the divisor into the MD register (including normalization of the divisor if it is not already in this condition), and the quotient length into the shift counter (which is assumed to count down). The high order bit position of the dividend (with a shifter setting of zero) and the high order bit position of the divisor enter the same column of the adder unless stated otherwise. Dividend shifting is to the left, which clears the right end of the MQ register as the operation proceeds. The quotient is developed at the right end of the MQ register and shifted along with the dividend. The dividend decoder is assumed to be on the high order end of the adder output, which means that the initial operation always starts with a forced zero shift, following which the decoder takes control of the shifting.

Some additional general rules that apply to all methods, particularly those that deal with starting and terminating a division, will be discussed following the detailed descriptions of the several methods.

Division Using Single Adder, One-Times Divisor, and Shifting Across Zeros and Ones.

Assume a dividend in true form. Since the high order bit of the divisor is required to be a one, if the high order bit of the dividend is a zero,the divisor is obviously larger than the dividend which will result in a zero quotient bit. A zero may therefore be placed in the quotient, and the dividend and quotient each shifted left one position before any addition is performed. If there are n leading zeros, and the decoder can recognize them, n positions may be shifted across in one operation, a zero also being inserted in the quotient for each position shifted.

With the dividend true and the high order bit a one, an addition must be performed to determine whether or not the dividend is larger than the divisor. If the result of the operation is true, the dividend was larger, and a one is entered in the quotient. If the result is complement, the dividend was smaller than the divisor, and a zero is entered in the quotient. In either case, the result of the addition replaces that part of the previous dividend in the MQ register that was used in the addition. If the result of the addition was a complement number, this will now make the entire new dividend a complement number, even though part of it did not go through the adder.

Shifting the dividend one position left is equivalent to dividing the divisor by two with respect to the original dividend. For a true dividend with a high order one , if one times the divisor results in a zero in that position of the quotient (divisor larger than dividend), then one-half of the divisor (next shift position) will always result in a one in the following bit position of the quotient.

(Dividend is equal to or greater than one-half, while one-half of divisor must be less than one-half.) If, after the first addition, the dividend had been returned to its original value, then, using the first addition as a point of reference, the second addition would have given a true result (indicating the one in the quotient) with a value equal to the original dividend minus one-half of the divisor. If, instead of returning to the original dividend, shifting, and adding complement, the complement result of the previous addition had been retained, shifted, and the true value of the divisor added to it, the result would have been (original dividend minus divisor) plus (one-half dividend). This would also be a true final result having the same value as was obtained by the previous method.

Assume that a partial division has been performed yielding a partial quotient of 0 1 1 1 1 and a corresponding partial dividend. This result could have been obtained by any of the following series of operations.

Dividend + (- 1/2 - 1/4 - 1/8 - 1/16) Divisor
Dividend + (- 1.0 + 1/2 - 1/4 - 1/8- 1/16) Divisor
Dividend + (- 1.0 + 1/4 - 1/8 - 1/16) Divisor
Dividend + (- 1.0 + 1/8 - 1/16) Divisor
Dividend + (- 1.0 + 1/16) Divisor

These are all equal to Dividend minus 15/16 Divisor. From this it may be stated that if a complement result is obtained under the condition that it is known that the next succeeding quotient bit is a one, then as many positions of the dividend may be shifted across, a one being entered in the quotient for each position shifted across, as is known will still result in a true dividend following the addition.

Since the high order position of the divisor, in its true form, always contains a one, a true result will always be obtained if the high order bit position of the complement dividend contains a one. This justifies shifting across all except the last one in a string of high order ones in a complement dividend, together with the entering of a one in the quotient for each position shifted across. It is also known that if an addition is performed without shifting across the final one, a true dividend will always be obtained together with another one in the quotient. If the complement result had been shifted one position farther, the new dividend obtained would be the same following the addition of the true divisor as would have been obtained following a one position shift of the true dividend and the addition of the complement of the divisor. Thus it is evident that with either true or complement dividends it is only necessary to perform an addition when it is not evident what the quotient bit should be. From this the following operating rules may be stated.

(1) When the dividend is true, shift across any leading zeros, entering a zero in the low order end of the quotient for each position shifted across except the last; then add the complement of the divisor.

a) If the result is true, enter a one in the low order position of the quotient, then shift across zeros.

b) If the result is complement, enter zero in the low order position of the quotient, then shift across ones.

(2) When the dividend is complement, shift across any leading ones, entering a one in the low order end of the quotient for each position shifted across except the last; then add the true divisor.

a) If the result is true, enter a one in the low order position of the quotient; then shift across zeros.

b) If the result is complement, enter a zero in the low order position of the quotient; then shift across ones.

If the decoder calls for a larger shift than can be obtained from the shifter in one operation, use the maximum shift available and suppress both the true and complement entry of the divisor to the adder. This will pass the high order part of the shifted dividend through the adder with zero added to it so that it is available to the decoder. If the dividend is complement, the output of the adder following this will be complement, which would normally result in the setting of a zero in the low order position of the quotient. However, this is in the middle of a shift across ones, not an addition to determine the proper quotient bit following a shift, and the dividend only goes through the adder because of the necessity of making it available to the decoder. Therefore, in this case, the low order bit of the quotient following the shift must be set to agree with the bits being shifted across. The same control that suppresses the entry of the divisor into the adder can also control this.

Some special rules are required to terminate the division and insure that the final remainder will be in true form. These are listed below.

(1) Dividend true, shift called for by decoder larger than allowed by shift counter. Treat in same manner as when shift called for is greater than capacity of shifter. Make shift allowed by shift counter, suppress entry of divisor into adder, set low order bit of quotient to agree with bits being shifted across. This will complete the division.

(2) Dividend true, shift called for by decoder equal to that allowed by shift counter. Treat in normal manner. If resulting adder output

is in true form, division is complete with its entry into the register. If resulting adder output is in complement form, one additional cycle is required to get remainder into true form. See (4) below.

(3) Dividend complement, shift called for by decoder equal to or greater than that allowed by shift counter register. Use allowed shift and proceed in normal manner. If resulting remainder is in true form, division is complete. If resulting remainder is in complement form, resulting quotient is complete, but one additional cycle is required to get remainder into true form. See (4) below. The latter condition can only occur when the shift called for and the shift counter register are equal.

(4) Dividend complement, shift counter register is zero. Take zero shift, add true value of divisor, suppress entry from adder output into low order bit position of quotient as the bit there is already correct (zero) and the true output of the adder would change it to a one.

If the following binary division is performed according to these rules, it will require fourteen add cycles to complete the operation.

```
                    011,100, 011, 011, 001, 001, 010, 110
110, 110 | 10, 111, 111,110, 111, 001, 111, 000, 100, 100
```

To compare this with the inverse operations required for multiplication, the quotient is shown below with the various additions and subtractions used shown above the corresponding bit positions, and the corresponding operations as determined from the multiplication rules shown below.

```
-        +        -     +  -     +        -     -  +  -  +  -     +  o     (14)
0  1  1, 1  0  0, 0  1  1, 0  1  1, 0  0  1, 0  0  1, 0  1  0, 1  1  0
-        +        -        +     +        -        -     -  -     +  o     (11)
```

Division Using Double Adder and One-Half, One, and Two Times Divisor

If a quotient contains a string of zeros followed by a string of ones, it is possible to shift across the ones only if the addition made after the shift across the zeros resulted in a complement dividend. If the result was a true dividend, then it is necessary to make a separate addition for each one in the string. This means that in some instances better results would have been obtained if the addition had been performed one position sooner than the position resulting from following the shift rules. This condition is most likely to occur with a small divisor, as a small divisor is less likely to produce a change in the sign of the dividend than a large divisor.

When a quotient contains two strings of ones separated by a single zero, mor efficient operation will be obtained if it is always treated as one string of ones with an interruption. This may be seen by comparing the fourth and fifth operations of the previous divide example with the fourth operation of the potential divide system obtained by an inversion of the multiplication rules and shown for comparison. In this case it is desired that the addition at the end of the first group of ones produce a complement result which will supply the single zero for the quotient and leave the remainder in complement form for shifting across ones again; the inverse applies if the quotient is two strings of zeros separated by a single one. To obtain this condition it is sometimes necessary to perform the addition one position later than the position given by the shift rules. However, if this extra length shift is taken at other times it may produce incorrect results. The failure to obtain optimum operations under these conditions is most likely to occur when the divisor is large because a large divisor has a greater probability of producing a change in the sign of the dividend.

It has been shown that the efficiency of the division operation may be improved if, on certain occasions, the addition following a shift could be made with the divisor one position to the left of the normal position, and on other occasions one position to the right of the normal position. By normal position is meant that position reached by shifting across all leading ones for a complement dividend or across all leading zeros for a true dividend. The divisor used in the normal position is designated as one times divisor, left of normal position as two-time divisor, and right of normal position as one-half times divisor.

One method of obtaining this improvement is by double addition. It requires that the main adder be slightly longer than twice the length of the divisor, or that there be two adders available. The procedure is to perform two additions simultaneously, then use the result that produces the largest shift. If a double length adder is available, the two additions may be performed in it as long as there is at least one position with no inputs to it between the two operations. One addition will always be performed with the divisor located, with reference, to the dividend, as called for by the shift decoder. The other addition will be performed using twice the divisor if the two high order bits of the divisor in its true form are 1 0 (value of divisor less than three-fourths), and one-half the divisor if the two high order bits are 1 1 (value of divisor equal to or greater than three-fourths). Thus a small divisor uses the larger multiple, while a large divisor uses the smaller multiple for the auxiliary addition.

The circuitry required is similar to that of Figure 5 except that the adder size is increased, gates are added to enter the dividend into the other half of the adder also, and to select two times or half times the divisor for entry there, the decoder is increased to decode and compare the two results, and a gate is added to permit a choice of the two outputs.

Although the two additions may be performed in two parts of one adder, the two parts will be called adder A and adder B. Adder A will correspond to the adder described in the previous method, while adder B will be the alternate adder. The output of adder B will be used only if its use results in a greater shift than would result from using adder A. If the shifts called for by the two adder outputs are the same, the adder A results will be used.

If the previously described example were performed using this method, the resulting operations would be exactly the same as those obtained by using the inverse of the multiplication rules. The rules for quotient development and division termination are very similar to those for the system using a single length adder, and will be developed when it is described.

Figure 9 is a table showing all possible results that can be obtained for a five-bit true divisor and complement dividend under the restrictions that a true divisor always has a high order one and a complement dividend is always used following shifting across all leading ones, which means that it will always have a high order zero. A corresponding table can be prepared for complement divisor and true dividend. If this is done and the two are compared, it will be found that for the same position the result on one table will be the exact inverse of that on the other table. For example, at column 3, row 10 of Figure 9 the result is 00110, while the corresponding position of the other table would be 11001.

The number of positions to be shifted is the same in both cases. The information of primary interest to be obtained from these tables is the number of shifts, which is shown in Figure 10.

From this table it is apparent that points of maximum shift lie along the diagonal representing equal values for divisor and dividend. Also, if random distribution of divisor bits between problems and dividend bits between and within problems is assumed, then the average shift per cycle will be 651/256 = 2.54 for a five-bit divisor used with a shifter capable of handling shifts of five or less. (It can be shown that the distribution of bits within a dividend does not remain completely random as the division progresses. However, the variations will not be sufficiently great to invalidate the results of the comparisons of efficiencies of different methods of division based on the assumption of complete randomness.)

Figure 11 shows a table of shifts that may be obtained when using one-half times the divisor or two times the divisor. Both are shown on the same table, half of the table being used for each. These results apply both for dividend complement with divisor true and for dividend true with divisor complement. On this and the preceding figure the pattern of shifts along any row should be noted, as each row contains a section of the pattern. The pattern goes both ways from the line of maximum shifts, and is one '5', one '4', two '3's, four '2's, eight '1's, and all that follow '0'. Any selection system used must not permit the selection of zero shift during normal operation, as this will result in an error in the problem.

When one-half or two times divisor is used, the dividend is positioned in the same manner as if one times the divisor were to be used, then the divisor is entered into the adder shifted one position to the left or right of where it would have been for one times. The columns of the output of the adder that are examined to determine the next shift are the same ones that would have been examined had one times the divisor been used. When preparing the table and using one-half times the divisor, the low order bit of the divisor is lost as a result of the right shift. This would not be the case in an actual operation, as the adder would have been extended by one position and an additional bit of the dividend would have been brought into the adder. When two times the divisor is used, the high order bit of the original divisor is entered into the overflow position of the adder, but for all the combinations for which two times the divisor would be used, this combines with the complement dividend to produce a true divisor with no overflow. Therefore this five-bit remainder used for the chart is correct.

Examples of the use of one times the divisor are shown below, followed by examples of one times and one-half times. The examples onthe left use one times, while the top right uses two times and the bottom right one-half times. The part of the result that is used in the tables is to the right of the binary point in each case. The part to the left is shown indirectly by the indication of true or complement result. The figure numbers, column numbers and row numbers refer to the table locations of the examples.

Figures 9 & 10

$$\begin{array}{l} 11.00010 \\ 00.10001 \\ \hline 11.\underline{1}0011 \end{array}$$

$$\begin{array}{l} 11.01110 \\ 00.11101 \\ \hline 00.\underline{0}1011 \end{array}$$

Figure 11

$$\begin{array}{ll} 11.00010 & \text{Column } 13 \\ 01.00010 & \text{Row } 1 \\ \cline{1-1} 00.\underline{001}00 & \end{array}$$

$$\begin{array}{ll} 11.01110 & \text{Column } 1 \\ 00.01110 & \text{Row } 13 \\ \cline{1-1} 11.\underline{111}00 & \end{array}$$

The underlined part of the result indicates the amount of shift that would result in each case.

Figure 12 is obtained by replacing all of the positions calling for a shift of one on Figure 10 with the shift called for on the corresponding position of Figure 11. The three sections are shown separated by heavy stepped lines. The circled numbers represent shifts that are the same on both figures. This represents the optimum combination that can be obtained when using one-half, one, and two times the divisor, and gives an average of 2.82 bits per cycle.

The heavy line between rows 7 and 8 represents the division that was made between the use of half times and two times divisor in the double adder method. As may be seen , the optimum use for each multiple is within this division, which means that the double-adder method of division will give the same results as are obtained from optimum use of these particular divisor multiples. An alternate selection rule which may be used with the double adder method for these particular multiples is: "If the output of the alternate adder does not call for a shift of two or more, use the output of the adder having the one times divisor input". This avoids the need for any compare circuits, and also gives correct results.

Division Using Single Adder With Half, One and Two Times Divisor

If only a single length adder is available, the use of the three divisor multiples to improve efficiency is still possible, although the improvement may be somewhat less. In this case the selection must be made by examining, or decoding, the high order bits of the divisor and dividend before each operation to determine what multiple to use. The degree of improvement will be dependent on the number of bits included, as will the complexity of the decoding system and the time required by it. The selection must be sufficiently accurate that it will never call for a multiple that will result in a zero shift. The dashed lines on Figure 12 that outline rectangles in the upper left and lower right corners indicate what may be expected from very simple decoding. This is based on the following rules: (1) If the high order bits of the divisor are 111 and the high order bits of the dividend are either 011 or 100, use the half times divisor multiple. (2) If the high order bits of the divisor are 100 and the high order bits of the dividend are either 000 or 111, use the two times divisor multiple. (3) If neither of these conditions exist, use the one times divisor multiple. This gives an average of 2.74 bits shifted per cycle as compared with 2.82 for the double adder.

Quotient Development and Termination When Using 1/2, 1.0, & 2.0 Multiples

When these multiples are used, an additional low order register position is required. Designate the two low order positions of this register as X and Y, where X is the position that is normally set by whether the output of the adder is true or complement when one times the divisor is used. Position Y is the next lower order position in the register.

When the half times divisor is used, it is in the same position with respect to the dividend that the one times divisor would have been had the previous shift been one greater. Therefore the quotient bit determined by the output of the adder when the half times divisor is used must be placed where it will enter the quotient adjacent to position X, which is position Y. The quotient bit placed in position X must be the same that would have been place there had one times the divisor been used, and will always be the same as the bits shifted across during the preceding shift.

The bit placed in position Y as a result of the use of the half times divisor is a correct quotient bit. In the event that its generation is followed by a shift of one, the information that the half times divisor was used must be stored so that on the next add cycle position X can be set from data that was in position Y instead of from the condition of the adder output.

It should be noted then when the remainder from the use of the half times divisor multiple is decoded to give the number of bits to shift across, the number will always be one greater than would have been obtained had the previous shift been one greater followed by the use of one times the divisor, which puts the end of the shift at the same place in either case.

Whenever the one times divisor is used, position Y is set to agree with the bits that will be shifted across on the next shift. It enters into all shifting operations except shifts of one. It may be shifted across position X, but never into it (except for the special condition described above.).

The two times multiple will be selected only when the one times multiple, if used, would not cause a reversal in dividend sign, but the use of the two times multiple will cause a reversal. Therefore, if the original dividend was true, X is set to a one; if it was complement, X is set to a zero. Y is set to agree with the bits that are to be shifted across as determined by the output of the adder using the two times multiple. This bit is not preserved in the event of a one position shift.

The above information may be summarized in the following table.

Original Dividend	Multiple Selected	X	Y	Y definite
True	half times	O	1	Yes
True	two times	1	1	No
Complement	half times	1	O	Yes
Complement	two times	O	O	No

To terminate a division follow the rules previously given, with the added restriction that if the shift called for is equal to the contents of the shift counter register, the choice of the divisor multiple is limited to the one times multiple.

Division Using Divisor Multiples of Three-fourths, One, and Three-Halves

It was previously stated that the largest shifts occured along the diagonal of equal values of divisor and dividend. Figure 11 shows that such diagonals for the half times or two times multiples would each intersect the rectangle at one corner only, the half times going through the corner at which the divisor has a value of 1. 0 and the dividend 0. 5, and the two times going through the corner at which the divisor has a value of 0. 5 and the dividend 1. 0. A multiple which would have its high points within the area so that the high values on both sides would be available should give a greater improvement in efficiency. To be of practical use, it should also be easy to generate. Such a multiple is three-halves times the divisor, which can be generated in one addition cycle by adding one times the divisor to one-half times the divisor. Three-fourths times the divisor can then be generated from it by shifting.

Figure 13 shows a shift table obtained when using three-fourths and three-halves divisor multiples with five-bit divisors and five-bit dividends. The line of maximum shifts varies somewhat from the theoretical line because of the limits in size and the effects of truncating the three-fourths times multiple of five bits. Without these limits the line of maximum shifts for the three-fourths times divisor multiple would go between the points of divisor equal to 2/3 dividend equal to 1/2 and divisor equal to 1.0 dividend equal to 3/4; for the three-halves times divisor multiple the line would go between the points of divisor equal to 1/2 dividend equal to 3/4 and divisor equal to 2/3 dividend equal to 1.0.

Figure 14 shows a combination of Figures 10 and 13 to give the optimum arrangement when using the 3/4, 1.0, 3/2 multiples. The heavy stepped lines show the separation between the areas of use of the three multiples. The circled numbers represent shifts that are the same in the two adjacent areas. The separation line could go on either side of these positions without changing the result. The heavy horizontal line at divisor equals three-fourths represents the separation between the inputs to the alternate adder when these multiples are used in the double adder method, and the numbers in squares in the seven positions below this line indicate the shifts these positions would have as part of the one times area, instead of the three-fourths times area. The optimum arrangement here for the five-bit divisor indicates an average of 3.57 bits per cycle, while the use of these multiples in the double adder method gives 3.51 bits per cycle.

Figure 15 shows a coding arrangement for multiple selection that gives the same results as are obtained from the double adder method. A simpler coding method, which uses the three-fourths times multiple when the high order bits of the divisor are 11 and the high order bits of the dividend are either 10 or 01, and uses the three-halves multiple when the high order divisor bits are 10 and the high order bits of the dividend are either 11 or 00, will give an average of 3.37 bits per cycle based on a similar table (not shown).

The use of the three-fourths, one, and three-halves divisor multiples requires an additional register position (Z) because the three-fourths multiple produces two advance quotient bits, three definite bits in all. These go into positions X, Y, and Z. The three-halves multiple produces two definite quotient bits in positions X and Y, and a tentative bit in position Z. The one-times multiple produces one definite quotient bit in position X and two tentative bits in positions Y and Z.

If the division example previously described were performed using the double-adder method with three-fourths, one, and three-halves divisor multiples, the number of operating cycles would be reduced from eleven to nine. One cycle would have to be added to this to allow for the generation of the three-halves times multiple of the divisor.

Figure 16 illustrates graphically the various conditions that may occur when using the 3/4, 1.0, 3/2 divisor multiples. It shows an initial true dividend with complement divisor multiples only, but the inverse can easily be found from this by reversing all directions and interchanging zeros and ones in the quotient bit columns.

In example 1 the initial dividend is between one-and-a-half and two times the divisor. Selection here would choose the use of the 3/2 divisor multiple which would give two definite quotient bits and one tentative (indicated by a circle). The 1.0 times multiple could be used, though it would be less efficient. It would give one definite quotient bit and two tentative bits. In this case the first tentative bit would be incorrect, and would be changed on the next cycle. The 3/4 multiple would not be selected for use with this initial condition.

In example 2 the initial dividend is greater than one time the divisor but less than one-and-a-half times the divisor. Either the 3/2 or 1.0 divisor multiple may be selected here, but not the 3/4 multiple as it would be less efficient than the 1.0 times multiple. Here again the 3/2 multiple gives two definite quotient bits and the 1.0 times multiple gives one.

Example 3 has a dividend less than one times the divisor but greater than 3/4 times. It may use either of these multiples, but not the 3/2 multiple. The 3/4 multiple gives three definite quotient bits, while the 1.0 multiple gives one definite and two tentative.

In example 4 the dividend has a value between 1/2 and 3/4 the divisor. This condition will always result in the choice of the 3/4 divisor multiple, though the 1.0 times will give correct results.

Example 5 shows a dividend having a value less than half the divisor. This condition could only arise as a result of an incorrect previous cycle as it would require a true dividend with a leading zero following the shift.

The use of the 3/4 multiple will never result in a following shift of only one. If it results in a shift of two, the fact that the 3/4 multiple was used must be remembered into the next cycle, and the entry into position X must be made from position Z instead of from data obtained in that cycle from the adder result. Similar precautions must be taken when using the 3/2 multiple to protect data from position Y in the event of a one-position shift.

Division termination procedure is the same as was previously described, with the additional requirement that the 3/2 multiple must not be used if the shift counter register agrees with the shift called for, and the 3/4 multiple must not be used if the shift counter register agrees with or is one greater than the shift called for by the decoder. In either case the one-times multiple should be substituted.

Comparative Evaluation of Various Methods of Division

The effectiveness of several methods of performing division has been compared on the basis of five-bit divisors. These results need to be modified to show the effect of larger divisors. A simple method of doing this which will yield a close approximation to the desired result may be developed from a study of the pattern of shift amount variations in Figure 10. From this it can be predicted that if a six-bit chart is constructed, it will show the same percentage of total operations for shifts of 1, 2, 3, and 4 positions. The present shift of 5, which actually represents five or greater, would split approximately evenly into five, and six or greater. The six or greater could then be split approximately evenly in six, and seven or greater. The accuracy of this even division increases as the number of positions in the square increases.

In a computer the need for large shifts occurs so infrequently that it is usually not considered practical to include a shifter capable of making, in one shift cycle, all shifts that may be required. Once the data has been expanded to include the possibility of long shifts, the effect of this on performance must be considered.

To permit easier expansion, the data for the five-bit divisor was transferred to a basis of 1000 operations rather than 256, the 1000 operations being obtained by using the percentage figures from the various tables with the decimal moved one position right. In each case the expansion was extended to include all shifts that would occur at least one-tenth of one percent of the time. The remaining shifts, amounting to one-tenth of one percent, were all assigned to the next shift length. All numbers of shifts were adjusted to be whole numbers. The average total positions shifted across for 1000 shifts was then obtained by multiplying each shift number by its frequency of occurrence, then adding these products together. This number divided by 1000 gave the average bits shifted across per cycle with no limitation on the shifter size.

Limiting the range of the shifter leaves the number of bits shifted across the same as for the operation with no limit, but it increases the number of shift cycles required to get across them. If a limit of four is assumed, a desired shift of five will require two operations, one shift of four and one shift of one. A desired shift of ten would require three operations, two shifts of four and one shift of two.

The results obtained in this manner for eight different division methods will be summarized in the following table. A description of the column headings is given below.

(1) Division using one times the divisor and shifting across zeros only. Data for this was obtained from Figure 10 by assigning shift values of one to all complement results when starting with a true dividend.

(2) Division using one times the divisor and shifting across ones and zeros, single addition.

(3) Division using one-half, one, and two times divisor with coded multiple selection.

(4) Division using one-half, one, and two times divisor with double addition, also with optimum selection.

(5) Division using three-fourths, one, and three-halves times divisor with simple (two by two) coding.

(6) Division using three-fourths, one, and three-halves times divisor with complex (four by eight) coding.

(7) Division using three-fourths, one, and three-halves times divisor with double addition.

(8) Division using three-fourths, one, and three-halves times divisor with optimum selection.

Average Bits Shifted Across Per Shift Cycle

Shifter Limit	1	2	3	4	5	6	7	8
None	1.86	2.66	2.86	2.94	3.59	3.77	3.75	3.82
8	1.85	2.64	2.84	2.92	3.54	3.72	3.59	3.76
6	1.83	2.54	2.78	2.86	3.40	3.55	3.54	3.60
4	1.76	2.39	2.53	2.61	2.98	3.07	3.08	3.03
5*	1.80	2.54	2.74	2.82	3.37	3.58	3.51	3.58

*Five-bit divisor

These figures are believed to represent an accurate comparison of the efficiencies of the different methods of division that have been described. The absolute accuracy is subject to the limitations previously explained.

ACKNOWLEDGEMENTS

Most of the material used in the preparation of this report was accumulated or developed during the design of the parallel arithmetic section of the IBM Stretch Computer. Particular mention should be made of the following original contributions.

The method of division described in the section "Division using single adder, one-times divisor, and shifting across zeros and ones" was proposed by D. W. Sweeney, and was described in an IBM internal paper titled "High speed arithmetic in a parallel device", by J. Cocke and D. W. Sweeney, February, 1957.

The method of division described in the section "Division using divisor of three-fourths, one, and three-halves" was proposed by J. R. Stewart, and a theoretical evaluation of its advantages made by C. Freiman.

The method of modifying the high order end of the adders described in the section "Component reduction with carry save adders" was proposed by F. R. Bielawa.

REFERENCES

1. Arthur W. Burks, Herman H. Goldstine, John von Neuman; "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument"; The Institute for Advanced Study, Princeton, New Jersey, 1947.

2. A. L. Leiner, J. L. Smith, and A. Weinberger; "System Design of Digital Computer at the National Bureau of Standards"; National Bureau of Standards Circular 591, February 1958.

3. Bruce Gilchrist, J. H. Pomerene, S. Y. Wong, "Fast Carry Logic for Digital Computers," *IRE Trans. on Electronic Computers,* Vol. EC-4, pp. 133-136, December 1955.

4. M. Lehman, "High-speed Digital Multiplication," *IRE Trans. on Electronic Computers* , Vol. EC-6, pp 204-205; September 1957.

5. James E. Robertson, "A New Class of Digital Division Methods," *IRE Trans. on Electronic Computers*, Vol. EC-7, pp. 218-222, September 1958.

6. E. Bloch, "The Engineering Design of the Stretch Computer", *Proc.* of the Eastern Joint Computer Conference, December 1959.

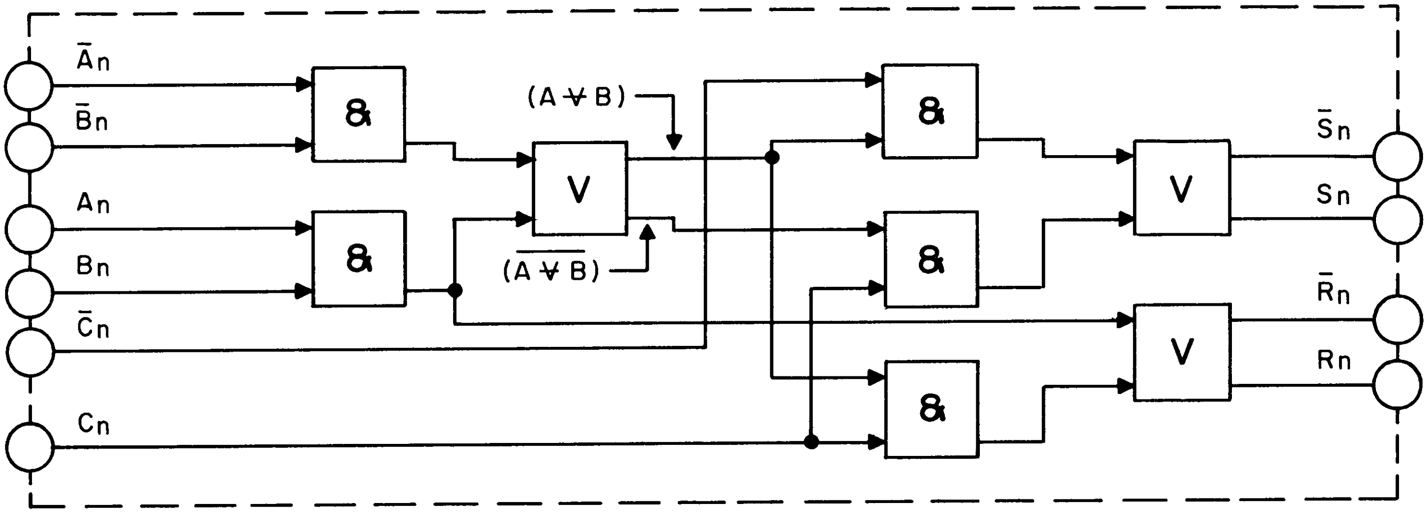
7. S. J. Campbell and G. H. Rosser, Jr., "An Analysis of Carry Transmission in Computer Addition". Preprints of papers presented at the 13th National Meeting of the ACM, June 11-13, 1958.

8. V. S. Burtsev, "Accelerating Multiplication and Division Operations in High-speed Digital Computers", The Institute of Exact Mechanics and Computing Technique, The Academy of Sciences of the USSR, Moscow, 1958.

9. J. E. Robertson, "Theory of Computer Arithmetic Employed in the Design of the New Computer at the University of Illinois", File #319, Digital Computer Laboratory, University of Illinois.

10. A. Avizienis, "A Study of Redundant Number Representation for Parallel Digital Computers", Report #101, Digital Computer Laboratory, University of Illinois.

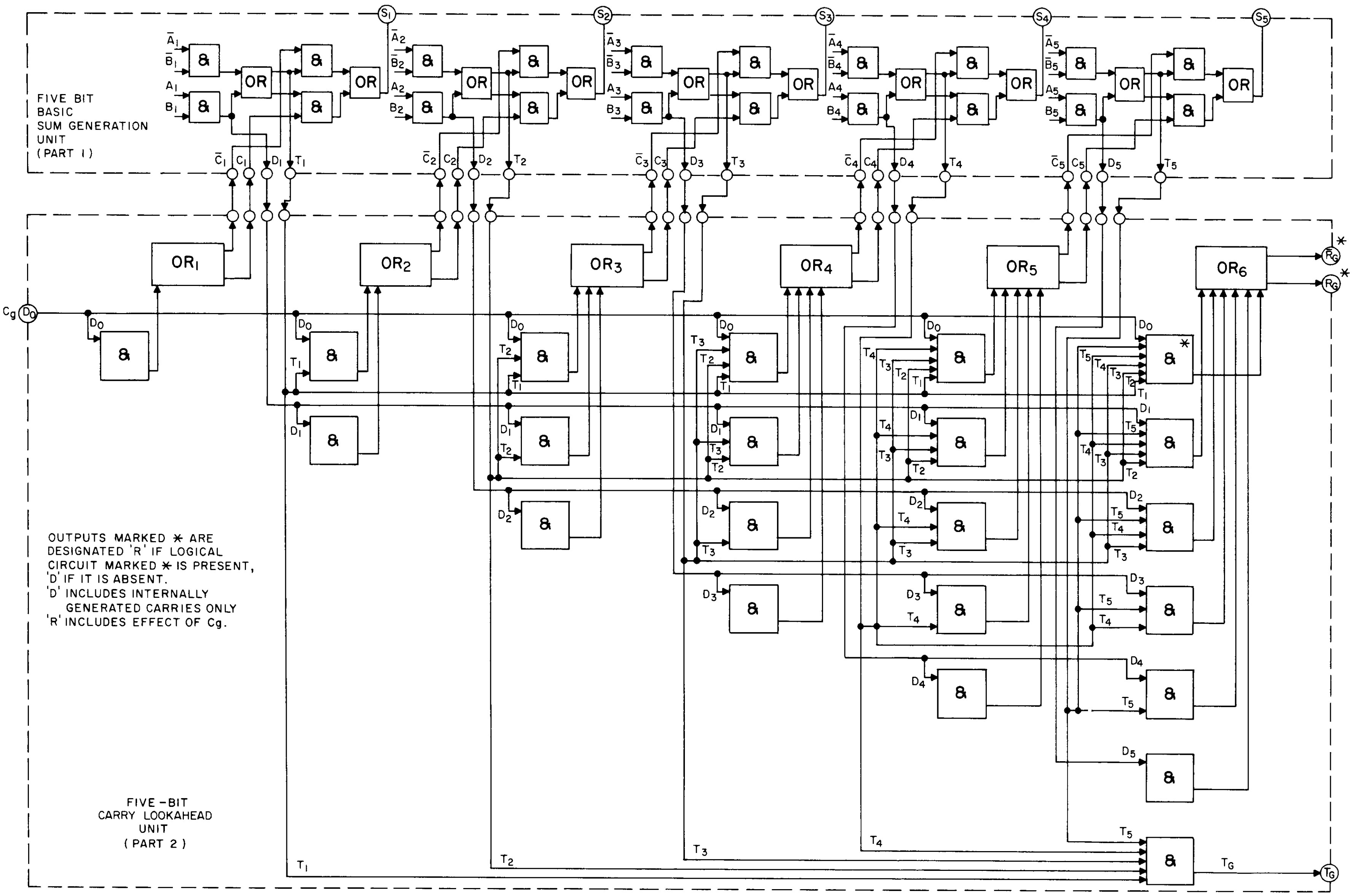
11. C. V. Frieman, "A Note on Statistical Analysis of Arithmetic Operations in Digital Computers", Paper to be published.



$S_n = A_n \veebar B_n \veebar C_n$

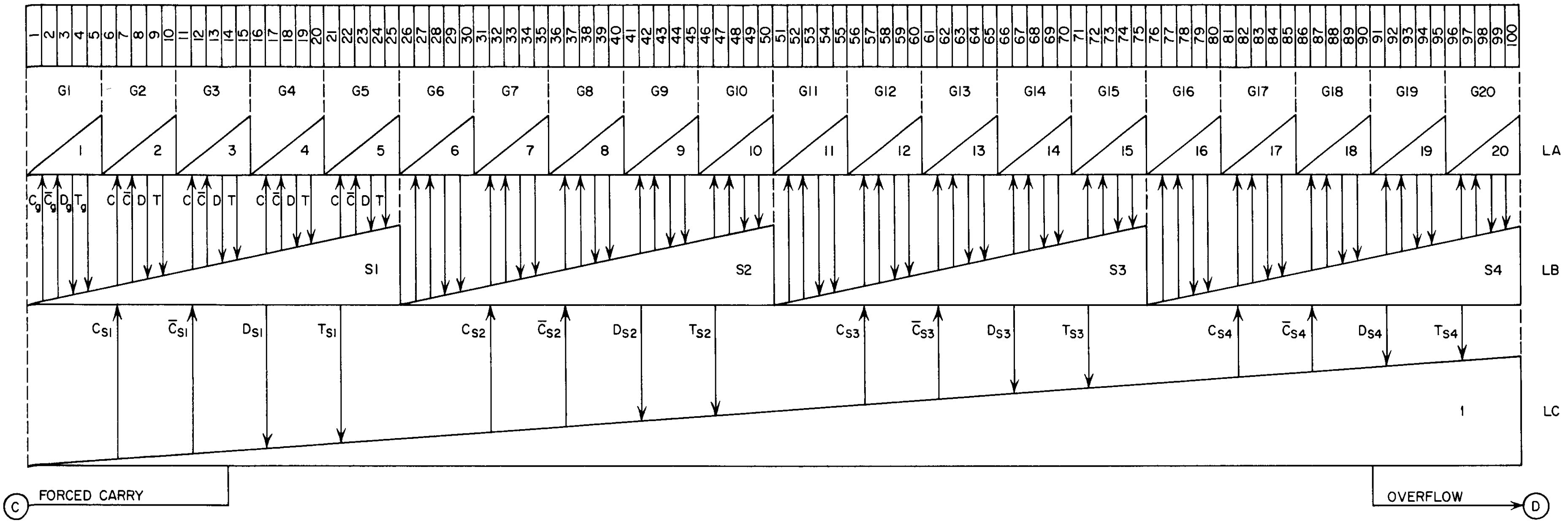
$R_n = (A_n \veebar B_n)\, C_n \vee A_n B_n$

FIGURE 1. Full adder, one stage.



OUTPUTS MARKED * ARE DESIGNATED 'R' IF LOGICAL CIRCUIT MARKED * IS PRESENT, 'D' IF IT IS ABSENT.
'D' INCLUDES INTERNALLY GENERATED CARRIES ONLY
'R' INCLUDES EFFECT OF Cg.

FIGURE 2. Five-bit adder group with full carry look-ahead.



TRANSIT TIME

A_I TO D_G = 4 UNITS

C_g TO S = 4 UNITS

C_g TO D_g = 2 UNITS

A_I TO T_g = 3 UNITS

D_g TO D_S = 2 UNITS

D_S TO C_S = 2 UNITS

C_S TO C_g = 2 UNITS

LOGICAL UNITS

BASIC ADDER 5-BIT GROUP = 30

5–INPUT LOOKAHEAD = 28

4–INPUT LOOKAHEAD = 22

	50 BIT ADDER	100 BIT ADDER
MAX TRANSIT TIME	12 UNITS	14 UNITS
LOGICAL UNITS		
BASIC SUM GENERATION UNITS	300	600
FIRST LEVEL CARRY	280	560
SECOND LEVEL CARRY	56	112
THIRD LEVEL CARRY	0	22
TOTAL	636	1294
LOGICAL UNITS/BIT	12.72	12.94

FIGURE 3. Carry-propagate adder with full carry look-ahead.

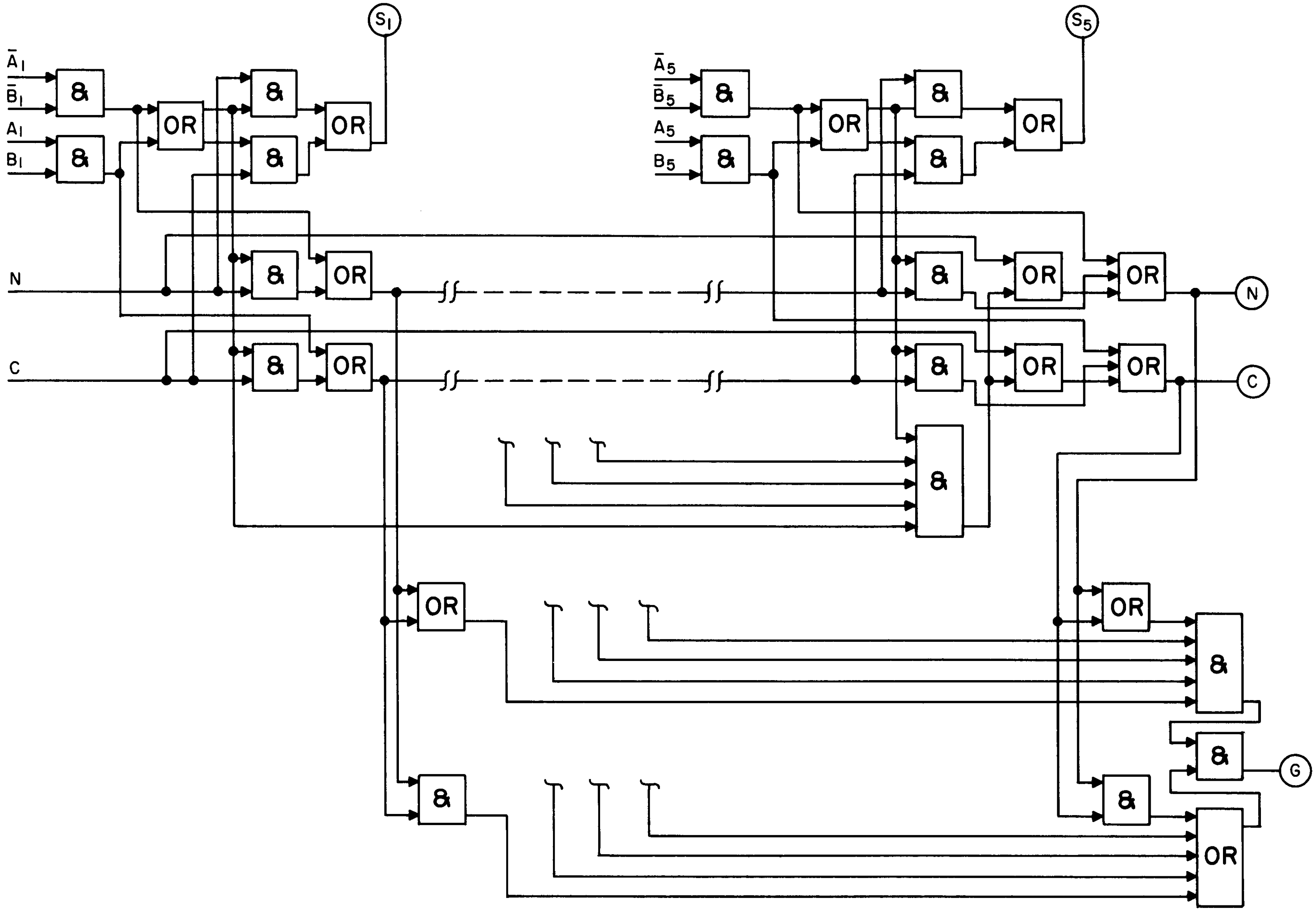


FIGURE 4. Completion recognition adder.

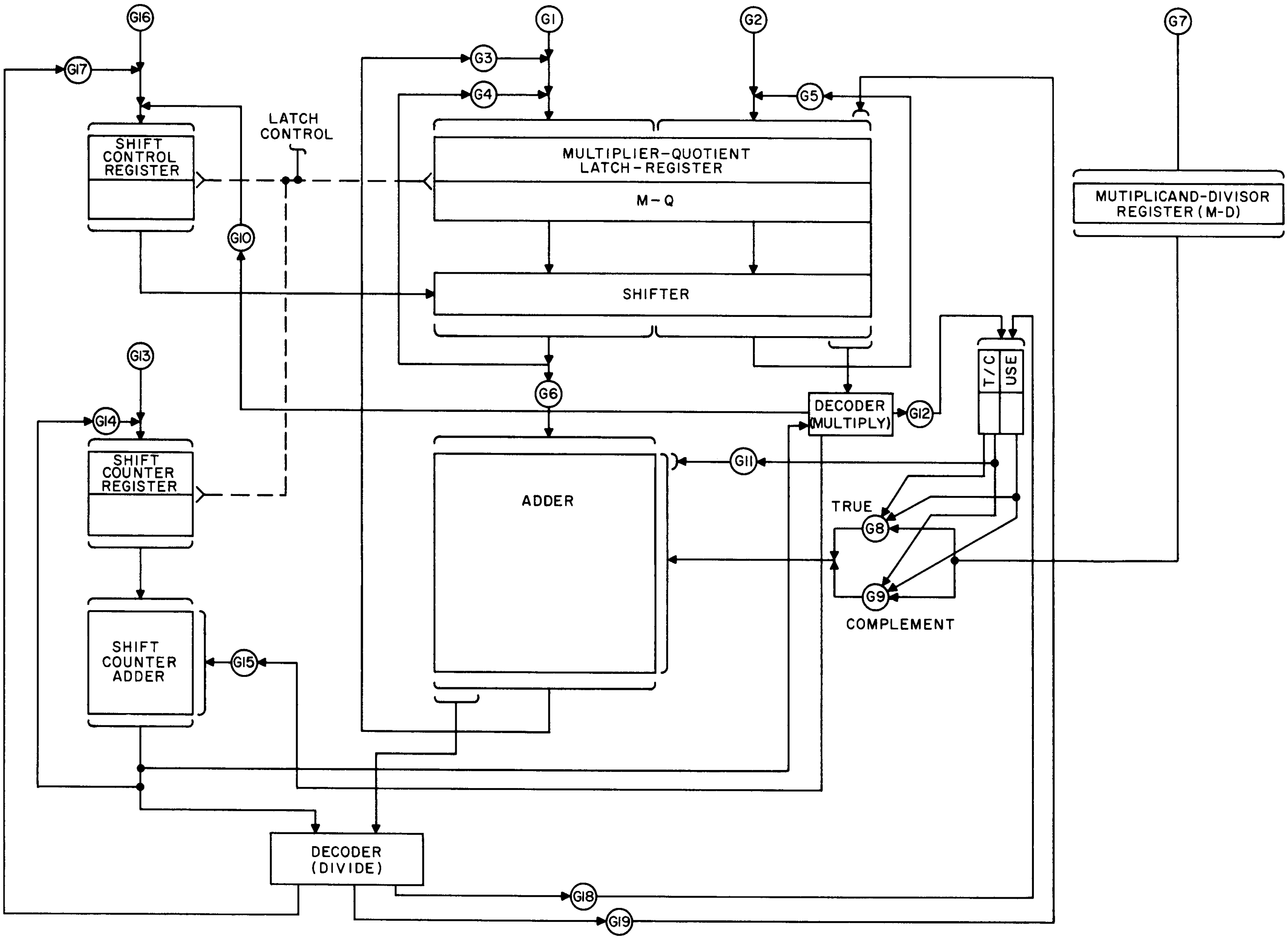


FIGURE 5. Computer arithmetic system.

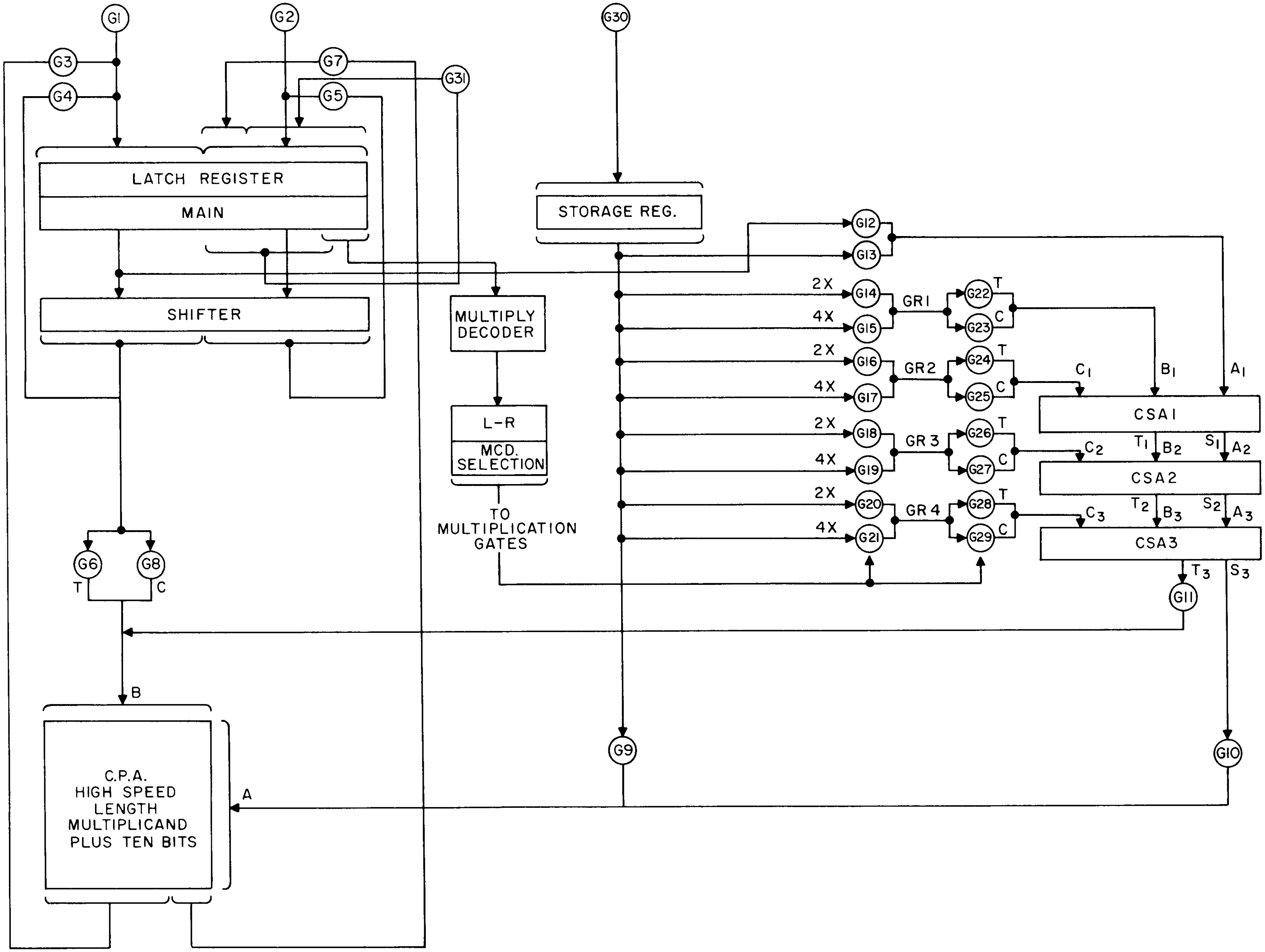
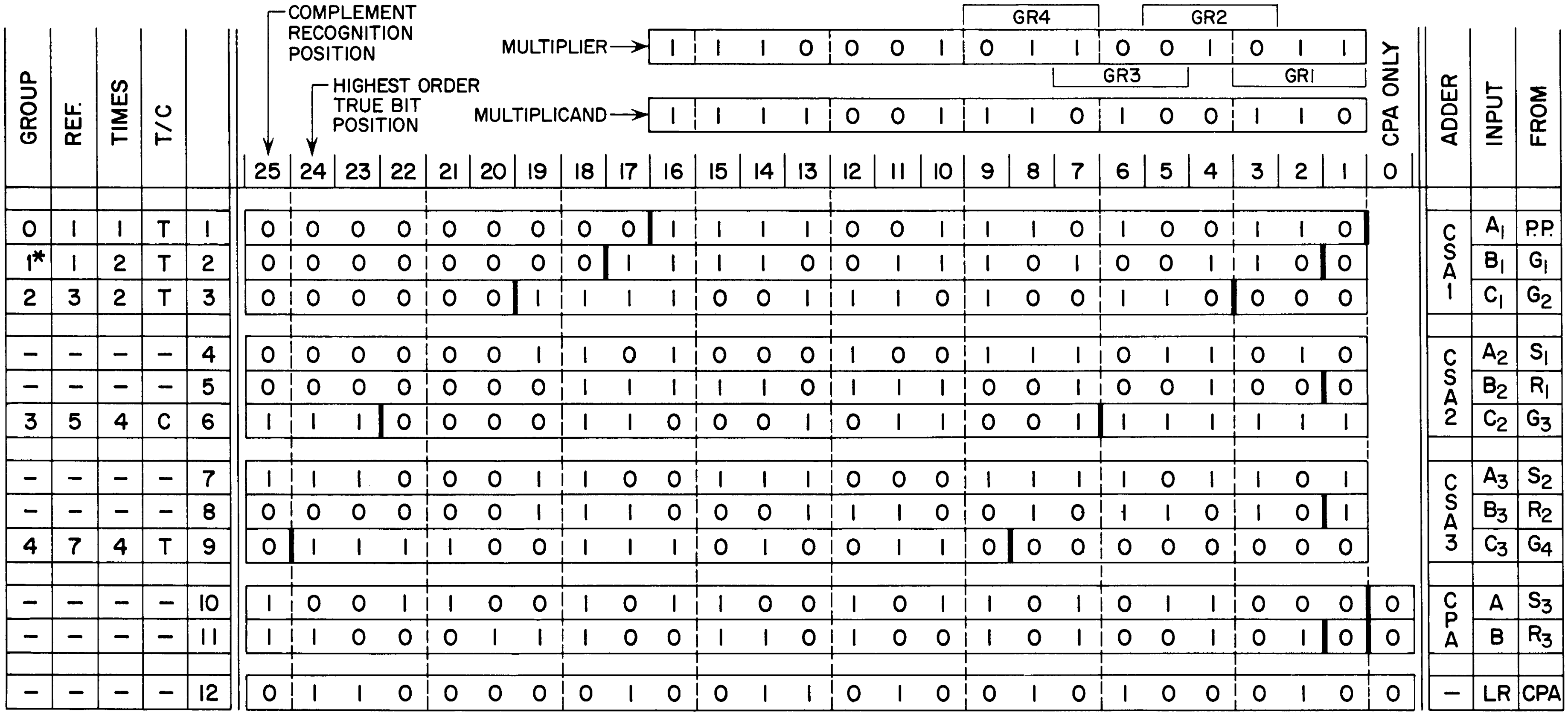


FIGURE 6. High-speed multiplication system.


COMPLEMENT RECOGNITION POSITION
HIGHEST ORDER TRUE BIT POSITION
GR4
GR2
GR3
GR1

MULTIPLIER → 1 1 1 0 0 0 1 0 1 1 0 0 1 0 1 1

MULTIPLICAND → 1 1 1 1 0 0 1 1 1 0 1 0 0 1 1 0

GROUP	REF.	TIMES	T/C		25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	CPA ONLY 0	ADDER	INPUT	FROM
0	1	1	T	1	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	1	1	0	1	0	0	1	1	0		CSA 1	A_1	P.P.
1*	1	2	T	2	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0			B_1	G_1
2	3	2	T	3	0	0	0	0	0	0	1	1	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0	0	0			C_1	G_2
–	–	–	–	4	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0	1	1	1	0	1	1	0	1	0		CSA 2	A_2	S_1
–	–	–	–	5	0	0	0	0	0	0	0	1	1	1	1	1	0	1	1	1	0	0	1	0	0	1	0	0	0			B_2	R_1
3	5	4	C	6	1	1	1	0	0	0	0	1	1	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1			C_2	G_3
–	–	–	–	7	1	1	1	0	0	0	1	1	0	0	1	1	1	0	0	0	1	1	1	1	0	1	1	0	1		CSA 3	A_3	S_2
–	–	–	–	8	0	0	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0	1	0	1	1	0	1	0	1			B_3	R_2
4	7	4	T	9	0	1	1	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0			C_3	G_4
–	–	–	–	10	1	0	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	1	0	1	1	0	0	0	0	CPA	A	S_3
–	–	–	–	11	1	1	0	0	0	1	1	1	0	0	1	1	0	1	0	0	1	0	1	0	0	1	0	1	0	0		B	R_3
–	–	–	–	12	0	1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0	1	0	0	0	1	0	0	–	LR	CPA

*SPECIAL DECODING

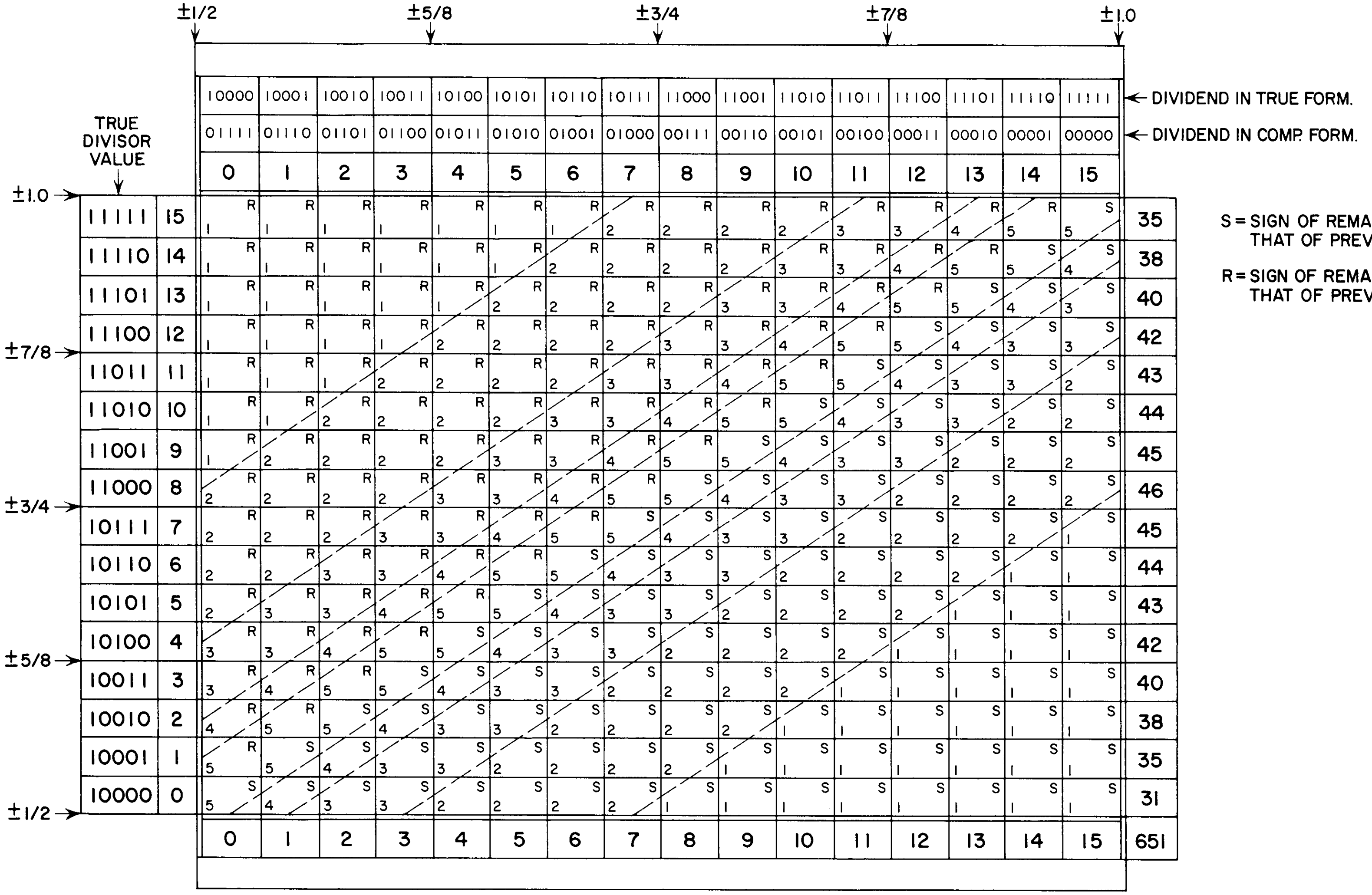
FIGURE 7. First cycle of multiplication example using carry-save adders.

		25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ADDER	INPUT	FROM
1								0	0	0	1	1	1	1	0	0	1	1	1	0	1	0	0		①			CSA 1	A_1	P.P.
2								0	0	1	1	1	1	0	0	1	1	1	0	1	0	0	1		②				B_1	G_1
3						⑨		1	1	1	1	0	0	1	1	1	0	1	0	0	1	1	0						C_1	G_2
4					⑫	0	0	1	1	0	1	0	0	0	1	0	0	1	1	1	0	③						CSA 2	A_2	S_1
5					⟨9⟩	0	0	0	1	1	1	1	1	0	1	1	1	0	0	1	0	④	⑤						B_2	R_1
6				⑩		0	0	0	1	1	0	0	0	1	0	1	1	0	0	1	1								C_2	G_2
7			⑬	0	0	0	0	1	1	0	0	1	1	1	0	0	0	1	1	⑥								CSA 3	A_3	S_2
8			⟨10⟩	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	1	⑦	⑧								B_3	R_2
9		⑪		1	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0										C_3	G_3
																					⟨6⟩	⟨3⟩			⟨1⟩		⑮			
10	⑭	0	0	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	1	1	1	1	1	1	0	0	CPA	A	S_3
11	⟨11⟩	0	1	0	0	0	1	1	1	0	0	1	1	0	1	0	0	1	0	0	1	0	0	1	0	0	0		B	R_3
																			⑯	⟨7⟩	⟨8⟩	⟨4⟩	⟨5⟩		⟨2⟩		⑮			
12		0	1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0	1	0	0	0	1	0	0	—	LR	CPA

FIGURE 8. Modified high-speed multiplication adder system.

			1/2				5/8				3/4				7/8			1.0	
			01111	01110	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	DIVIDEND COMPLEMENT
	DIVISOR TRUE		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1.0	11111	15	01110	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	11111	15
	11110	14	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	11111	11110	14
	11101	13	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	11111	11110	11101	13
	11100	12	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	11111	11110	11101	11100	12
7/8	11011	11	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	11111	11110	11101	11100	11011	11
	11010	10	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	11111	11110	11101	11100	11011	11010	10
	11001	9	01000	00111	00110	00101	00100	00011	00010	00001	00000	11111	11110	11101	11100	11011	11010	11001	9
	11000	8	00111	00110	00101	00100	00011	00010	00001	00000	11111	11110	11101	11100	11011	11010	11001	11000	8
3/4	10111	7	00110	00101	00100	00011	00010	00001	00000	11111	11110	11101	11100	11011	11010	11001	11000	10111	7
	10110	6	00101	00100	00011	00010	00001	00000	11111	11110	11101	11100	11011	11010	11001	11000	10111	10110	6
	10101	5	00100	00011	00010	00001	00000	11111	11110	11101	11100	11011	11010	11001	11000	10111	10110	10101	5
	10100	4	00011	00010	00001	00000	11111	11110	11101	11100	11011	11010	11001	11000	10111	10110	10101	10100	4
5/8	10011	3	00010	00001	00000	11111	11110	11101	11100	11011	11010	11001	11000	10111	10110	10101	10100	10011	3
	10010	2	00001	00000	11111	11110	11101	11100	11011	11010	11001	11000	10111	10110	10101	10100	10011	10010	2
	10001	1	00000	11111	11110	11101	11100	11011	11010	11001	11000	10111	10110	10101	10100	10011	10010	10001	1
1/2	10000	0	11111	11110	11101	11100	11011	11010	11001	11000	10111	10110	10101	10100	10011	10010	10001	10000	0
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

FIGURE 9. Division table, divisor true, dividend complement, using one times divisor.


±1/2
±5/8
±3/4
±7/8
±1.0
±1.0
±7/8
±3/4
±5/8
±1/2

		10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111	← DIVIDEND IN TRUE FORM.
		01111	01110	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	← DIVIDEND IN COMP. FORM.
TRUE DIVISOR VALUE		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
11111	15	1 R	1 R	1 R	1 R	1 R	1 R	1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	35
11110	14	1 R	1 R	1 R	1 R	1 R	1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	38
11101	13	1 R	1 R	1 R	1 R	1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	40
11100	12	1 R	1 R	1 R	1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	42
11011	11	1 R	1 R	1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	43
11010	10	1 R	1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	44
11001	9	1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	45
11000	8	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	46
10111	7	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	1 S	45
10110	6	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	1 S	1 S	44
10101	5	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	1 S	1 S	1 S	43
10100	4	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	1 S	1 S	1 S	1 S	42
10011	3	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	1 S	1 S	1 S	1 S	1 S	40
10010	2	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	1 S	1 S	1 S	1 S	1 S	1 S	38
10001	1	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	1 S	1 S	1 S	1 S	1 S	1 S	1 S	35
10000	0	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	1 S	1 S	1 S	1 S	1 S	1 S	1 S	1 S	31
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	651

S = SIGN OF REMAINDER SAME AS THAT OF PREVIOUS DIVIDEND

R = SIGN OF REMAINDER REVERSE THAT OF PREVIOUS DIVIDEND

	SHIFTS					
SHIFT LENGTH	1	2	3	4	5	TOTAL
NUMBER	64	80	52	29	31	256
PERCENT	25.0	31.3	20.4	11.3	12.0	100.0

$\frac{651}{256}$ = 2.54 BITS/CYCLE

$\frac{48}{2.54}$ = 19.0 AVERAGE SHIFT CYCLES FOR 48 BIT QUOTIENT WITH 5 BIT DIVISOR

FIGURE 10. Division table using one times divisor with five-bit divisor.

±1/2 ±5/8 ±3/4 ±7/8 ±1.0

1/2 TIMES DIVISOR

			10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111	DIVIDEND IN TRUE FORM.
			01111	01110	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	DIVIDEND IN COMP. FORM.
	TRUE DIVISOR VALUE		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
±1.0	11111	15	S 4	S 3	S 3	S 2	S 2	S 2	S 2	S 1	S 1	S 1	S 1	S 1	S 1	S 1	S 1		15
	11110	14	S 4	S 3	S 3	S 2	S 2	S 2	S 2	S 1	S 1	S 1	S 1	S 1	S 1	S 1		R 0	14
	11101	13	S 3	S 3	S 2	S 2	S 2	S 2	S 1	S 1	S 1	S 1	S 1	S 1	S 1		R 0	R 0	13
	11100	12	S 3	S 3	S 2	S 2	S 2	S 2	S 1	S 1	S 1	S 1	S 1	S 1		R 0	R 0	R 0	12
±7/8	11011	11	S 3	S 2	S 2	S 2	S 2	S 1	S 1	S 1	S 1	S 1	S 1		R 0	R 0	R 0	R 0	11
	11010	10	S 3	S 2	S 2	S 2	S 2	S 1	S 1	S 1	S 1	S 1		R 0	R 0	R 0	R 0	R 0	10
	11001	9	S 2	S 2	S 2	S 2	S 1	S 1	S 1	S 1	S 1		R 0	R 0	R 0	R 0	R 0	R 0	9
	11000	8	S 2	S 2	S 2	S 2	S 1	S 1	S 1	S 1		R 0	R 0	R 0	R 0	R 0	R 0	R 0	8
±3/4	10111	7	S 2	S 2	S 2	S 1	S 1	S 1	S 1		R 0	R 0	R 0	R 0	R 0	R 0	R 1	R 1	7
	10110	6	S 2	S 2	S 2	S 1	S 1	S 1		R 0	R 0	R 0	R 0	R 0	R 1	R 1	R 1	R 1	6
	10101	5	S 2	S 2	S 1	S 1	S 1		R 0	R 0	R 0	R 0	R 1	R 1	R 1	R 1	R 1	R 1	5
	10100	4	S 2	S 2	S 1	S 1		R 0	R 0	R 0	R 1	R 1	R 1	R 1	R 1	R 1	R 1	R 1	4
±5/8	10011	3	S 2	S 1	S 1		R 0	R 0	R 1	R 1	R 1	R 1	R 1	R 1	R 1	R 1	R 2	R 2	3
	10010	2	S 2	S 1		R 0	R 1	R 1	R 1	R 1	R 1	R 1	R 1	R 1	R 2	R 2	R 2	R 2	2
	10001	1	S 1		R 1	R 1	R 1	R 1	R 1	R 1	R 1	R 1	R 2	R 2	R 2	R 2	R 3	R 3	1
±1/2	10000	0		R 1	R 1	R 1	R 1	R 1	R 1	R 1	R 2	R 2	R 2	R 2	R 3	R 3	R 4	R 5	0
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

2 TIMES DIVISOR

FIGURE 11. Division tables using 2.0 and 1/2 times divisor.

±1/2 ±5/8 ±3/4 ±7/8 ±1.0

← 1/2 TIMES DIVISOR → ← 1 TIMES DIVISOR →

	TRUE DIVISOR VALUE		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	OPTIMUM AND DOUBLE ADDER	CODED MULTIPLE SELECTION
		DIVIDEND IN TRUE FORM	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111		
		DIVIDEND IN COMP. FORM	01111	01110	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000		
±1.0	11111	15	4 S	3 S	3 S	2 S	2 $\boxed{1}$ S	2 $\boxed{1}$ S	2 $\boxed{1}$ S	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	46	43
	11110	14	4 S	3 S	3 S	2 S	2 $\boxed{1}$ S	2 $\boxed{1}$ S	② R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	48	46
	11101	13	3 S	3 S	2 S	2 S	2 $\boxed{1}$ S	② R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	47	46
	11100	12	3 S	3 S	2 S	2 S	② R	② R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	48	48
±7/8	11011	11	3 $\boxed{1}$ S	2 $\boxed{1}$ S	2 $\boxed{1}$ S	② R	② R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	47	43
	11010	10	3 $\boxed{1}$ S	2 $\boxed{1}$ S	② R	② R	② R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	47	44
	11001	9	2 $\boxed{1}$ S	② R	② R	② R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	46	45
	11000	8	② R	② R	② R	② R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	46	46
±3/4	10111	7	② R	② R	② R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	① R	45	45
	10110	6	② R	② R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	① R	① R	44	44
	10101	5	② R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	① R	① R	① R	43	43
	10100	4	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	① R	① R	① R	① R	42	42
±5/8	10011	3	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	① R	① R	① R	2 R	2 R	42	42
	10010	2	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	① R	① R	2 R	2 R	2 R	2 R	42	42
	10001	1	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	① R	2 $\boxed{1}$ R	2 $\boxed{1}$ R	2 R	2 R	3 R	3 R	43	41
	10000	0	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	2 $\boxed{1}$ R	2 $\boxed{1}$ R	2 $\boxed{1}$ R	2 $\boxed{1}$ R	3 R	3 R	4 R	5 R	46	42
±1/2			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	722	702

← 1 TIMES DIVISOR → ← 2 TIMES DIVISOR →

OPTIMUM & DOUBLE ADDER

$\frac{722}{256}$ = 2.82 = BITS/CYCLE

$\frac{48}{2.82}$ = 17.0 = AVERAGE SHIFT CYCLES FOR 48 BIT QUOTIENT WITH 5 BIT DIVISOR

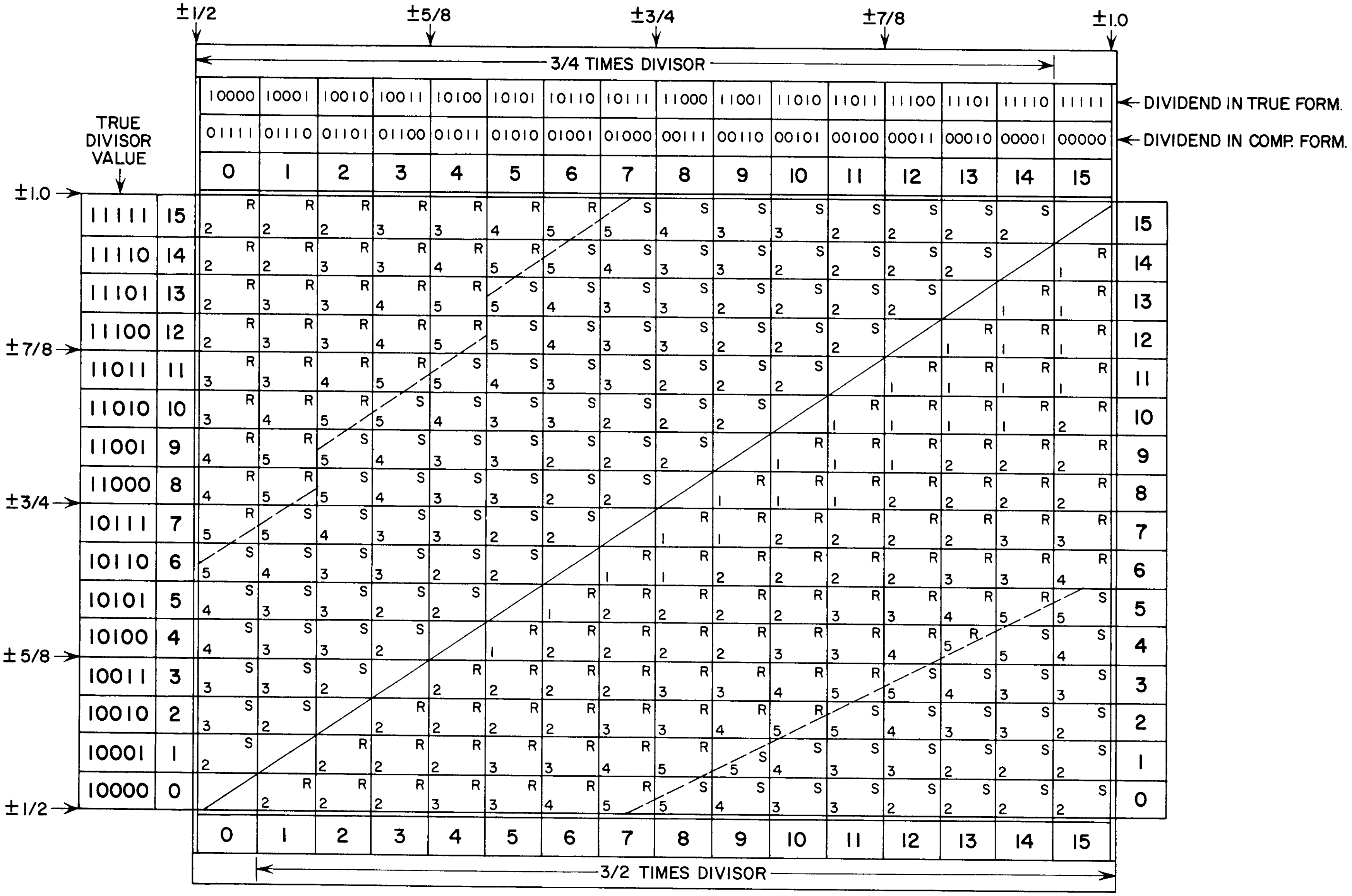
CODED MULTIPLE SELECTION

$\frac{702}{256}$ = 2.74 BITS/CYCLE

$\frac{48}{2.74}$ = 17.5 = AVERAGE SHIFT CYCLES FOR 48 BIT QUOTIENT WITH 5 BIT DIVISOR

	SHIFTS - DOUBLE ADDER						SHIFTS - CODED SELECTION					
SHIFT LENGTH	1	2	3	4	5	TOTAL	1	2	3	4	5	TOTAL
NUMBER	16	110	66	32	32	256	34	94	64	32	32	256
PERCENT	6.2	43.0	25.8	12.5	12.5	100.0	13.3	36.7	25.0	12.5	12.5	100.0

FIGURE 12. Division table using 2.0, 1.0, 1/2 times divisor with optimum coding..



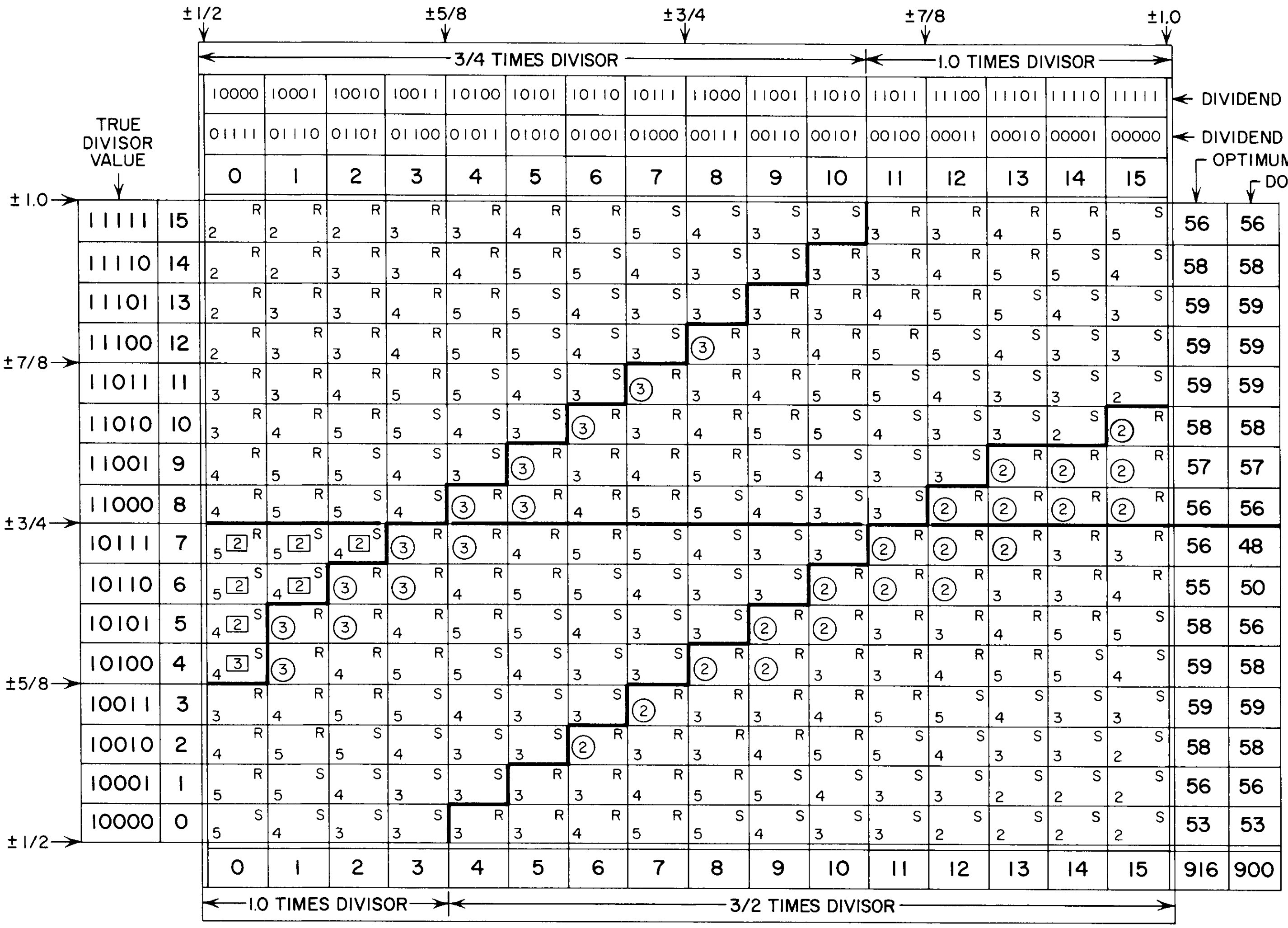
±1/2 ±5/8 ±3/4 ±7/8 ±1.0

3/4 TIMES DIVISOR

	TRUE DIVISOR VALUE																		
			10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111	DIVIDEND IN TRUE FORM.
			01111	01110	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	DIVIDEND IN COMP. FORM.
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
±1.0	11111	15	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S		15
	11110	14	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S		1 R	14
	11101	13	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S		1 R	1 R	13
±7/8	11100	12	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S		1 R	1 R	1 R	12
	11011	11	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S		1 R	1 R	1 R	1 R	11
	11010	10	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S		1 R	1 R	1 R	1 R	2 R	10
	11001	9	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S		1 R	1 R	1 R	2 R	2 R	2 R	9
±3/4	11000	8	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S		1 R	1 R	1 R	2 R	2 R	2 R	2 R	8
	10111	7	5 R	5 S	4 S	3 S	3 S	2 S	2 S		1 R	1 R	2 R	2 R	2 R	2 R	3 R	3 R	7
	10110	6	5 S	4 S	3 S	3 S	2 S	2 S		1 R	1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	6
	10101	5	4 S	3 S	3 S	2 S	2 S		1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	5
±5/8	10100	4	4 S	3 S	3 S	2 S		1 R	2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	4
	10011	3	3 S	3 S	2 S		2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	3
	10010	2	3 S	2 S		2 R	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2
	10001	1	2 S		2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	1
±1/2	10000	0		2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	0
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

3/2 TIMES DIVISOR

FIGURE 13. Division tables using 3/2 and 3/4 times divisor.



	TRUE DIVISOR VALUE		±1/2				±5/8				±3/4				±7/8			±1.0		
			3/4 TIMES DIVISOR											1.0 TIMES DIVISOR						
		DIVIDEND IN TRUE FORM	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111		
		DIVIDEND IN COMP. FORM	01111	01110	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	OPTIMUM	DOUBLE ADDITION
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
±1.0	11111	15	2 R	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	3 R	3 R	4 R	5 R	5 S	56	56
	11110	14	2 R	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	3 R	3 R	4 R	5 R	5 S	4 S	58	58
	11101	13	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	3 R	3 R	4 R	5 R	5 S	4 S	3 S	59	59
±7/8	11100	12	2 R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	③ R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	59	59
	11011	11	3 R	3 R	4 R	5 R	5 S	4 S	3 S	③ R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	59	59
	11010	10	3 R	4 R	5 R	5 S	4 S	3 S	③ R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	② R	58	58
	11001	9	4 R	5 R	5 S	4 S	3 S	③ R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	② R	② R	② R	57	57
±3/4	11000	8	4 R	5 R	5 S	4 S	③ R	③ R	4 R	5 R	5 S	4 S	3 S	3 S	② R	② R	② R	② R	56	56
	10111	7	5 $\boxed{2}$ R	5 $\boxed{2}$ S	4 $\boxed{2}$ S	③ R	③ R	4 R	5 R	5 S	4 S	3 S	3 S	② R	② R	② R	3 R	3 R	56	48
	10110	6	5 $\boxed{2}$ S	4 $\boxed{2}$ S	③ R	③ R	4 R	5 R	5 S	4 S	3 S	3 S	② R	② R	② R	3 R	3 R	4 R	55	50
	10101	5	4 $\boxed{2}$ S	③ R	③ R	4 R	5 R	5 S	4 S	3 S	3 S	② R	② R	3 R	3 R	4 R	5 R	5 S	58	56
±5/8	10100	4	4 $\boxed{3}$ S	③ R	4 R	5 R	5 S	4 S	3 S	3 S	② R	② R	3 R	3 R	4 R	5 R	5 S	4 S	59	58
	10011	3	3 R	4 R	5 R	5 S	4 S	3 S	3 S	② R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	59	59
	10010	2	4 R	5 R	5 S	4 S	3 S	3 S	② R	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	58	58
	10001	1	5 R	5 S	4 S	3 S	3 S	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	56	56
±1/2	10000	0	5 S	4 S	3 S	3 S	3 R	3 R	4 R	5 R	5 S	4 S	3 S	3 S	2 S	2 S	2 S	2 S	53	53
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	916	900
			1.0 TIMES DIVISOR				3/2 TIMES DIVISOR													

OPTIMUM

$\frac{916}{256} = 3.57 =$ BITS/CYCLE

$\frac{48}{3.57} = 13.4 =$ AVERAGE SHIFT CYCLES FOR 48 BIT QUOTIENT WITH 5 BIT DIVISOR

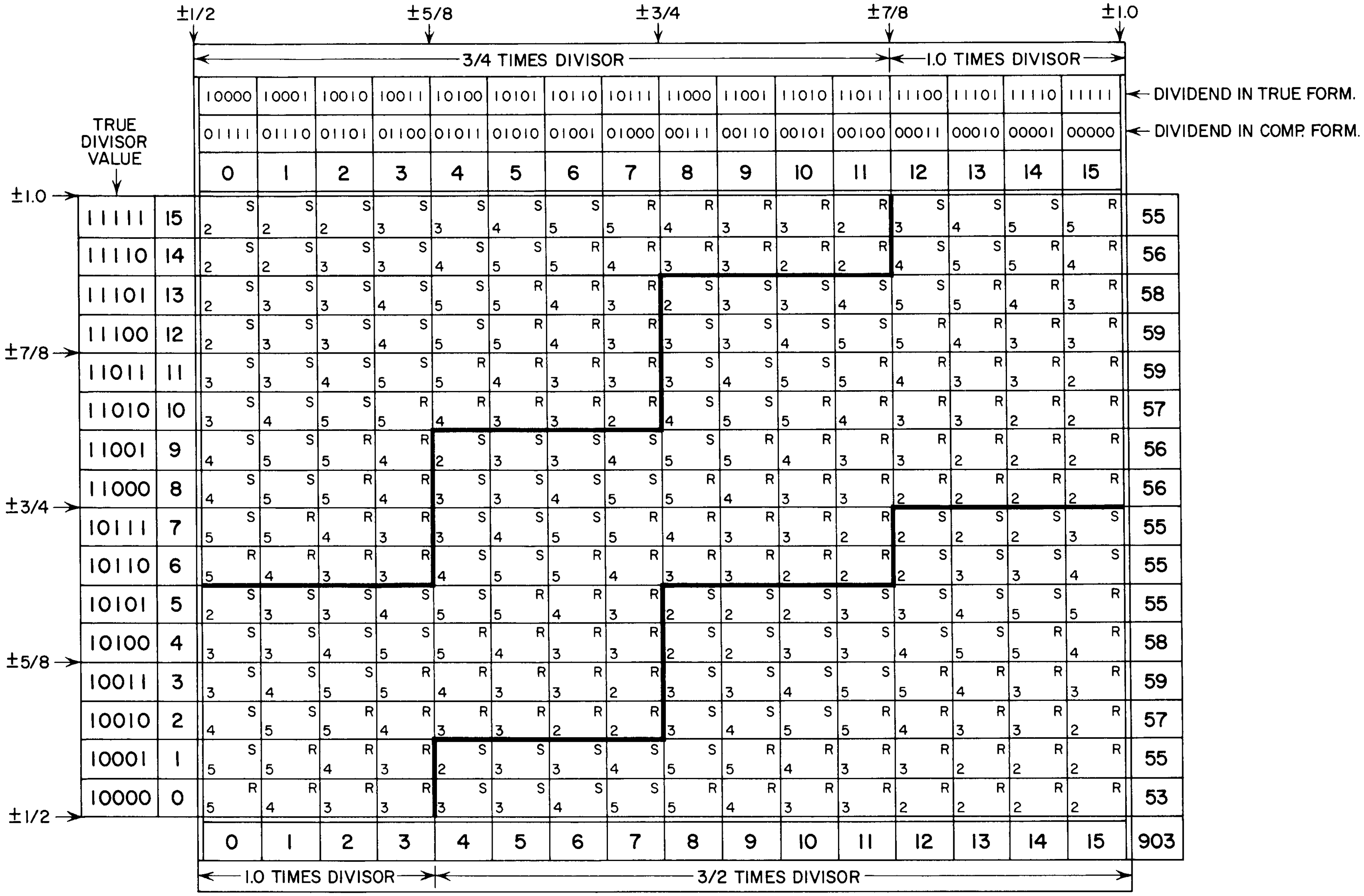
DOUBLE ADDER

$\frac{900}{256} = 3.51$ BITS/CYCLE

$\frac{40}{3.51} = 13.7 =$ AVERAGE SHIFT CYCLES FOR 48 BIT QUOTIENT WITH 5 BIT DIVISOR

	SHIFTS - OPTIMUM						SHIFTS - DOUBLE ADDER					
SHIFT LENGTH	1	2	3	4	5	TOTAL	1	2	3	4	5	TOTAL
NUMBER	0	37	96	61	62	256	0	43	97	57	59	256
PERCENT	0.0	14.5	37.5	23.8	24.2	100.0	0.0	16.8	37.9	22.3	23.0	100.0

FIGURE 14. Division table using 3/2, 1.0, 3/4 times divisor with optimum coding.



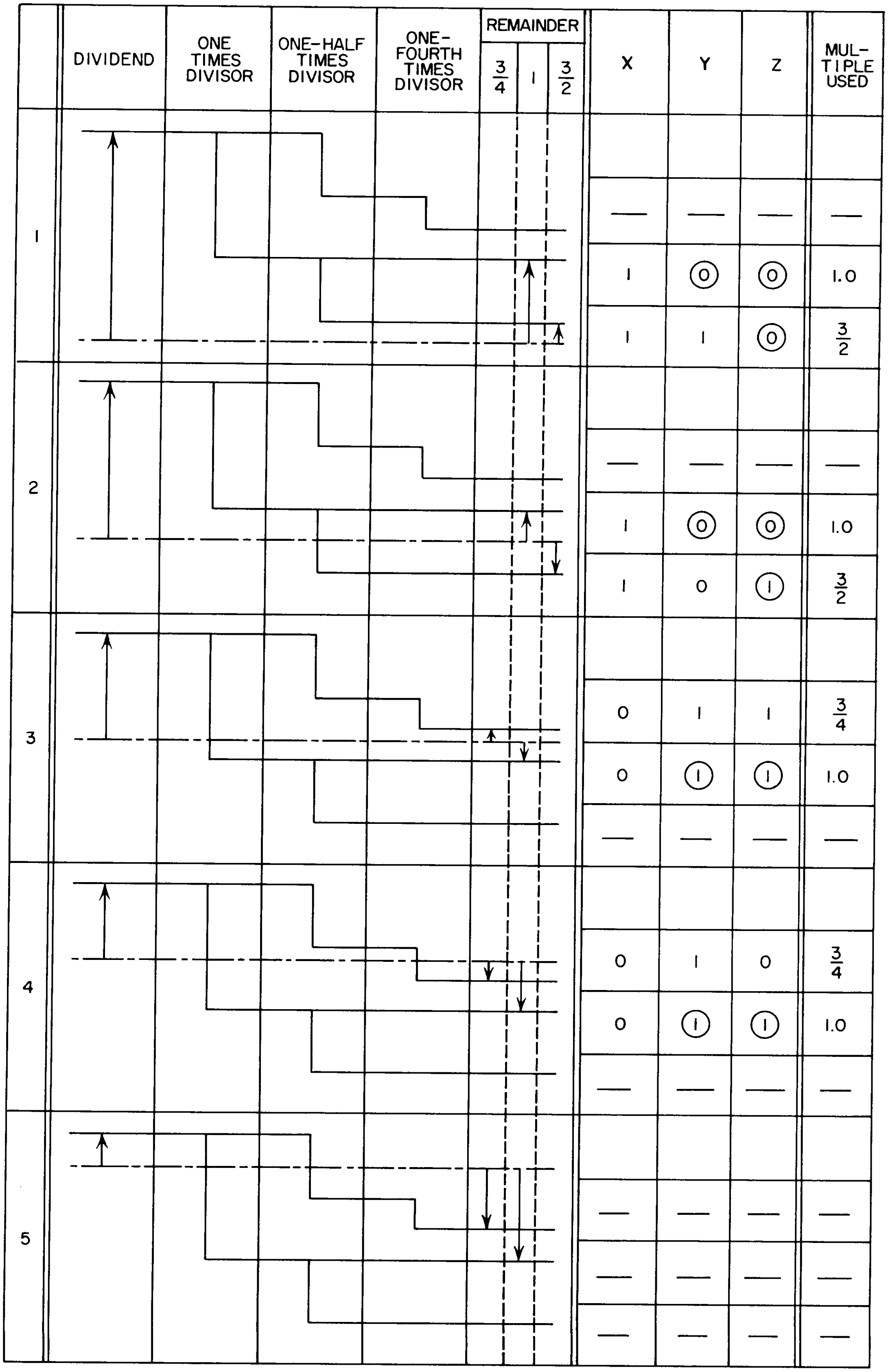
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
			±1/2				±5/8				±3/4				±7/8			±1.0	
			3/4 TIMES DIVISOR												1.0 TIMES DIVISOR				
			10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111	← DIVIDEND IN TRUE FORM.
	TRUE DIVISOR VALUE		01111	01110	01101	01100	01011	01010	01001	01000	00111	00110	00101	00100	00011	00010	00001	00000	← DIVIDEND IN COMP. FORM.
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
±1.0 →	11111	15	S 2	S 2	S 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	S 3	S 4	S 5	R 5	55
	11110	14	S 2	S 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	R 2	S 4	S 5	R 5	R 4	56
	11101	13	S 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	S 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	58
	11100	12	S 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	59
±7/8 →	11011	11	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	59
	11010	10	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	S 4	S 5	R 5	R 4	R 3	R 3	R 2	R 2	57
	11001	9	S 4	S 5	R 5	R 4	S 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	R 2	R 2	56
	11000	8	S 4	S 5	R 5	R 4	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	R 2	R 2	R 2	56
±3/4 →	10111	7	S 5	R 5	R 4	R 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	S 2	S 2	S 2	S 3	55
	10110	6	R 5	R 4	R 3	R 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	R 2	S 2	S 3	S 3	S 4	55
	10101	5	S 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	S 2	S 2	S 2	S 3	S 3	S 4	S 5	R 5	55
	10100	4	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	S 2	S 2	S 3	S 3	S 4	S 5	R 5	R 4	58
±5/8 →	10011	3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	59
	10010	2	S 4	S 5	R 5	R 4	R 3	R 3	R 2	R 2	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	57
	10001	1	S 5	R 5	R 4	R 3	S 2	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	R 2	R 2	55
±1/2 →	10000	0	R 5	R 4	R 3	R 3	S 3	S 3	S 4	S 5	R 5	R 4	R 3	R 3	R 2	R 2	R 2	R 2	53
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	903
			1.0 TIMES DIVISOR				3/2 TIMES DIVISOR												

	SHIFTS					
SHIFT LENGTH	1	2	3	4	5	TOTAL
NUMBER	0	96	174	118	124	512
PERCENT	0.0	18.8	33.9	23.1	24.2	100.0

$\frac{903}{256}$ = 3.52 BITS/CYCLE

$\frac{48}{3.52}$ = 13.6 AVERAGE SHIFT CYCLES FOR 48 BIT QUOTIENT WITH 5 BIT DIVISOR

FIGURE 15. Division table using 3/2, 1.0, 3/4 times divisor with four-way by sixteen-way coding.



X	Y	Z	MULTIPLE USED
—	—	—	—
1	⓪	⓪	1.0
1	1	⓪	$\frac{3}{2}$
—	—	—	—
1	⓪	⓪	1.0
1	0	①	$\frac{3}{2}$
0	1	1	$\frac{3}{4}$
0	①	①	1.0
—	—	—	—
0	1	0	$\frac{3}{4}$
0	①	①	1.0
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—

FIGURE 16. Quotient development using 3/4, 1.0, 3/2 times divisor.

APPENDIX

Readers who do not already have a fairly extensive background in computer binary arithmetic may find that some of the assumed facts in the main text are not obvious. This appendix will attempt to remedy this by covering some of these items in more detail.

The Full Adder

The full adder, one version of which is illustrated in Figure 1,may be described by the following table.

INPUT	OUTPUT	SUM	CARRY	NOT SUM	NOT CARRY
A B C	- S R	S	R	$\bar{S}$	$\bar{R}$
0 0 0	- 0 0	- - -	- - -	$\bar{A}\,\bar{B}\,\bar{C}$	$\bar{A}\,\bar{B}\,\bar{C}$
1 0 0	- 1 0	$A\,\bar{B}\,\bar{C}$	- - -	- - -	$A\,\bar{B}\,\bar{C}$
0 1 0	- 1 0	$\bar{A}\,B\,\bar{C}$	- - -	- - -	$\bar{A}\,B\,\bar{C}$
1 1 0	- 0 1	- - -	$A\,B\,\bar{C}$	$A\,B\,\bar{C}$	- - -
0 0 1	- 1 0	$\bar{A}\,\bar{B}\,C$	- - -	- - -	$\bar{A}\,\bar{B}\,C$
1 0 1	- 0 1	- - -	$A\,\bar{B}\,C$	$A\,\bar{B}\,C$	- - -
0 1 1	- 0 1	- - -	$\bar{A}\,B\,C$	$\bar{A}\,B\,C$	- - -
1 1 1	- 1 1	A B C	A B C	- - -	- - -

From the above it may be seen that, with the application of the rules of Boolian algebra, the SUM and CARRY outputs may be described by any of the following equations.

$$S = (A\,\bar{B}\,\bar{C}) \vee (\bar{A}\,B\,\bar{C}) \vee (\bar{A}\,\bar{B}\,C) \vee (A\,B\,C)$$

$$S = (A\,\bar{B} \vee \bar{A}\,B\,)\,\bar{C} \vee (\bar{A}\,\bar{B} \vee A\,B)\,C$$

$$S = (A \veebar B) \veebar C$$

$$R = (\,A\,B\,\bar{C}) \vee (A\,\bar{B}\,C) \vee (\bar{A}\,B\,C\,) \vee (A\,B\,C)$$

$$R = (A \veebar B\,)\,C \vee (A\,B)$$

$$R = (A\,B) \vee (A\,C\,) \vee (B\,C)$$

Assume that an N-position adder is made by connecting N full adders of the type shown in Figure 1 in series with the R output of each stage connected to the C input of the next higher order stage. In such an adder, if one input number were all *ones*, and the other number were a single *one* in the lowest order position, a carry would be generated in the first position, then go through each position in turn, producing a change in each position it passed through, until it finally reached the Nth position. Such an addition would require 2N logical transit times to complete. Usually, however, carries will be generated at several points within the adder simultaneously, and each generation point will be the start of a ripple path. Each ripple path will continue until it reaches a stage with a *zero zero* input (which will not pass a carry) or a *one one* (which was the starting point of another ripple path). The required time to complete the addition will then be determined by the longest ripple path within the adder for that particular pair of input numbers. This can be shown, assuming random distribution within the numbers being added, to have an average value of less than seven for a hundred-bit adder. If means are available to recognize when all carry rippling is completed, this represents a considerable time reduction over the time required for a full-length ripple carry.

Completion Recognition Adders

It was stated in the description of the operation of a completion recognition adder that to prevent false indications of completion the two inputs must enter the adder simultaneously; once the operation has started no changes may be made in the inputs; and both inputs must be changed to all *zeros* or all *ones* before the next addition may be performed. The reason for this may be seen from the following examples. Assume that the addition shown on the left is performed; then, without first clearing the adder, the input is changed as shown to give the resulting sum. In the first example

$$\begin{array}{cl@{\qquad\qquad}cl}
0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0 & \text{A} & 0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0 & \text{A} \\
0\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,0 & \text{B} & 0\,0\,0\,1\,1\,0\,0\,1\,1\,0\,1\,0 & \text{B} \\
\hline
0\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,0 & \text{S} & 1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 & \text{S}
\end{array}$$

a no-carry signal generated in position one ripples down through positions two through ten into position eleven. There are transmit signals generated in positions two through eleven inclusive and a new no-carry signal originated in position twelve. Since the second example involves a change in position two only, other transmit signals are not disturbed, and the no-carry lines of the various stages do not change simultaneously. The change from ON to OFF, however, ripples from one stage to the next in the same manner that the change from ON to OFF did in the first example. Simultaneously with the termination of the no-carry signal from position two, a carry signal is generated and started along the carry line. Thus the two changes are moving from one stage to the next simultaneously, one going ON at the same time the other goes OFF. This could result in the completion signal not going OFF at all, even though the sum is in the process of being changed. Changing the inputs to all zeros or all ones causes all of the transmit signals to go OFF, which means that there are no ripple groups to go OFF in sequence when the next number is entered. Obviously, making a change in either of the inputs after an addition has started would be the same as going from one completed addition to the next without clearing in between. The term"clear" means making the inputs to the adder either all zeros or all ones.

Multiplication

It was stated earlier that an average shift of three bits may be expected when using the variable shift method of multiplication described. The basis for this statement is shown below. The table shows the eight possible combinations of the next four bits following a shift across zeros.

(1)	$0\ 0 - \underline{1\ 1\ 1}\ 1$	3+
(2)	$0\ 0 - \underline{1\ 1\ 1}\ 0$	3+
(3)	$0\ 0 - \underline{1\ 1\ 0}\ 1$	3
(4)	$0\ 0 - \underline{1\ 1}\ 0\ 0$	2
(5)	$0\ 0 - 1\ \underline{0}\ 1\ 1$	1
(6)	$0\ 0 - 1\ \underline{0\ 1}\ 0$	2
(7)	$0\ 0 - 1\ \underline{0\ 0}\ 1$	2+
(8)	$0\ 0 - 1\ \underline{0\ 0}\ 0$	2+

Shift----------	(1)	(2)	(3)	(2+)	(3+)
Fraction------	1/8	1/4	1/8	1/4	1/4

From the table it may be seen that fifty percent of the following shifts are across zeros and fifty percent across ones. These in turn split into two groups each, one of which contains known numbers of shifts, while the other calls for a known minimum plus a number to be determined from additional bits. The two following tables describe the breakdown of the two incomplete groups. The first table refers to the group shifting across ones, while the second table refers to the group shifting across zeros.

(1 - 1)	0 0 - 1 1 1 - 1 1 1	(3+)	2+
(1 - 2)	0 0 - 1 1 1 - 1 1 0	(3+)	2+
(1 - 3)	0 0 - 1 1 1 - 1 0 1	(3+)	2

(1 - 4)	0 0 - 1 1 1 - 1 0 0	(3+)	1
(2 - 5)	0 0 - 1 1 1 - 0 1 1	(3+)	1
(2 - 6)	0 0 - 1 1 1 - 0 1 0	(3+)	1
(2 - 7)	0 0 - 1 1 1 - 0 0 1	(3+)	0
(2 - 8)	0 0 - 1 1 1 - 0 0 0	(3+)	0

Additional Shift ---	(0)	(1)	(2)	(2+)
Fraction ---------	1/4	3/8	1/8	1/4

(7 - 1)	$0\ 0 - 1\ \underline{0\ 0} - 1\ 1\ 1$	(2+)	0
(7 - 2)	$0\ 0 - 1\ \underline{0\ 0} - 1\ 1\ 0$	(2+)	0
(7 - 3)	$0\ 0 - 1\ \underline{0\ 0} - \underline{1}\ 0\ 1$	(2+)	1
(7 - 4)	$0\ 0 - 1\ \underline{0\ 0} - \underline{1}\ 0\ 0$	(2+)	1
(8 - 5)	$0\ 0 - 1\ \underline{0\ 0} - \underline{0}\ 1\ 1$	(2+)	1
(8 - 6)	$0\ 0 - 1\ \underline{0\ 0} - \underline{0\ 1}\ 0$	(2+)	2
(8 - 7)	$0\ 0 - 1\ \underline{0\ 0} - \underline{0\ 0}\ 1$	(2+)	2+
(8 - 8)	$0\ 0 - 1\ \underline{0\ 0} - \underline{0\ 0}\ 0$	(2+)	2+

Additional Shift ----	(0)	(1)	(2)	(2+)
Fraction ----------	1/4	3/8	1/8	1/4

The additional shift fractions are seen to be the same for both groups.

From the three preceding tables one table can be made that will give the fraction of the total number of shifts made that should have each shift value for random bit distribution. The one-position shift is taken from the first table, and is 1/8 of the total shifts. The two-position shift is determined by a combination of two figures. From the first table it is known that 1/4 of the shifts will be two-position and another quarter will be two or more (2+). From the third table it can be seen that one quarter of the latter are plus zero, which means that the total having a shift of two will be one-quarter plus one-quarter of one quarter, or five-sixteenths. The three position shift is composed of 1/8 from the first table, (1/4 x 3/8) from (2 + 1) and (1/4 x 1/4) from (3 + 0).

SHIFT								
1						1/8	=	1/8
2				1/4) +	(1/4	x 1/4)	=	5/16
3		1/8 +	(1/4	x 3/8) +	(1/4	x 1/4)	=	9/32
4	(1/4	x 1/8) +	(1/4	x 3/8) +	(1/16	x 1/4)	=	9/64
5	(1/4	x 1/8) +	(1/16	x 3/8) +	(1/16	x 1/4)	=	9/128
6	(1/16	x 1/8) +	(1/16	x 3/8) +	(1/64	x 1/4)	=	9/256
7	(1/16	x 1/8) +	(1/64	x 3/8) +	(1/64	x 1/4)	=	9/512
8	(1/64	x 1/8) +	(1/64	x 3/8) +	(1/256	x 1/4)	=	9/1024
9	(1/64	x 1/8) +	(1/256	x 3/8) +	(1/256	x 1/4)	=	9/2048
10	(1/256	x 1/8) +	(1/256	x 3/8) +	(1/1024	x 1/4)	=	9/4096

The four-position shift requires a double extension of the table for its third term. It is composed of (1/4 x 1/8) from (2 + 2), (1/4 x 3/8) from (3 + 1), and (1/4 x 1/4 x 1/4) from (2 + 2 + 0). This procedure may be continued indefinitely, but an examination of the sums at the right of the last table indicates a uniform progression that makes this unnecessary. Instead, the following rule may be used.

Then operating with multipliers having random bit distribution, the average shift distribution will be as follows. 1/8 will be shifts of one. 5/16 will be shifts of two. This leaves 9/16 which will be shifts of three or greater. Of these, half will be three-shift, half of the remainder will be four-shift, half of the remainder following that will be five-shift, etc.

This rule was developed on the assumption of starting by shifting across zeros. An initial assumption of shifting across ones would duplicate these results, giving exactly the same ratios. This must be true since the only difference in the rules for handling ones and zeros is that the locations of the words one and zero interchanged.

To determine the average shift, start with an assumed number of shifts, apply the preceding rule to determine the distribution among the various shift amounts, multiply the number of times each shift amount occurs by that shift amount, then add these products to determine the total number of positions shifted across. Divide the total number of shifts into the total number of positions shifted across to determine the average shift. It will be necessary to have one group which will include all shifts greater than some particular amount and assume some average value for these. This procedure is illustrated below. A total of 4096 shifts is assumed. All shifts of one through twelve are determined, and the remainder are grouped as greater than twelve.

SHIFTS	Number of Shifts	Number of Bits
1	512	512
2	1280	2560
3	1152	3456
4	576	2304
5	288	1440
6	144	864
7	72	504
8	36	288
9	18	162
10	9	90
11	4.5	49.5
12	2.25	27
12+	2.25	31.5
	4096.	12288.

This gives an average shift of three (12,288/4096) bits per shift cycle, based on the assumption that the shifter can supply any shift called for in one cycle. Usually this is not the case, and the figure must be modified accordingly. For example, if the maximum shift available is four, a desired shift of six would be obtained by one shift of four and one of two, while a desiredshift of thirteen would be three shifts of four followed by one of one. To correct for this, leave the number of bits shifted across the same, but increase the number of shifts required in the following manner. Divide the table into groups, each group having a size equal to the maximum allowable shift. Keep the number of shifts in the first group as it is, double the number in the second group, triple the number in the third group, etc. Add these results together, then divide the sum into the total number of bits to get the average shift. For example, assume a maximum shift of four.

Group 1	512	+	1280	+	1152	+	576	=	3520	3,520
Group 2	288	+	144	+	72	+	36	=	540	1,080
Group 3	18	+	9	+	4.5	+	2.25	=	33.75	101.25
Group 4	2.25							=	2.25	9
Sum										4,710.25

This gives an average shift of (12,288/4,710.25) = 2.60 bits.

A maximum shift of five gives (12,288/4378) = 2.81, while a maximum shift of six gives(12,288/4242.25) = 2.90. There is an additional slight adjustment due to the fact that the multiplier has a finite length which is not included.

Multiplication Using Carry-Save Adders

Figure 17 illustrates two versions of a combined carry-save carry-propagate adder. Each position has two carry input connections, one for use under carry-save conditions and the other for use under carry-propagate conditions. Carry look-ahead circuits can be used with the carry-propagate input.

Figure 18 illustrates the modifications required in the system when using three-bit multiplier groups. Figure 19 shows the use of storage registers on the outputs of the last carry-save adder. This costs an additional carry-save adder and two additional registers, but will give some increase in speed. The part of the partial product that is completed goes through the carry-propagate adder each cycle from the two registers, but the remainder of the partial product goes directly back to the carry-save adders from the S and R registers. The contents of the S and R registers (except the low-order group) only go to the CPA after the last pass through the carry-save adders.

Division

Figure 20 illustrates graphically the quotient development when using 1/2, 1.0, and 2.0 times the divisor. Examples are shown only for the condition of initial dividend true. Examples for the initial condition of dividend complement can be obtained from this by inverting all signs and interchanging ones and zeros in the list of quotient bits.

Example 1 assumes the initial dividend to be greater than twice the divisor. Since the divisor has a minimum value of one-half, this would make the initial dividend equal to or greater than one, which is contrary to the rules of division. Therefore, this condition would not occur.

In Example 2, the initial dividend is greater than one-times and less than two-times the divisor. The multiple chosen would have to be one of these two, the choice depending on which gives the smaller remainder. Either choice results in a one being entered as the quotient bit in position X. The selection of the one-times multiple would leave a true remainder, which will result in shifting across the zeros the next cycle, and will therefore result in a tentative zero being entered into position Y. If the two-times multiple is chosen, the remainder will be complement, which means shifting across zeros, and results in a tentative one being placed in position Y.. Obviously one of these tentative choices will be incorrect and will be changed on the next cycle. The circle around the bit value in column Y indicates that it is tentative and may be changed.

Example 3 shows a dividend less than one-time but greater than one-half times the divisor. Here the multiple chosen must be one of these two. Both must result in a zero quotient bit in position X. If the one-half times multiple is used, the fact that it results in a true remainder indicates that the next lower order quotient bit is a one. This is entered into position Y and is a correct quotient bit. If the next shift should be a one-position shift, the bit that was developed and placed in position Y must be preserved and entered into position X in place of the bit indicated by the output of the adder. If the remainder from the use of the one-times multiple is used, the one shown in position Y is tentative and may be changed if a one-position shift should result.

Example 4 shows an initial dividend having a value less than one-half that of the divisor. Since the true value of the divisor must have a leading one, this would require that the true value of the dividend be 0. 01xxx. Division rules require that a true dividend is always shifted across all leading zeros before performing an addition. Therefore this condition could only occur as a result of an incorrect shift, or as a result of starting with an unnormalized divisor. Results will be correct, but the number of cycles will be increased.

Figure 21 is an example of a division performed using one-times the divisor with shifting across ones and zeros. The divisor is shown at the top, while the initial dividend is shown in columns 3 through 31 of row 1. The leftmost unnumbered column contains three pieces of information about each addition. The top number of each section is the column number of Figure 10 that applies to that particular dividend (the row number will be 11 for the complete problem). The sign indicates whether an addition or subtraction (complement addition) is being performed. The third number is the sequence number for the operation. The shift counter operation is shown in the unnumbered column at the right. The complete operation requires 14 add cycles.

It should be noted that following cycle 13 the shift decoder will indicate six leading zeros, but the shift counter output is one, so the shift called for will be one.

Figure 22 shows the same problem performed using 2. 0, 1. 0, 1/2 times divisor with double addition. The information in the leftmost column refers to Figures 10, 11, and 12, and the letter indicates which adder output is used. The complete add operation requires eleven add cycles.

Figure 23 repeats the example using 3/2, 1. 0, 3/4 times the divisor with double addition. Figures 10, 13, and 14 are the reference figures. The number of addition cycles is reduced to nine. It should be noted that the low-order bit positions of Adder A serve no purpose except to supply the same low-order remainder positions from both adders. On the last cycle, one of the quotient bits goes through this adder. These two low-order positions are required in Adder B when the 3/4 times multiple is used.

Division Using Carry-Save Adder

It is possible to perform division using a carry-save adder, but it is very improbable that any increase in speed or reduction in cost will result from doing so.

Assuming the variable shift method of division that was described for use with one-times the divisor and shifting across ones and zeros, it is necessary to decode the high-order bits following an addition to determine the number of positions to shift. It is also necessary to know whether the result is in true or complement form. This may be done by connecting the output of the high-order bits of the carry-save adder to a short, high-speed, carry-propagate adder, and the output of this into a decoder. This represents four levels through the CSA plus six levels through the CPA, or a total of ten levels from the input of the decoder, as compared to fourteen levels for a hundred-bit carry-propagate adder with full carry look-ahead.

The first problem when using a carry-save adder for division is to determine whether the result is true or complement when it is near zero in absolute value. For example, assume that the following is the result of an addition, that the maximum available shift is five, and that the number of positions normally examined in the decoder is seven.

Sum..................	0 ' 1 1 1 1 1 1 1 ' 1 1 1 0 1
Carry................	0 ' 0 0 0 0 0 0 0 ' 0 0 1 0 1
Short Adder Sum	0 ' 1 1 1 1 1 1 1
Full Adder Sum	1 ' 0 0 0 0 0 0 0 ' 0 0 0 1 0

An examination of the seven-bit output of the short adder indicates that the result is complement and calls for a zero in the low-order position of the quotient, followed by a shift of greater than five with ones inserted in the quotient for each position shifted across. The full adder output shows that the correct procedure would have been to enter a one in the low-order position of the quotient, followed by a shift of more than five positions with zeros entered in the quotient for each position shifted across. This could have been recognized only by having a circuit that would recognize that a carry would be produced when the low-order parts of the carry-save adder outputs were combined. Circuits that will do this are of the same order of complexity and/ or require as much time as a full carry look-ahead for the adder.

The alternative solution to this problem is to act on the assumption that the information received from the short-adder is correct, then make such corrections as are found necessary. This can be done, and the procedure and equipment required will be described.

Without counting the overflow position, the short-adder sum should contain two more positions than the maximum capacity of the shifter. If its output is anything except all ones, then the indicated condition of true or complement is correct; but if the output is all ones, then the indicated condition of complement output may or may not be correct. This can result in the following condition in the quotient.

Developed Quotient............. 0 0 1 1 1 1 1 0 0 0 1

Correct Quotient............... 0 1 0 0 0 0 0 0 0 0 1

This condition will only occur when the string of ones is longer than the maximum allowable shift of the shifter, and the output of the short adder will contain the correct output on the cycle in which it includes the end of the string. Thus the example of the short adder output given previously would have been decoded as shift five positions across ones, following the setting of a zero in the quotient, and turn on the trigger indicating that an incomplete shift across ones is in process. Following the shift, the next short adder output would be 1 ' 0 0 0 0 0 1 0. The fact that this is a true result while the status trigger indicates an incomplete shift across ones would indicate that an incorrect choice was made and must be corrected. To correct the quotient, add one to it in the same column as the lowest order of the incorrect ones. One way to do this is to stop and enter the quotient into a carry-propagate adder together with the one to be added to it. This is not necessarily an unreasonable solution, as it should not occur on an average of more than once in 250 add cycles. The alternative is to have a double quotient register, each equipped with a shifter, and use one of them to store these additional ones, then perform the addition at the end. Since it is highly improbable that there would be more than one of these correction bits in any one quotient, it would probably be no more time-consuming and definitely less expensive to perform the addition to the quotient when the need was discovered.

There are two other possibilities to consider in relation to the decoding. The first of these is the following:

Short adder result............ 0 ' 1 1 1 0 1 1 1

Possible correct result .... 0 ' 1 1 1 1 0 0 0

This would be decoded as shift three across ones. It is possible that it should be shift four across ones. Shifting less than the maximum allowable amount will not cause an error, although it may result in an additional cycle. Therefore, this condition would be handled as though it were known that the short-adder result was the correct result.

The other possibility is shown next:

Short adder result .... 1 ' 0 0 0 0 1 1 1

Possible correct result .... 1 ' 0 0 0 1 0 0 0

Shifting too far before performing an addition can mix up both the remainder and the quotient. Therefore, if the short-adder output is true and composed of a single string of zeros followed by a single string of ones, the shift to be taken will be one less than the number of leading zeros, but never less than one. If there are any zeros following the highest order one , then the shift will be equal to the number of leading zeros.

The termination rules are the same as when using the carry-propagate adder, with the following exceptions..

(1) Following the last addition in the carry-save adder, the sum and carry outputs must be combined in a carry-propagate adder if a final remainder in normal form is desired.

(2) If an indeterminant result is obtained on what is supposed to be the termination cycle, the result must be checked in the carry-propagate adder to determine whether the remainder is true or complement and whether the low-order bits of the quotient are correct or not.

Figure 17 shows a possible combined carry-save carry-propagate adder which can can have carry look-ahead included if desired. If this were used in place of the adder shown in Figure 2, it would also be necessary to add another register, together with a shifter capable of performing those shifts used in divide and having a length sufficient to handle either the sum or carry output of the carry-save adder. Gating circuits would also be required with it.

The general impression is that in parallel binary operations, in division as in multiplication the use of a single carry-save adder is not a profitable investment. Also, it does not appear feasible to use carry-save adders in series for division as is done in multiplication. However, the advantages, disadvantages, and problems associated with the use of this type of adders in division have been described here for the benefit of those interested.

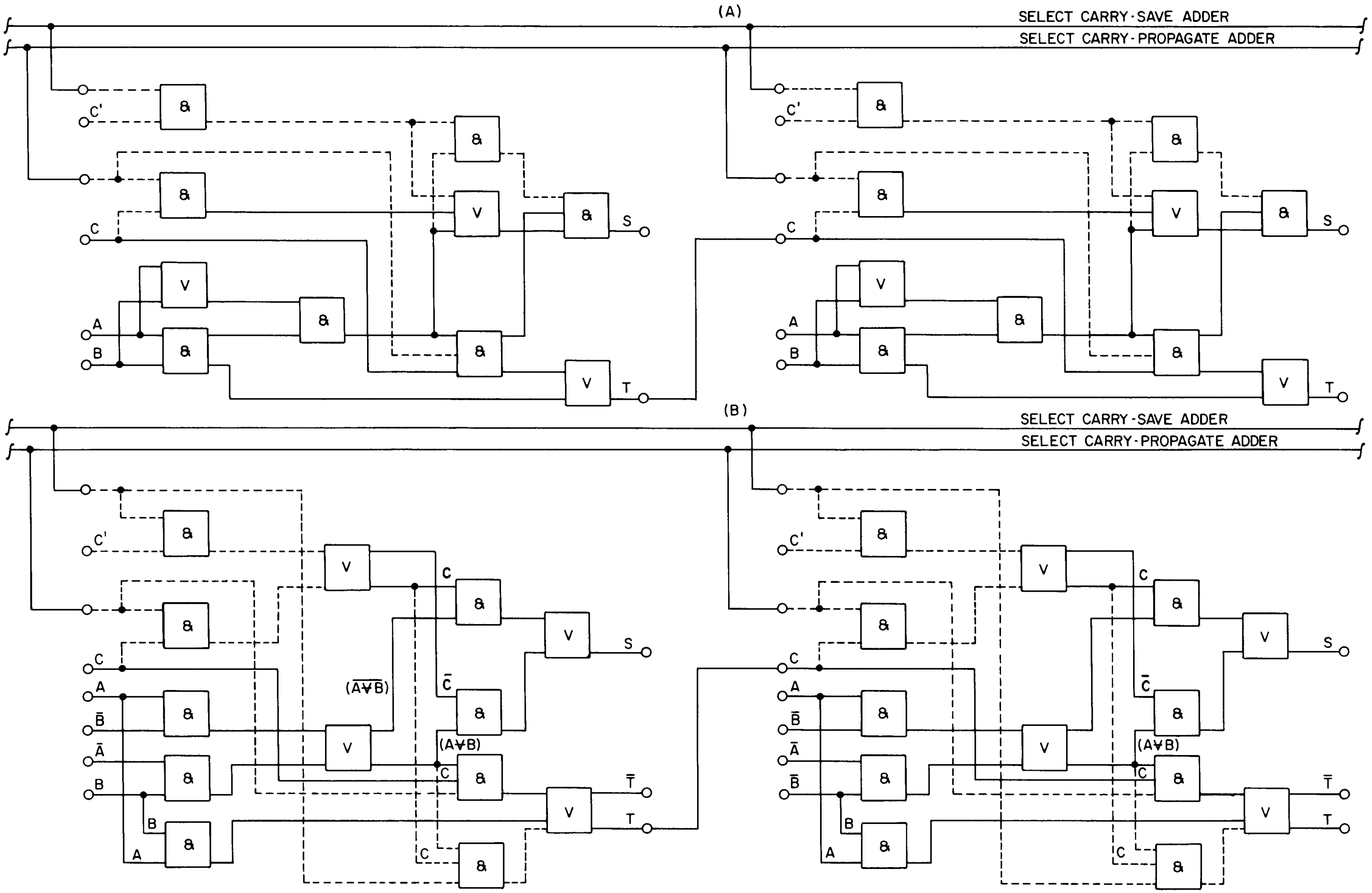


FIGURE 17. Carry-save/carry-propagate adders. Two versions of combined carry-save/carry-propagate adders are shown. Dotted lines are additional connections required to allow choice. Additional circuits do not increase the carry path when in the CPA mode. Carry path may be same as sum path in CSA mode. Separate carry input positions are provided for the two modes, but the same carry output terminal (T) is used for both.

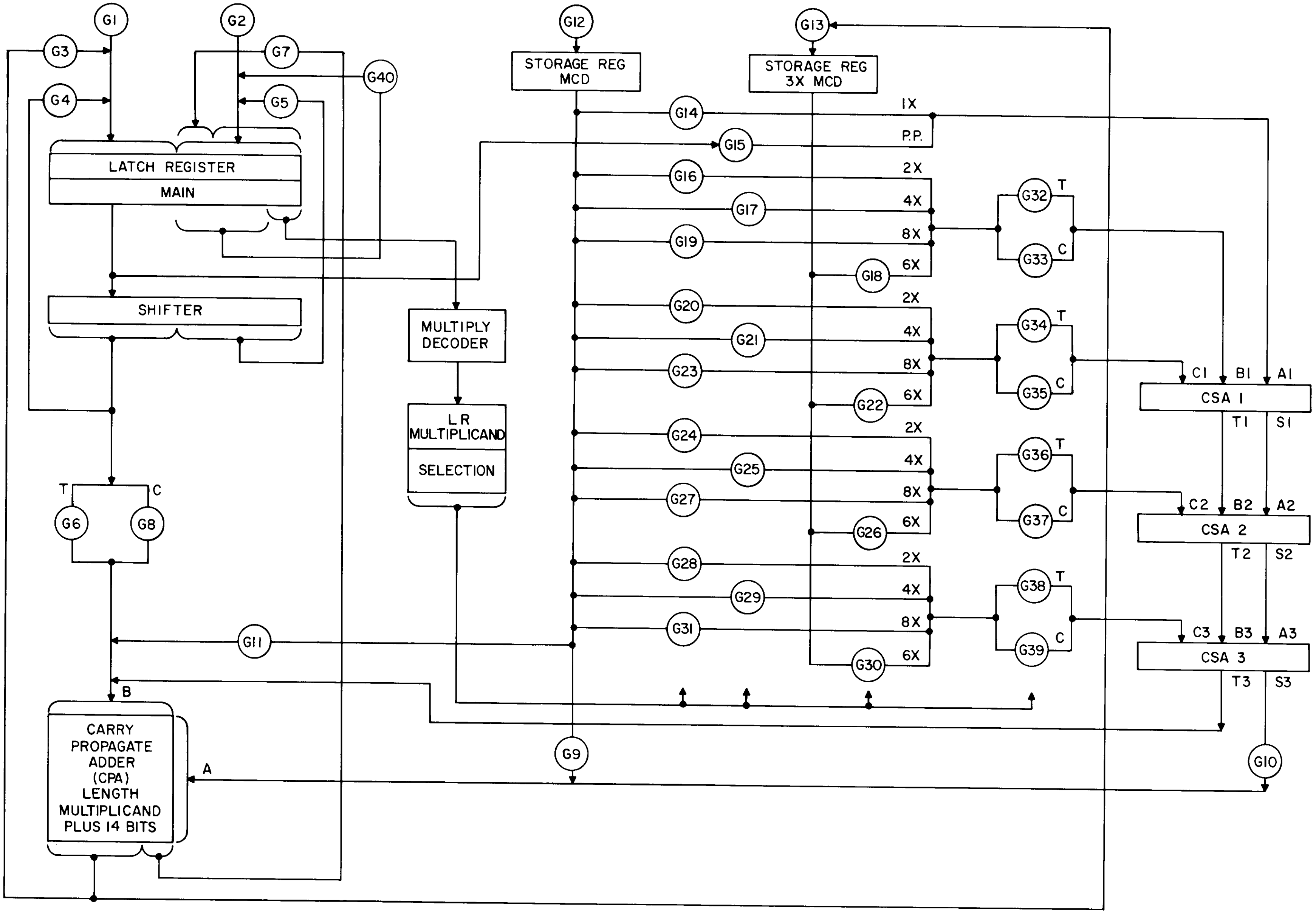


FIGURE 18. Multiplication system using carry-save adders and handling 12 multiplier bits per cycle.

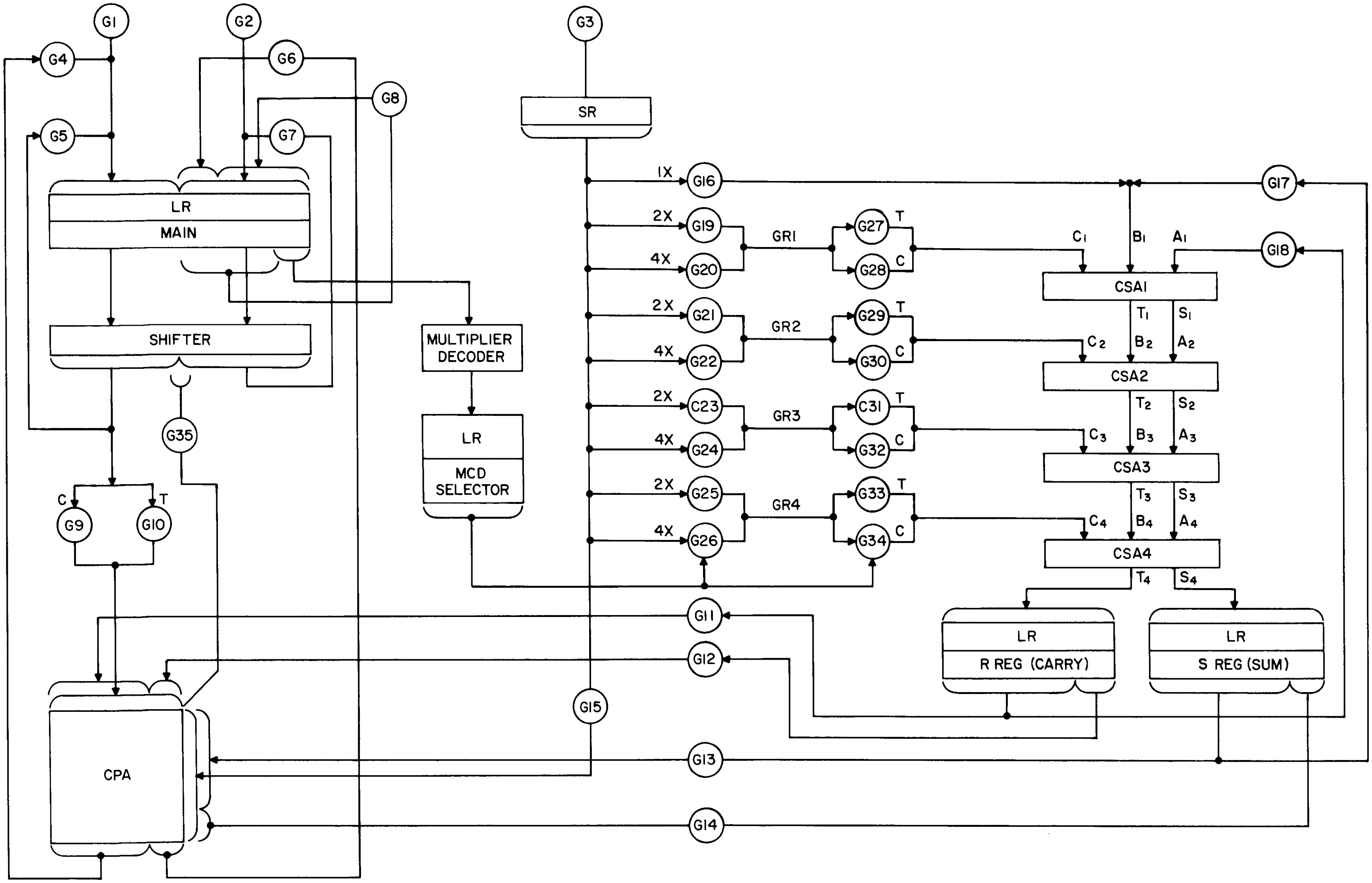
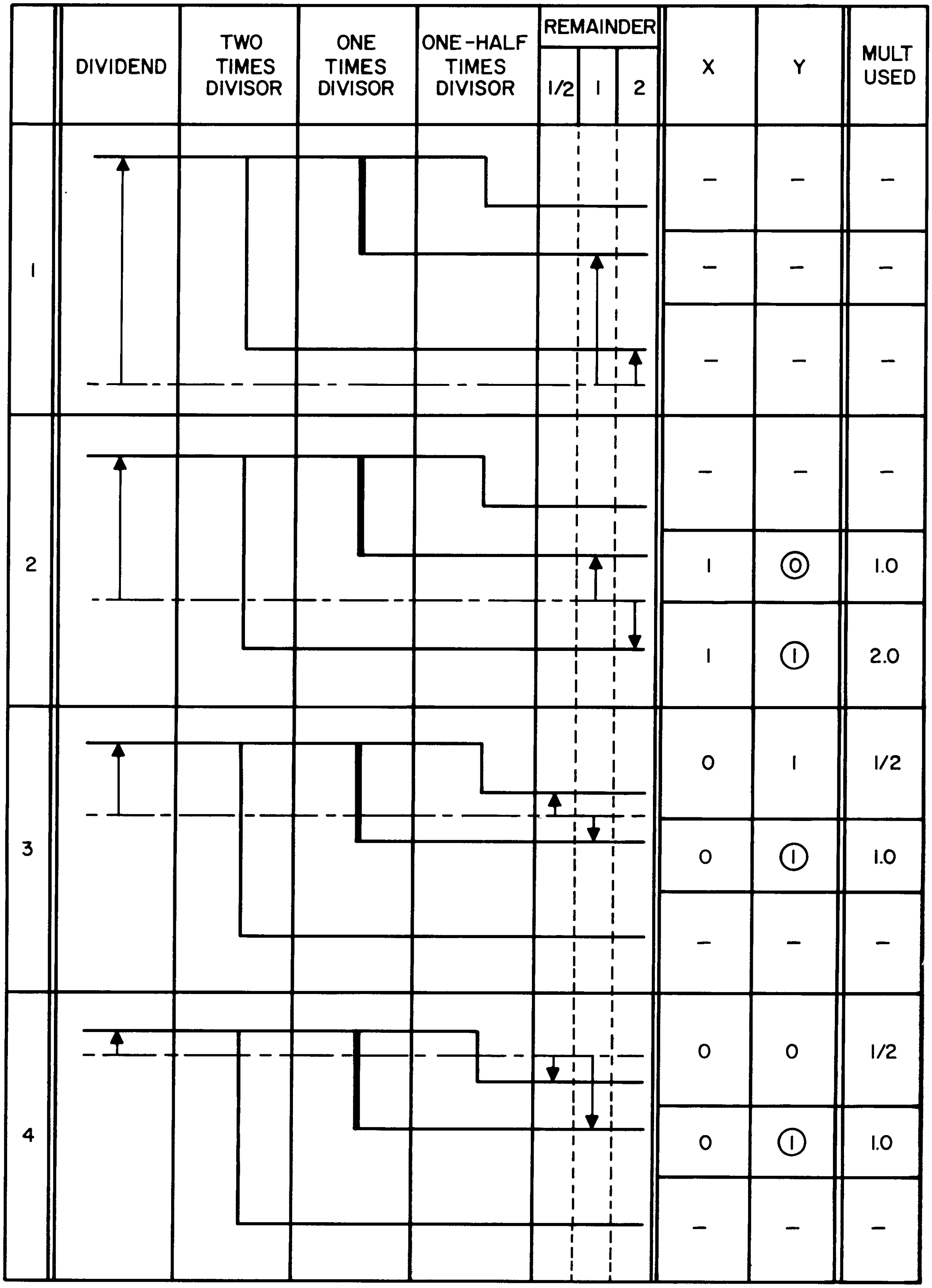
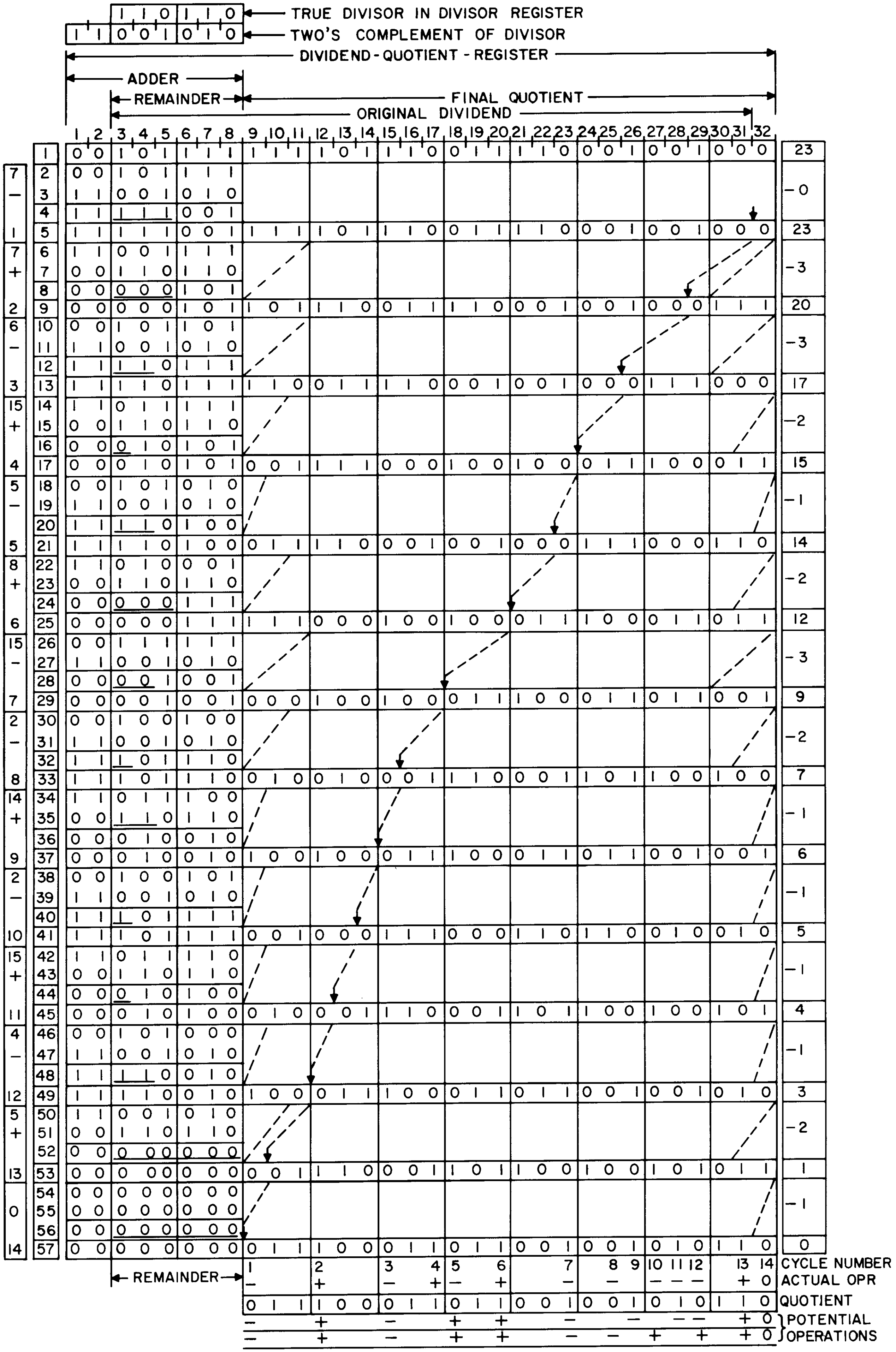


FIGURE 19. Multiplication system using carry-save adders and sum and carry registers.



	X	Y	MULT USED
1	–	–	–
	–	–	–
	–	–	–
2	–	–	–
	1	⓪	1.0
	1	①	2.0
3	0	1	1/2
	0	①	1.0
	–	–	–
4	0	0	1/2
	0	①	1.0
	–	–	–

FIGURE 20. Quotient development using 1/2, 1.0, 2.0 times divisor.


1 1 0 1 1 0 TRUE DIVISOR IN DIVISOR REGISTER
1 1 0 0 1 0 1 0 TWO'S COMPLEMENT OF DIVISOR
DIVIDEND-QUOTIENT-REGISTER
ADDER
REMAINDER
FINAL QUOTIENT
ORIGINAL DIVIDEND
REMAINDER
CYCLE NUMBER
ACTUAL OPR
QUOTIENT
POTENTIAL OPERATIONS

		1 2	3 4 5	6 7 8	9 10 11	12 13 14	15 16 17	18 19 20	21 22 23	24 25 26	27 28 29	30 31 32	
	1	0 0	1 0 1	1 1 1	1 1 1	1 0 1	1 1 0	0 1 1	1 1 0	0 0 1	0 0 1	0 0 0	23
7	2	0 0	1 0 1	1 1 1									−0
−	3	1 1	0 0 1	0 1 0									
	4	1 1	1 1 1	0 0 1									
1	5	1 1	1 1 1	0 0 1	1 1 1	1 0 1	1 1 0	0 1 1	1 1 0	0 0 1	0 0 1	0 0 0	23
7	6	1 1	0 0 1	1 1 1									−3
+	7	0 0	1 1 0	1 1 0									
	8	0 0	0 0 0	1 0 1									
2	9	0 0	0 0 0	1 0 1	1 0 1	1 1 0	0 1 1	1 1 0	0 0 1	0 0 1	0 0 0	1 1 1	20
6	10	0 0	1 0 1	1 0 1									−3
−	11	1 1	0 0 1	0 1 0									
	12	1 1	1 1 0	1 1 1									
3	13	1 1	1 1 0	1 1 1	1 1 0	0 1 1	1 1 0	0 0 1	0 0 1	0 0 0	1 1 1	0 0 0	17
15	14	1 1	0 1 1	1 1 1									−2
+	15	0 0	1 1 0	1 1 0									
	16	0 0	0 1 0	1 0 1									
4	17	0 0	0 1 0	1 0 1	0 0 1	1 1 1	0 0 0	1 0 0	1 0 0	0 1 1	1 0 0	0 1 1	15
5	18	0 0	1 0 1	0 1 0									−1
−	19	1 1	0 0 1	0 1 0									
	20	1 1	1 1 0	1 0 0									
5	21	1 1	1 1 0	1 0 0	0 1 1	1 1 0	0 0 1	0 0 1	0 0 0	1 1 1	0 0 0	1 1 0	14
8	22	1 1	0 1 0	0 0 1									−2
+	23	0 0	1 1 0	1 1 0									
	24	0 0	0 0 0	1 1 1									
6	25	0 0	0 0 0	1 1 1	1 1 1	0 0 0	1 0 0	1 0 0	0 1 1	1 0 0	0 1 1	0 1 1	12
15	26	0 0	1 1 1	1 1 1									−3
−	27	1 1	0 0 1	0 1 0									
	28	0 0	0 0 1	0 0 1									
7	29	0 0	0 0 1	0 0 1	0 0 0	1 0 0	1 0 0	0 1 1	1 0 0	0 1 1	0 1 1	0 0 1	9
2	30	0 0	1 0 0	1 0 0									−2
−	31	1 1	0 0 1	0 1 0									
	32	1 1	1 0 1	1 1 0									
8	33	1 1	1 0 1	1 1 0	0 1 0	0 1 0	0 0 1	1 1 0	0 0 1	1 0 1	1 0 0	1 0 0	7
14	34	1 1	0 1 1	1 0 0									−1
+	35	0 0	1 1 0	1 1 0									
	36	0 0	0 1 0	0 1 0									
9	37	0 0	0 1 0	0 1 0	1 0 0	1 0 0	0 1 1	1 0 0	0 1 1	0 1 1	0 0 1	0 0 1	6
2	38	0 0	1 0 0	1 0 1									−1
−	39	1 1	0 0 1	0 1 0									
	40	1 1	1 0 1	1 1 1									
10	41	1 1	1 0 1	1 1 1	0 0 1	0 0 0	1 1 1	0 0 0	1 1 0	1 1 0	0 1 0	0 1 0	5
15	42	1 1	0 1 1	1 1 0									−1
+	43	0 0	1 1 0	1 1 0									
	44	0 0	0 1 0	1 0 0									
11	45	0 0	0 1 0	1 0 0	0 1 0	0 0 1	1 1 0	0 0 1	1 0 1	1 0 0	1 0 0	1 0 1	4
4	46	0 0	1 0 1	0 0 0									−1
−	47	1 1	0 0 1	0 1 0									
	48	1 1	1 1 0	0 1 0									
12	49	1 1	1 1 0	0 1 0	1 0 0	0 1 1	1 0 0	0 1 1	0 1 1	0 0 1	0 0 1	0 1 0	3
5	50	1 1	0 0 1	0 1 0									−2
+	51	0 0	1 1 0	1 1 0									
	52	0 0	0 0 0	0 0 0									
13	53	0 0	0 0 0	0 0 0	0 0 1	1 1 0	0 0 1	1 0 1	1 0 0	1 0 0	1 0 1	0 1 1	1
	54	0 0	0 0 0	0 0 0									−1
0	55	0 0	0 0 0	0 0 0									
	56	0 0	0 0 0	0 0 0									
14	57	0 0	0 0 0	0 0 0	0 1 1	1 0 0	0 1 1	0 1 1	0 0 1	0 0 1	0 1 0	1 1 0	0
					1	2	3 4	5 6	7	8 9	10 11 12	13 14	CYCLE NUMBER
					−	+	− +	− +	−	−	− − −	+ 0	ACTUAL OPR
					0 1 1	1 0 0	0 1 1	0 1 1	0 0 1	0 0 1	0 1 0	1 1 0	QUOTIENT
					−	+	−	+ +	−	−	− −	+ 0	POTENTIAL
					−	+	−	+ +	−	−	+ +	+ 0	OPERATIONS

FIGURE 21. Divide example using one times divisor with shifting across ones and zeros.

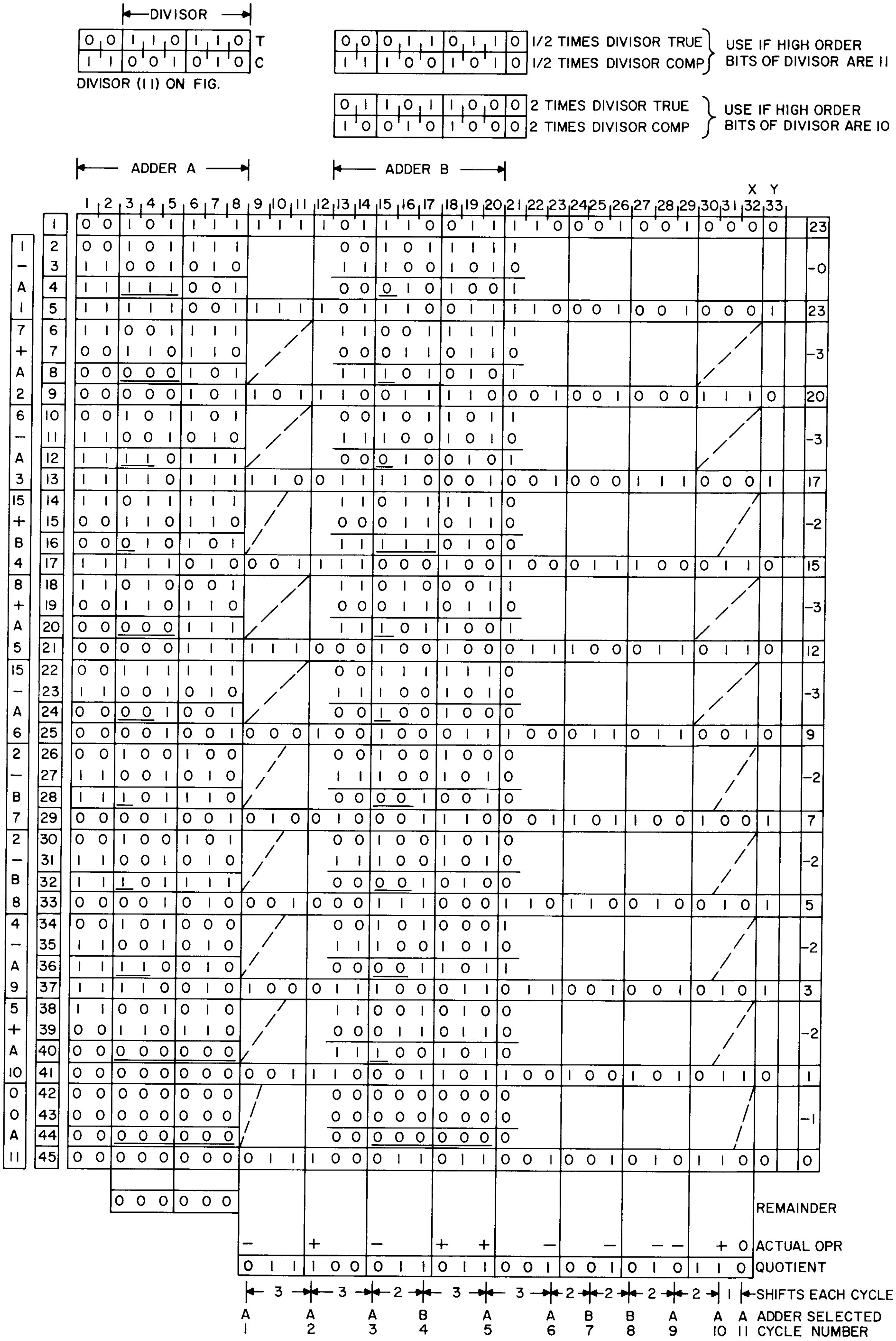


FIGURE 22. Divide example using 2.0, 1.0, 1/2 times divisor with double addition.

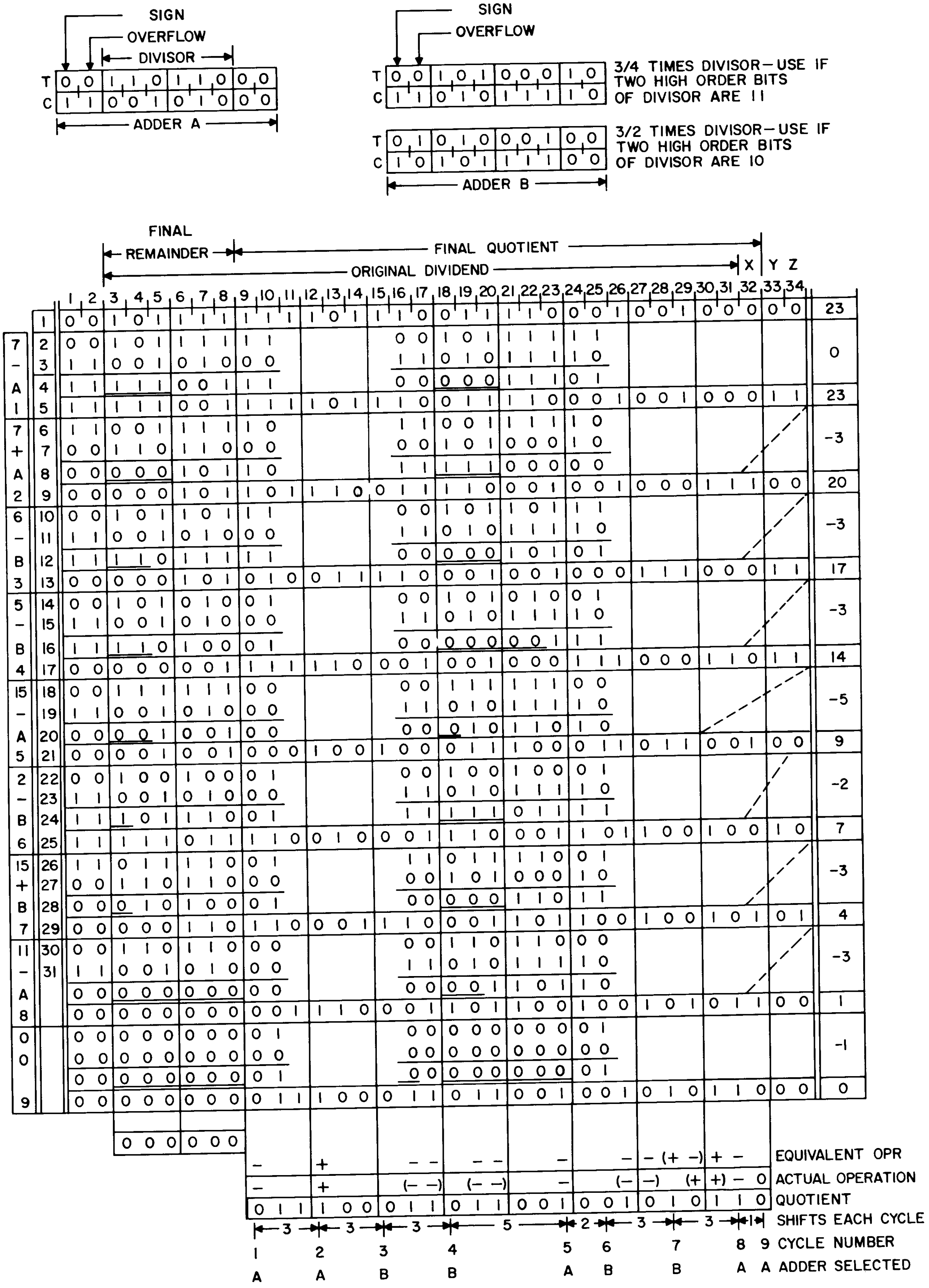


FIGURE 23. Divide example using 3/2, 1, 3/4 times divisor with double addition.

TO: Technical Report Distribution List

SUBJECT: Addendum to TR 00.740, "High-Speed Arithmetic in Binary Computers," by O. L. MacSorley

The attached is an addendum to the above Technical Report, offering further explanation of the Appendix section. Please insert this material in your copy or copies of Technical Report 00.740.

Laboratory Publications, Department 225
Product Development Laboratory
Data Systems Division, Poughkeepsie
October 19, 1960

ADDENDUM

HIGH-SPEED ARITHMETIC IN BINARY COMPUTERS

by

O. L. MacSorley
TR 00.740

In the Appendix to the above report is a proof of the statement that when using the variable shift method of multiplication (described in the report) with an infinite shifter and random bit distribution assumed in the multiplier, the average number of multiplier bits used per shift is three. Following this is a development of the modification required when the maximum allowable shift for any one operation is given a finite limit.

In the text, rules are given for two methods of variable shift multiplication. One assumes operation starting at the high-order end of the multiplier, the other at the low-order end of the multiplier. The proof given in the Appendix is based on the assumption that multiplication is performed starting at the high-order end.

The results obtained when starting from the low-order end of the multiplier are the same for an assumed infinite shifter. However, they vary somewhat when a finite limit is placed on the shifter, as shown in the following development.

Assume that a shift across zeros has just been completed. This means that the low-order bit of the multiplier is now a one. If the low-order one is followed by another one , a subtraction will be performed and the next shift will be across ones; if the low-order one is followed by a zero, an addition will be performed and the next shift will be across zeros. The size of the shift will be one more than the number of successive ones or zeros (whichever is being shifted across) following the low-order one. This rule never results in a shift of one being called for, as long as an infinite shifter is assumed (except possibly on the starting cycle).

The following table shows the eight possible combinations of the four low-order bits of the multiplier following a completed shift across zeros.

						Bits Shift	Shift Across
(1)	1	1	1	1	(0)	3+	1
(2)	0	1	1	1	(0)	3	1
(3)	1	0	1	1	(0)	2	1
(4)	0	0	1	1	(0)	2	1
(5)	1	1	0	1	(0)	2	0
(6)	0	1	0	1	(0)	2	0
(7)	1	0	0	1	(0)	3	0
(8)	0	0	0	1	(0)	3+	0

Summary:-	Shift	2	3	3+
.	Fraction	1/2	1/4	1/4

From this table it may be seen that one-half of the resulting shifts are across zeros and one-half are across ones. These groups in turn split into two groups each; one contains a known number of bits for each shift, the other contains shifts having a known minimum value but requiring the decoding of additional multiplier bits to determine the total value. The eight possible combinations of the next three multiplier bits are shown in the table below. The number of additional bits shift resulting from each combination is given for shifting across zeros and for shifting across ones. The fraction producing a particular shift is the same for both possibilities.

				Across Zeros	Across Ones
(1)	1	1	1	3+	1
(2)	0	1	1	3	1
(3)	1	0	1	2	1
(4)	0	0	1	2	1
(5)	1	1	0	1	2

				Across Zeros	Across Ones
(6)	0	1	0	1	2
(7)	1	0	0	1	3
(8)	0	0	0	1	3+

Summary:-	Additional Shifts	1	2	3	3+
	Fraction	1/2	1/4	1/8	1/8

From these two tables, a new table may be computed that will give the fraction of the total number of shifts made that should have each shift value when using a multiplier with infinite length and random bit distribution. The first table indicates that one-half of all of the shifts made will be across two bits, one-quarter of them will be across three bits, and one-quarter will be across more than three bits. The second table breaks down the last mentioned quarter (more than three bits). It shows that one-half of this quarter (one-eighth of the total) will be across four bits, one-quarter (one-sixteenth of the total) will be across five bits, one-eighth (one-thirtysecond of the total) across six bits, and one-eighth across more than six bits. This final thirty-second of the total may be split up according to the next three bits of the multiplier by again applying the second table in the same manner.

It is evident from the above that, for this method of decoding the multiplier, the fraction of the total shifts that will have any particular value is always one-half of the fraction having the next smaller shift value, to the limit that half of the shifts have a value of two and none of them have a value of one or zero. $\left[F = 1/(2^{n-1}) \text{ for } n > 1\right]$.

The preceding was developed on the assumption that a shift across zeros had just been completed. Had the initial assumption been that a shift across ones had just been completed, the results would have been identical, since the rules for handling ones and zeros are identical except that the locations of the words one and zero in the rules are interchanged.

To determine the average shift, start with an assumed number of shifts. From the results just developed compute the number of these that will have each shift value. Multiply the number of times each shift amount occurs by that shift amount, then add these products to determine the total number of positions shifted across. Divide the total number of shifts into the total number of positions shifted across to determine the average shift. It will be necessary to have one group which will include all shifts greater than some particular amount and assume some average value for these. The larger the initial number of shifts assumed, the smaller will be the potential error due to this. The example given below assumes a total of 8192 shifts.

SHIFT	FRACTION	NUMBER OF SHIFTS	NUMBER OF BITS
1		000	000
2	1/2	4,096	8,192
3	1/4	2,048	6,144
4	1/8	1,024	4,096
5	1/16	512	2,560
6	1/32	256	1,536
7	1/64	128	896
8	1/128	64	512
9	1/256	32	288
10	1/512	16	160
11	1/1024	8	88
12	1/2048	4	48
13	1/4096	2	26
14	1/8192	1	14
14+	1/8192	1	16 *
	1	8,192	24,576

This gives an average of three bits per shift cycle (24,576/8,192) when using a shifter that can supply any desired shift in one cycle. Usually a more limited shifter is used, and it is desired to know how the average shift size varies as the limit on the shifter size is varied. For example, if the maximum shift available is four, a desired shift of six would be performed as one shift of four followed by a shift of two; while a desired shift of thirteen would be three shifts of four followed by one shift of one. This results in an increased number of shifts for the same number of bits shifted across.

* Average value

The modified number of shift cycles resulting from any particular limitation on the shifter size may be easily determined from the preceding table in the following manner. Divide the table into groups, each group having a size equal to the maximum size of the shifter. For example, for a maximum shift of five, group 1 would contain shifts one through five, group 2 would contain shifts six through ten, group 3 would contain shifts eleven through fifteen, etc. All shifts in group 1 will still be performed in one shifting cycle each with the limited shifter, but those in group 2 will now require two cycles each, those in group 3 will require three cycles each, etc.

From the preceding table add all of the shifts contained in group 1 and multiply the result by one, add all of the shifts contained in group 2 and multiply the result by two, etc. Add the resulting products, then divide this into the total number of bits as determined in the table. This will give the modified bits per shift. An example of such a computation for a shifter with a maximum limit of three bits per cycle is shown below.

Group 1	000+	4096+	2048	6144	6144
Group 2	1024+	512+	256	1792	3584
Group 3	128+	64+	32	224	672
Group 4	16+	8+	4	28	112
Group 5	2+	1+	1	4	20
				8192	10532

Bits per shift cycle = (24, 576/10, 532) = 2. 32 for a maximum shift of 3.

The following are the average bits per shift for various limits of shifter size. These results were computed in the same manner as shown in the previous example.

Shifter Limit	Bits per Shift
3	2.32
4	2.65
5	2.82
6	2.90
7	2.95
8	2.97
Infinite	3.00

These figures do not include an adjustment for the fact that the multiplier has a finite length. Such an adjustment would cause a slight reduction in these values.

IBM®
Prepared by Laboratory Communications, Data Systems Division, Product Development Laboratory
International Business Machines Corporation, Poughkeepsie, New York
Printed in U. S. A. 1959